New York University Tandon School of Engineering
Computer Science and Engineering
Big Data CS-GY 6513 Spring 2023

AUTO-REGRESSIVE TEXT GEN

Team Members

Arpan Chatterjee | ac9839 | ac9839@nyu.edu
Shubhankar Vashishta | sv2229 | sv2229@nyu.edu

1. Introduction:

Autoregressive text generation refers to a technique for generating text where each word or character in the sequence is generated based on a probability distribution conditioned on the previously generated words or characters.

The project's goal is to generate a paragraph using a frequency dictionary of the next word based on a 3-gram model from the existing e-books collected from the Project Gutenberg digital library. This digital library contains over 70,000 books. All of these books have been used as the raw data for our project.

The project uses a probabilistic assumption to generate new data. This predicts the probabilistic maximum likelihood estimate to generate the subsequent words after the initial prompt.

To make this system convenient to end users, we will implement a Web front-end of this system with the help of Streamlit library. In the web pages we would provide users with several options to generate text from a selected book from an author of their choice.

2. Data Collection:

The raw data is collected from the Project Gutenberg digital library. This data comprises over 70,000 books in the .txt format.

Project Gutenberg Website: https://www.gutenberg.org/

An example of a book in .txt format: https://www.gutenberg.org/cache/epub/100/pg100.txt

The above link has the metadata given below:

Book:- The Complete Works of William Shakespeare
Author:- William Shakespeare
Release Date:- January 1994
Language:- English

We stored all of the 70708 available books, from similar urls as shown above, and stored it in Google Drive. After storing all of these books, we will compute the posterior probability of a given word given the k-grams of each of the books separately and store them in a dictionary. As computing the dictionary for each of the 70708 books is computationally expensive, we will compute this once and store the dictionaries in separate pickle files.

2.1 PySpark : Creating 3-grams

A function named ‘word_sequence_from_file’ is created. The purpose of this function is to generate a k-gram frequency dictionary from a text file. The k-gram frequency dictionary is a data structure that stores the frequency of occurrence of each k-gram in the text file.

The first few lines of the code import the necessary libraries and set up a Spark context. The SparkConf() function is used to create a Spark configuration object that is used to specify the application name. The SparkContext.getOrCreate(conf=conf) function is used to create a new Spark context or retrieve an existing one. The textFile() function is used to read the text file as an RDD, which is a distributed data structure that can be processed in parallel across multiple nodes in a cluster.

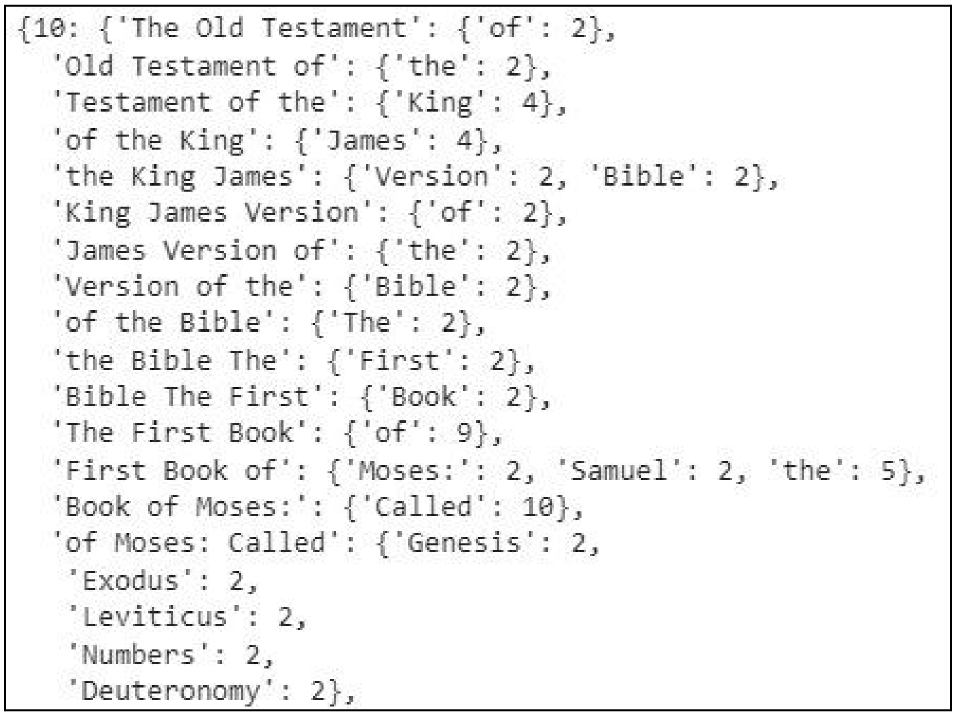
The next line of the code merges all the lines of the RDD into a single string using the reduce() function. The resulting string is split into a sequence of words using the split() function. The k variable is set to the desired length of each k-gram, which in this case is 3.

The freq_dict variable is a Python dictionary that will be used to store the k-gram frequency counts. The for loop iterates through each word in the word_sequence list and creates a k-gram for every three consecutive words. The join() function is used to concatenate the three words into a single k-gram string.

If the k-gram is not already in the freq_dict dictionary, a new entry is created for it. If the next word in the sequence is not already in the frequency count for that k-gram, it is added with a count of 1. If the word is already in the frequency count for that k-gram, the count is incremented by 1.

Finally, the Spark context is stopped and the resulting k-gram frequency dictionary is returned from the function. This code is an example of how PySpark can be used to process large datasets in a distributed manner to efficiently generate k-gram frequency dictionaries.

The output generated by this function is given below:



```
{10: {'The Old Testament': {'of': 2},
  'Old Testament of': {'the': 2},
  'Testament of the': {'King': 4},
  'of the King': {'James': 4},
  'the King James': {'Version': 2, 'Bible': 2},
  'King James Version': {'of': 2},
  'James Version of': {'the': 2},
  'Version of the': {'Bible': 2},
  'of the Bible': {'The': 2},
  'the Bible The': {'First': 2},
  'Bible The First': {'Book': 2},
  'The First Book': {'of': 9},
  'First Book of': {'Moses:': 2, 'Samuel': 2, 'the': 5},
  'Book of Moses:': {'Called': 10},
  'of Moses: Called': {'Genesis': 2,
   'Exodus': 2,
   'Leviticus': 2,
   'Numbers': 2,
   'Deuteronomy': 2},
```

2.2 Storage of 3-gram Frequency Dictionary in Pickle Files

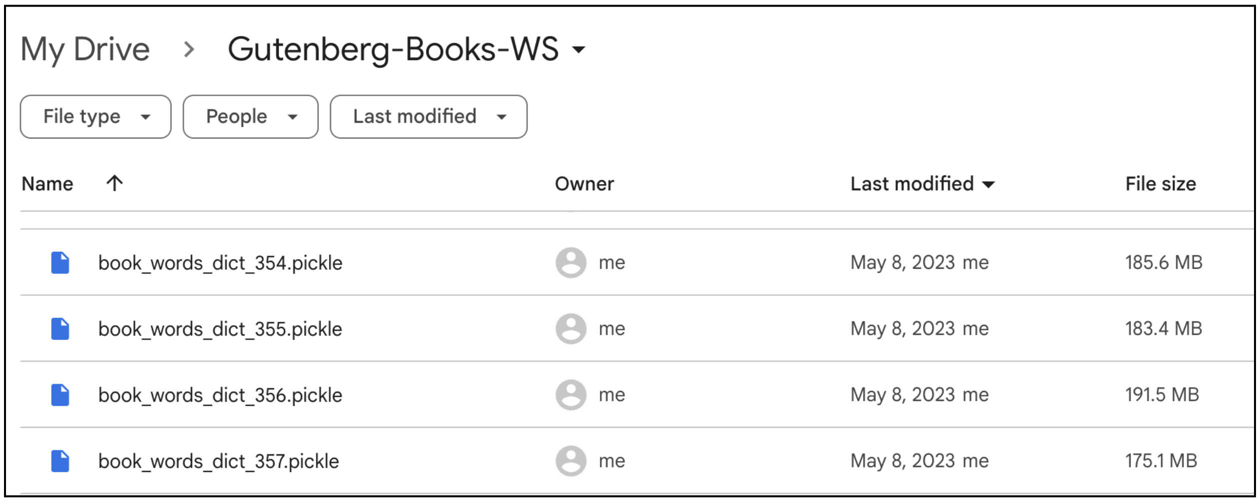
PySpark is implemented by a function named `process_book` that processes a text file to generate a k-gram frequency dictionary. The function takes a single argument `bid`, which is an integer representing the ID of a book. The book text file path is constructed by appending the `bid` to a directory path.

The code then creates a Spark context by defining a Spark configuration object using the `SparkConf()` function, which sets the application name. The Spark context is created or retrieved using the `SparkContext.getOrCreate(conf=conf)` function.

Next, a list of book IDs is generated using the `range()` function and the `parallelize()` function is used to create an RDD from the list. The `flatMap()` function is then used to apply the `process_book()` function to each book ID in the RDD, resulting in a list of tuples containing the book ID and the frequency dictionary generated from the book text file.

The `filter()` function is used to remove any tuples where the frequency dictionary is `None`, which indicates that an error occurred while processing the book text file. The resulting RDD is then converted to a dictionary using the `collectAsMap()` function and stored in the `books_rdd` variable.

Finally, the Spark context is stopped using the `sc.stop()` function. This code is an example of how PySpark can be used to efficiently process and analyze large amounts of text data in a distributed manner. By leveraging the power of Spark, it is possible to process thousands of books in parallel, generating a k-gram frequency dictionary for each one, and store the resulting dictionaries in a single data structure.



Name	Owner	Last modified	File size
book_words_dict_354.pickle	me	May 8, 2023 me	185.6 MB
book_words_dict_355.pickle	me	May 8, 2023 me	183.4 MB
book_words_dict_356.pickle	me	May 8, 2023 me	191.5 MB
book_words_dict_357.pickle	me	May 8, 2023 me	175.1 MB

2.3 Metadata Collection

Python script is written that reads text files containing books from a specified directory and extracts metadata from each book, including the title, author, release date, language, and encoding. The metadata for each book is stored in a dictionary and written to a series of pickle files.

The script starts by setting the directory path where the book text files are stored and defining an empty dictionary named `book_dict`. The variable `c` is also initialized to 70.

Next, a loop is used to iterate over each book ID in the range of 1 to 70709. For each book ID, the path to the corresponding text file is constructed by appending the `bid` to the directory path.

The script then attempts to open the file using a `with` statement. If the file is successfully opened, the script loops over each line in the file using the `enumerate()` function. If the line number is 23, the script extracts the metadata from the previous lines and stores it in a dictionary with the book ID as the key.

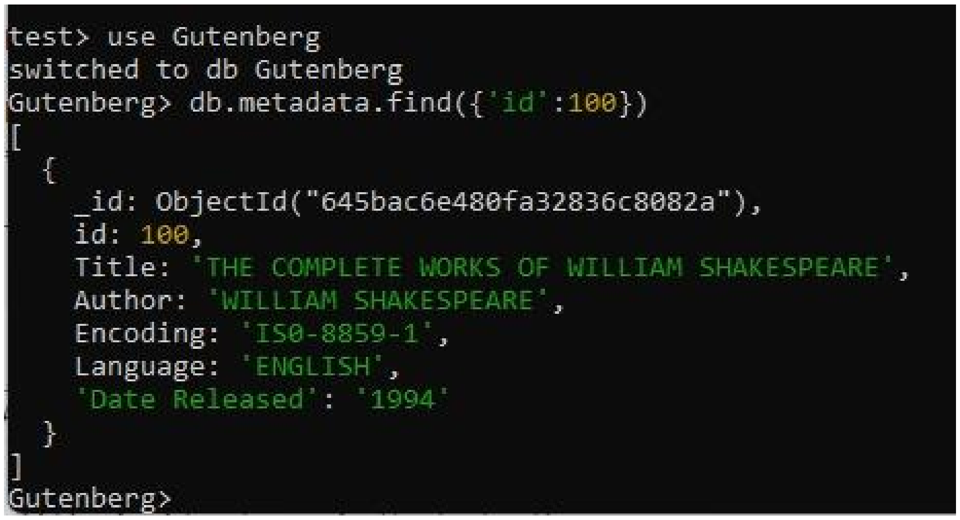
The metadata is extracted by checking each line for specific keywords, such as "title:", "author:", "release date:", "language:", and "character set encoding:". If a line contains one of these keywords, the script extracts the corresponding metadata value by removing the keyword prefix and any trailing newline characters.

If an error occurs while opening the file or extracting the metadata, the script simply moves on to the next book ID without raising an exception.

Finally, if the book ID is a multiple of 1000 or is the last book ID in the range, the script increments the `c` variable and constructs a pickle file name using the current value of `c`. The `book_dict` dictionary is then written to the pickle file using the `pickle.dump()` function and `book_dict` is reset to an empty dictionary.

This script provides an efficient way to extract metadata from a large collection of books and store it in a structured format that can be easily queried or analyzed later.

This metadata collected is stored inside MongoDB for querying the database in the Streamlit application. A database 'Gutenberg' is created inside MongoDB. The collection that stores the documents is called 'metadata'. A screen grab of a document inside 'metadata' is shown:-



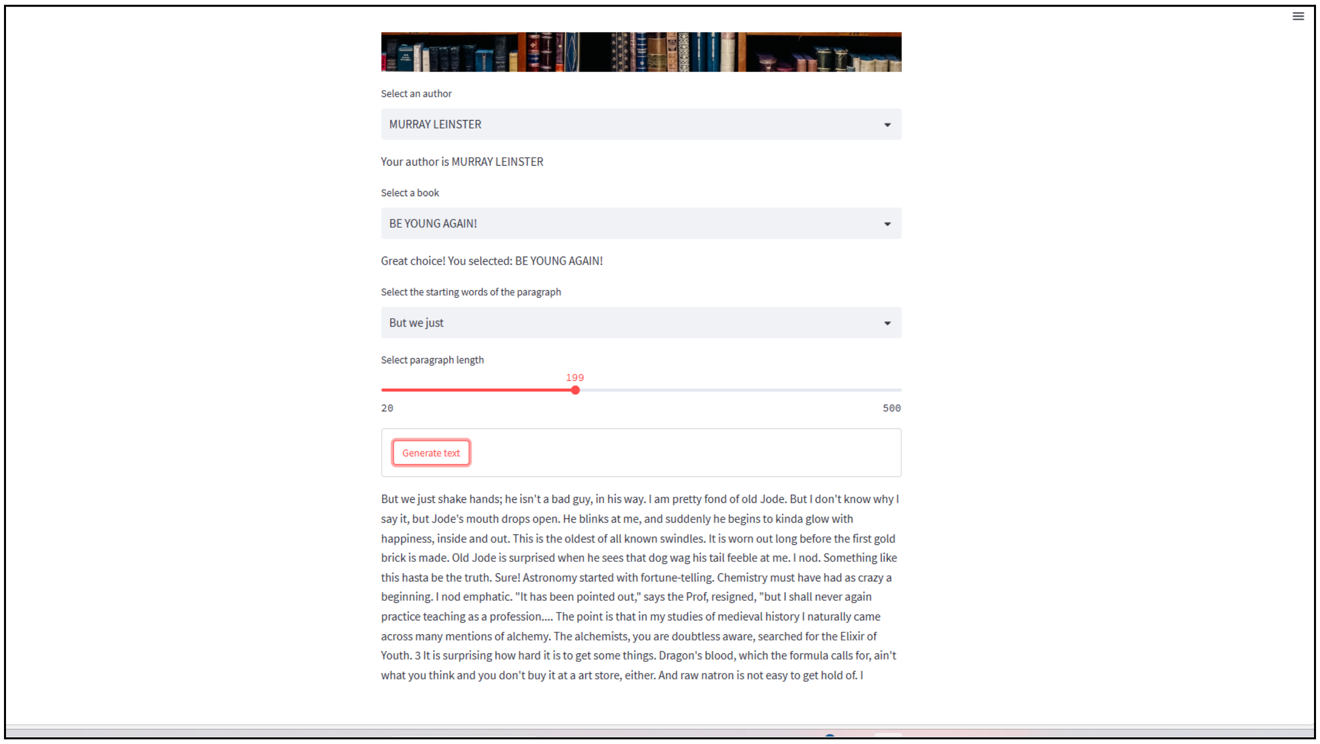
```
test> use Gutenberg
switched to db Gutenberg
Gutenberg> db.metadata.find({'id':100})
[
  {
    _id: ObjectId("645bac6e480fa32836c8082a"),
    id: 100,
    Title: 'THE COMPLETE WORKS OF WILLIAM SHAKESPEARE',
    Author: 'WILLIAM SHAKESPEARE',
    Encoding: 'ISO-8859-1',
    Language: 'ENGLISH',
    'Date Released': '1994'
  }
]
Gutenberg>
```

3. User Interface - Streamlit Application

Streamlit is a Python framework that enables easy and interactive creation of web applications. It can be used to create user interfaces for various machine learning and data analysis tasks. In this case, it is being used as an interface between the user and our application, where the user can interact with our MongoDB to select an author and any of their books.

The Streamlit application provides the user with a user-friendly interface where they can select an author and a book by the author. The user can choose from a list of authors, and then select a book by that author. Once the user selects a book, they can specify the length of the generated text and the initial input - starting 3 words of the generated text.

Once the user provides all the required inputs, the application generates a text of the specified length based on the selected book and the initial input provided by the user. The generated text can be used for various purposes, such as testing language models or generating content for websites or blogs.


Select an author

MURRAY LEINSTER

Your author is MURRAY LEINSTER

Select a book

BE YOUNG AGAIN!

Great choice! You selected: BE YOUNG AGAIN!

Select the starting words of the paragraph

But we just

Select paragraph length

199

20 500

But we just shake hands; he isn't a bad guy, in his way. I am pretty fond of old Jode. But I don't know why I say it, but Jode's mouth drops open. He blinks at me, and suddenly he begins to kinda glow with happiness, inside and out. This is the oldest of all known swindles. It is worn out long before the first gold brick is made. Old Jode is surprised when he sees that dog wag his tail feeble at me. I nod. Something like this hasta be the truth. Sure! Astronomy started with fortune-telling. Chemistry must have had as crazy a beginning. I nod emphatic. "It has been pointed out," says the Prof, resigned, "but I shall never again practice teaching as a profession.... The point is that in my studies of medieval history I naturally came across many mentions of alchemy. The alchemists, you are doubtless aware, searched for the Elixir of Youth. 3 It is surprising how hard it is to get some things. Dragon's blood, which the formula calls for, ain't what you think and you don't buy it at a art store, either. And raw natron is not easy to get hold of. I

The Streamlit application uses multiple functions to produce the desired output. A brief description of each of these functions has been described below:-

3.1 predict_next_word:

Function predict_next_word `this_kgram` and `freq_dict`. The `this_kgram` parameter represents the current k-gram sequence for which we want to predict the next word. The `freq_dict` parameter is a dictionary that contains the frequency of all possible words that occur after a given k-gram.

The function first checks if the current `this_kgram` exists in the `freq_dict`. If it does not exist, the function returns `None`. If it does exist, the function creates a `weighted_list` of all the possible words that could occur after the `this_kgram`. The weight of each word in the list is determined by the frequency of its occurrence after the given k-gram.

Finally, the function uses the `random.choice` method to select a word from the `weighted_list` and returns it as the predicted next word.

3.2 predict_paragraph:

The function predict_paragraph takes a starting k-gram, a k value, a frequency dictionary, and a generated length as input and returns a paragraph generated using the given parameters.
The function starts with the provided starting k-gram and predicts the next word using the predict_next_word function. It then appends the predicted word to the paragraph and forms a new k-gram by considering the last k words in the generated paragraph. This process is repeated until the generated paragraph reaches the desired length.

If the predict_next_word function returns None, which means that there are no words in the frequency dictionary for the given k-gram, the function will return the generated paragraph as is.
The function then returns the generated paragraph as a list of words.

3.3 get_authors:

This function retrieves the list of authors from a MongoDB database. The code connects to the local instance of the database, then uses an aggregation pipeline to group the books in the database by author, count the number of books by each author, and sort them in descending order by count. The pipeline then projects only the author names and returns the result. The code then takes the top 4 authors with the most books and returns their names as a list.

3.4 get_books_of_author:

This function connects to a MongoDB database hosted on the local machine and retrieves a list of books written by a specific author. The function takes an author name as input and queries the "metadata" collection of the "Gutenberg" database to find all documents with a matching "Author" field. The function then returns a list of book titles written by that author, by extracting the "Title" field from each of the matching documents.

3.5 get_book_id:

This function takes a book title as its input and returns the corresponding book ID. The function queries a MongoDB database named 'Gutenberg' and its 'metadata' collection for a document that matches the title input. It then extracts the 'id' field from the document and returns it. The function returns a list of book IDs, although there should only be one ID per book title.

3.6 get_random_ngram:

This function takes a dictionary freq_dict_test as an input parameter. It randomly selects 30 keys from the dictionary using random.choices() and stores them in the list kgram_start_choices. It then removes any key that contains a punctuation character using string.punctuation module and any() function. Finally, it returns the cleaned list of kgram_start_choices. The purpose of this function is to provide a list of potential n-grams that can be used as a starting point for generating a new paragraph of text.

4. Examples of User Input and the Generated Paragraph:

4.1 Selected Author: Charles Dickens; Selected Book: Oliver Twist

Starting words of the paragraph: “The husband carried”
Paragraph Length: 94

Generated paragraph: The husband carried a lantern, from which, however, no light yet shone; and trudged on, a few paces in front, as though the way being dirty to give his wife the benefit of treading in his heavy footprints. They went on, in profound silence; every now and then, a powerful inducement, which he could not know what they had done for him, and in bewailing his cruel separation from them. The hope of eventually clearing himself with them, too, and explaining how he had been consigned by the wisdom and mercy of the board.

4.2 Selected Author: Randall Garrett; Selected Book: Needler

Starting words of the paragraph: “I added my”
Paragraph Length: 112

Generated paragraph: I added my recommendation and sent it on to the psychological effects." Kiffer Samm looked up at Gowlan, who had elected to follow him. "Any sign of the ship changed. There was nothing to do now but wait. It was up to the trackers, and the trackers can't fire at indetectable objects. And you can't detect a meteorite in no-space drive. "Of course, I suppose we could send out some torpedoes and try to hit them, but that would be sort of wasteful." "Then the guns aren't tested in no-space, huh?" Roysland said grinning. "Then somebody's been falsifying reports to my office." Kiffer grinned back. "Sure," he said, "they're tested, but without the robot trackers;

4.2 Selected Author: John Allison; Selected Book: Dropped stitches in Tennessee history

Starting words of the paragraph: “The failure to”
Paragraph Length: 96

Generated paragraph:The failure to deliver these goods naturally exasperated the Cherokees, and caused them to commit depredations, from which the western settlers were of course the sufferers. At this session, the North Carolina act had subjected them to the payment of the sum due Andrew Jackson, as a full compensation for services as Attorney General for the District of Mero, for the purpose of publishing (for the first time, the question as to whether treason could be committed have been so extended as to include the court-yard.

4. Proposed Future Work:

4.1 Use a BERT Model for Autoregressive Text Generation:

A state-of-the-art model like BERT would be able to generate more comprehensible and meaningful phrases and sentences than what we have generated using the maximum likelihood estimate probabilistic model, which performs fairly well.

4.2 Generate paragraphs from multiple books:

Currently we are generating paragraphs from the context of one book. We could expand this to generate paragraphs based on the writing style of an author by making all of their writings available as the context instead of a single book. Moreover we could categorize the book genre and generate text based on a category