Final Report

Ramyak Bilas and Naman Nimbale

2024-12-18

Final Report: N-gram Markov Language Model

The project can be found at github link

Aim of the Project

The aim of this project is to develop an **N-gram Markov Language Model** to analyze and generate natural language text based on statistical properties of sequential word patterns. By leveraging the Markov assumption, the project seeks to:

1. Use publicly available text data, preprocess it for training.
2. Convert the text into n_gram tokens.
3. Calculate the transition probabilities of for words as well as punctuation from the training data.
4. Train N-gram Markov models for various models and generate text using the transition probabilities.

How an N-gram Markov Language Model Works

An **N-gram Markov Language Model** is a statistical approach to modeling and generating text by predicting the next word based on a fixed sequence of preceding words (known as the context). The model relies on the **Markov assumption**, which simplifies the dependencies in the language by considering only a limited history of `N-1` words.

1. Basic idea of the language model

N-gram An N-gram is a sequence of `N` consecutive words from a given text. For example: - **Unigram** (1-gram): `["I", "love", "coding"]` - **Bigram** (2-gram): `[("I", "love"), ("love", "coding")]` - **Trigram** (3-gram): `[("I", "love", "coding")]`

Markov assumption The Markov assumption states that the probability of a word depends only on the previous `N-1` words, not on the entire sentence or history. Formally:

$$P(w_t \mid w_{t-1}, w_{t-2}, \dots, w_1) \approx P(w_t \mid w_{t-1}, w_{t-2}, \dots, w_{t-N+1})$$

This simplifies the computation while retaining sufficient context for generating coherent text.

2. Transition probabilities

The model estimates the **transition probability** $P(w_t \mid w_{t-1}, \dots, w_{t-N+1})$, which represents the likelihood of a word w_t appearing after a specific sequence of preceding words.

The transition probabilities are calculated from the training text corpus using **relative frequencies**:

$$P(w_t \mid w_{t-1}, \dots, w_{t-N+1}) = \frac{\text{Count}(w_{t-N+1}, \dots, w_{t-1}, w_t)}{\text{Count}(w_{t-N+1}, \dots, w_{t-1})}$$

where the numerator counts the number of the N-gram $(w_{t-N+1}, \dots, w_t)$ and the denominator counts the number of $(N-1)$-gram $(w_{t-N+1}, \dots, w_{t-1})$.

3. Punctuation probabilities

To reduce the number of tokens we treat all the punctuation as same word. So the prediction of a punctuation is done in two steps: use the **transition probability** to generate text where all punctuation are treated as a single word, and use **punctuation transition** to predict what the given punctuation is after a text is generated. Since punctuation depends strongly on the last word, and we only allow four punctuation (? , ! .), the **punctuation probabilities** are calculated from training text:

$$P(\text{punctuation}|w) = \frac{\text{Count}(w, \text{punctuation})}{\text{Count}(w)}$$

Design Decisions

Text preprocessing

First we set the directory. Since our text files were stored with .txt extensions, we use the regular expression \.txt (\ to escape .) and check for txt files in the directory.

```r
directory <- "/Users/ramyakbilas/Downloads/S610_Markov_Model"
txt_files <- list.files(path = directory, pattern = "\\.txt$", full.names = TRUE)
print(txt_files)
```

```
## [1] "/Users/ramyakbilas/Downloads/S610_Markov_Model/Alice_in_Wonderland.txt"
## [2] "/Users/ramyakbilas/Downloads/S610_Markov_Model/Wizard_of_Oz.txt"
```

We create a named list with the content of the the txt files in the previous directory. Function **read_txt_files** takes input a list of paths of the .txt files (as in **txt_files** from the previous block), reads the files and converts to named list with name of the books and their content. We store this in the list **file_content**.

```r
# Function to read the content of each file
read_txt_files <- function(files) {
  # Read each file and return as a named list
  file_contents <- lapply(files, readr::read_file)
  names(file_contents) <- basename(files)  # Assign file names as list names
  return(file_contents)
}

# Get the contents of the .txt files
file_contents <- read_txt_files(txt_files)
```

```r
# Print the first 500 characters of the text
print(substr(file_contents[[1]], 1, 500))
```

```
## [1] "The Project Gutenberg eBook of Alice's Adventures in Wonderland\n\n\nThis ebook is for the use
```

```r
typeof(file_contents)
```

```
## [1] "list"
```

We clean the text and normalize the texts. First we convert the text to lower case, then we replace everything except alphanumeric characters , . ! ? " ' into a space. We also replace all the newline \n with space. This introduces a lot of spaces so we replace multiple strings of spaces to a single space. Function **clean_text** is the function that does that; take in a list where the elements are texts (**file_contents** in this case) and returns normalized text (**cleaned_txt** here).

```r
clean_text <- function(text) {

  # Convert all characters to lowercase
  text <- tolower(text)

  #sub replaces the first pattern by the second string in the txt
  # Replace all non-alphanumeric characters (except , . ! ? "') with a space
  text <- gsub("[^[:alnum:],.!?\"\']", " ", text)

  # add spaces before and after , . ? !
  #text <- gsub("([,\\.\\?!])", " \\1 ", text)

  # Replace newlines (\\n) with a space
  text <- gsub("\n", " ", text)

  # Replace multiple spaces with a single space
  text <- gsub("\\s+", " ", text)

  # Trim leading and trailing spaces
  text <- trimws(text)

  # Return the cleaned text
  return(text)
}

cleaned_text <- clean_text(file_contents)
print(substr(cleaned_text[[1]], 1, 900))
```

```
## [1] "the project gutenberg ebook of alice's adventures in wonderland this ebook is for the use of an
```

Tokenization

The function **tokenize_words** tokenizes words. It takes in a list containing text (**cleaned_text**) and returns a list of words (which includes punctuation) which we store it in **tokens_clean**.

```r
# Function to tokenize text into words
tokenize_words <- function(text) {

  # Split the text wherever there is one or more whitespace characters
  tokens <- unlist(strsplit(text, "\\s+"))

  # Return the vector of word tokens
  return(tokens)
}

tokens_clean <- tokenize_words(cleaned_text)

print(tokens_clean[1:100])
```

```
##   [1] "the"                 "project"             "gutenberg"
##   [4] "ebook"               "of"                  "alice's"
##   [7] "adventures"          "in"                  "wonderland"
##  [10] "this"                "ebook"               "is"
##  [13] "for"                 "the"                 "use"
##  [16] "of"                  "anyone"              "anywhere"
##  [19] "in"                  "the"                 "united"
##  [22] "states"              "and"                 "most"
##  [25] "other"               "parts"               "of"
##  [28] "the"                 "world"               "at"
##  [31] "no"                  "cost"                "and"
##  [34] "with"                "almost"              "no"
##  [37] "restrictions"        "whatsoever."         "you"
##  [40] "may"                 "copy"                "it,"
##  [43] "give"                "it"                  "away"
##  [46] "or"                  "re"                  "use"
##  [49] "it"                  "under"               "the"
##  [52] "terms"               "of"                  "the"
##  [55] "project"             "gutenberg"           "license"
##  [58] "included"            "with"                "this"
##  [61] "ebook"               "or"                  "online"
##  [64] "at"                  "www.gutenberg.org."  "if"
##  [67] "you"                 "are"                 "not"
##  [70] "located"             "in"                  "the"
##  [73] "united"              "states,"             "you"
##  [76] "will"                "have"                "to"
##  [79] "check"               "the"                 "laws"
##  [82] "of"                  "the"                 "country"
##  [85] "where"               "you"                 "are"
##  [88] "located"             "before"              "using"
##  [91] "this"                "ebook."              "title"
##  [94] "alice's"             "adventures"          "in"
##  [97] "wonderland"          "author"              "lewis"
## [100] "carroll"
```

The function **replace_punctuation_with_tag** takes text and replaces all the punctuation marks (? , . !) with a special character . We use this on **cleaned_text** and store the resultant on **modified_text**.

```r
replace_punctuation_with_tag <- function(text) {

  # Replace punctuation (, . ? !) with </s>
  text <- gsub("[,\\.\\?!]", " </s> ", text)

  # Replace multiple spaces with a single space
  text <- gsub("\\s+", " ", text)

  return(text)
}

modified_text <- replace_punctuation_with_tag(cleaned_text)
print(substr(modified_text[[1]], 1, 1000))
```

```
## [1] "the project gutenberg ebook of alice's adventures in wonderland this ebook is for the use of any
```

Punctuation Probabilities

Here we take all the last word of a sentence and the punctuation pair, and calculate the P(punctuation | last_word) and put them in a data frame. **punctuation_transition** function takes in text (here **cleaned_text**) and returns a data frame containg 3 columns: last words, punctuation and P(punctuation | last_word) respectively (we store the output in **punctuations_df**).

```r
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
punctuation_transition <- function(text) {
  # Split text into sentences using punctuation as delimiters
  sentences <- unlist(strsplit(text, "(?<=[,\\.\\?!])\\s*", perl = TRUE))

  # Initialize vectors to store results
  last_words <- c()
  punctuations <- c()

  # Extract the last word and punctuation for each sentence
  for (sentence in sentences) {
    # Use regex to find the last word and punctuation
    match <- regmatches(sentence, gregexpr("\\b\\w+\\b[,\\.\\?!]?", sentence))[[1]]
    if (length(match) > 0) {
      last_part <- match[length(match)]
      last_word <- gsub("[,\\.\\?!]", "", last_part)  # Remove punctuation
      punctuation <- gsub(".*(,|\\.|\\?|!)$", "\\1", last_part)  # Extract punctuation

      if (punctuation %in% c(",", ".", "?", "!")) {
        last_words <- c(last_words, last_word)
        punctuations <- c(punctuations, punctuation)
      }
    }
  }

  # Create a data frame
  df <- data.frame(
    last_word = last_words,
    punctuation = punctuations,
    stringsAsFactors = FALSE
  )

  # Calculate probability of punctuation following each word
  prob_df <- df %>%
    group_by(last_word, punctuation) %>%
    summarise(count = n(), .groups = 'drop') %>%
    group_by(last_word) %>%
    mutate(probability = count/sum(count)) %>%
    ungroup() %>%
    select(last_word, punctuation, probability)

  return(prob_df)
}

punctuations_df <- punctuation_transition(cleaned_text)
print(punctuations_df)
```

```
## # A tibble: 3,121 x 3
##    last_word punctuation probability
##    <chr>     <chr>             <dbl>
##  1 1         ,                0.0118
##  2 1         .                0.988
##  3 10        ,                1
##  4 1887      .                1
##  5 1900      .                1
##  6 2         .                1
##  7 2001      ,                1
##  8 27        ,                1
##  9 3         ,                0.4
## 10 3         .                0.6
## # i 3,111 more rows
```

```
#input_text <- "Hello, how are you? I am fine, thank you! How about you? It's great."
#result_df <- punctuation_transition(input_text)
#print(result_df)
```

Transition Probabilities

The function **generate_n_grams** takes in tokens (as generated by the function we defined before) and returns a list of n-grams where each entry is a pair of context and predicted token. Here we use the **modified_text** tokens and store the n_grams in **n_grams_modified**.

```
generate_n_grams <- function(tokens, n=2){
  n_grams <- lapply(1:(length(tokens)-n+1), function(i) tokens[i:(i+n-1)])

  n_grams <- lapply(n_grams, function(x) list(x[1:(n-1)], x[n]))

  n_grams <- lapply(n_grams, function(n_gram) {
    c(paste(n_gram[[1]], collapse = " "),
      n_gram[[2]])
  })

  if (length(tokens) == 0 || length(tokens) < n) {
    return(list())  # Return an empty list
  }
  return(n_grams)
}

tokens_modified <- tokenize_words(modified_text)
n_grams_modified <- generate_n_grams(tokens_modified, 3)
print(n_grams_modified[500:510])
```

```
## [[1]]
## [1] "take out" "of"
##
## [[2]]
## [1] "out of" "it"
##
## [[3]]
## [1] "of it" "</s>"
##
## [[4]]
## [1] "it </s>" "and"
##
## [[5]]
## [1] "</s> and" "burning"
##
## [[6]]
## [1] "and burning" "with"
##
## [[7]]
## [1] "burning with" "curiosity"
##
## [[8]]
## [1] "with curiosity" "</s>"
##
## [[9]]
## [1] "curiosity </s>" "she"
##
## [[10]]
## [1] "</s> she" "ran"
##
## [[11]]
## [1] "she ran" "across"
```

The function **generate_transition_prob** takes in **n_grams** and returns a data frame with previous, current, and their transition probabilities. Here we apply the function on **n_grams_modified** and store the outcome in **transition_prob_modified**.

```
generate_transition_prob <- function(n_grams){
  #Creates a data frame of previous (context) and current (predicted token).
  df <- do.call(rbind, lapply(n_grams, function(x) {
    data.frame(previous = x[1], current = x[2])
  }))

  transition_prob <- df %>%
    group_by(previous, current) %>%
    summarise(count = n(), .groups = 'drop') %>%
    group_by(previous) %>%
    mutate(prob = count/sum(count)) %>%
    ungroup() %>%
    select(previous, current, prob)

  return(transition_prob)
}

transition_prob_modified <- generate_transition_prob(n_grams_modified)
print(transition_prob_modified)
```

```
## # A tibble: 57,302 x 3
##    previous   current        prob
##    <chr>      <chr>         <dbl>
##  1 0 contents chapter      1
##  2 000 are    particularly 1
##  3 1 </s>     1            0.0235
##  4 1 </s>     1993         0.0118
##  5 1 </s>     a            0.0235
##  6 1 </s>     b            0.0235
##  7 1 </s>     c            0.0471
##  8 1 </s>     d            0.0235
##  9 1 </s>     e            0.518
## 10 1 </s>     f            0.259
## # i 57,292 more rows
```

Training a model

We now train a model depending on the txt files in the directory and a choice of n for n-gram. The function **train_model** is basically consolidating what we did before. It takes in the directory path and the n of n-gram, and returns two data frames containing the punctuation transition probability and n-gram transition probability.

```
train_model <- function(directory, n=2){

  # get the names of the .txt files in the directory
  txt_files <- list.files(path = directory, pattern = "\\.txt$", full.names = TRUE)

  #we store all the text content
  file_contents <- read_txt_files(txt_files)

  # cleans the texts
  cleaned_text <- clean_text(file_contents)

  # store the punctuation transitions
  punctuations_df <- punctuation_transition(cleaned_text)

  # This block generates the transition probability for the n-gram
  modified_text <- replace_punctuation_with_tag(cleaned_text)
  tokens_modified <- tokenize_words(modified_text)
  n_grams_modified <- generate_n_grams(tokens_modified, n)
  transition_prob_modified <- generate_transition_prob(n_grams_modified)

  model <- list(punctuations_df = punctuations_df, transition_df = transition_prob_modified)

  return(model)
}

n_gram_model_3 <- train_model(directory, 3)
```

Text Generation

The **generate__text** function takes in model name, length of text to generate, context, and n to generate text. What it does is generates using n-gram markov model a sequence of texts of max length = len where we donot count the special character . Then we use the punctuation transition to replace with the appropriate punctuation depending on the last word preceeding . Note that the output is a character vector containing each individual word and punctuations.

```
generate_text <- function(model, len=50, feed = "", n = 2){
  transition_df <- model$transition_df
  punctuations_df <- model$punctuations_df

  # This code block fixes the first n-gram either from feed

  # Check the validity of the feed
  if (feed != ""){
    feed <- tolower(feed)
    feed <- unlist(strsplit(feed, "\\s+"))
    if (length(feed) >= len){
      stop("The length of the feed is longer than the length of the text requested")
    }
    else if (length(feed) < n-1){
      stop("The length of the feed is shorter than the n-1")
    }
    else{
```

```r
    first_ngram <- paste(tail(feed, n - 1), collapse = " ")
  }
}
# If feed is empty picks a n-gram at random
else {
  first_ngram <- sample(transition_df$previous, 1)
}

first_ngram <- unlist(strsplit(first_ngram, "\\s+"))

text <- character(0)
text <- c(text, first_ngram)

word_count <- length(text)
i <- length(text) + 1

while (word_count < len){

  previous_ngram <- tail(text, n-1)
  previous_ngram <- paste(previous_ngram, collapse = " ")
  previous_ngram <- as.character(previous_ngram)

  next_words <- transition_df %>%
    filter(previous == previous_ngram)

  if (nrow(next_words) == 0){
    stop("You input some words not included in the training data")
    #break
  }

  current_word <- sample(next_words$current, 1, prob = next_words$prob)


  if (current_word != "</s>") {
    word_count <- word_count + 1
  }

  text <- c(text, current_word)
  i <- i + 1
}


for (i in 1:length(text)){
  if (text[i] == "</s>"){
    prev_word <- text[i-1]
    available_punctuation <- punctuations_df%>%
      filter(last_word %in% prev_word)
    #Note to self: %in% above instead of == fixes the error. Don't know why

    punctuation <- sample(available_punctuation$punctuation, 1, prob = available_punctuation$probabil
    text[i] <- punctuation
  }
}
```

```r
  return(text)
}

random_text <- generate_text(n_gram_model_3, 50, "", 3)
random_text
```

```
##  [1] "night"    "."        "each"     "in"       "his"      "pockets"
##  [7] ","        "and"      "very"     "neatly"   "and"      "simply"
## [13] "arranged" "the"      "only"     "one"      "way"      "of"
## [19] "speaking" "to"       "it"       ","        "and"      "that"
## [25] "will"     "protect"  "him"      "."        "indeed"   ","
## [31] "he"       "d"        "do"       "almost"   "anything" "you"
## [37] "want"     "to"       "be"       "ashamed"  "of"       "yourself"
## [43] "."        "i"        "should"   "think"    "you"      "are"
## [49] "really"   "brighter" "than"     "he"       "needs"    "."
## [55] "and"      "each"
```

The **readable_text** takes in a character vector with many words, and collapses them into a readable text, but before doing that it capitalizes the first letter at the beginning of every sentence and takes care of proper spacing near punctuations.

```r
#Takes character vector and capitalizes the first letter
capitalize_first_letter <- function(text){
  text <- paste0(toupper(substr(text, 1, 1)),
                 substr(text, 2, nchar(text)))
  return(text)
}

# This takes the character vector and gives us a paragharph which follows basic english syntaxes
readable_text <- function(text){

  #capitalize the first letters of the first word
  text[1] <- capitalize_first_letter(text[1])

  #capitalize the first letter of any word succeeding . or ? or !
  for (i in 2:length(text)){
    if (text[i-1] %in% c(".", "?", "!")){
      text[i] <- capitalize_first_letter(text[i])
    }
  }

  # remove space between the word and the punctuation following it
  for (i in 2:length(text)){
    if (text[i] %in% c(".", "?", "!", ",")){
      text[i - 1] <- paste0(text[i - 1], text[i])
      text[i] <- ""
    }
  }

  # collapse the character vector by keeping space between each word
  text <- paste(text, collapse = " ")
  text <- gsub("\\s+", " ", text)

  return(text)
}

readable_text(random_text)
```

```
## [1] "Night. Each in his pockets, and very neatly and simply arranged the only one way of speaking to
```

The following function ***main()* makes it user interactive to train a model or use a model to generate text, basically consolidating everything we have done before. First it asks you to enter N of the N-gram one wants to build, context for text generation, and the length of text to generate. Then it prints all the pretrained N-gram models in the directory (all models have the N in their name), asking you if you want to train a new one or use an existing one, and output the text.

```
main <- function(){
  while(TRUE){
    n <- as.integer(readline("What is the n in n-gram: "))
    feed <- readline("Any context: ")
    length <- as.integer(readline("Enter the total length of text you want to generate: "))

    pattern <- paste0("*", "n_", n, "\\.rds")
    models <- list.files(path = directory, pattern = pattern, full.names = TRUE)
    cat("The following models are available:\n")
      for (i in 1:length(models)){
        cat(models[i], "\n")
      }

    new_model <- readline("Do you want to train a new model? (y/n): ")

    if (new_model == "y") {
      model <- train_model(directory, n)

      # Prompt user for model name
      model_name <- readline("Enter a name for the model: ")
      model_name <- paste(model_name, "n", n, sep = "_")

      model_path <- paste0(directory, "/" , model_name, ".rds")

      # Save the model
      saveRDS(model, model_path)
      cat("Model saved as:", model_path, "\n")

      sample <- generate_text(model, length, n, feed = feed)
      cat(generate_readable_text(sample))
    }else {

      # list all valid models
      #models <- list.files(path = directory, pattern = "*.rds$", full.names = TRUE)

      #cat("The following models are available:\\n")
      #for (i in 1:length(models)){
      #  cat(models[i], "\\n")
      #}

      model_name <- readline("Please enter the name of the model (including extension): ")
      # load the model
      model_path <- paste0(directory, "/" , model_name)
      model <- readRDS(model_path)
      sample <- generate_text(model, length, n, feed = feed)
      cat(generate_readable_text(sample))
    }

    exit <- readline("Do you want to exit? (y/n): ")
    if (exit == "y"){
      break
    }
  }
}
```

Outputs

Here are a few outputs for various contexts and N-grams.

For N=2 and context=Alice:

> Alice! And as i think i am ashamed of green glasses on each tale chapter x the widest array of finding that will not agree that this time without lobsters, so, take on again. And it would change, the water, there she was perfectly round, and over again, i was ready for some of them but i wonder what shall only it? Where the tarts, you may converse with your knocking, the throne seemed to do for the first at the gryphon. And morcar, and growing, who did, please. With another, said alice to finish the king, but just in

For N=3 and context=Alice to:

> Alice to herself. As they were obliged to camp out that night under a tree. Standing this time tugging hard at work mending the woodman happy. They walked along at a brisk pace. Said the scarecrow. Is to do as you go i certainly shall if i could kill toto! Once more they could, for a work with his band may thereafter be free for evermore. The players, and if you were down here, we will go back to my jaw, has he? Said a kind of courage? I m perfectly sure i shall be as friendly as the hours

For N=4 and context=Alice to herself:

> Alice to herself, for this palace and the emerald city and the winkies gave them three cheers and many good wishes to carry with them. As if she were saying lessons? And began bowing to the king crow said it is only a child. Said the mock turtle nine the next. The lion was angry at this speech, but could not because his head was quite bulged out at the door. And glancing around saw. To his surprise, that before the throne was a ball of cotton, but when it was daylight, the girl asked, i feel like a new

For N=8 and no context:

> Kansas for good and all, thank you? Said dorothy gratefully. You are all very kind to me, but i should like to start as soon as possible. We shall go tomorrow morning, returned the scarecrow. So now let us all get ready, for it will be a long journey, chapter xix attacked by the fighting trees the next morning dorothy kissed the pretty green girl good bye! And they all shook hands with the soldier with the green whiskers, he said, and ask his advice. So the soldier was summoned and entered the throne room timidly, for while oz

For N=20 and no context:

> Thick that the ground was carpeted with them, there were big yellow and white and blue and purple blossoms, besides great clusters of scarlet poppies. Which were so brilliant in color they almost dazzled dorothy s eyes. Aren t they beautiful, the girl asked. As she breathed in the spicy scent of the bright flowers, i suppose so, answered the scarecrow, when i have brains, i shall probably like them better, if i only had a heart. I should love them? Added the tin woodman, i always did like flowers. Said the lion. They seem so helpless and frail. But there are none in the forest so bright as these? They now came upon more and more of the big scarlet poppies, and fewer and fewer of the other flowers and soon they found themselves in the midst of a great meadow of poppies. Now it is well known that when there are many of these flowers together their odor is so powerful that anyone who breathes it falls asleep, and if the sleeper is not carried away from the scent of the flowers, he sleeps on and on forever. But dorothy did not know this, nor could she

We notice that as the value of N increases, the contextual meaning of text is preserved for longer duration of time. An observation in favour of this is that for lower values of N we switch from Alice in Wonderland to Wizard of Oz quite frequently in a moderate size text, but this is rare for a larger value of N.

Further Improvements

One problem which still persists is to generate text following a context such that the last N-1 letters of the context do not occur in the training data. There can be a few improvements in the readable text fucntion by making it recognize proper nouns and capitalizing them. Other than the four common punctuation (? , ! .) rest of them are lost during cleaning. One can try to keep a few more important one.

Evidence of Functionality using Test cases

All functions were thoroughly tested, and the results confirm that they operate as intended, handling both typical and edge cases appropriately. The test cases are defined in test_cases.R file. The below are the brief description of all the test cases:

1. Transition probabilities validation: Verified that all transition probabilities in the model lie within the valid range of 0 to 1. The function behaves as expected.

2. Probability sum validation: Ensured the sum of probabilities for each “previous” state is approximately 1, accounting for floating-point precision. The function produces correct results.

3. Last word probabilities validation: Checked that all probabilities associated with the last words are within the range of 0 to 1. The function works correctly.

4. Word tokenizer validation: Confirmed that tokenize_words correctly splits a given text into individual words and retains punctuation. The function behaves as intended.

5. Punctuation transition validation: Verified that punctuation_transition accurately extracts last words, punctuation marks, and probabilities from a text. The function performs as expected.

6. n-gram generation validation: Tested generate_n_grams to ensure it generates correct n-grams for a given sequence of tokens. The output matches the expected results.

7. n-gram generation with empty input: Confirmed that generate_n_grams returns an empty list when provided with an empty token input. The function handles edge cases gracefully.

8. n-gram generation with single token: Verified that generate_n_grams returns an empty list when the token count is less than n. The function operates correctly for this scenario.

9. Word tokenizer with special characters: Ensured that tokenize_words accurately handles text containing special characters and punctuation, preserving their placement. The function behaves as expected.

10. Transition probability calculation: Tested generate_transition_prob to ensure it calculates correct transition probabilities for given n-grams. The function produces valid results.

11. Transition probability with empty input: Verified that generate_transition_prob handles an empty input list by returning an empty data frame. The function handles edge cases correctly.