Statistical Language Models Using N-Grams for Syllables and Characters

Report on Turkish Language Modeling with N-Grams

Using Character and Syllable-Based Approaches

Kerem Göbekcioğlu

31 October 2024

Abstract

In this project, we developed two statistical language models for Turkish, each with a different focus: one using syllables and the other using individual characters as the basis for N-grams. N-gram language models predict the likelihood of a sequence of units—whether characters or syllables—based on the frequency of past occurrences of similar patterns. This helps us understand and generate text by learning from the structure and style of real language data.

The syllable-based model focuses on segmenting Turkish words into syllables, which captures more phonetic and linguistic detail and aligns with the natural syllabic structure of Turkish. The character-based model, on the other hand, looks at each letter individually, building a model from single letters, pairs of letters, and triplets. This approach has the advantage of working with finer details, although it may lack some of the language flow captured by syllable units.

The objectives of this project include preparing a Turkish dataset, building and storing N-gram tables efficiently, applying smoothing techniques, calculating perplexity, and generating random sentences to assess the language models' capabilities.

Contents

1 Introduction **2**

2 Design and Implementation **2**

1 Introduction

In this project, we aimed to construct statistical language models tailored to the Turkish language, focusing on two distinct approaches: character-based and syllable-based N-gram modeling. N-gram models predict the probability of a unit (character or syllable) based on its preceding sequence, effectively learning from language patterns.

The syllable-based model attempts to represent Turkish more naturally by using syllables as atomic units, aligning with the phonological structure of the language. This allows for better capture of rhythmic and grammatical tendencies. In contrast, the character-based model operates on a finer granularity by analyzing letters individually, enabling learning at a more basic level, though often lacking in capturing holistic language flow.

These models are trained on a cleaned Turkish Wikipedia dataset, where preprocessing steps include text normalization, character and syllable segmentation, and train-test splitting. Both models are evaluated using Good-Turing smoothing and perplexity calculation, and their performance is compared through the quality of generated sentences and statistical metrics.

2 Design and Implementation

Part 1: Data Preparation

1. **Dataset Cleaning and Converting to Lowercase**

This function, `clean_text`, reads a Turkish Wikipedia dump file line by line, cleans each line by removing XML tags, empty lines, and extra whitespace, converts the text to lowercase, and writes the cleaned output to a new file. This prepares the text for further processing by standardizing and removing unwanted content.

```python
def clean_text(input_file_path, output_file_path):
    with open(input_file_path, 'r', encoding='utf-8') as input_file:
        with open(output_file_path, 'w', encoding='utf-8') as output_file:
            for line in input_file:
                # Strip leading and trailing whitespace (including newlines)
                stripped_line = line.strip()

                # Skip lines starting with <doc> or </doc>, and empty lines
                if stripped_line.startswith('<doc>') or stripped_line.startswith('</doc>') or not stripped_line:
                    continue

                # Convert the line to lowercase
                lower_line = stripped_line.lower()

                # Write the cleaned line to the output file
                output_file.write(lower_line + '\n')

```

```
# Call the function with your file paths
input_file_path = "turkish_wikipedia_dump/wiki_00"
output_file_path = "cleaned_wiki_00.txt"
clean_text(input_file_path, output_file_path)
```

2. **Characterisation**

The split_text_to_characters function reads a cleaned text file, then splits each character with a space in between for easy tokenization, while preserving spaces between words. This prepares the data for character-based N-gram analysis by making each character distinct and sequential in the output file.

```
def split_text_to_characters(input_file_path, output_file_path):
    with open(input_file_path, 'r', encoding='utf-8') as
        input_file:
         text = input_file.read()

    with open(output_file_path, 'w', encoding='utf-8') as
        output_file:
         for char in text:
             if char != ' ':
                 output_file.write(char + ' ')
             else:
                 output_file.write(char)

# Example usage
input_file_path = 'cleaned_wiki_00.txt'
output_file_path = 'character_based_model.txt'

split_text_to_characters(input_file_path, output_file_path)
print("Character-based model created successfully.")
```

3. **Data Splitting**

The split_dataset function divides a dataset into training and testing sets based on a specified ratio (default is 95% training, 5% testing). The function reads the input file, shuffles the data to randomize it, and then splits it at the calculated index. The training and testing sets are saved into separate files, allowing the model to be trained and evaluated on distinct data portions.

```
import random

def split_dataset(input_file_path, train_file_path,
    test_file_path, train_ratio=0.95):
    # Read the processed data from the input file
    with open(input_file_path, 'r', encoding='utf-8') as file:
        lines = file.readlines()

    # Shuffle the lines to randomize the data (optional)
    random.shuffle(lines)

    # Calculate the split index
    split_index = int(len(lines) * train_ratio)

    # Split the data into training and testing sets
    train_data = lines[:split_index]
    test_data = lines[split_index:]

    # Save the training data to a new file
    with open(train_file_path, 'w', encoding='utf-8') as train_file:
        train_file.writelines(train_data)

    # Save the testing data to a new file
    with open(test_file_path, 'w', encoding='utf-8') as test_file:
        test_file.writelines(test_data)

    print(f"Data split complete. Training set: {len(train_data)} lines, Testing set: {len(test_data)} lines.")

# Example usage
input_file_path_1 = "syllable_based_model.txt"
train_file_path_1 = "syllable_based_model_train.txt"
test_file_path_1 = "syllable_based_model_test.txt"
split_dataset(input_file_path_1, train_file_path_1, test_file_path_1)

input_file_path_2 = "character_based_model.txt"
train_file_path_2 = "character_based_model_train.txt"
test_file_path_2 = "character_based_model_test.txt"
split_dataset(input_file_path_2, train_file_path_2, test_file_path_2)
```

Part 2: N-Gram Calculation

1. Building N-Gram Tables

a. Syllabification of the Dataset Characterization part was already done so before building N-Gram tables, dataset should be splitted into syllables. For this operation, I have found a code example which syllabifies Turkish words and treats non-Turkish words as Turkish words. Of course, if it is not Latin alphabet like Kiril or Arabic, it does not perform the operation since it does not recognize the characters. For punctuations, at the beginning I thought they were not necessary for sentence generation but then it glazed my mind without punctuation in written sentences, there could be a lot of misunderstanding. While we are speaking, we make emphasis so that the person we are speaking understand us correctly. Emphasis is made by punctuation in written documents, so I have treated them as a character also as a syllable, too. The code for syllabification was written by PHP and with ChatGPT's help, it has been transformed to Python code because my workspace was built on Python.

We have characterized and syllabified our dataset, then it has been splitted to 95% for train and 5% for testing. Now we can build our N-Gram tables.

```
import string
```

```python
import re

class Hececi:
    def __init__(self):
        self.sesliler = ["a", "e", "  ", "i", "o", "  ", "u", "  ", "A", "E", "I", "  ", "O", "  ", "U", "  "]
        self.punctuation = list(string.punctuation)  # List of standard punctuation marks

    def hecelere_ayir(self, text):
        kelime_dizisi = []  # List to store words, punctuation, and numbers

        tokens = re.findall(r'\w+|[^\w\s]', text, re.UNICODE)

        for token in tokens:
            if token in self.punctuation:
                kelime_dizisi.append(token)
            elif token.isdigit():
                kelime_dizisi.append(token)
            else:
                heceler = self._hecele(token)
                kelime_dizisi.append(" ".join(heceler))

        return " ".join(kelime_dizisi)

    def _hecele(self, sozcuk):
        heceler = []

        while len(sozcuk) > 0:
            if not self._sesli_var_mi(sozcuk):
                if len(heceler) > 0:
                    heceler[-1] += sozcuk
                else:
                    heceler.append(sozcuk)
                break

            en_sagdaki_sesli_konumu = self._en_sagdakini_bul(sozcuk)

            if en_sagdaki_sesli_konumu == 0 or self._sesli_mi(sozcuk[en_sagdaki_sesli_konumu - 1]):
                hece = sozcuk[en_sagdaki_sesli_konumu:]
            else:
                hece = sozcuk[en_sagdaki_sesli_konumu - 1:]

            heceler.insert(0, hece)
            sozcuk = sozcuk[:-len(hece)]

        return heceler

    def _en_sagdakini_bul(self, kelime):
        for i in range(len(kelime) - 1, -1, -1):
            if kelime[i] in self.sesliler:
                return i
        return -1

    def _sesli_mi(self, harf):
        return harf in self.sesliler

    def _sesli_var_mi(self, kelime):
        return any(self._sesli_mi(harf) for harf in kelime)
```

```python
# Function to read from file in chunks and write the syllable
   output incrementally
def process_text_in_chunks(input_file_path, output_file_path,
   chunk_size=100000):
    hececi = Hececi()
    word_counter = 0
    remainder = ""

    with open(input_file_path, 'r', encoding='utf-8') as
       input_file, open(output_file_path, 'w', encoding='utf-8')
       as output_file:
        while True:
            chunk = input_file.read(chunk_size)
            if not chunk:
                break

            chunk = remainder + chunk

            last_space = chunk.rfind(" ")
            if last_space != -1:
                remainder = chunk[last_space + 1:]
                chunk = chunk[:last_space]

            syllable_text = hececi.hecelere_ayir(chunk)
            output_file.write(syllable_text + "\n")

            word_count_in_chunk = len(chunk.split())
            word_counter += word_count_in_chunk

            if word_counter % 1000000 < word_count_in_chunk:
                print(f"Processed {word_counter} words/syllables
                   ...")

        if remainder:
            syllable_text = hececi.hecelere_ayir(remainder)
            output_file.write(syllable_text + "\n")
            word_counter += len(remainder.split())

        print(f"Finished processing. Total words/syllables
            processed: {word_counter}")

# Example usage
input_file_path = "cleaned_wiki_00.txt"
output_file_path = "syllable_based_model.txt"
process_text_in_chunks(input_file_path, output_file_path)
```

b. N-Gram Tables

In this part, functions will be analysed one by one and how an efficient storage method been achieved will be explained.

1. N-Gram Generation Function The `generate_ngrams` function is designed to create 1-gram, 2-gram, and 3-gram tables, which count the occurrences of consecutive tokens (either syllables or characters) within the dataset. To manage memory more effectively, `defaultdict` is used to store N-grams as keys and their counts as values. This data structure automatically initializes counts for new Ngrams to zero, providing a sparse storage solution that minimizes memory usage by only storing observed combinations. Additionally, tuples are used as dictionary keys, which provide a compact and efficient format for representing N-grams. The use of `tqdm` for progress tracking also offers real-time feedback on the progress of N-gram generation, which is particularly helpful when processing large datasets.

```python
def generate_ngrams(tokens, n):
    ngrams = defaultdict(int)  # Default dic to store N-Grams and
        their counts
    for i in tqdm(range(len(tokens) - n + 1), desc=f"Generating {
       n}-Grams"):
        ngram = tuple(tokens[i:i + n])  # Get the N-Gram as a
           tuple
        ngrams[ngram] += 1  # Increment the count for this N-Gram
    print(f"{n}-grams generated")
    return ngrams
```

2. Top N-Gram Finding To find the most frequent N-grams, the `get_top_n_ngrams` function identifies the top N entries without sorting the entire dataset. This function uses `heapq.nlargest`, which is optimized to select the largest items efficiently, focusing only on the most relevant data. By avoiding unnecessary sorting and computation, this approach reduces both memory and processing requirements. Extracting the most frequent N-grams is essential for identifying common patterns and making later stages of analysis more efficient and focused.

```python
def get_top_n_ngrams(ngrams, top_n=20):
    top_ngrams = heapq.nlargest(top_n, ngrams.items(), key=lambda
        x: x[1])
    return top_ngrams
```

3. JSON Serialization and Conversion The `convert_ngrams_to_list` function prepares the N-gram dictionary for JSON serialization by converting each N-gram and its count into a list format. This process ensures compatibility with JSON, allowing for lightweight and portable storage. Converting data in this way also supports efficient storage by retaining only observed N-grams, preventing the storage of empty or irrelevant data points.

```python
def convert_ngrams_to_list(ngrams):
    if isinstance(ngrams, dict):
        return [[list(key), value] for key, value in ngrams.items()]
    elif isinstance(ngrams, list):
        return [[list(key), value] for key, value in ngrams if isinstance(key, tuple) and isinstance(value, int)]
    else:
        raise ValueError("Expected a dictionary or a list of tuples")
```

4. N-Gram Processing The function `process_syllable_based` and `process_character_based` processes both syllable and character based text, generating N-gram tables for 1-grams, 2-grams, and 3-grams. For each N-gram level, the function captures the top 20 most frequent entries and returns them separately. This separation allows the program to store each N-gram's top N, making each most used N-Grams easy to read and load independently. This design minimizes memory use by allowing access to specific N-gram levels without overloading memory resources.

```python
def process_syllable_based(file_path, top_n=20):
    with open(file_path, 'r', encoding='utf-8') as f:
        text = f.read().strip()

    syllables = text.split()
    unigrams = generate_ngrams(syllables, 1)
    bigrams = generate_ngrams(syllables, 2)
    trigrams = generate_ngrams(syllables, 3)

    top_20_unigrams = get_top_n_ngrams(unigrams, 20)
    top_20_bigrams = get_top_n_ngrams(bigrams, 20)
    top_20_trigrams = get_top_n_ngrams(trigrams, 20)

    return top_20_unigrams, top_20_bigrams, top_20_trigrams, unigrams, bigrams, trigrams
```

```python
def process_character_based(file_path):
    with open(file_path, 'r', encoding='utf-8') as f:
        text = f.read().strip()
    print("text read")

    characters = text
    unigrams = generate_ngrams(characters, 1)
    print("character uni")
    bigrams = generate_ngrams(characters, 2)
    print("character bi")
    trigrams = generate_ngrams(characters, 3)
    print("character tri")

    top_20_unigrams = get_top_n_ngrams(unigrams, 20)
    top_20_bigrams = get_top_n_ngrams(bigrams, 20)
    top_20_trigrams = get_top_n_ngrams(trigrams, 20)

    return top_20_unigrams, top_20_bigrams, top_20_trigrams,
       unigrams, bigrams, trigrams
```

5. JSON File Creation JSON files are used to store N-Grams because they share same structure with dictionaries. This way made sense to me so that I choose it. Accessing and reading N-Grams from a JSON file is much more easier than a regular text file. For syllable-based models, since that NGram datas are so large I have used separate files for each N-Gram.

```
output_file_path_2 = "character_n_grams_test.json"
with open(output_file_path_2, 'w', encoding='utf-8') as
   output_file:
    json.dump({
        "top_20_unigrams": convert_ngrams_to_list(top_20_unigrams
           ),
        "unigrams": convert_ngrams_to_list(char_unigrams),
        "top_20_bigrams": convert_ngrams_to_list(top_20_bigrams),
        "bigrams": convert_ngrams_to_list(char_bigrams),
        "top_20_trigrams": convert_ngrams_to_list(top_20_trigrams
           ),
        "trigrams": convert_ngrams_to_list(char_trigrams)
    }, output_file, ensure_ascii=False, indent=4)
print("N-Gram generation completed and saved to file.")
```

```
top_20_unigrams, top_20_bigrams, top_20_trigrams,
   syllable_unigrams, syllable_bigrams, syllable_trigrams =
   process_syllable_based(syllable_file)

output_prefix = "syllable_test_data"  # Prefix for the output
   files
with open(f"{output_prefix}.json", 'w', encoding='utf-8') as
   output_file:
    json.dump({
        "top_20_unigrams": convert_ngrams_to_list(top_20_unigrams
           ),
        "unigrams": convert_ngrams_to_list(syllable_unigrams)
    }, output_file, ensure_ascii=False, indent=4)

with open(f"{output_prefix}_bigrams.json", 'w', encoding='utf-8')
    as output_file:
    json.dump({
        "top_20_bigrams": convert_ngrams_to_list(top_20_bigrams),
        "bigrams": convert_ngrams_to_list(syllable_bigrams)
```

```python
    }, output_file, ensure_ascii=False, indent=4)

with open(f"{output_prefix}_trigrams.json", 'w', encoding='utf-8') as output_file:
    json.dump({
        "top_20_trigrams": convert_ngrams_to_list(top_20_trigrams),
        "trigrams": convert_ngrams_to_list(syllable_trigrams)
    }, output_file, ensure_ascii=False, indent=4)
```

In summary, this code achieves efficient storage through multiple methods: `defaultdict` structures help reduce memory by focusing only on observed N-grams; tuple keys are used for N-grams, providing a compact way to represent N-gram sequences; and JSON output keeps the storage organized and accessible. By splitting the data into separate files for each N-gram level, the code minimizes memory usage and enables selective access to specific N-gram levels. This storage method strikes a balance between handling large-scale data and the constraints of available memory, enabling effective N-gram analysis on sizable datasets.

c. Counting Characters and Syllables

This code builds and tests N-Gram models by generating, counting, and evaluating the top N-Grams from text data. It processes data into syllable-based and character-based models separately, calculates 1-Gram, 2-Gram, and 3-Gram counts, and stores the results. The code starts by reading from a specified file for each model type (syllable or character) and uses the `generate_ngrams` function to create N-Grams. These N-Grams are then stored in a dictionary, allowing efficient counting and retrieval.

To display progress, the `tqdm` library is used, showing the process of generating unigrams, bigrams, and trigrams for each N-Gram type. After generating N-Grams, the code sorts and selects the top 10 most frequent N-Grams, ensuring only the most common patterns are stored. This makes it easier to identify high-frequency patterns and validate that N-Gram generation has been correctly implemented. Additionally, the code counts how many N-Grams appear at least five times, as well as how many unique N-Grams (those that appear only once) are in the text.

The output of each function includes the top N-Grams for both syllable-based and character-based models, along with various statistics such as counts of frequently occurring and unique N-Grams. This structure allows testing to verify that N-Gram sorting is accurate and that the model accurately captures the most common N-Grams across both syllables and characters. Some of the functions that used here will be explained in next part of the report.

```python
import json
from collections import defaultdict
from tqdm import tqdm  # Progress bar

def generate_ngrams(tokens, n):
    ngrams = defaultdict(int)  # Dictionary to store N-Grams and their counts
    for i in tqdm(range(len(tokens) - n + 1), desc=f"Generating {n}-Grams"):
        ngram = tuple(tokens[i:i + n])  # Get the N-Gram as a tuple
        ngrams[ngram] += 1  # Increment the count for this N-Gram
    print(f"{n}-grams generated")
    return ngrams

# Function to get the top N most frequent N-Grams
def get_top_n_ngrams(ngrams, top_n=10):
    return sorted(ngrams.items(), key=lambda x: x[1], reverse=True)[:top_n]

# Function to count N-Grams that appear at least min_count times
def count_ngrams_with_min_count(ngrams, min_count=5):
    return sum(1 for count in ngrams.values() if count >= min_count)

# Function to count unique N-Grams (appear only once)
def count_unique_ngrams(ngrams):
    return sum(1 for count in ngrams.values() if count == 1)

# Convert the N-Gram dictionary into a list of lists for JSON serialization
def convert_ngrams_to_list(ngrams):
    return [[list(key), value] for key, value in ngrams.items()]
```

```python
def process_character_based(file_path):
    with open(file_path, 'r', encoding='utf-8') as f:
        text = f.read().strip()
    characters = text  # Characters are not splitted because they had been already splitted
    print("characters generated")

    unigrams = generate_ngrams(characters, 1)
    bigrams = generate_ngrams(characters, 2)
    trigrams = generate_ngrams(characters, 3)

    return unigrams, bigrams, trigrams
```

```python
def process_syllable_based(file_path):
    with open(file_path, 'r', encoding='utf-8') as f:
        text = f.read().strip()  # Read the file content

    syllables = text.split()  # This ensures that we're counting syllables, not characters
    print("syllables generated")

    unigrams = generate_ngrams(syllables, 1)
    bigrams = generate_ngrams(syllables, 2)
    trigrams = generate_ngrams(syllables, 3)

    return unigrams, bigrams, trigrams
```

2. Smoothing

The Good-Turing smoothing technique addresses how we handle words or N-grams that appear very infrequently or have not appeared at all in the training data. The process begins by calculating what's called the "frequency of frequencies," where we determine how many N-grams occur once, twice, and so on. This helps us redistribute some probability from the more frequent N-grams to account for the unobserved ones.

For each N-gram, this adjusted count replaces the original to create a smoother probability distribution across all N-grams. After adjusting, we calculate each N-gram's probability by dividing the adjusted count by the total tokens in the dataset. To account for N-grams not present in our training data, we use the unseen probability where N_1 represents N-grams that appear exactly once. This "unseen" probability prevents the model from giving a probability of zero to new sequences, making it more adaptable and realistic for real-world use.

This method ensures a balanced distribution, supporting both common and rare language patterns effectively.

Good-Turing Smoothing Equations

1. **Adjusted Count for Observed N-Grams:**

$$P(r) = (r+1) \cdot \left(\frac{N_{r+1}}{N_r} \right)$$

where:

- r is the observed frequency of an N-gram,
- N_r is the number of N-grams that appear r times,
- N_{r+1} is the number of N-grams that appear $r+1$ times.

2. **Probability Calculation:**

$$P(\text{N-gram}) = \frac{\text{Adjusted Count of N-gram}}{\text{Total Tokens}}$$

3. **Unseen N-Gram Probability:**

$$P(\text{unseen}) = \frac{N_1}{\text{Total Tokens}}$$

where N_1 is the number of N-grams that appear exactly once.

Good-Turing Python Implementation

```python
def good_turing_smoothing(ngrams):
    # Frequency of Frequencies
    frequency_of_frequencies = defaultdict(int)
    for count in ngrams.values():
        frequency_of_frequencies[count] += 1

    # Applying Good-Turing adjustment
    adjusted_counts = {}
    total_tokens = sum(ngrams.values())

    for ngram, count in ngrams.items():
        if count + 1 in frequency_of_frequencies:
            # Applying Good-Turing adjustment for observed N-
               Grams
            adjusted_count = (count + 1) * (
                frequency_of_frequencies[count + 1] /
                   frequency_of_frequencies[count]
            )
        else:
            # If no N_{r+1}, use the original count
            adjusted_count = count
        adjusted_counts[ngram] = adjusted_count

    probabilities = {}
    for ngram, adjusted_count in adjusted_counts.items():
        probabilities[ngram] = adjusted_count / total_tokens

    # Probability for unseen N-Grams
    unseen_probability = (
        frequency_of_frequencies[1] / total_tokens
        if 1 in frequency_of_frequencies
        else 0
    )

    return probabilities, unseen_probability
```

3. Perplexity Calculation

In calculating perplexity for an N-gram model, this function evaluates how well the model predicts a sequence of text. Here's how it works:

The function iterates through the test data to check each N-gram's probability. If the N-gram is not found in the training data, we use a small unseen probability, determined by smoothing techniques like Good-Turing. By taking the log of each probability and summing them up, we can calculate an average log probability. Then, we exponentiate this value to the base of 2 to get a final perplexity score, which reflects the model's predictive accuracy: lower perplexity means better predictions.

This function captures how well the language model aligns with real data by considering both seen and unseen sequences.

Mathematical Formulation

Perplexity is defined as the inverse probability of the test set, normalized by the number of words:

$$PP(W) = P(w_1 w_2 \dots w_N)^{-\frac{1}{N}} = \left(\prod_{i=1}^{N} \frac{1}{P(w_i \mid w_{i-1}, \dots, w_{i-n+1})} \right)^{\frac{1}{N}}$$

Minimizing perplexity is equivalent to maximizing the model's probability over the test set.

Python Implementation

```python
def calculate_perplexity(test_data, ngram_probabilities,
   unseen_probability, n):
    log_prob_sum = 0
    N = len(test_data)
    print("calculating perplexity")

    for i in range(N - n + 1):
        # Get the current n-gram from the test data
        current_ngram = tuple(test_data[i:i + n])

        # Get the probability from the model or use the unseen
           probability
        probability = ngram_probabilities.get(current_ngram,
           unseen_probability)

        # Add the log probability to the sum
        if probability > 0:
            log_prob_sum += math.log2(probability)
        else:
            log_prob_sum += math.log2(unseen_probability)

    # Calculating perplexity
    perplexity = 2 ** (-log_prob_sum / (N - n + 1))
    return perplexity
```

Part 3: Random Sentence Generation

The `generate_random_sentence_with_smoothing` function creates random sentences by using the n-grams generated from a training dataset and applying Good-Turing smoothing. Starting with a random n-gram, it progressively builds a sentence by selecting new n-grams that continue from the last part of the current sentence. The function attempts to use the top 5 highest-probability n-grams to form logical continuations of the sentence. If there are no fitting candidates in the top 5, it falls back on a random n-gram, weighted by the unseen probability calculated through smoothing. This method prioritizes frequent n-grams while allowing flexibility with unseen options when needed, making sentences both coherent and somewhat creative. However, by focusing heavily on smoothing, it can sometimes produce sentences that are disjointed if high-probability sequences aren't available.

Function 1: With Good-Turing Smoothing

```python
def generate_random_sentence_with_smoothing(ngram_probabilities,
   unseen_probability, n, sentence_length=10):
    sentence = []

    # Start with a random N-Gram from the model
    current_ngram = random.choice(list(ngram_probabilities.keys()
        ))
    sentence.extend(current_ngram)

    while len(sentence) < sentence_length:
        # Filter to get only N-Grams that continue from the last
            part of the sentence
        candidates = [
            (ngram, prob) for ngram, prob in ngram_probabilities.
                items()
            if ngram[:n - 1] == tuple(sentence[-(n - 1):])
        ]

        # Use the top 5 candidates based on their probabilities
        if candidates:
            top_candidates = sorted(candidates, key=lambda x: x
                [1], reverse=True)[:5]
            next_ngram = random.choices(
                [ngram for ngram, _ in top_candidates],
                weights=[prob for _, prob in top_candidates]
            )[0]
        else:
            next_ngram = random.choices(
                list(ngram_probabilities.keys()),
                weights=[unseen_probability] * len(
                    ngram_probabilities)
            )[0]

        sentence.append(next_ngram[-1])

    return " ".join(sentence)
```

Function 2: With Top-5 N-Gram Filtering

The `generate_random_sentence_with_top_5` function, in contrast, uses only the top 5 ngrams for each context when building sentences. The function first organizes the n-grams into a dictionary that lists the top 5 options based on each context (or prefix). This setup helps the function quickly access the most likely next words, ensuring that each choice closely aligns with the most probable sequences in the dataset. In cases where no top 5 ngrams fit the current context, the function randomly selects a new context to proceed. This approach maintains a higher degree of fluency since it heavily favors the most probable ngrams. However, it restricts creative randomness as it doesn't consider unseen n-grams unless forced to change contexts.

```python
def get_top_5_ngrams(ngram_probabilities):
    top_5_ngrams = {}
    for ngram, prob in ngram_probabilities.items():
        prefix = ngram[:-1]
        if prefix not in top_5_ngrams:
            top_5_ngrams[prefix] = []
        top_5_ngrams[prefix].append((ngram, prob))
        top_5_ngrams[prefix] = sorted(top_5_ngrams[prefix], key=
            lambda x: x[1], reverse=True)[:5]
    return top_5_ngrams

def generate_random_sentence_with_top_5(ngram_probabilities,
    unseen_probability, n, sentence_length=10):
    top_5_ngrams = get_top_5_ngrams(ngram_probabilities)
    sentence = []

    current_ngram = random.choice(list(ngram_probabilities.keys()
        ))
    sentence.extend(current_ngram)

    while len(sentence) < sentence_length:
        context = tuple(sentence[-(n - 1):])
        candidates = top_5_ngrams.get(context, [])

        if candidates:
            next_ngram = random.choices(
                [ngram for ngram, _ in candidates],
                weights=[prob for _, prob in candidates]
            )[0]
        else:
            new_context = random.choice(list(top_5_ngrams.keys())
                )
            next_ngram = random.choice(top_5_ngrams[new_context])
                [0]

        sentence.append(next_ngram[-1])

    return " ".join(sentence)
```

Function 3: Hybrid Method

The `generate_random_sentence_hybrid` function combines aspects of both smoothing and the top 5 n-gram approach. It starts with a context-driven selection but includes an unseen probability in its weighting to introduce more randomness. If the unseen probability is chosen, the function uses a completely random n-gram as the next word. By mixing the top 5 method with the unseen probability option, this hybrid approach ensures that the generated sentences can flow smoothly while also bringing in new and unexpected elements, even when frequent n-grams dominate the data. It strikes a balance between probability-driven structure and the variety provided by Good-Turing smoothing, creating sentences that feel both coherent and novel.

```python
def generate_random_sentence_hybrid(ngram_probabilities,
    unseen_probability, n, sentence_length=10):
```

```
    top_5_ngrams = get_top_5_ngrams(ngram_probabilities)
    sentence = []

    current_ngram = random.choice(list(ngram_probabilities.keys()
       ))
    sentence.extend(current_ngram)

    while len(sentence) < sentence_length:
        context = tuple(sentence[-(n - 1):])
        candidates = top_5_ngrams.get(context, [])

        if candidates:
            candidate_ngrams = [ngram for ngram, _ in candidates]
                + [None]
            weights = [prob for _, prob in candidates] + [
               unseen_probability]
            next_ngram = random.choices(candidate_ngrams, weights
               =weights)[0]

            if next_ngram is None:
                next_ngram = random.choice(list(
                   ngram_probabilities.keys()))
        else:
            next_ngram = random.choice(list(ngram_probabilities.
               keys()))

        sentence.append(next_ngram[-1])

    return " ".join(sentence)
```

Each approach brings trade-offs: the smoothing method promotes randomness but can lack cohesion; the top 5 method prioritizes fluency but may sound repetitive; and the hybrid model balances predictability with variability, making it the most flexible option for sentence generation. The use of all three approaches offers a comprehensive view of sentence generation strategies based on probability and context, each model revealing different strengths in terms of coherence, creativity, and adherence to the most probable structures in the dataset.

Results and Tables

Unseen Probabilities

Model	**Unseen Probability**
Syllable Unigram	0.0005530626502600843
Syllable Bigram	0.008692685027445477
Syllable Trigram	0.055009020202446836
Character Unigram	2.1472796057755156e-06
Character Bigram	4.295982199510243e-06
Character Trigram	3.458557386806077e-05

Table 1: Unseen N-Gram Probabilities for Syllable-based and Character-based Models

Perplexity Values

Model	**Perplexity**
Character Unigram	9.259611117781004
Character Bigram	232775.63745403048
Character Trigram	28913.789669177255
Syllable Unigram	467.7130390529446
Syllable Bigram	115.03925399539486
Syllable Trigram	18.178836782124364

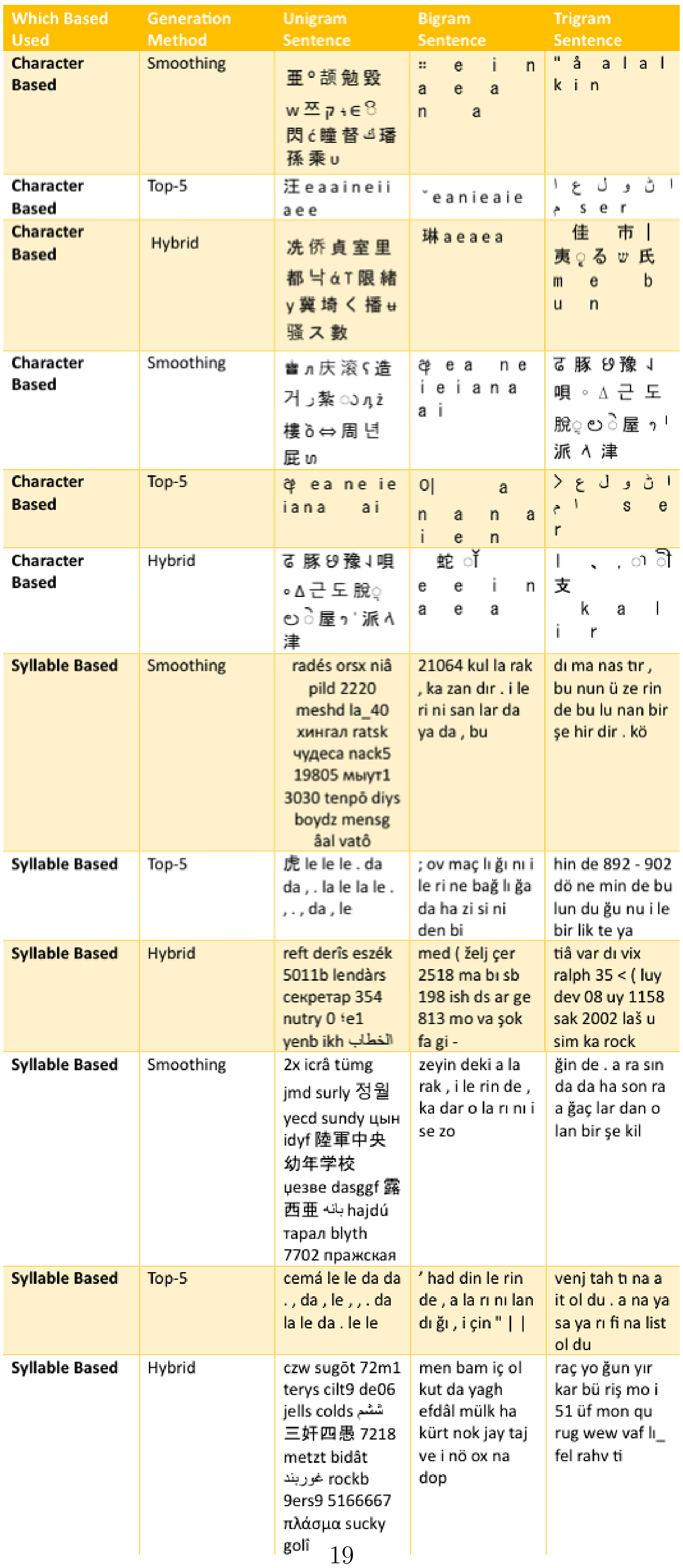
Table 2: Perplexity Values for Different N-Gram Levels

The results demonstrate clear distinctions in how well each model handles Turkish language data. The character-based model shows a low perplexity at the unigram level (9.26), while the syllable-based model is much higher (467.71). This suggests that, for predicting individual characters, the character model has a better fit with the test data.

However, as we look at the bigram and trigram levels, the syllable-based model performs significantly better. For bigrams, the syllable-based model has a perplexity of 115.04 compared to the character-based model's 232,775.64, and for trigrams, the syllable-based perplexity drops to 18.18, while the character-based perplexity remains much higher at 28,913.79.

These results indicate that syllable-based models capture meaningful context in Turkish, especially at the bigram and trigram levels. Turkish words are often built around consistent syllable patterns, which the syllable-based model successfully learns. The character-based model, in contrast, struggles as it tries to predict dependencies across multiple characters, showing high perplexity scores that indicate it has difficulty with longer character sequences.

Overall, these findings suggest that syllable-based bigrams and trigrams are more effective for Turkish language modeling, as they provide clearer patterns for prediction and produce more cohesive sentence structures.



Which Based Used	Generation Method	Unigram Sentence	Bigram Sentence	Trigram Sentence
Character Based	Smoothing	亜ᵒ頡勉毀 w쯔ק ᵻ∈ꝸ 閃¿瞳督ᐊ璠 孫乗ᴜ	∷ e i n a e a n a	" å a l a l k i n
Character Based	Top-5	汪eaaineii aee	ˇeanieaie	ا ع ل و ڽ ا م s e r
Character Based	Hybrid	冼侨貞室里 都낙άт限緒 y冀埼く播ᵾ 骚ス數	琳aeaea	佳 市 | 夷ួる ಲ 氏 m e b u n
Character Based	Smoothing	ᄈл庆滚ʕ造 거ᓗ紮ᦃ ɲż 楼ò⇔周 넌 屁ഗ	ಥ e a n e i e i a n a a i	ᄒ 豚 ꓘ 豫 ᑊ 唄 ◦ ᐃ 근 도 脫ួ ᧐ ᳵ 屋 ᥕ ˡ 派 ᐱ 津
Character Based	Top-5	ಥ ea ne ie iana ai	이 a n a n a i e n	> ع ل و ڽ ا م ا s e r
Character Based	Hybrid	ᄒ豚ꓘ豫ᑊ唄 ◦ᐃ근도脫ួ ᧐ᳵ屋ᥕ'派ᐱ 津	蛇 ়ў e e i n a e a	l ، , ়ा ়ी 支 k a l i r
Syllable Based	Smoothing	radés orsx niâ pild 2220 meshd la_40 хинган ratsk чудеса nack5 19805 мыут1 3030 tenpō diys boydz mensg åal vatô	21064 kul la rak , ka zan dır . i le ri ni san lar da ya da , bu	dı ma nas tır , bu nun ü ze rin de bu lu nan bir şe hir dir . kö
Syllable Based	Top-5	虎 le le le . da da , . la le la le . , . , da , le	; ov maç lı ğı nı i le ri ne bağ lı ğa da ha zi si ni den bi	hin de 892 - 902 dö ne min de bu lun du ğu nu i le bir lik te ya
Syllable Based	Hybrid	reft derîs eszék 5011b lendàrs секретар 354 nutry 0 ؛e1 yenb ikh الخطاب	med (želj çer 2518 ma bı sb 198 ish ds ar ge 813 mo va şok fa gi -	tiâ var dı vix ralph 35 < (luy dev 08 uy 1158 sak 2002 laš u sim ka rock
Syllable Based	Smoothing	2x icrâ tümg jmd surly 정월 yecd sundy цын idyf 陸軍中央幼年学校 цезве dasggf 露西亜 بانه hajdú тарал blyth 7702 пражская	zeyin deki a la rak , i le rin de , ka dar o la rı nı i se zo	ğin de . a ra sın da da ha son ra a ğaç lar dan o lan bir şe kil
Syllable Based	Top-5	cemá le le da da . , da , le , , . da la le da . le le	' had din le rin de , a la rı nı lan dı ğı , i çin " | |	venj tah tı na a it ol du . a na ya sa ya rı fi na list ol du
Syllable Based	Hybrid	czw sugōt 72m1 terys cilt9 de06 jells colds شمش 三奸四愚 7218 metzt bidât غوربند rockb 9ers9 5166667 πλάσμα sucky golî	men bam iç ol kut da yagh efdâl mülk ha kürt nok jay taj ve i nö ox na dop	raç yo ğun yır kar bü riş mo i 51 üf mon qu rug wew vaf lı_ fel rahv ti

Figure 1: Sentence generation examples using different methods and models.

The results reflect distinct differences in sentence generation quality across character-based and syllable-based models, as well as among different generation methods (smoothing, top-5, and hybrid). Character-based models produce highly varied sentences, but the randomness is especially noticeable in unigram and bigram sentences. For example, character-based sentences generated with smoothing show isolated characters without meaningful structure. Even with the top-5 approach, the output lacks coherent patterns, producing disjointed sequences that rarely form recognizable words. This pattern continues in the hybrid approach, though it sometimes introduces unexpected character combinations, demonstrating the challenges of modeling Turkish at a purely character level.

In contrast, syllable-based models offer more linguistically structured sentences, especially in bigram and trigram forms. Sentences generated from the syllable-based model with top-5 selection are more likely to contain Turkish syllables that fit together in ways that mimic real words. For instance, the top-5 and hybrid bigram outputs sometimes resemble familiar syllable patterns, though they still lack full word or sentence coherence. The syllable-based trigrams with top-5 selection are even closer to actual phrases, creating repetitive but recognizable structures that reflect common Turkish syllable combinations.

The hybrid method for both character and syllable models adds an element of variability by incorporating unseen probabilities, which makes it less repetitive than top-5 alone but also introduces more randomness. This effect is more beneficial for syllable-based models, where it generates sentences that are diverse yet somewhat structured.

Analysis and Conclusion

The perplexity values show a clear distinction between how each model handles Turkish language text, particularly at different n-gram levels.

Character-Based Model

- **Unigram Level:** The character-based model achieves a relatively low perplexity of around 9.26. This suggests that single character prediction, based on individual character probabilities, is relatively straightforward. However, this model does not provide contextual depth, limiting its usefulness for more complex language modeling.

- **Bigram and Trigram Levels:** The perplexity values increase dramatically for bigrams (232,775.64) and trigrams (28,913.79). This substantial rise reflects the difficulty of accurately predicting sequences of characters. Character sequences in Turkish lack standalone structure, making it hard for this model to capture linguistic coherence.

Syllable-Based Model

- **Unigram Level:** The syllable-based model's unigram perplexity is higher than the character model's, at around 467.71, due to a greater variety of syllables. However, syllables carry more linguistic information, making them stronger modeling units.

- **Bigram and Trigram Levels:** Perplexity decreases considerably to 115.04 and 18.18 for bigrams and trigrams, respectively. Syllables allow the model to capture realistic and contextually sound Turkish sequences.

Quality of Generated Sentences

Character-Based Model:

- **Unigrams:** Generated sentences tend to be random and lack coherence due to limited contextual understanding.
- **Bigrams and Trigrams:** While showing slight improvement, these models still produce nonsensical output, reflecting their poor ability to represent Turkish patterns.

Syllable-Based Model:

- **Unigrams:** Although still random, syllable-based unigram outputs are closer to word-like structures than their character-based counterparts.
- **Bigrams and Trigrams:** These levels generate far more fluent and realistic Turkish phrases. Trigram sequences especially resemble natural Turkish speech patterns.

Model Suitability for Turkish Language Modeling

The analysis suggests that **syllable-based trigrams** are the most suitable for modeling Turkish. They offer a strong balance between predictability and linguistic coherence. These trigrams effectively capture context, structure, and produce fluent, human-like phrases.

In contrast, character-based models—especially at higher n-gram levels—struggle with the structure of the Turkish language. For Turkish, syllable-based bigrams and trigrams provide the best performance,

References

1. Turkish Syllabification Code by Hasan LaIin (10.04.2005)
2. `https://drive.google.com/file/d/1_vBwTMssj0jPPjEvbBJragz9x4lx8Zeq/view`
3. Turkish Syllabification with PHP:
   `https://tdsoftware.wordpress.com/2011/09/11/php-ile-turkce-sozluk-hecelemek/`
4. A YouTube video uploaded by André Ribeiro Miranda (2013):
   NLP - 2.7 - Good-Turing Smoothing
   `https://www.youtube.com/watch?v=GwP8gKa-ij8`