<https://g.co/gemini/share/edd54f025dc0>

**Write a python program that can take: filename: str**

**start\_word: str**

**output\_length: str and give a sentence with output\_length no of words by autocompleting after start\_word using markov chains**

Here's a Python program that takes a filename, start\_word, and output length as input and generates a sentence using Markov chains:

Python

from collections import defaultdict

def build\_markov\_chain(filename):

"""

Builds a Markov chain from the text in a file.

Args:

filename: The name of the file to read.

Returns:

A dictionary where keys are pairs of words and values are lists of following words.

"""

markov\_chain = defaultdict(list)

with open(filename, 'r') as f:

text = f.read().lower()

words = text.split()

for i in range(len(words) - 2):

pair = (words[i], words[i + 1])

markov\_chain[pair].append(words[i + 2])

return markov\_chain

def generate\_sentence(markov\_chain, start\_word, output\_length):

"""

Generates a sentence using the Markov chain.

Args:

markov\_chain: The Markov chain dictionary.

start\_word: The word to start the sentence with.

output\_length: The desired length of the sentence.

Returns:

A string containing the generated sentence.

"""

sentence = [start\_word]

current\_pair = (start\_word, None)

for \_ in range(output\_length - 1):

következő\_words = markov\_chain.get(current\_pair, [])

if not következő\_words:

break

next\_word = következő\_words[random.randint(0, len(következő\_words) - 1)]

sentence.append(next\_word)

current\_pair = (current\_pair[1], next\_word)

return " ".join(sentence)

if \_\_name\_\_ == "\_\_main\_\_":

filename = input("Enter filename: ")

start\_word = input("Enter starting word: ")

output\_length = int(input("Enter desired sentence length: "))

markov\_chain = build\_markov\_chain(filename)

sentence = generate\_sentence(markov\_chain, start\_word, output\_length)

print(sentence)

Use code with caution.

This program defines two functions:

build\_markov\_chain: This function reads text from a file, builds a dictionary where keys are pairs of consecutive words and values are lists of words that follow the pair.

generate\_sentence: This function takes the Markov chain, a starting word, and desired output length as input. It iterates through the chain, randomly choosing the next word based on the previous pair, until the desired length is reached.

The program then prompts the user for the filename, starting word, and desired sentence length, builds the Markov chain, and generates a sentence.

Note:

This program uses random.randint to choose the next word. You can explore other probability distribution functions for more sophisticated weighting.

Make sure the input file contains enough text data for the Markov chain to work effectively.

The provided words ('következő\_words') are in Hungarian and mean "next words". You can replace them with their English equivalents.

profile picture

**use random.choice instead of következő\_words[random.randint(0, len(következő\_words) - 1)]**

Sure, here is the modified generate\_sentence function that uses random.choice instead of következő\_words[random.randint(0, len(következő\_words) - 1)]:

Python

def generate\_sentence(markov\_chain, start\_word, output\_length):

"""

Generates a sentence using the Markov chain.

Args:

markov\_chain: The Markov chain dictionary.

start\_word: The word to start the sentence with.

output\_length: The desired length of the sentence.

Returns:

A string containing the generated sentence.

"""

sentence = [start\_word]

current\_pair = (start\_word, None)

for \_ in range(output\_length - 1):

next\_word = random.choice(markov\_chain.get(current\_pair, []))

if not next\_word:

break

sentence.append(next\_word)

current\_pair = (current\_pair[1], next\_word)

return " ".join(sentence)

Use code with caution.

This function uses the random.choice function to randomly choose the next word from the list of possible words that follow the current pair. This is a more concise way of achieving the same result as the previous code.

Here is an example of how to use the program:

Python

filename = "text.txt"

start\_word = "the"

output\_length = 10

markov\_chain = build\_markov\_chain(filename)

sentence = generate\_sentence(markov\_chain, start\_word, output\_length)

print(sentence)

Use code with caution.

This code will generate a sentence of 10 words, starting with the word "the", using the Markov chain built from the text in the file "text.txt".

Note:

The random.choice function is a good choice for this task because it is efficient and easy to use.

You can also use other probability distribution functions to weight the possible next words differently.

**consider phrases of length n (2) to generate the next word.**

The provided code can be modified to consider phrases of length n (2) for generating the next word. Here's how:

1. Update build\_markov\_chain:

Python

def build\_markov\_chain(filename, n=2):

"""

Builds a Markov chain from the text in a file, considering phrases of length n.

Args:

filename: The name of the file to read.

n: The length of the phrase to consider (default: 2).

Returns:

A dictionary where keys are tuples of n words and values are lists of following words.

"""

markov\_chain = defaultdict(list)

with open(filename, 'r') as f:

text = f.read().lower()

words = text.split()

for i in range(len(words) - n):

phrase = tuple(words[i:i + n]) # Create a tuple of n words

markov\_chain[phrase].append(words[i + n])

return markov\_chain

Use code with caution.

This update changes the key in the markov\_chain dictionary from pairs of words to tuples of n words.

The loop iterates through the text, creating a tuple of n consecutive words and appending the next word as the value.

2. Update generate\_sentence:

Python

def generate\_sentence(markov\_chain, start\_word, output\_length, n=2):

"""

Generates a sentence using the Markov chain, considering phrases of length n.

Args:

markov\_chain: The Markov chain dictionary.

start\_word: The word to start the sentence with (can be a string or tuple of length n).

output\_length: The desired length of the sentence.

n: The length of the phrase to consider (default: 2).

Returns:

A string containing the generated sentence.

"""

sentence = list(start\_word) # Convert start\_word to a list

current\_phrase = tuple(sentence[-n:]) # Create initial phrase tuple

for \_ in range(output\_length - n):

next\_word = random.choice(markov\_chain.get(current\_phrase, []))

if not next\_word:

break

sentence.append(next\_word)

current\_phrase = tuple(sentence[-n:]) # Update current phrase

return " ".join(sentence)

Use code with caution.

This update allows start\_word to be either a single word (converted to a list) or a tuple of n words.

The loop creates a phrase tuple from the last n words in the sentence (sentence[-n:]).

It retrieves possible next words based on the current phrase and updates the sentence and phrase for the next iteration.

Using the updated functions:

Python

filename = "text.txt"

start\_word = ("the", "cat") # Starting phrase (tuple of length 2)

output\_length = 10

markov\_chain = build\_markov\_chain(filename, n=2)

sentence = generate\_sentence(markov\_chain, start\_word, output\_length, n=2)

print(sentence)

Use code with caution.

This code will generate a sentence using phrases of length 2 (bigrams) to predict the next word, starting with the phrase "the cat".

**turn start\_word to start\_phrase of length n which is always in the markov\_chain**

Here's how to modify the code to ensure the provided start\_phrase (of length n) is always present in the Markov chain:

1. Update generate\_sentence:

Python

def generate\_sentence(markov\_chain, start\_phrase, output\_length, n=2):

"""

Generates a sentence using the Markov chain, considering phrases of length n.

Args:

markov\_chain: The Markov chain dictionary.

start\_phrase: A tuple of words with length n, ensuring it's present in the chain.

output\_length: The desired length of the sentence.

n: The length of the phrase to consider (default: 2).

Returns:

A string containing the generated sentence, or None if start\_phrase not found.

"""

if start\_phrase not in markov\_chain:

return None # Handle non-existent start\_phrase

sentence = list(start\_phrase) # Convert start\_phrase to a list

current\_phrase = tuple(sentence[-n:])

for \_ in range(output\_length - n):

next\_word = random.choice(markov\_chain.get(current\_phrase, []))

if not next\_word:

break

sentence.append(next\_word)

current\_phrase = tuple(sentence[-n:])

return " ".join(sentence)

Use code with caution.

The function now checks if the provided start\_phrase exists in the markov\_chain before proceeding.

If the phrase is not found, the function returns None to indicate an error.

2. Ensure start\_phrase validity in your script:

Python

filename = "text.txt"

n = 2

# Look for a valid starting phrase in the text (replace with your logic)

# This part ensures the start\_phrase is found in the chain

candidate\_phrases = []

with open(filename, 'r') as f:

text = f.read().lower().split()

for i in range(len(text) - n + 1):

candidate\_phrases.append(tuple(text[i:i + n]))

start\_phrase = None

# Choose the first valid phrase (or implement logic to pick a suitable one)

for phrase in candidate\_phrases:

if phrase in markov\_chain:

start\_phrase = phrase

break

if not start\_phrase:

print("Error: Could not find a valid starting phrase of length", n)

else:

output\_length = 10

sentence = generate\_sentence(markov\_chain, start\_phrase, output\_length, n=2)

if sentence:

print(sentence)

else:

print("Sentence generation failed.")

Use code with caution.

This code snippet demonstrates how to find a valid start\_phrase from the text file.

It iterates through the text, creating candidate phrases of length n.

It checks if any of these phrases exist in the markov\_chain.

If a valid phrase is found, it's used as the start\_phrase.

This ensures that the provided start\_phrase will always be a valid starting point for the sentence generation. You can modify the logic for choosing a suitable start\_phrase based on your specific needs.

**give me some test cases for this code**

Test Cases for Markov Chain Text Generation

Here are some test cases for the code that generates text using Markov chains:

1. Basic Functionality:

Input:

Filename: "text.txt" containing a variety of sentences.

start\_phrase: ("the", "cat")

output\_length: 5

Expected Output: The program should generate a sentence starting with "the cat" and containing 3 additional words (total length 5). The generated words should be grammatically correct and follow the context of "the cat".

2. Non-existent Start Phrase:

Input:

Filename: "text.txt"

start\_phrase: ("never", "seen") # May not exist in the text

output\_length: 7

Expected Output: The program should print an error message indicating that the provided starting phrase is not found in the Markov chain.

3. Short Text File:

Input:

Filename: "short\_text.txt" containing just a few sentences.

start\_phrase: ("a", "sunny")

output\_length: 10

Expected Output: The program might generate a shorter sentence than the desired length (less than 10 words) due to the limited data in the text file. The generated sentence should still be grammatically correct and follow the context of "a sunny".

4. Edge Case - Repeated Words:

Input:

Filename: "repetitive\_text.txt" containing many sentences with repeated words (e.g., "the quick brown fox jumps over the quick brown fox").

start\_phrase: ("the", "quick")

output\_length: 8

Expected Output: The program might generate a repetitive sentence due to the repetitive nature of the text file. It's important to consider the quality of the input text for meaningful results.

5. Testing with Different n-gram Values:

Test Case 1 (bigrams, n=2):

Use the same inputs as Test Case 1 but with n=2 (considering bigrams).

The generated sentence might differ slightly compared to using n=1 (unigrams) due to the additional context provided by bigrams.

Test Case 2 (trigrams, n=3):

Use the same inputs as Test Case 1 but with n=3 (considering trigrams).

The generated sentence might be even more semantically relevant due to the larger context used (trigrams). However, the program might struggle to find enough valid trigram phrases in the text, potentially leading to shorter sentences or errors.

Additional Considerations:

You can create test cases with different text files containing specific vocabulary or sentence structures to test the program's ability to adapt to different writing styles.

It's important to visually inspect the generated sentences to ensure they are grammatically correct and make sense in the context of the starting phrase.

By running these test cases, you can verify that the code functions as expected and identify potential issues with the text data or the chosen n-gram size.

**give me three file contents to make the chains and test cases for each of them**

Three Text Files and Test Cases for Markov Chain Text Generation

Here are three text files with different content and corresponding test cases for your Markov chain text generation code:

File 1: short\_story.txt (Simple story with various sentence structures)

The cat chased the ball under the couch. It couldn't reach it, so it meowed loudly. The girl heard the meow and came running. She reached under the couch and pulled out the ball. The cat happily batted the ball around the room.

Test Case 1:

Input:

Filename: "short\_story.txt"

start\_phrase: ("The", "cat")

output\_length: 7

Expected Output: The program should generate a sentence starting with "The cat" and containing 5 additional words (e.g., "chased the ball under").

Test Case 2:

Input:

Filename: "short\_story.txt"

start\_phrase: ("The", "girl")

output\_length: 8

Expected Output: The program should generate a sentence starting with "The girl" and containing 7 additional words (e.g., "heard the meow and came").

File 2: news\_article.txt (Formal text with factual information)

A new study has found that regular exercise can improve brain health. The study, published in the journal Nature, involved over 10,000 participants. The participants were followed for a period of five years. During the study, the participants were asked to complete a series of cognitive tests. The results of the study showed that those who exercised regularly performed better on the cognitive tests.

Test Case 1:

Input:

Filename: "news\_article.txt"

start\_phrase: ("A", "new")

output\_length: 8

Expected Output: The program should generate a sentence starting with "A new" and containing 7 additional words (e.g., "study has found that regular").

Test Case 2:

Input:

Filename: "news\_article.txt"

start\_phrase: ("The", "study")

output\_length: 6

Expected Output: The program should generate a sentence starting with "The study" and containing 5 additional words (e.g., "published in the journal").

File 3: song\_lyrics.txt (Informal text with rhymes and repetition)

We are young, we are strong

We will live forever

Singing our song, all night long

We are the champions, my friend

And we'll keep on fighting 'til the end

We are the champions

We are the champions

No time for losers

Cause we are the winners

Test Case 1:

Input:

Filename: "song\_lyrics.txt"

start\_phrase: ("We", "are")

output\_length: 6

Expected Output: The program should generate a sentence starting with "We are" and containing 4 additional words (e.g., "young, we are strong").

Test Case 2:

Input:

Filename: "song\_lyrics.txt"

start\_phrase: ("We", "are", "the")

output\_length: 5

Expected Output: The program should generate a sentence starting with "We are the" and containing 4 additional words (e.g., "champions, my friend").

These test cases explore how the code handles different text styles and vocabulary choices. You can modify the test cases and file content to further test the program's capabilities.

Sources

www.thundercc.com/boys/article.aspx?id=1024….

1

Three Text Files and Test Cases for Markov Chain Text Generation

Here are three text files with different content and corresponding test cases for your Markov chain text generation code:

File 1: short\_story.txt (Simple story with various sentence structures)

The cat chased the ball under the couch. It couldn't reach it, so it meowed loudly. The girl heard the meow and came running. She reached under the couch and pulled out the ball. The cat happily batted the ball around the room.

Test Case 1:

Input:

Filename: "short\_story.txt"

start\_phrase: ("The", "cat")

output\_length: 7

Expected Output: The program should generate a sentence starting with "The cat" and containing 5 additional words (e.g., "chased the ball under").

Test Case 2:

Input:

Filename: "short\_story.txt"

start\_phrase: ("The", "girl")

output\_length: 8

Expected Output: The program should generate a sentence starting with "The girl" and containing 7 additional words (e.g., "heard the meow and came").

File 2: news\_article.txt (Formal text with factual information)

A new study has found that regular exercise can improve brain health. The study, published in the journal Nature, involved over 10,000 participants. The participants were followed for a period of five years. During the study, the participants were asked to complete a series of cognitive tests. The results of the study showed that those who exercised regularly performed better on the cognitive tests.

Test Case 1:

Input:

Filename: "news\_article.txt"

start\_phrase: ("A", "new")

output\_length: 8

Expected Output: The program should generate a sentence starting with "A new" and containing 7 additional words (e.g., "study has found that regular").

Test Case 2:

Input:

Filename: "news\_article.txt"

start\_phrase: ("The", "study")

output\_length: 6

Expected Output: The program should generate a sentence starting with "The study" and containing 5 additional words (e.g., "published in the journal").

File 3: song\_lyrics.txt (Informal text with rhymes and repetition)

We are young, we are strong

We will live forever

Singing our song, all night long

We are the champions, my friend

And we'll keep on fighting 'til the end

We are the champions

We are the champions

No time for losers

Cause we are the winners

Test Case 1:

Input:

Filename: "song\_lyrics.txt"

start\_phrase: ("We", "are")

output\_length: 6

Expected Output: The program should generate a sentence starting with "We are" and containing 4 additional words (e.g., "young, we are strong").

Test Case 2:

Input:

Filename: "song\_lyrics.txt"

start\_phrase: ("We", "are", "the")

output\_length: 5

Expected Output: The program should generate a sentence starting with "We are the" and containing 4 additional words (e.g., "champions, my friend").

These test cases explore how the code handles different text styles and vocabulary choices. You can modify the test cases and file content to further test the program's capabilities.

Sources

www.thundercc.com/boys/article.aspx?id=1024