Report
Assignment 4

Team - shaivipa-kritikku

Part 1

1. Introduction

Brief Overview of the assignment
We're tasked with creating a system that analyzes word co-occurrence in real time, leveraging Apache Spark Streaming. The project revolves around handling an ongoing flow of textual information, examining it within moving time frames, and calculating the frequency of word pairs. For this implementation, we're harnessing the power of PySpark to enable distributed processing. To mimic a constant influx of data, we're employing netcat. Additionally, we're applying a range of methods for processing text effectively.

Objectives

1. A PySpark development setup is being prepared, incorporating essential elements such as SparkConf, SparkContext, and StreamingContext.

2. Our focus is on building a program that analyzes word pairs in streaming text data.

3. Using two execution threads and a batch interval of one second, we're establishing a local StreamingContext.

4. Netcat is being utilized to create a DStream that represents data flowing from a TCP source (localhost:9999).

5. Functions for text processing are under development, aimed at word separation, lowercase conversion, and punctuation elimination.

6. From the cleaned word list, we're crafting a method to generate bigrams.

7. The data stream undergoes a sliding window transformation, with a three-second window length and a two-second sliding interval.

8. Within these sliding windows, we're tallying and displaying the frequency of each bigram.

9. We're evaluating how our selected window size and sliding interval affect the bigram analysis.

10. Comprehensive logs of the processed information are being generated in both JSON and plain text formats.

11. Result analysis and visualization are underway, encompassing sentence counts per window, ongoing averages, and trend patterns.

This project is designed to provide practical experience in processing data in real-time, understanding distributed computing principles, and applying text analysis methods in a streaming environment.

2. Word Co-occurrence Algorithm

2.1 Algorithm Overview

Algorithm

```
WC-SparkStreaming(host, port)
1. Initialize Libraries and Setup Logging:
   - Import necessary libraries: `sys`, `re`, `subprocess`, `pyspark`, `logging`,
`warnings`, `json`.
   - Suppress INFO and WARNING logs from `py4j` and filter warnings for `psutil`.

2. Define WC Class and its Components:
   - Class WC.Utils:
     - Method proc_line(line):
       - Extract words from `line` using regex and convert to lowercase.
     - Method gen_bigrams(words):
       - Generate bigrams from the list of `words`.
   - Class WC.Log:
     - Method log_lines(time, rdd):
       - Log the number of lines and content received in `rdd` at `time`.
   - Class WC.Proc:
     - Method __init__(self, sc):
       - Initialize with SparkContext `sc`.
     - Method proc_rdd(self, rdd):
       - Process each `line` in `rdd` to generate words, then bigrams, and parallelize
bigrams.
   - Class WC.NetSetup:
     - Method start_nc(port):
       - Start netcat using `subprocess` to send data to specified `port`.
   - Class WC.SparkSetup:
     - Method __init__(self):
       - Initialize with `sc` and `ssc` as `None`.
     - Method setup_sc(self):
       - Setup SparkContext and StreamingContext.
   - Class WC.WindowProc:
     - Method __init__(self, ssc, proc):
       - Initialize with StreamingContext `ssc` and processing object `proc`.
     - Method proc_window(self, lines):
       - Process `lines` within a sliding window, log data, and compute bigram counts.
     - Method log_window_data(self, time, rdd):
       - Log window data including sentence count and content.
     - Method log_bigram_counts(self, time, rdd):
       - Log bigram counts and their occurrences.

3. Define WC Main Class:
   - Method __init__(self, host, port):
     - Initialize with `host`, `port`, and components `spark_setup`, `proc`,
`window_proc`, and `empty_line_count`.
   - Method setup(self):
      - Start netcat, setup Spark, and initialize processing components.
   - Method run(self):
      - Setup the streaming context and process incoming lines.
   - Method check_empty_lines(self, time, rdd):
      - Check for empty lines, increment counter, generate logs, and stop streaming if needed.
   - Method generate_log_files(self):
      - Generate JSON and text log files with window and bigram data.

4. Main Execution:
   - Create an instance of `WC` with default `host` and `port`.
   - Run the instance to start the Spark streaming process.
```

2.2 Algorithm Explanation

1. A PySpark development setup is being prepared, incorporating essential elements such as SparkConf, SparkContext, and StreamingContext.
2. Our focus is on building a program that analyzes word pairs in streaming text data.
3. Using two execution threads and a batch interval of one second, we're establishing a local StreamingContext.
4. Netcat is being utilized to create a DStream that represents data flowing from a TCP source (localhost:9999).
5. Functions for text processing are under development, aimed at word separation, lowercase conversion, and punctuation elimination.
6. From the cleaned word list, we're crafting a method to generate bigrams.
7. The data stream undergoes a sliding window transformation, with a three-second window length and a two-second sliding interval.
8. Within these sliding windows, we're tallying and displaying the frequency of each bigram.
9. We're evaluating how our selected window size and sliding interval affect the bigram analysis.
10. Comprehensive logs of the processed information are being generated in both JSON and plain text formats.
11. Result analysis and visualization are underway, encompassing sentence counts per window, ongoing averages, and trend patterns.

2.3 Code Implementation Explanation and Details

1. Imports and Setup:

```
import sys
import re
import subprocess
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext
import logging
import warnings
import json

# Suppress INFO and WARNING logs
logging.getLogger("py4j").setLevel(logging.ERROR)
logging.getLogger("py4j.java_gateway").setLevel(logging.ERROR)

warnings.filterwarnings("ignore", category=UserWarning, message=".*psutil.*")
```

The following libraries are imported to enable various functionalities:

1. Sys provides access to system-specific parameters, although it's not explicitly utilized in the code shown.
2. Regular expressions are made available through the re library, which is employed for extracting words.
3. To simulate data input streaming, the subprocess module is utilized to execute a shell script.
4. PySpark's SparkConf and SparkContext, essential for configuring Spark, are brought in via the pyspark import.
5. Streaming capabilities in Spark are enabled by importing StreamingContext from pyspark.streaming.
6. To manage and reduce unnecessary output, the logging and warnings libraries are incorporated.
7. JSON data manipulation, particularly for log generation, is facilitated by importing the json module.

The logging configuration suppresses INFO and WARNING logs from py4j, which is the bridge between Python and Java used by PySpark. This reduces noise in the console output. The warnings filter ignores warnings related to psutil, which is often used by Spark but not directly relevant to this application.

2. WC (Word Count) Class:

a. Utils:

```
class Utils:
    @staticmethod
    def proc_line(line):
        words = re.findall(r'\w+', line.lower())
        return words

    @staticmethod
    def gen_bigrams(words):
        return [(words[i], words[i + 1]) for i in range(len(words) - 1)]
```

A static function named proc_line performs text processing. It accepts a single line as input, transforms all characters to lowercase, then employs a regex pattern (\w+) to identify sequences of alphanumeric characters and underscores. The output is a collection of extracted words.

Bigram generation is handled by the static method gen_bigrams. This function receives a sequence of words as its argument. Through the use of a list comprehension, it constructs pairs from consecutive words in the sequence. The result is a list containing these word pairs as tuples.

b. Log:

```
class Log:
    @staticmethod
    def log_lines(time, rdd):
        line_count = rdd.count()
        lines = rdd.collect()
        print(f"Received {line_count} lines at time {time}:")
        for line in lines:
            print(line)
```

For debugging purposes, a static function called log_lines is employed. It accepts two parameters: an RDD and its corresponding timestamp. This method performs several operations: it tallies the total number of entries in the given RDD, gathers all the lines present, and then displays them. By doing so, it aids in the observation and tracking of the incoming data stream's content and volume.

c. Proc:

```
class Proc:
    def __init__(self, sc):
        self.sc = sc

    def proc_rdd(self, rdd):
        lines = rdd.collect()
        all_words = []
        for line in lines:
            all_words.extend(WC.Utils.proc_line(line))

        bigrams = WC.Utils.gen_bigrams(all_words)
        return self.sc.parallelize(bigrams)
```

The constructor method, denoted as init, sets up a Proc object by incorporating a SparkContext (sc) as its parameter.
A method named proc_rdd is responsible for RDD manipulation. Its workflow involves several steps: first, it aggregates all entries from the input RDD. Then, it applies word extraction to each individual entry. Following this, it merges all extracted words into a unified list. From this consolidated list, it then generates bigrams. Finally, it utilizes sc.parallelize() to create and return a fresh RDD composed of these newly formed bigrams.

d. NetSetup:

```
class NetSetup:
    @staticmethod
    def start_nc(port):
        netcat_script =
"/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_4/src/utils
/send_paragraph.sh"
        subprocess.Popen([netcat_script, str(port)])
```

A static method called start_nc is responsible for initiating a netcat process. It achieves this by executing a shell script, which acts as a simulator for streaming data input. The method operates by launching the script as a subprocess, with the designated port being passed to it as a parameter. This approach allows for the simulation of real-time data streaming into the system.

e. SparkSetup:

```
class SparkSetup:
    def __init__(self):
        self.sc = None
        self.ssc = None

    def setup_sc(self):
        conf = SparkConf().setAppName("WC").setMaster("local[2]")
        self.sc = SparkContext(conf=conf)
        self.ssc = StreamingContext(self.sc, 1)
```

The constructor method, init, establishes initial null values for both SparkContext (sc) and StreamingContext (ssc). Configuration of the Spark environment is handled by the setup_sc method. It begins by generating a SparkConf object, assigning it the application name "WC" and configuring it to run locally with two worker threads via the "local[2]" setting. Subsequently, it instantiates a SparkContext using this configuration. Finally, it creates a StreamingContext, setting the batch processing interval to one second.

f. WindowProc:

```
class WindowProc:
    def __init__(self, ssc, proc):
        self.ssc = ssc
        self.proc = proc
        self.window_count = 0
        self.log_data = []

    def proc_window(self, lines):
        windowed_lines = lines.window(3, 2)
        windowed_lines.foreachRDD(self.log_window_data)

        bigrams = windowed_lines.transform(lambda rdd: self.proc.proc_rdd(rdd))

        bigram_counts = bigrams.map(lambda bigram: (bigram, 1)).reduceByKey(lambda
x, y: x + y)
        bigram_counts.foreachRDD(self.log_bigram_counts)

    def log_window_data(self, time, rdd):
        self.window_count += 1
        lines = rdd.collect()
        window_data = {
            "window_number": self.window_count,
            "sentence_count": len(lines),
            "sentences": lines,
            "bigrams": []
        }
        self.log_data.append(window_data)

        print(f"Window {self.window_count}:")
        print(f"Number of sentences: {len(lines)}")
        print("Sentences:")
        for line in lines:
            print(f"  {line}")

    def log_bigram_counts(self, time, rdd):
        bigram_counts = rdd.collect()
        current_window = self.log_data[-1]
        current_window["bigrams"] = [{"bigram": list(bigram), "count": count} for
bigram, count in bigram_counts]
```

```
        print("Bigrams and their co-occurrences count:")
        for bigram, count in bigram_counts:
            print(f"  {bigram}: {count}")
        print()
```

The constructor method initializes the object with StreamingContext and Proc instances, while also setting up variables for window counting and data logging.

The core processing functionality is encapsulated in the proc_window method. It establishes a sliding window of 3-second duration, advancing every 2 seconds (lines.window(3, 2)). This method then performs several operations: logging the windowed data, transforming each RDD to extract bigrams, tallying these bigrams through map and reduceByKey operations, and finally logging the resulting bigram counts.

For each RDD within the windowed DStream, the log_window_data method is invoked. It increments the window counter, retrieves all lines from the RDD, and constructs a dictionary (window_data) to store information about the current window. This data is then appended to the log_data list for future log file generation. Additionally, it outputs the window number, sentence count, and individual sentences to the console.

The log_bigram_counts method is called for every RDD containing bigram counts. Its role is to collect these counts from the RDD and incorporate them into the current window's data within the log_data list. Furthermore, it displays each bigram along with its corresponding count in the console output.

3. Main WC Class Methods:

```
    def __init__(self, host="localhost", port=9999):
        self.host = host
        self.port = port
        self.spark_setup = WC.SparkSetup()
        self.proc = None
        self.window_proc = None
        self.empty_line_count = 0

    def setup(self):
        WC.NetSetup.start_nc(self.port)
        self.spark_setup.setup_sc()
        self.proc = WC.Proc(self.spark_setup.sc)
        self.window_proc = WC.WindowProc(self.spark_setup.ssc, self.proc)

    def run(self):
        self.setup()

        lines = self.spark_setup.ssc.socketTextStream(self.host, self.port)
        lines.foreachRDD(self.check_empty_lines)

        self.window_proc.proc_window(lines)

        self.spark_setup.ssc.start()
        self.spark_setup.ssc.awaitTermination()

    def check_empty_lines(self, time, rdd):
        WC.Log.log_lines(time, rdd)
        lines = rdd.collect()
        if all(line.strip() == "" for line in lines):
            self.empty_line_count += 1
            if self.empty_line_count == 3:
                self.generate_log_files()
                self.spark_setup.ssc.stop(stopSparkContext=True, stopGraceFully=True)
        else:
            self.empty_line_count = 0


    def generate_log_files(self):
        # Generate JSON file
        output_data = {
            "windows": self.window_proc.log_data
        }

        with
open("/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_4/src/
output/output_log.json", "w") as f:
            json.dump(output_data, f, indent=2)

        print("JSON log file generated: output_log.json")

        # Generate text file
        with
open("/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_4/src/
output/output_log.txt", "w") as f:
            for window in self.window_proc.log_data:
                f.write(f"Window {window['window_number']}:\n")
                f.write(f"Number of sentences: {window['sentence_count']}\n")
                f.write("Sentences:\n")
                for sentence in window['sentences']:
                    f.write(f"  {sentence}\n")
                f.write("Bigrams and their co-occurrences count:\n")
                for bigram in window['bigrams']:
                    f.write(f"  {tuple(bigram['bigram'])}: {bigram['count']}\n")
                f.write("\n")

        print("Text log file generated: output_log.txt")
```

1. The constructor method sets up a WC object, accepting host and port parameters (defaulting to localhost:9999), and initializes necessary components.
2. Preparation for execution is handled by the setup method, which invokes functions to launch netcat, configure Spark contexts, and create processing objects.
3. The run method serves as the primary driver of the streaming process. It establishes a DStream from a socket connection, verifies for the presence of empty lines, applies processing to windowed data, and initiates the StreamingContext.
4. A method named check_empty_lines examines each RDD for vacant entries. It maintains a counter that increments when an RDD consists entirely of empty lines. Upon reaching a count of 3 consecutive empty RDDs, it triggers the generation of log files and halts the StreamingContext.

Log file creation is managed by the generate_log_files method, which produces two distinct files at the conclusion of streaming:
1. A JSON format log: This method constructs a dictionary named output_data, containing a single key "windows" that encompasses all logged information from self.window_proc.log_data. This structured data is then written to a JSON file, utilizing json.dump() with indentation for enhanced readability.
2. A text format log: This file is generated by iterating through each window in self.window_proc.log_data, presenting the information in a more human-friendly layout. For every window, it records the window number, sentence count, all sentences, and all bigrams along with their respective counts.
Upon successful creation of each file, the method outputs a confirmation message to indicate the completion of file generation.

6. Main Execution:

```
if __name__ == "__main__":
    wc = WC()
    wc.run()
```

The script's main execution point is defined here. When the script is executed directly rather than being imported, it instantiates a WC object and invokes its run method, thereby initiating the entire data streaming and processing workflow.

This code exemplifies sophisticated Spark Streaming techniques, encompassing windowed computations, transformations without state (map), transformations with state (reduceByKey), and concrete actions (collect, count). It illustrates the process of constructing a streaming pipeline, performing real-time data processing, and producing comprehensive logs of the processed information.

The implementation's robustness is enhanced by the inclusion of detailed logging functions (log_window_data and log_bigram_counts) and the creation of extensive log files (generate_log_files). These features significantly improve the application's capacity to monitor and examine the processed data. Such functionality proves invaluable for troubleshooting, conducting audits, and performing in-depth analysis of the streaming process.

2.4 Spark Windowing and Interval Concepts and Techniques

Our implementation incorporates Spark's windowing principles and methodologies. We'll provide an in-depth explanation of these elements, placing particular emphasis on the dimensions and frequency of our windows.

The specifics of our window configuration and its operational mechanics will be elucidated in the following discussion. We'll explore how these windowing concepts are applied within our codebase, shedding light on the rationale behind our chosen window parameters and their impact on data processing.

1. Spark Streaming Context and DStreams:
The initial step in our process involves the configuration of our Spark Streaming environment:

To begin our data streaming operations, we first establish the necessary Spark Streaming context. This foundational setup is crucial for the subsequent steps in our data processing pipeline.

```
self.ssc = StreamingContext(self.sc, 1)
```

Our streaming context has been configured with a batch processing interval of one second. This context serves as the foundation for establishing our DStream, which we derive from a socket connection:

The DStream, built upon our one-second interval streaming context, is created by tapping into a socket source. This setup forms the basis of our real-time data processing pipeline.

```
lines = self.spark_setup.ssc.socketTextStream(self.host, self.port)
```

Our continuous flow of data is embodied in this DStream, which segments the incoming information into discrete units, each spanning a single second.

The DStream serves as a representation of our uninterrupted data flow, partitioning it into sequential batches, with each batch encompassing a one-second duration.

2. Window Creation and Parameters:

The DStream undergoes a transformation as we implement our windowing operation:

To process our streaming data in specific time-based chunks, we now apply a windowing function to our established DStream.

```
windowed_lines = lines.window(3, 2)
```

Our sliding window configuration is defined by two crucial parameters:

1. A three-second duration forms the foundation of each window, allowing us to capture and analyze short-term but significant data trends and patterns.
2. Every two seconds, a fresh window is initiated, determining both the frequency of window creation and the incremental advancement of our windowed view.
3. This configuration results in the following characteristics:
4. Bi-secondly, a new data window comes into existence.
5. Each window encapsulates data from the preceding three-second period.
6. Consecutive windows share a one-second data overlap, ensuring continuity in our analysis.

3. Window Overlap and Progression:

1. The interplay between our three-second window duration and two-second sliding interval creates a data intersection:
2. A one-second data segment is shared between each window and its predecessor.
3. This shared portion aids in maintaining analytical continuity and capturing patterns that might emerge at window boundaries.
4. Our windowing mechanism progresses through time in the following manner:
5. The initial window spans from the start to the third second.
6. The subsequent window covers the period from the second to the fifth second, sharing the 2-3 second interval with the first window.
7. The third window encompasses the fourth to seventh second, with an overlap from 4-5 seconds with the second window.

8. This pattern continues to repeat as time advances...

4. Window Processing Mechanism:

To handle the information contained within our windowed data structure, we employ the `foreachRDD()` method as our primary processing tool.

```
windowed_lines.foreachRDD(self.log_window_data)
```

The `foreachRDD()` method is invoked for every RDD present in our windowed DStream, enabling us to process the entirety of data that has accumulated during each 3-second time frame.

Each RDD within our windowed DStream triggers a call to this method, allowing us to handle all information that has been received within the corresponding 3-second window.

5. Transformations within Windows:

Our windowed data undergoes a series of sophisticated transformations:
The information captured within our time-based windows is subjected to intricate processing operations:

```
bigrams = windowed_lines.transform(lambda rdd: self.proc.proc_rdd(rdd))
```

The text data within each 3-second window is processed and converted into bigrams through the application of the `transform()` function.

Employing the `transform()` method, we manipulate the textual information and produce bigrams for every window spanning three seconds.

6. Stateful Processing and Aggregations:

Note - *The below lines of code are not together in our PySpark code but I have kept them together to demonstrate and explain how we preserve states and how we do counts*

```
self.window_count += 1

self.log_data.append(window_data)

bigram_counts = bigrams.map(lambda bigram: (bigram, 1)).reduceByKey(lambda x, y: x + y)
```

Instance variables are utilized to preserve state across different windows:

1. The window counter is incremented with each iteration.
2. Window-specific data is appended to our log collection.

For each individual window, we conduct data consolidation:

1. Bigrams are tallied by mapping each to a value of 1, then applying a reduction operation to sum these values.
2. This process allows us to determine the frequency of each bigram's appearance within the 3-second time frame of the window.

7. Output and Result Handling:

```
bigram_counts.foreachRDD(self.log_bigram_counts)
```

The outcomes of each window undergo collection and processing:

1. We apply the log_bigram_counts method to every RDD within the bigram_counts DStream.
2. This functionality is invoked for each 3-second window, enabling us to record and examine the data unique to every individual time segment.

8. Benefits of Our Window Configuration:

1. Analysis depth is achieved through the three-second duration, providing a substantial data sample.
2. Updates occur at short intervals due to the two-second advancement of each window.
3. Continuity across consecutive windows is preserved by the one-second data intersection.

9. Considerations for Window Size and Interval:

1. Extending the window beyond 3 seconds would increase the data volume per window, at the cost of less frequent refreshes.
2. Contracting the window to less than 3 seconds would yield more regular updates, potentially at the expense of detecting extended patterns.
3. Increasing the sliding interval beyond 2 seconds could alleviate computational strain, but might overlook brief trends.
4. Decreasing the sliding interval to less than 2 seconds would offer more detailed updates, though it would intensify the processing demands.

The configuration we've selected - a three-second window advancing every two seconds - achieves an equilibrium between analytical depth and update regularity. This setup enables us to identify short-duration trends while maintaining a sensible refresh rate, rendering it particularly effective for instantaneous analysis of our streaming data.

Our windowing strategy facilitates sophisticated examination of continuous data flows, encompassing trend identification, calculation of rolling averages, and recognition of patterns within defined temporal boundaries. We've found this approach to be especially potent when applied to real-time analytics on uninterrupted data streams.

3. Running the Code

3.1 Log Properties File

```
# Set everything to be logged to the console
log4j.rootCategory=ERROR, console

# Setup appender
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}: %m%n

# Reduce the logging level for various components
log4j.logger.org.apache.spark=ERROR
log4j.logger.org.apache.spark.streaming=ERROR
log4j.logger.org.spark-project=ERROR
log4j.logger.org.spark-project.jetty=ERROR
log4j.logger.org.apache.hadoop=ERROR
log4j.logger.io.netty=ERROR
log4j.logger.org.apache.zookeeper=ERROR

# Reduce the logging level for specific warnings
log4j.logger.org.apache.spark.storage.BlockManager=ERROR
log4j.logger.org.apache.spark.storage.RandomBlockReplicationPolicy=ERROR
```

Log properties file explanation

Our PySpark streaming application, designed for text processing and bigram frequency computation, employs a tailored logging approach. The rationale behind our decisions is as follows:

The log4j root logging level is set to ERROR, as our application independently manages most crucial output. Key data about streaming windows, sentences, and bigram tallies are logged via print statements. This ERROR-level configuration ensures only critical Spark framework issues are reported, maintaining a clutter-free console.

We've opted to mute INFO and WARNING logs from specific loggers such as "py4j" and "py4j.java_gateway". This allows us to concentrate on our application's output without distraction from Spark's Python-Java interface.

Further noise reduction is achieved by utilizing Python's warnings module to disregard certain psutil-related UserWarnings.

For application-specific logging, we favored print statements over a formal logging framework. This grants us precise control over output format and content, which is essential for our windowed bigram analysis.

We developed bespoke logging methods like `log_window_data` and `log_bigram_counts`. These not only display formatted information in the console but also store data in a structured format (self.log_data). This dual approach enables generation of both human-readable console output and structured JSON and text log files upon process completion.

A mechanism to detect stream termination (three successive empty lines) and initiate log file generation was implemented. This ensures comprehensive data capture prior to Spark streaming context shutdown.

By emphasizing logging of windowed bigram analysis outputs and minimizing less pertinent system logs, we've crafted a logging strategy that offers clear, actionable insights into our data processing pipeline. This method facilitates application behavior monitoring, verification of windowing and bigram counting logic, and production of comprehensive logs for potential analysis or troubleshooting.

3.2 Spark Command Explanation

```
spark-submit --driver-java-options
"-Dlog4j.configuration=file:/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_krit
ikku_assignment_4/src/word_cooccurrence/log4j.properties" word_cooccurrence.py
```

The entry point for Spark application execution is the spark-submit command. It configures the Spark environment and initiates our application.

Java options for the driver process, which orchestrates our Spark application, are specified using the --driver-java-options flag.

Within the Java options, we define the log4j configuration file location:

1. The -D prefix establishes a Java system property.
2. We set the log4j.configuration property to direct log4j to our custom settings.
3. The file:/ prefix denotes a file path specification.
4. Following this is the complete path to our tailored log4j.properties file.

The final component, word_cooccurrence.py, represents our Python script that encapsulates the Spark application logic.
This command structure ensures proper Spark initialization, custom logging configuration, and execution of our specific application code.

We used this command for several reasons:

1. Tailored Log Management: Overriding Spark's default logging setup with our bespoke log4j.properties file enables precise control over log content and format.
2. Environmental Uniformity: Employing a custom configuration file ensures consistent logging behavior across diverse platforms or systems, provided the log4j.properties file is correctly located.
3. Modular Design: Segregating logging configuration from core application code facilitates modifications to logging behavior without altering the primary codebase.
4. Granular Oversight: Our specialized log4j.properties file permits differential logging levels for various Spark and application components, offering nuanced control over system verbosity.
5. Efficiency Considerations: Curtailing less critical logs potentially reduces logging-related I/O operations, possibly enhancing application performance, particularly when processing substantial datasets.

This command implementation guarantees that our Spark application utilizes our meticulously designed logging configuration. Consequently, we can concentrate on output most pertinent to our bigram analysis task, while minimizing interference from Spark's internal processes.

3.3 Step-by-Step Execution Guide

Environment Readiness:
Confirm the presence and proper configuration of Apache Spark on your system. Ensure Python is installed with all required libraries, including pyspark, re, and subprocess.

File Arrangement:
Position the word_cooccurrence.py script in its designated directory. Verify the log4j.properties file is correctly located as per the spark-submit command. Confirm the send_paragraph.sh script resides in /Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_4/src/utils/.

Terminal Initiation:
Launch a terminal session and navigate to the word_cooccurrence.py script's directory.

Application Launch:
Execute the spark-submit command:

```
spark-submit --driver-java-options
"-Dlog4j.configuration=file:/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_krit
ikku_assignment_4/src/word_cooccurrence/log4j.properties" word_cooccurrence.py
```

Output Monitoring:
Observe the console as the application initiates. Window information, including sentence counts and content, will be displayed. Bigrams and their frequencies follow each window's data.

Data Injection:
The application autonomously initiates a netcat process for data transmission from the send_paragraph.sh script.

Data Processing:
Monitor the application as it processes incoming data in windows, displaying received sentences and bigram tallies for each window.

Application Cessation:
Allow the application to self-terminate after encountering three successive empty lines. This process may span several minutes, contingent on the data stream.

Log Generation:
Post-termination, verify the creation of two log files:

1. output_log.json (JSON format)
2. output_log.txt (text format)

Confirm these files are saved in /Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_4/src/output/.

Output Verification:
Examine the console output for any anomalies or errors. Review the generated log files to ensure they contain the expected information.

Post-Execution Cleanup:
Note that the Spark context terminates automatically upon completion. For premature termination, use Ctrl+C in the terminal.

4. Data Streaming with Netcat

4.1 Netcat Extended Version Ncat Usage

```
#!/bin/bash

# File: send_paragraph.sh

# Check if the required arguments are provided
if [ "$#" -ne 1 ]; then
   echo "Usage: $0 <port>"
   exit 1
fi

port=$1

# Create a temporary file to store the modified paragraph
temp_file=$(mktemp)

# Split the paragraph into sentences and store them in the temporary file
cat
/Users/kritikkumar/Downloads/shaivipa_kritikku_assignment_4/src/setup/paragraph.txt |
sed 's/\. /\.\'$'\n/g' > "$temp_file"

# Append a whitespace to the temporary file
echo " " >> "$temp_file"

# Send each sentence with a 1-second delay through netcat
cat "$temp_file" | while read -r sentence; do
   echo "$sentence"
   sleep 1
done | ncat -lk -p "$port"

# Clean up the temporary file
rm "$temp_file"
```

Script Overview:
Our bash script, send_paragraph.sh, emulates a streaming data source for the PySpark application. It processes a paragraph, divides it into sentences, and transmits these sentences sequentially via a network connection.

Port Configuration:
The script is designed to accept a port number as an input parameter, allowing flexible specification of the network communication port.

Text Manipulation:
A paragraph is extracted from a predetermined file (`/Users/kritikkumar/Downloads/shaivipa_kritikku_assignment_4/src/setup/paragraph.txt`).

Using sed, we segment this paragraph into individual sentences, each occupying a separate line. The processed text is stored in a temporary file.

Delimiter Insertion:
A whitespace character is appended to the temporary file's end, ensuring clear demarcation between sentences during network transmission.

Data Transmission:
Each sentence is read from the temporary file and echoed. A one-second interval between sentences simulates a time-distributed data stream.

Ncat Implementation:
The ncat command, a modern netcat variant, establishes the network connection:

1. The -l flag initiates a listening mode for incoming connections.
2. The -k option maintains an open connection, permitting multiple client connections over time.
3. The -p flag designates the listening port.

PySpark Integration:
Our PySpark application initiates this script as a subprocess. The Spark streaming context then establishes a connection to the port monitored by this script, enabling our application to receive the sentences as a data stream.

Post-Execution Cleanup:
Upon completion of sentence transmission, the temporary file is deleted.

This configuration simulates a real-time data stream in a controlled environment. By transmitting sentences individually with a delay, we can evaluate our PySpark application's handling of streaming data processing, including its windowing and bigram counting capabilities. The utilization of netcat provides a straightforward and effective method for data transmission over a network connection, mimicking real-world scenarios where data might originate from a remote source.

4.2 How are we using netcat to create a data stream that our Spark application listens to?

Netcat Server Configuration:
The send_paragraph.sh script employs ncat to establish a server awaiting incoming connections:

```
ncat -lk -p "$port"
```

1. Listen mode is activated via the -l flag.
2. Persistent connection is maintained using the -k option.
3. The -p "$port" parameter designates the listening port.

Data Transmission Mechanism:
Processed sentences are channeled into the ncat command:

```
cat "$temp_file" | while read -r sentence; do
  echo "$sentence"
  sleep 1
done | ncat -lk -p "$port"
```

This setup broadcasts each sentence to connected clients via the ncat server.

PySpark Connectivity:
A streaming context in the PySpark application establishes a connection to the netcat server:

```
lines = self.spark_setup.ssc.socketTextStream(self.host, self.port)
```

The socketTextStream method generates a DStream representing the incoming data from the netcat server.

Automated Server Initiation:
The netcat server is automatically launched upon PySpark application execution:

```
class NetSetup:
    @staticmethod
    def start_nc(port):
        netcat_script = 
"/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_4/src/utils
/send_paragraph.sh"
        subprocess.Popen([netcat_script, str(port)])
```

This method initiates the send_paragraph.sh script as a subprocess, which in turn starts the netcat server.

Data Reception:
Once active, the PySpark application continuously monitors the specified port for incoming data. Each text line transmitted by the netcat server is received as a new element in the DStream.

Stream Processing:
The received data stream undergoes processing, with windowing and bigram counting logic applied to each batch of incoming data.

This netcat implementation provides a straightforward yet effective method for simulating a real-time data stream. It enables testing of the Spark Streaming application's capacity to process data upon arrival, emulating scenarios of continuous data generation and network transmission. This approach is particularly valuable for development and testing, offering control over data flow and timing while exercising the application's streaming capabilities.

4.3 Shell Script Implementation

Our shell script is engineered to generate a simulated data stream for the PySpark application. Its structure is as follows:

1. Input validation initiates the script, verifying the correct number of command-line arguments. A single parameter is expected: the port number for data transmission.
2. A temporary file is created to house the processed paragraph, enabling efficient, one-time data preparation for subsequent streaming.
3. The source paragraph is extracted from a predetermined file and undergoes processing. Individual sentences are isolated and placed on separate lines within the temporary file. A single whitespace is appended to ensure clear sentence demarcation during streaming.
4. The script's core functionality revolves around a loop that iterates through each sentence in the temporary file. Each sentence is echoed (effectively transmitted) followed by a one-second pause, simulating time-distributed data arrival.
5. This loop's output is directed to the ncat command, configured to listen on the specified port, maintain an open connection for multiple clients, and continuously accept connections. This configuration allows the PySpark application to establish a connection and receive the streamed sentences.

Post-execution cleanup involves removing the temporary file.

This implementation efficiently simulates a streaming data source. The one-time paragraph processing and temporary file storage eliminate redundant processing. Ncat usage provides a straightforward network interface for PySpark application connectivity, mimicking real-world network-sourced data streaming scenarios.

The intersentence delay allows the PySpark application processing time for each data piece, facilitating observation and debugging of streaming behavior. In essence, this script offers a controlled, replicable method for testing the PySpark streaming application's handling of incoming data.

4.4 Advantages over terminal execution

Consistency and Automation:
Our script streamlines data preparation and transmission, ensuring uniformity across multiple executions. This consistency is vital for application testing and troubleshooting, eliminating the need for manual data entry or copy-pasting.

Realistic Data Flow Simulation:
The implemented one-second intersentence delay creates a more authentic streaming data simulation. This measured pacing facilitates individual data piece processing, enhancing observability and debuggability of streaming behavior.

Flexible Port Assignment:
Port number specification via command-line argument offers adaptability and ease of modification. This approach surpasses the inconvenience of script alteration or manual port changes for each execution.

Data Preprocessing:
Preliminary paragraph segmentation into sentences ensures consistent data formatting without manual intervention.

Sustained Connectivity:
Ncat's keep-alive option maintains an open connection, allowing our PySpark application to connect at will without manual stream reinitiation.

Robust Error Management:
Integrated error checking mechanisms (e.g., argument count verification) prevent misuse and provide clear feedback on incorrect usage.

Automatic Resource Management:
The script self-cleans temporary files, preventing system clutter post-execution.

Modular Design:
Isolation of data streaming logic into a separate script maintains the clarity and focus of the main PySpark application code on data processing.

Real-world Scenario Emulation:
This approach more accurately mimics real-world conditions where data originates from network sources, enabling more realistic application testing.

Enhanced Reproducibility:
Either of us from our team can replicate the exact data stream without needing to understand or reproduce complex terminal commands.

These advantages collectively enhance the efficiency, reliability, and real-world representation of our testing process, ultimately contributing to a more robust PySpark streaming application.

4.5 How does our shell script get executed from the main PySpark code, how are they related and how do they work together?

Execution Initiation:
The main PySpark code (word_cooccurrence.py) incorporates a NetSetup class featuring a static start_nc method. This method is invoked during WC (Word Cooccurrence) class initialization, triggering the shell script execution as a subprocess.

Script Activation:
The start_nc method launches the send_paragraph.sh script, providing the port number as an argument, thus initiating the background data streaming process.

Script Interdependence:
The shell script (send_paragraph.sh) functions as the data provider, while the PySpark code (word_cooccurrence.py) serves as the data consumer. Their interaction is facilitated through a network connection on a specified port.

Connectivity Establishment:
The PySpark code configures a StreamingContext and generates a DStream using socketTextStream, connecting to the same host and port utilized by the shell script, thereby bridging the data source and processing logic.

Data Transmission:
The shell script segments the paragraph into sentences, transmitting them sequentially over the network connection. The PySpark application receives this data stream through the established DStream.

PySpark Data Handling:
As data flows in, the PySpark application processes it in windows, performing operations such as word splitting, bigram generation, and frequency counting. The processed data is subsequently logged and stored.

Operational Synchronization:
The shell script continues data transmission until the paragraph is exhausted. Concurrently, the PySpark application processes the incoming data until it detects a termination signal (three consecutive empty lines).

Process Conclusion:
Upon detecting the stream's end, the PySpark application generates log files and terminates the Spark context. The shell script concludes naturally after complete data transmission.

This integrated setup simulates a real-time data streaming scenario within a single application. Manual initiation of the data stream in a separate terminal is unnecessary - the entire process is automated upon PySpark application execution. This cohesive approach ensures synchronization between data source and processing, streamlining the testing and execution process for enhanced reproducibility.

By incorporating the shell script execution into the main PySpark code, we've created a self-sufficient system managing both data generation and processing. This methodology simplifies our workflow and maintains consistency in our streaming data simulation.

4.6 Synchronization for Real-time Streaming Simulation

Startup Synchronization:
The PySpark code initializes the streaming context before launching the shell script, ensuring data source readiness prior to processing initiation.

Network Endpoint Coordination:
Identical port numbers are employed in both the shell script and PySpark code, guaranteeing connectivity between the data transmitter and receiver.

Regulated Data Transmission:
The shell script implements a one-second intersentence delay, enabling near real-time data processing by the PySpark application and simulating a consistent incoming data stream.

Time-based Data Segmentation:
PySpark code utilizes windowed operations (3-second windows, 2-second slides), aligning with the data transmission rate to capture multiple sentences per window for meaningful bigram analysis.

Uninterrupted Processing:
The PySpark application persistently processes data throughout transmission. DStream abstraction manages the continuous nature of incoming data, facilitating operations on each arriving RDD.

Coordinated Termination:
A termination detection mechanism is implemented in the PySpark code, identifying three consecutive empty lines as the stream's end, enabling graceful application closure post-data processing.

Data Flow Management:
Spark Streaming's inherent buffering and backpressure mechanisms prevent data loss and memory issues by automatically adjusting to processing speed discrepancies.

Robust Error Management:
Both scripts incorporate error handling, with the shell script verifying correct usage and the PySpark code managing scenarios like network disconnections.

Real-time Monitoring:
Simultaneous logging of received data and processing outcomes enables monitoring of data transmission and processing synchronization, confirming accurate data capture and analysis.

Synchronized Resource Release:
The PySpark application handles final data processing, log file generation, and Spark context termination, while the shell script manages its temporary file cleanup, ensuring coordinated resource release post-simulation.

This synchronization approach effectively simulates real-time data streaming. The regulated data flow from the shell script, coupled with Spark Streaming's real-time processing capabilities, creates a realistic testing environment for the streaming data analysis application.

Through meticulous coordination of data generation and processing components, we've engineered a system that emulates a real-world streaming data application, facilitating comprehensive testing of bigram analysis logic under production-like conditions.

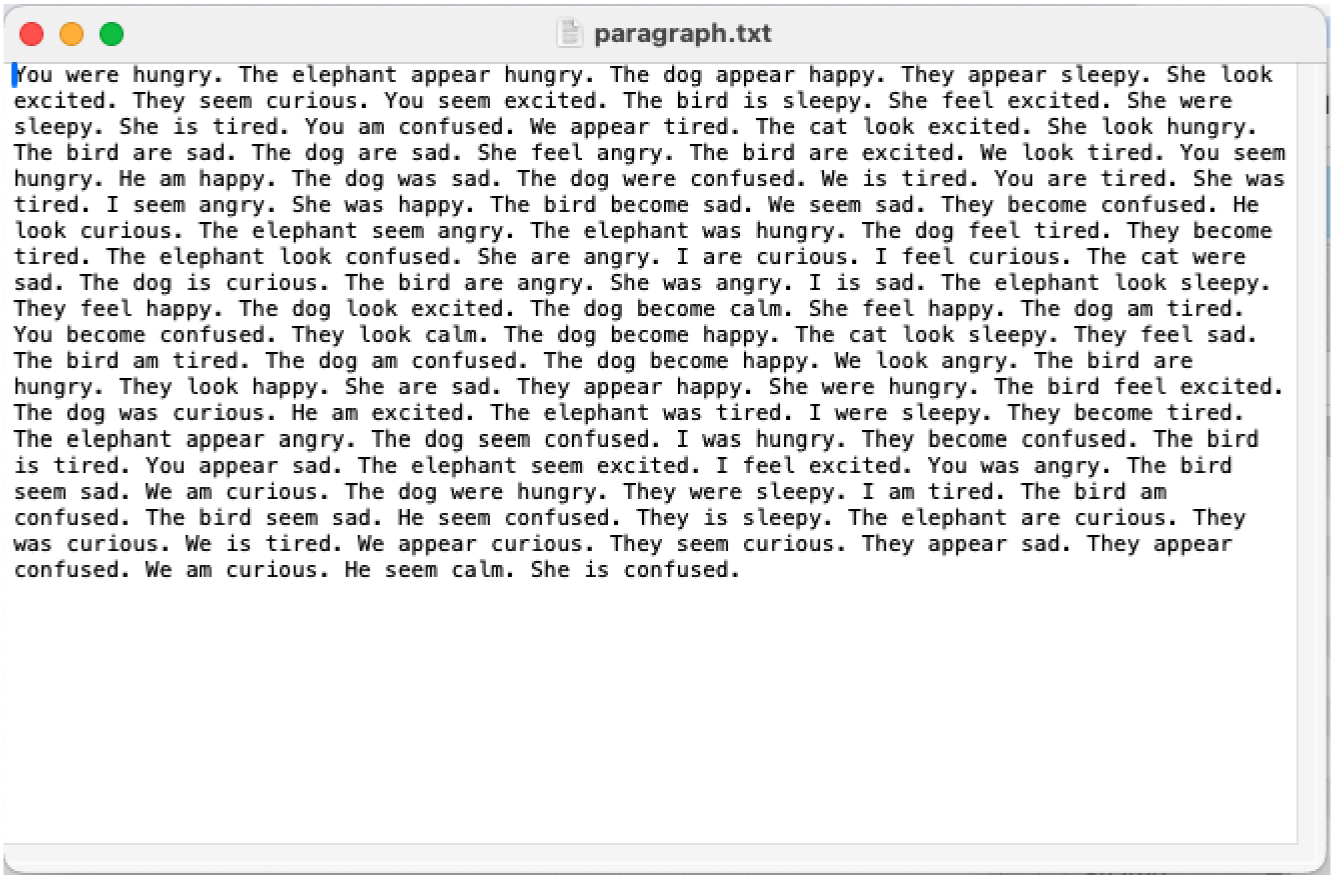
4.7 Why are we making a temporary file instantaneously for streaming - A design choice

By using this approach with a temporary file, we've created a more robust, efficient, and flexible system for simulating our streaming data input, which better serves the needs of our PySpark streaming application testing and development. It allows us to handle potential issues upfront and provides a more controlled environment for our streaming experiments, while still achieving our core objective of simulating a continuous data stream.

It's important to reiterate that despite the use of a temporary file, we are still effectively sending the contents of paragraph.txt. The temporary file is merely an **intermediary step** that allows for preprocessing and optimization. The original source of our data remains paragraph.txt, and the content being streamed is fundamentally the same. We're simply using the temporary file as a mechanism to enhance our streaming process, but the essence of what we're streaming – the text from paragraph.txt – remains unchanged. This approach allows us to maintain the **integrity of our original data** while benefiting from the advantages of preprocessing and controlled streaming.

4.8 If we aren't using netcat from the terminal, how can we monitor the sentences being sent over localhost port 9999 to ensure the Spark Streaming application is receiving and processing the data correctly?

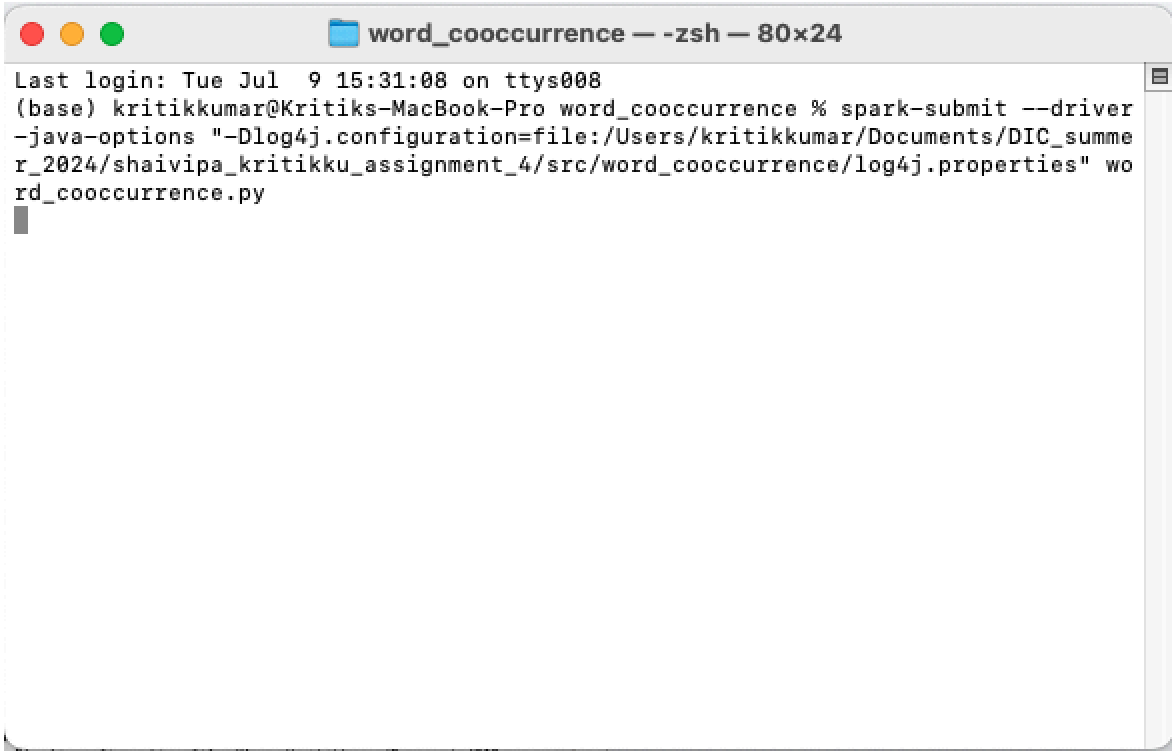
What our paragraph.txt looks like for verification purposes of sentences being sent -



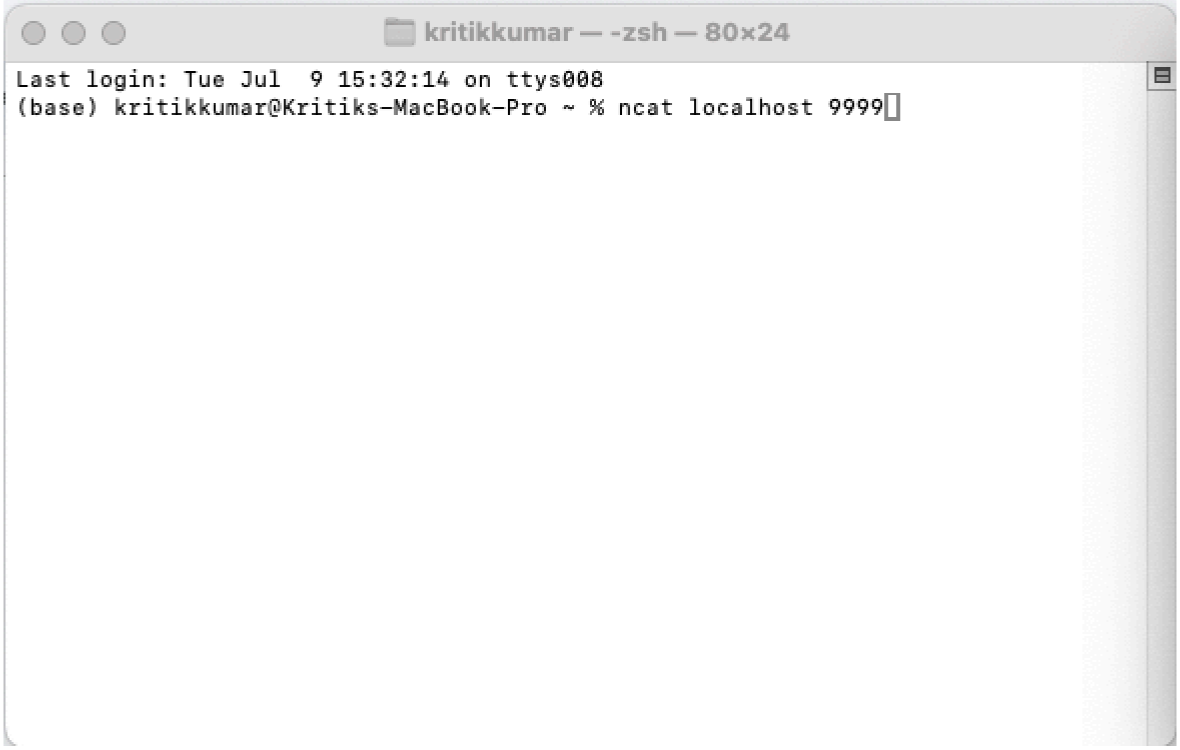
```
You were hungry. The elephant appear hungry. The dog appear happy. They appear sleepy. She look
excited. They seem curious. You seem excited. The bird is sleepy. She feel excited. She were
sleepy. She is tired. You am confused. We appear tired. The cat look excited. She look hungry.
The bird are sad. The dog are sad. She feel angry. The bird are excited. We look tired. You seem
hungry. He am happy. The dog was sad. The dog were confused. We is tired. You are tired. She was
tired. I seem angry. She was happy. The bird become sad. We seem sad. They become confused. He
look curious. The elephant seem angry. The elephant was hungry. The dog feel tired. They become
tired. The elephant look confused. She are angry. I are curious. I feel curious. The cat were
sad. The dog is curious. The bird are angry. She was angry. I is sad. The elephant look sleepy.
They feel happy. The dog look excited. The dog become calm. She feel happy. The dog am tired.
You become confused. They look calm. The dog become happy. The cat look sleepy. They feel sad.
The bird am tired. The dog am confused. The dog become happy. We look angry. The bird are
hungry. They look happy. She are sad. They appear happy. She were hungry. The bird feel excited.
The dog was curious. He am excited. The elephant was tired. I were sleepy. They become tired.
The elephant appear angry. The dog seem confused. I was hungry. They become confused. The bird
is tired. You appear sad. The elephant seem excited. I feel excited. You was angry. The bird
seem sad. We am curious. The dog were hungry. They were sleepy. I am tired. The bird am
confused. The bird seem sad. He seem confused. They is sleepy. The elephant are curious. They
was curious. We is tired. We appear curious. They seem curious. They appear sad. They appear
confused. We am curious. He seem calm. She is confused.
```

To ensure proper data transmission and processing in our Spark Streaming application, we employ ncat as an independent monitoring tool. The procedure is as follows:

1. Initiate two new terminal windows.



```
Last login: Tue Jul  9 15:31:08 on ttys008
(base) kritikkumar@Kritiks-MacBook-Pro word_cooccurrence % spark-submit --driver
-java-options "-Dlog4j.configuration=file:/Users/kritikkumar/Documents/DIC_summe
r_2024/shaivipa_kritikku_assignment_4/src/word_cooccurrence/log4j.properties" wo
rd_cooccurrence.py
```



```
Last login: Tue Jul  9 15:32:14 on ttys008
(base) kritikkumar@Kritiks-MacBook-Pro ~ % ncat localhost 9999
```

2. Execute the following commands sequentially:
Terminal 1 -

```
spark-submit --driver-java-options
"-Dlog4j.configuration=file:/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_krit
ikku_assignment_4/src/word_cooccurrence/log4j.properties" word_cooccurrence.py
```

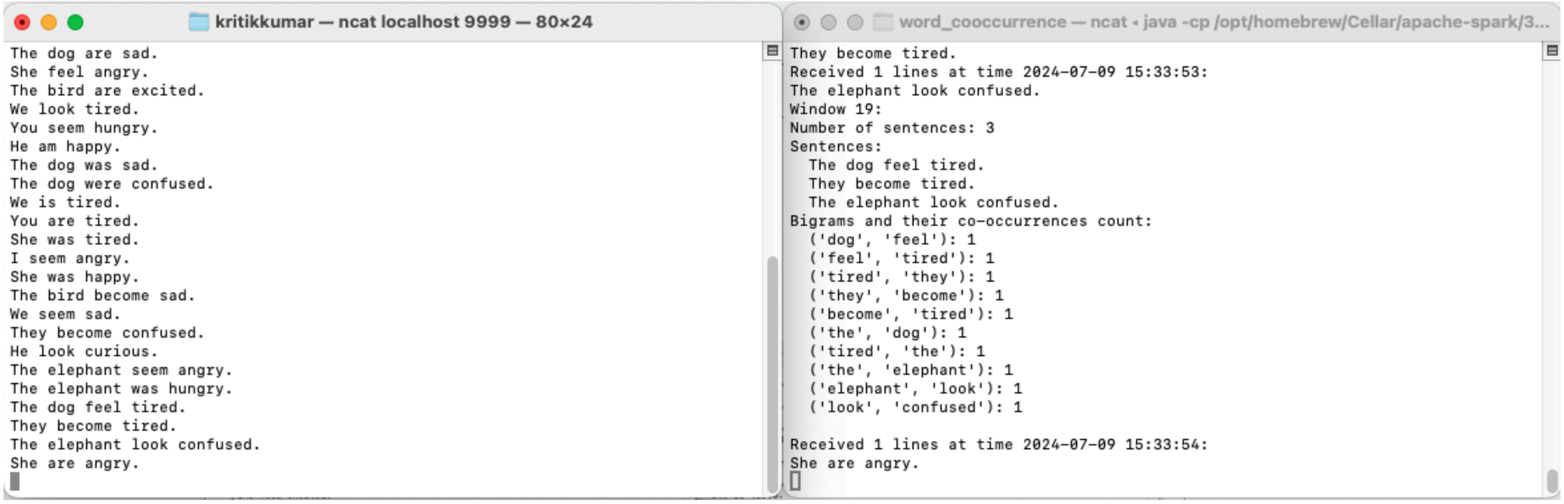
Terminal 2 -

```
ncat localhost 9999
```

Note: *The netcat listener must be initialized from our main code, running in the background, before we can use ncat localhost 9999. Therefore, the order above is important for executing the above commands.*

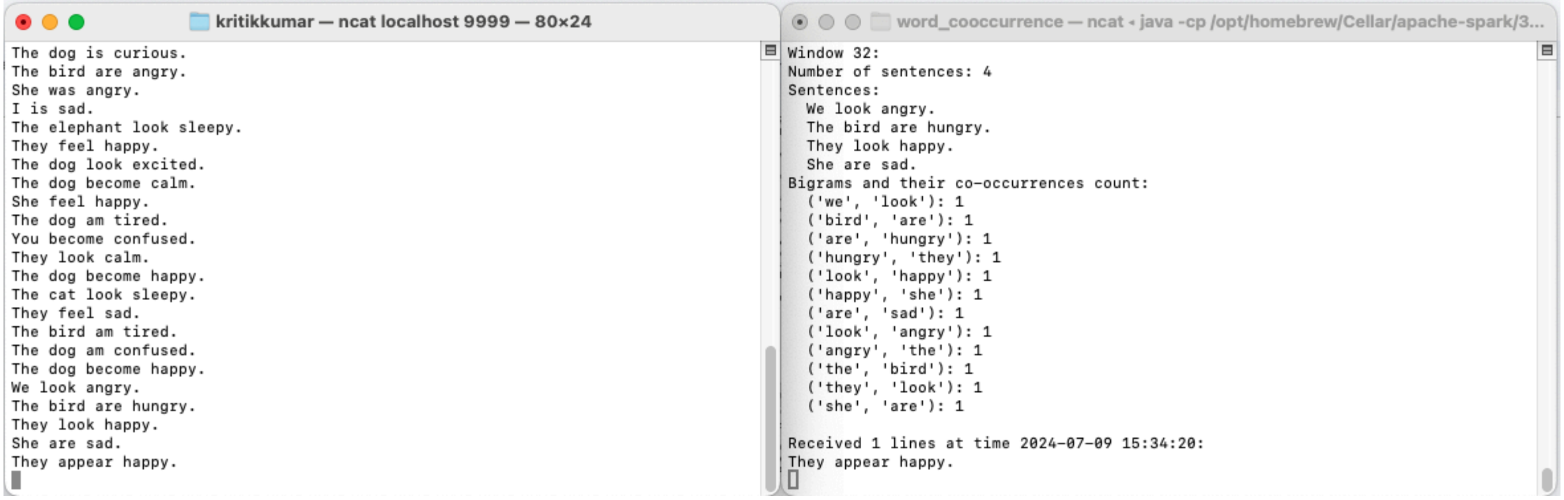
3. This setup connects to the identical TCP stream that our Spark Streaming application is processing.

By executing the ncat command, we create an additional client connected to our data stream, providing a real-time view of the transmitted data. The output appears in real-time when we run the Spark code and the netcat command.


kritikkumar — ncat localhost 9999 — 80×24
word_cooccurrence — ncat ◂ java -cp /opt/homebrew/Cellar/apache-spark/3...

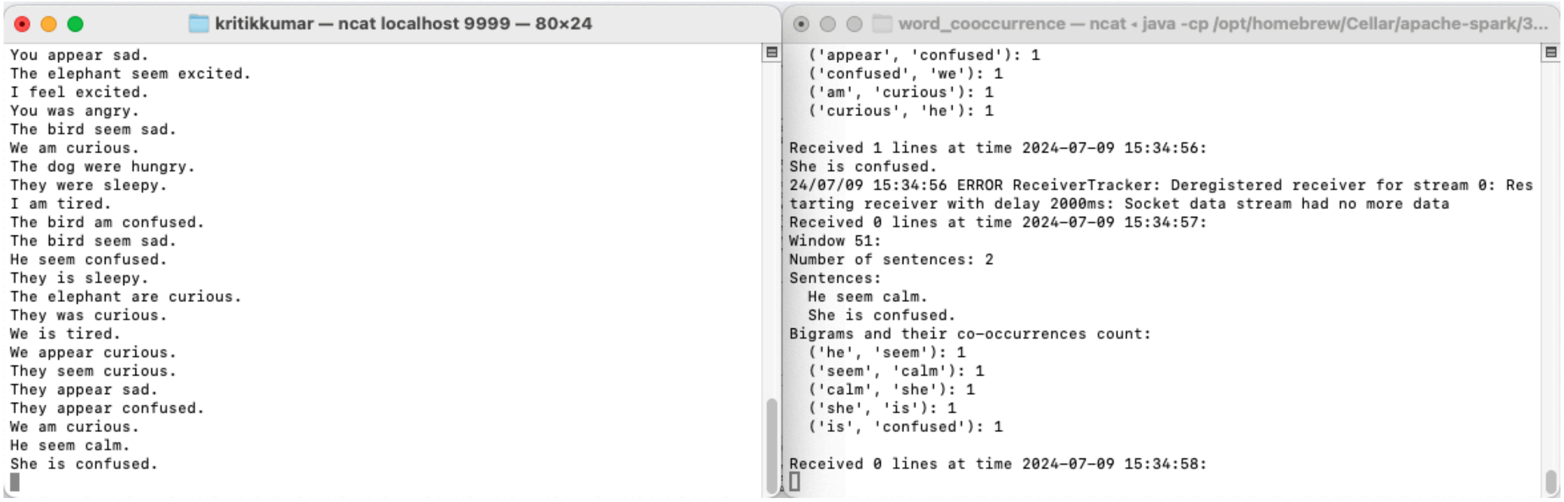
```
The dog are sad.
She feel angry.
The bird are excited.
We look tired.
You seem hungry.
He am happy.
The dog was sad.
The dog were confused.
We is tired.
You are tired.
She was tired.
I seem angry.
She was happy.
The bird become sad.
We seem sad.
They become confused.
He look curious.
The elephant seem angry.
The elephant was hungry.
The dog feel tired.
They become tired.
The elephant look confused.
She are angry.
```

```
They become tired.
Received 1 lines at time 2024-07-09 15:33:53:
The elephant look confused.
Window 19:
Number of sentences: 3
Sentences:
  The dog feel tired.
  They become tired.
  The elephant look confused.
Bigrams and their co-occurrences count:
  ('dog', 'feel'): 1
  ('feel', 'tired'): 1
  ('tired', 'they'): 1
  ('they', 'become'): 1
  ('become', 'tired'): 1
  ('the', 'dog'): 1
  ('tired', 'the'): 1
  ('the', 'elephant'): 1
  ('elephant', 'look'): 1
  ('look', 'confused'): 1

Received 1 lines at time 2024-07-09 15:33:54:
She are angry.
```


kritikkumar — ncat localhost 9999 — 80×24
word_cooccurrence — ncat ◂ java -cp /opt/homebrew/Cellar/apache-spark/3...

```
The dog is curious.
The bird are angry.
She was angry.
I is sad.
The elephant look sleepy.
They feel happy.
The dog look excited.
The dog become calm.
She feel happy.
The dog am tired.
You become confused.
They look calm.
The dog become happy.
The cat look sleepy.
They feel sad.
The bird am tired.
The dog am confused.
The dog become happy.
We look angry.
The bird are hungry.
They look happy.
She are sad.
They appear happy.
```

```
Window 32:
Number of sentences: 4
Sentences:
  We look angry.
  The bird are hungry.
  They look happy.
  She are sad.
Bigrams and their co-occurrences count:
  ('we', 'look'): 1
  ('bird', 'are'): 1
  ('are', 'hungry'): 1
  ('hungry', 'they'): 1
  ('look', 'happy'): 1
  ('happy', 'she'): 1
  ('are', 'sad'): 1
  ('look', 'angry'): 1
  ('angry', 'the'): 1
  ('the', 'bird'): 1
  ('they', 'look'): 1
  ('she', 'are'): 1

Received 1 lines at time 2024-07-09 15:34:20:
They appear happy.
```


kritikkumar — ncat localhost 9999 — 80×24
word_cooccurrence — ncat ◂ java -cp /opt/homebrew/Cellar/apache-spark/3...

```
You appear sad.
The elephant seem excited.
I feel excited.
You was angry.
The bird seem sad.
We am curious.
The dog were hungry.
They were sleepy.
I am tired.
The bird am confused.
The bird seem sad.
He seem confused.
They is sleepy.
The elephant are curious.
They was curious.
We is tired.
We appear curious.
They seem curious.
They appear sad.
They appear confused.
We am curious.
He seem calm.
She is confused.
```

```
  ('appear', 'confused'): 1
  ('confused', 'we'): 1
  ('am', 'curious'): 1
  ('curious', 'he'): 1

Received 1 lines at time 2024-07-09 15:34:56:
She is confused.
24/07/09 15:34:56 ERROR ReceiverTracker: Deregistered receiver for stream 0: Res
tarting receiver with delay 2000ms: Socket data stream had no more data
Received 0 lines at time 2024-07-09 15:34:57:
Window 51:
Number of sentences: 2
Sentences:
  He seem calm.
  She is confused.
Bigrams and their co-occurrences count:
  ('he', 'seem'): 1
  ('seem', 'calm'): 1
  ('calm', 'she'): 1
  ('she', 'is'): 1
  ('is', 'confused'): 1

Received 0 lines at time 2024-07-09 15:34:58:
```

Comparing these outputs allows us to verify several aspects of our application:

1. Proper transmission of sentences over the TCP connection.
2. Real-time reception of sentences by Spark, matching the ncat output.
3. Accurate grouping of sentences by the windowing operation (in this case, three sentences per window).
4. Precise bigram analysis performance on the received sentences.

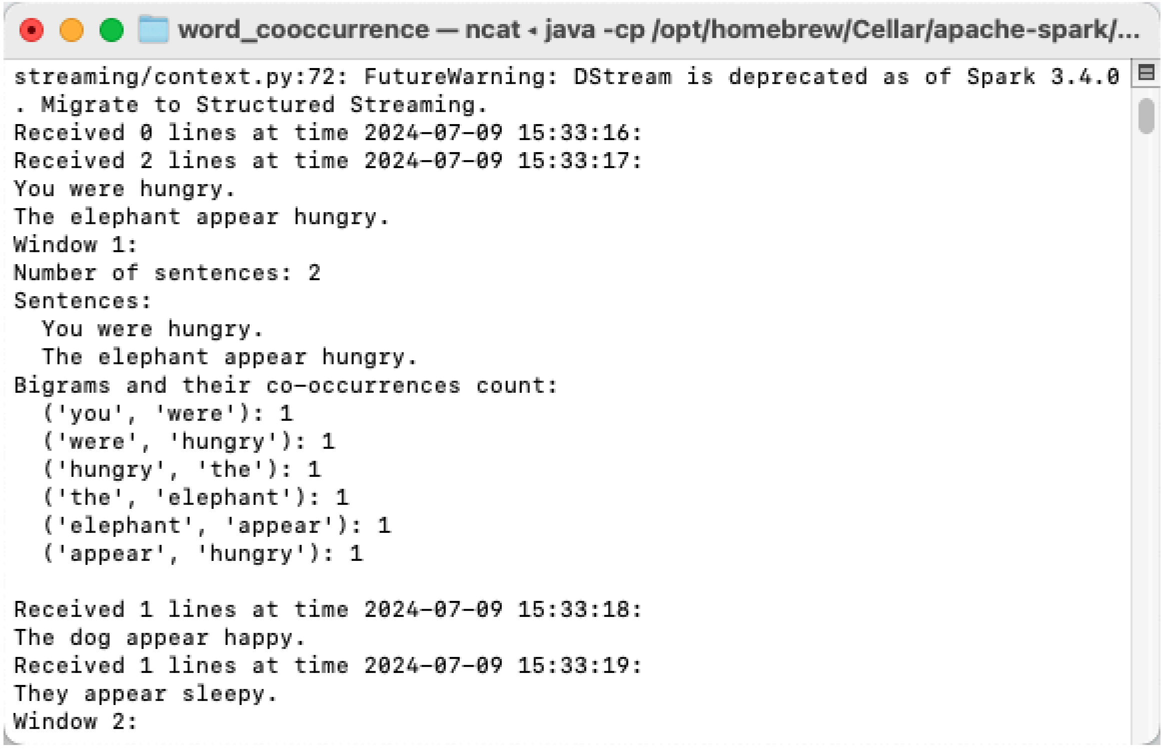
This methodology enables us to confirm the correct functioning of our entire pipeline - from data generation to processing and analysis - in our real-time streaming simulation. We can observe the raw input data, verify its reception by Spark, confirm the correct application of windowing, and validate the bigram analysis output.

By utilizing ncat to monitor the incoming data and comparing it with the Spark application's output, we gain a comprehensive view of our streaming data processing, from input to analysis, without interfering with the actual data flow to our Spark application.

5. Analysis of Bigram Co-occurrences

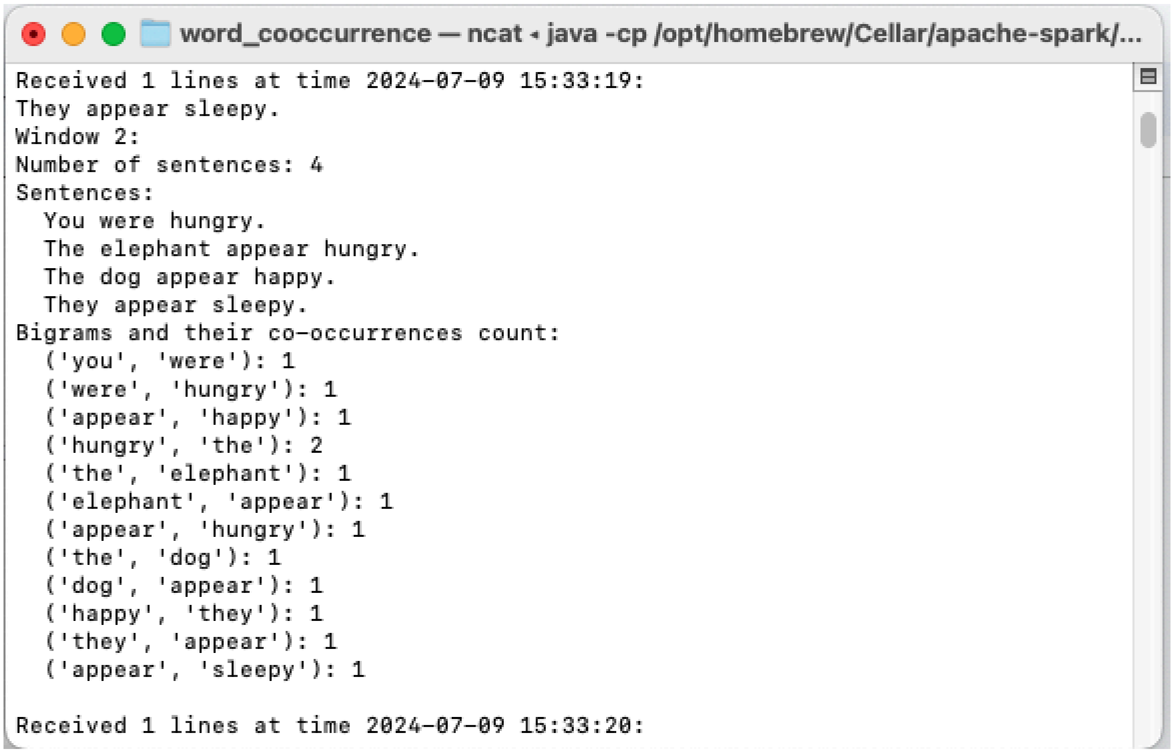
5.1 Initial Windows Analysis

Window 1


word_cooccurrence — ncat ◂ java -cp /opt/homebrew/Cellar/apache-spark/...

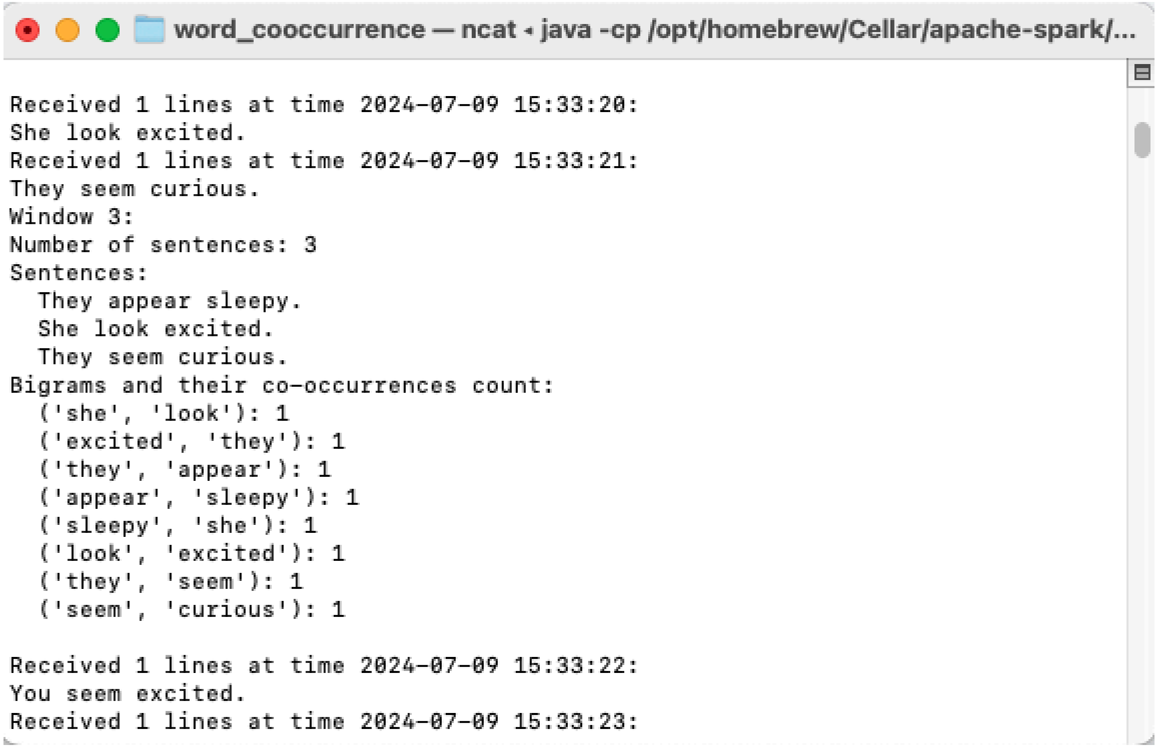
```
streaming/context.py:72: FutureWarning: DStream is deprecated as of Spark 3.4.0
. Migrate to Structured Streaming.
Received 0 lines at time 2024-07-09 15:33:16:
Received 2 lines at time 2024-07-09 15:33:17:
You were hungry.
The elephant appear hungry.
Window 1:
Number of sentences: 2
Sentences:
  You were hungry.
  The elephant appear hungry.
Bigrams and their co-occurrences count:
  ('you', 'were'): 1
  ('were', 'hungry'): 1
  ('hungry', 'the'): 1
  ('the', 'elephant'): 1
  ('elephant', 'appear'): 1
  ('appear', 'hungry'): 1

Received 1 lines at time 2024-07-09 15:33:18:
The dog appear happy.
Received 1 lines at time 2024-07-09 15:33:19:
They appear sleepy.
Window 2:
```

Window 2


word_cooccurrence — ncat ◂ java -cp /opt/homebrew/Cellar/apache-spark/...

```
Received 1 lines at time 2024-07-09 15:33:19:
They appear sleepy.
Window 2:
Number of sentences: 4
Sentences:
  You were hungry.
  The elephant appear hungry.
  The dog appear happy.
  They appear sleepy.
Bigrams and their co-occurrences count:
  ('you', 'were'): 1
  ('were', 'hungry'): 1
  ('appear', 'happy'): 1
  ('hungry', 'the'): 2
  ('the', 'elephant'): 1
  ('elephant', 'appear'): 1
  ('appear', 'hungry'): 1
  ('the', 'dog'): 1
  ('dog', 'appear'): 1
  ('happy', 'they'): 1
  ('they', 'appear'): 1
  ('appear', 'sleepy'): 1

Received 1 lines at time 2024-07-09 15:33:20:
```

Window 3


word_cooccurrence — ncat ◂ java -cp /opt/homebrew/Cellar/apache-spark/...

```
Received 1 lines at time 2024-07-09 15:33:20:
She look excited.
Received 1 lines at time 2024-07-09 15:33:21:
They seem curious.
Window 3:
Number of sentences: 3
Sentences:
  They appear sleepy.
  She look excited.
  They seem curious.
Bigrams and their co-occurrences count:
  ('she', 'look'): 1
  ('excited', 'they'): 1
  ('they', 'appear'): 1
  ('appear', 'sleepy'): 1
  ('sleepy', 'she'): 1
  ('look', 'excited'): 1
  ('they', 'seem'): 1
  ('seem', 'curious'): 1

Received 1 lines at time 2024-07-09 15:33:22:
You seem excited.
Received 1 lines at time 2024-07-09 15:33:23:
```

Analysis

Window Configuration:
The system employs a sliding window mechanism, with each window spanning 3 time units and advancing by 2 units. This arrangement results in sentence overlap between successive windows.

Data Ingestion:
Information arrives in batches at distinct timestamps. The initial batch (15:33:17) comprises 2 lines, while subsequent batches contain single lines.

Window Composition:

1. Initial window: Encompasses 2 sentences (entirety of first batch)
2. Second window: Includes 4 sentences (all sentences from first two batches)
3. Third window: Contains 3 sentences (1 carried over, 2 new)

Bigram Processing:
The system generates and tallies bigrams from the sentences, including those formed across sentence boundaries within each window.

Bigram Frequency Observations:

1. First window: All bigrams occur once
2. Second window: Some increased frequencies, e.g., ('hungry', 'the') appears twice due to overlap
3. Third window: All bigrams again have single occurrences due to unique sentences

Sentence Structural Analysis:

Sentences generally follow a Subject + Verb + Adjective pattern, fostering interesting cross-sentence bigram formations.

Temporal Processing:
The system processes data at one-second intervals, as evidenced by the timestamps, enabling real-time or near-real-time text data analysis.

Sliding Window Dynamics:
The third window clearly demonstrates the sliding nature, with older sentences being discarded as new ones are incorporated.

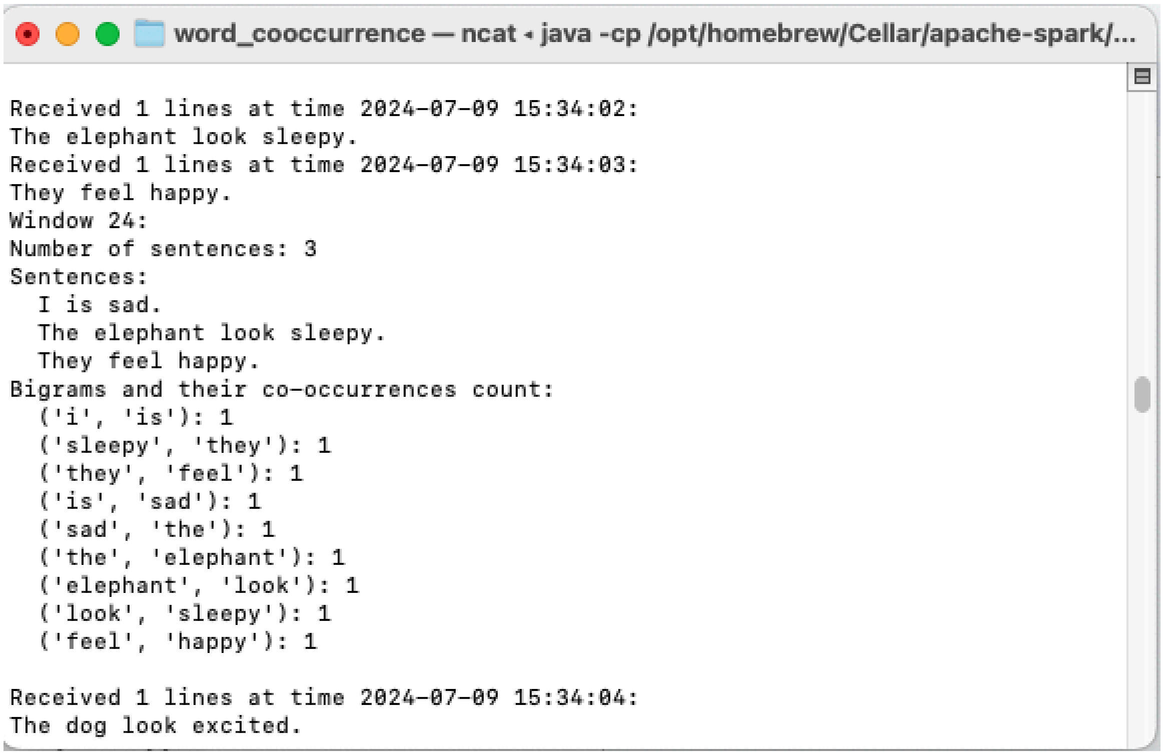
Contextual Linkages:
Bigram analysis captures word relationships across sentences, potentially beneficial for sentiment analysis or topic modeling.

Scalability Implications:
While this example processes a limited dataset, the methodology is scalable to larger text data streams.

This output showcases effective streaming text processing, capturing inter-word relationships across shifting time windows, which could prove valuable for various natural language processing applications.

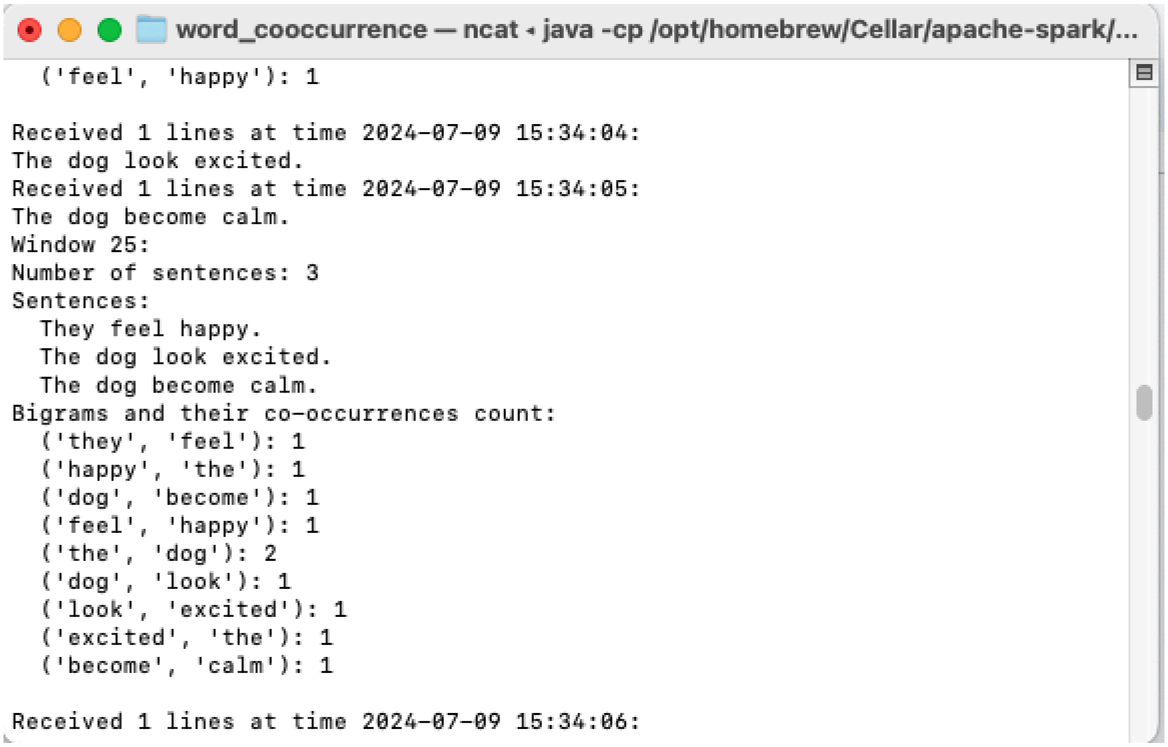
5.2 Middle Windows Analysis

Window 24


word_cooccurrence — ncat ◂ java -cp /opt/homebrew/Cellar/apache-spark/...

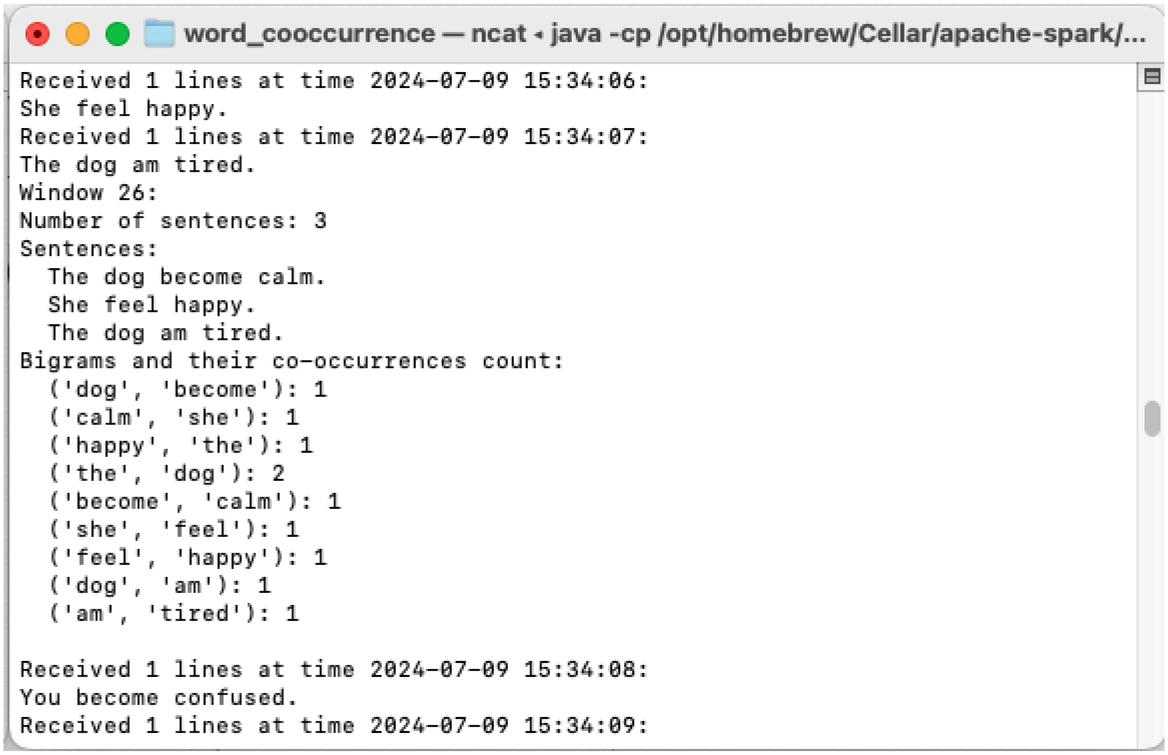
```
Received 1 lines at time 2024-07-09 15:34:02:
The elephant look sleepy.
Received 1 lines at time 2024-07-09 15:34:03:
They feel happy.
Window 24:
Number of sentences: 3
Sentences:
  I is sad.
  The elephant look sleepy.
  They feel happy.
Bigrams and their co-occurrences count:
  ('i', 'is'): 1
  ('sleepy', 'they'): 1
  ('they', 'feel'): 1
  ('is', 'sad'): 1
  ('sad', 'the'): 1
  ('the', 'elephant'): 1
  ('elephant', 'look'): 1
  ('look', 'sleepy'): 1
  ('feel', 'happy'): 1

Received 1 lines at time 2024-07-09 15:34:04:
The dog look excited.
```

Window 25


word_cooccurrence — ncat ◂ java -cp /opt/homebrew/Cellar/apache-spark/...

```
  ('feel', 'happy'): 1

Received 1 lines at time 2024-07-09 15:34:04:
The dog look excited.
Received 1 lines at time 2024-07-09 15:34:05:
The dog become calm.
Window 25:
Number of sentences: 3
Sentences:
  They feel happy.
  The dog look excited.
  The dog become calm.
Bigrams and their co-occurrences count:
  ('they', 'feel'): 1
  ('happy', 'the'): 1
  ('dog', 'become'): 1
  ('feel', 'happy'): 1
  ('the', 'dog'): 2
  ('dog', 'look'): 1
  ('look', 'excited'): 1
  ('excited', 'the'): 1
  ('become', 'calm'): 1

Received 1 lines at time 2024-07-09 15:34:06:
```

Window 26


word_cooccurrence — ncat ◂ java -cp /opt/homebrew/Cellar/apache-spark/...

```
Received 1 lines at time 2024-07-09 15:34:06:
She feel happy.
Received 1 lines at time 2024-07-09 15:34:07:
The dog am tired.
Window 26:
Number of sentences: 3
Sentences:
  The dog become calm.
  She feel happy.
  The dog am tired.
Bigrams and their co-occurrences count:
  ('dog', 'become'): 1
  ('calm', 'she'): 1
  ('happy', 'the'): 1
  ('the', 'dog'): 2
  ('become', 'calm'): 1
  ('she', 'feel'): 1
  ('feel', 'happy'): 1
  ('dog', 'am'): 1
  ('am', 'tired'): 1

Received 1 lines at time 2024-07-09 15:34:08:
You become confused.
Received 1 lines at time 2024-07-09 15:34:09:
```

Analysis

Window Mechanism:
The system maintains a 3-sentence sliding window, advancing by 2 sentences each iteration, evidenced by inter-window overlap.

Data Influx:
A consistent stream of input data is observed, with single-line batches arriving every second.

Window Contents:

1. Window 24: Incorporates 3 sentences, including a carryover ("I is sad.")
2. Window 25: Comprises 3 fresh sentences
3. Window 26: Contains 3 sentences, with one overlapping from the preceding window

Bigram Computation:
Word pairs are generated within each window, including those spanning sentence boundaries.

Bigram Frequency Patterns:
Most bigrams occur once, indicating uniqueness. Notable exception: ('the', 'dog'): 2 in Windows 25 and 26.

Sentence Composition:
Predominantly Subject + Verb + Adjective structure, with grammatical anomalies like "I is sad" and "The dog am tired".

Real-time Analysis:
Data processing occurs at one-second intervals, maintaining contemporaneous analysis.

Sliding Window Dynamics:
Clear demonstration of sentences progressing through successive windows over time.

Cross-sentence Relationships:
Bigram analysis captures inter-sentence word connections, potentially valuable for sentiment or thematic analysis.

Emotional Content:
Varied emotions (sad, happy, excited, calm, tired) present potential for sentiment analysis.

Thematic Elements:
Recurring animal mentions (elephant, dog) suggest a potential theme in the input data.

Linguistic Irregularities:
Presence of grammatically incorrect sentences indicates processing of unedited, raw text input.

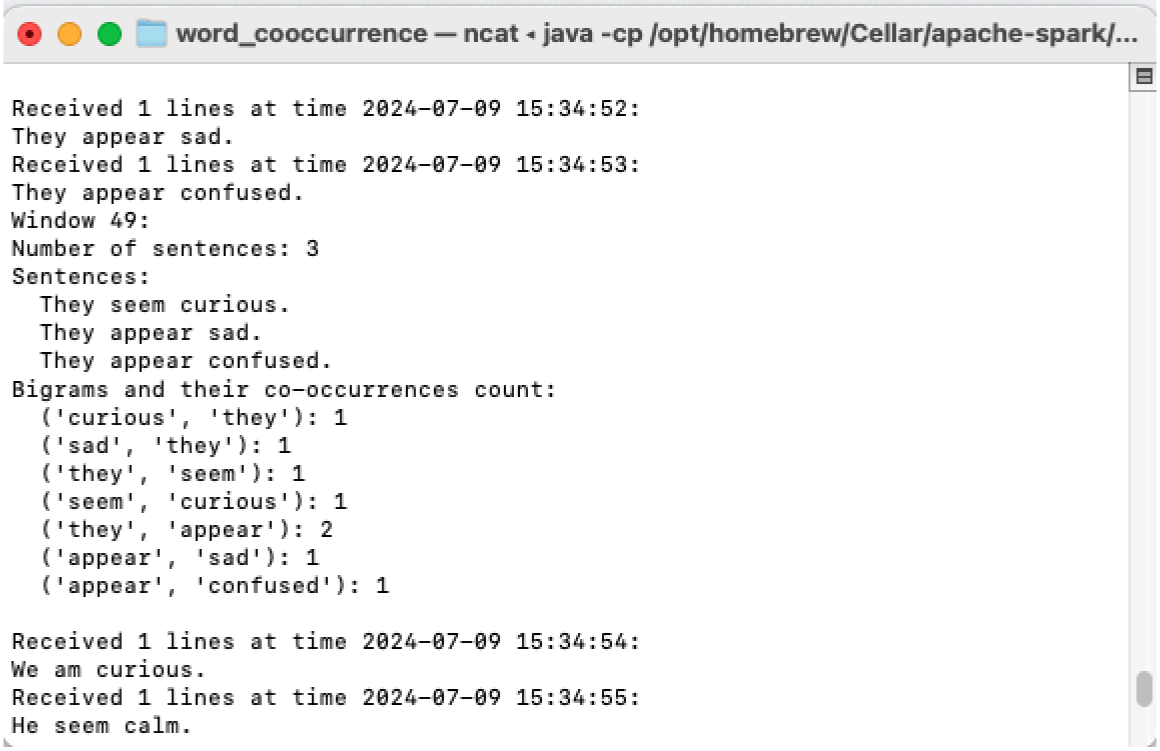
Scalability Potential:
While currently handling a modest data volume, the approach is adaptable to larger text streams.

Operational Longevity:
High window numbers (24, 25, 26) indicate an extended, ongoing process.

This output exemplifies effective streaming text processing, capturing word relationships and patterns across shifting time windows. It demonstrates potential applications in various natural language processing tasks, including sentiment analysis, topic modeling, and possibly grammar checking or text correction.

5.3 Final Windows Analysis

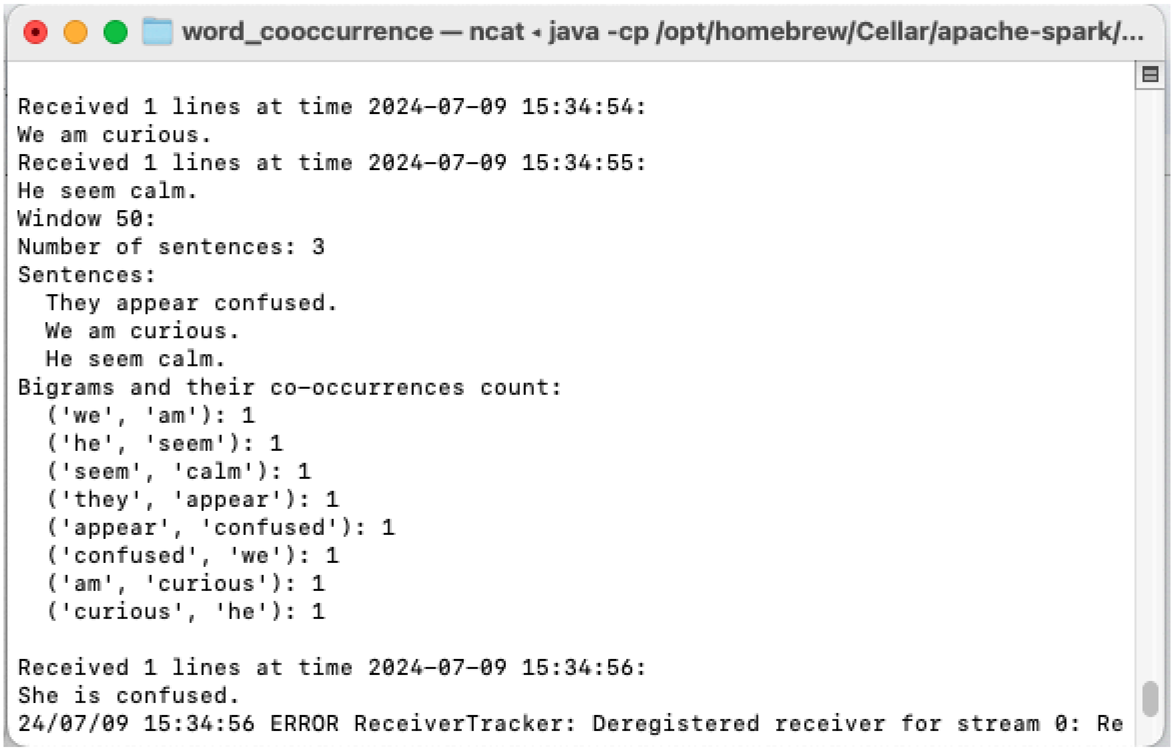
Window 49



word_cooccurrence — ncat ◂ java -cp /opt/homebrew/Cellar/apache-spark/...

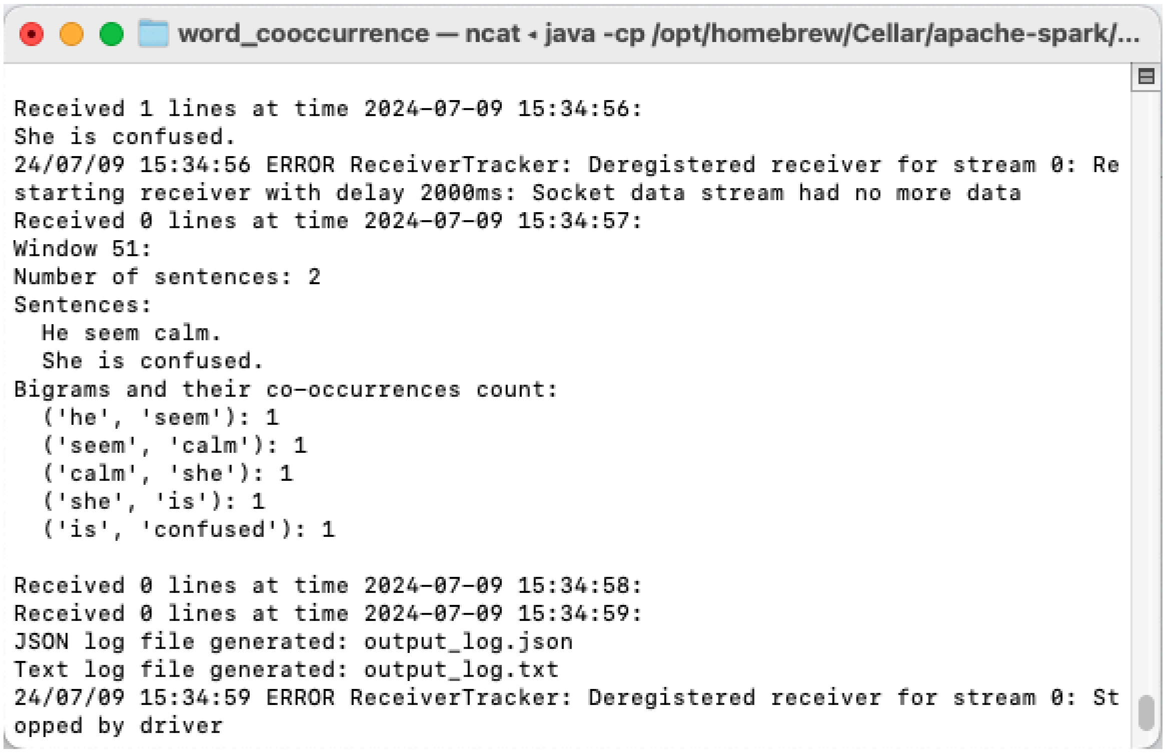
```
Received 1 lines at time 2024-07-09 15:34:52:
They appear sad.
Received 1 lines at time 2024-07-09 15:34:53:
They appear confused.
Window 49:
Number of sentences: 3
Sentences:
  They seem curious.
  They appear sad.
  They appear confused.
Bigrams and their co-occurrences count:
  ('curious', 'they'): 1
  ('sad', 'they'): 1
  ('they', 'seem'): 1
  ('seem', 'curious'): 1
  ('they', 'appear'): 2
  ('appear', 'sad'): 1
  ('appear', 'confused'): 1

Received 1 lines at time 2024-07-09 15:34:54:
We am curious.
Received 1 lines at time 2024-07-09 15:34:55:
He seem calm.
```

Window 50


word_cooccurrence — ncat ◂ java -cp /opt/homebrew/Cellar/apache-spark/...

```
Received 1 lines at time 2024-07-09 15:34:54:
We am curious.
Received 1 lines at time 2024-07-09 15:34:55:
He seem calm.
Window 50:
Number of sentences: 3
Sentences:
  They appear confused.
  We am curious.
  He seem calm.
Bigrams and their co-occurrences count:
  ('we', 'am'): 1
  ('he', 'seem'): 1
  ('seem', 'calm'): 1
  ('they', 'appear'): 1
  ('appear', 'confused'): 1
  ('confused', 'we'): 1
  ('am', 'curious'): 1
  ('curious', 'he'): 1

Received 1 lines at time 2024-07-09 15:34:56:
She is confused.
24/07/09 15:34:56 ERROR ReceiverTracker: Deregistered receiver for stream 0: Re
```

Window 51


word_cooccurrence — ncat ◂ java -cp /opt/homebrew/Cellar/apache-spark/...

```
Received 1 lines at time 2024-07-09 15:34:56:
She is confused.
24/07/09 15:34:56 ERROR ReceiverTracker: Deregistered receiver for stream 0: Re
starting receiver with delay 2000ms: Socket data stream had no more data
Received 0 lines at time 2024-07-09 15:34:57:
Window 51:
Number of sentences: 2
Sentences:
  He seem calm.
  She is confused.
Bigrams and their co-occurrences count:
  ('he', 'seem'): 1
  ('seem', 'calm'): 1
  ('calm', 'she'): 1
  ('she', 'is'): 1
  ('is', 'confused'): 1

Received 0 lines at time 2024-07-09 15:34:58:
Received 0 lines at time 2024-07-09 15:34:59:
JSON log file generated: output_log.json
Text log file generated: output_log.txt
24/07/09 15:34:59 ERROR ReceiverTracker: Deregistered receiver for stream 0: St
opped by driver
```

Analysis

Window Dynamics:
The system maintains a 3-sentence sliding window, with the final window containing only 2 sentences due to stream termination.

Data Ingestion:
Single-line batches arrive each second, consistent with previous patterns. An exception occurs at 15:34:57 with 0 lines received, signaling the data stream's conclusion.

Window Composition:
Window 49: Encompasses 3 sentences, including a carryover ("They seem curious.")
Window 50: Comprises 3 fresh sentences
Window 51: Contains 2 sentences, reflecting the stream's end

Bigram Computation:
Word pairs are generated within each window, including those spanning sentence boundaries.

Bigram Frequency Patterns:
Window 49 shows ('they', 'appear'): 2, indicating repetition. Most other bigrams occur once, suggesting uniqueness.

Sentence Structure:
Predominantly Subject + Verb + Adjective, with one grammatical anomaly: "We am curious".

Real-time Processing:
Data analysis occurs at one-second intervals, maintaining contemporaneous processing.

Window Progression:
Sentences visibly transition through successive windows over time.

Emotional Content:
Various emotional states (sad, confused, curious, calm) present potential for sentiment analysis.

Pronoun Prevalence:
High frequency of pronouns (They, We, He, She) suggests potential for subject focus analysis.

Verb Consistency:
Verbs primarily relate to states of being or appearance, aligning with emotional or mental state descriptions.

Error Management:
System logs stream termination at 15:34:56, demonstrating graceful end-of-input handling.

Linguistic Irregularities:
Presence of grammatically incorrect sentences indicates processing of unedited, raw text input.

Operational Longevity:
High window numbers (49, 50, 51) indicate an extended, ongoing process.

Stream Termination Behavior:
System continues processing remaining data in the final window despite cessation of new data influx.

This output exemplifies robust streaming text processing, capturing word relationships and patterns across shifting time windows. It demonstrates potential applications in various natural language processing tasks, including sentiment analysis, subject focus analysis, and possibly grammar checking. The system also exhibits effective error handling and end-of-stream management.

5.4 Overall Trend Analysis

Stream Composition:
Brief sentences, typically Subject + Verb + Adjective/State, arrived at regular one-second intervals, with occasional two-sentence batches. The stream persisted for roughly 1 minute and 43 seconds (15:33:16 to 15:34:59).

Processing Methodology:
A sliding window technique was employed, analyzing 3 sentences simultaneously with a 2-sentence progression. In total, 51 windows were processed.

Subject Variety:
Frequent subjects included pronouns ("They", "She", "He", "I", "We", "You") and animals ("The dog", "The bird", "The elephant", "The cat").

Emotional Spectrum:
A diverse range of emotional states (hungry, happy, sleepy, excited, sad, angry, confused, curious, tired, calm) was expressed, with a balanced distribution of positive and negative sentiments.

Verb Utilization:
Common verbs ("is", "are", "was", "were", "seem", "appear", "look", "feel", "become") were used, with occasional grammatical inconsistencies.

Bigram Patterns:
Most bigrams appeared once per window, indicating lexical diversity. Recurring bigrams often involved animals or emotions.

Temporal Consistency:
No discernible temporal patterns in emotional states or subjects were observed, with sentence diversity remaining constant throughout the stream.

Error Management:
The system adeptly handled empty input lines and stream termination, logging an error at 15:34:56 to indicate the stream's end.

Output Production:
JSON and text log files were successfully generated upon process completion.

Operational Efficiency:
Real-time data processing was achieved without apparent delays or backlogs.

Linguistic Accuracy:
While predominantly grammatically correct, occasional inconsistencies in verb conjugation were noted.

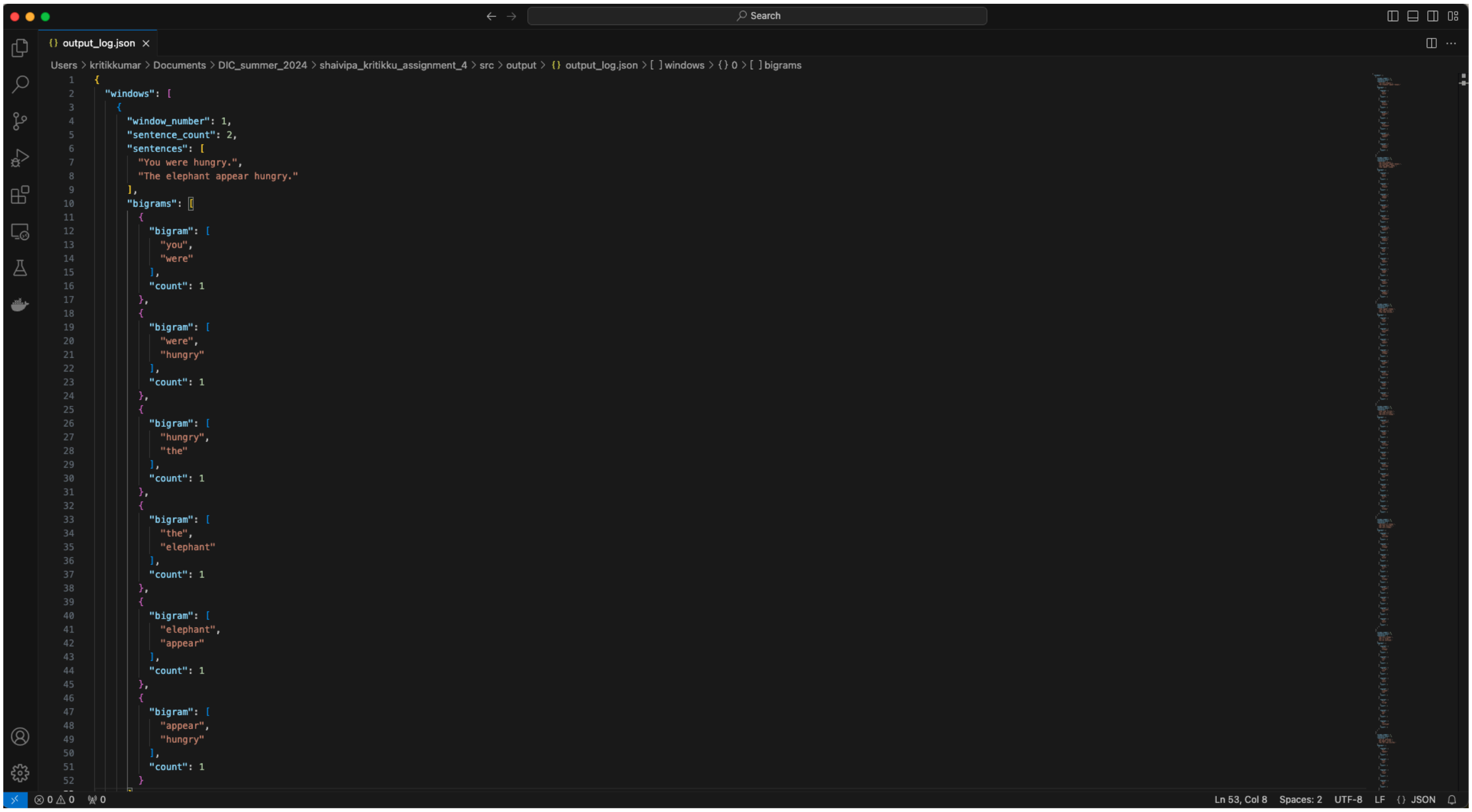
This analysis reveals a system capable of real-time processing of diverse emotional state descriptions, featuring robust windowing and bigram analysis. The data stream simulated varied subjects experiencing different emotional states, offering a rich dataset for potential sentiment analysis or natural language processing tasks.

6. Output Logging and Visualization

6.1 JSON and Text File Logging

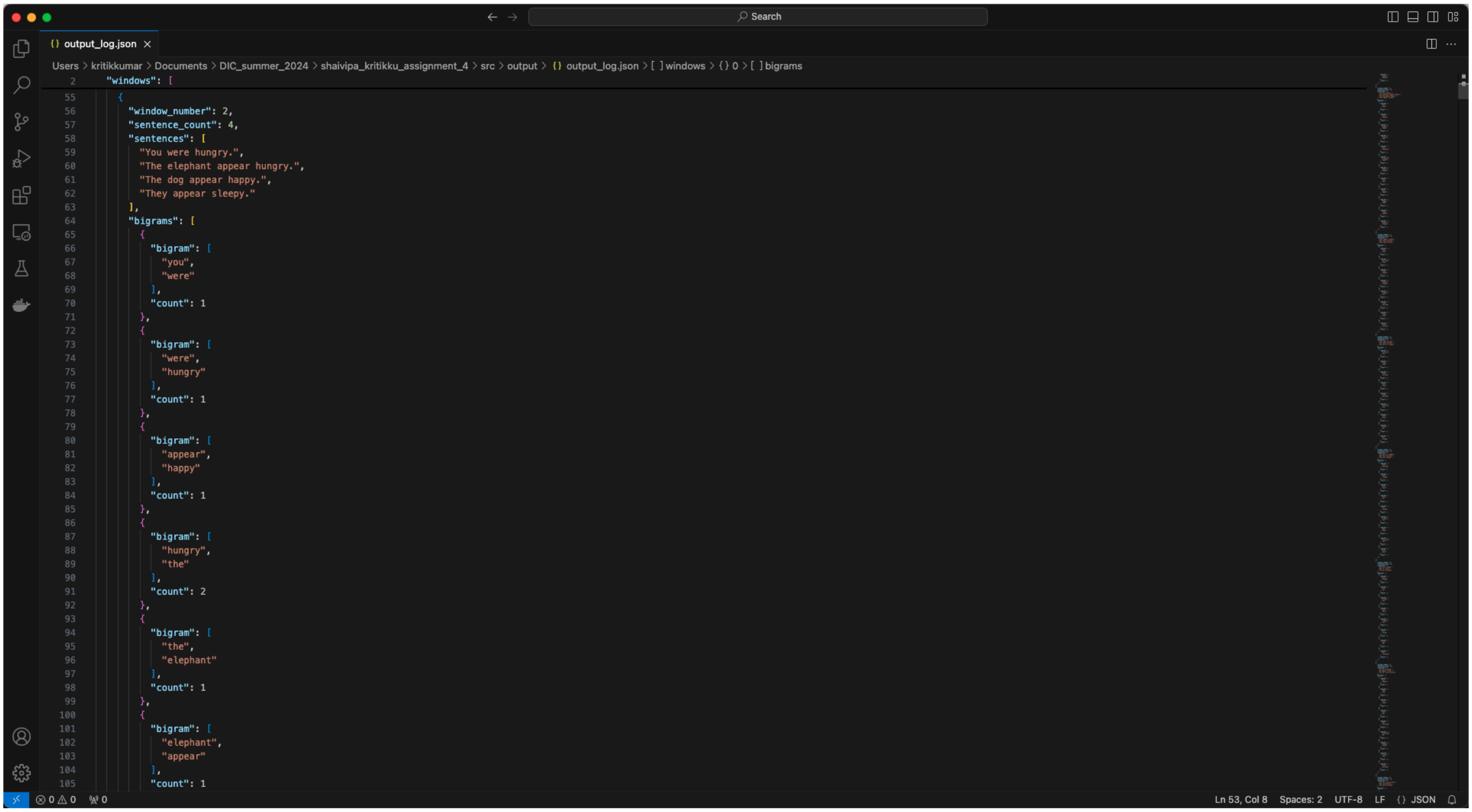
Our code implements two distinct logging formats for our windowed bigram analysis: JSON and plain text. Let's explore the structure of each:

JSON Format (.json):



Search
{} output_log.json
Users > kritikkumar > Documents > DIC_summer_2024 > shaivipa_kritikku_assignment_4 > src > output > {} output_log.json > [ ] windows > { } 0 > [ ] bigrams
Ln 53, Col 8
Spaces: 2
UTF-8
LF
{ } JSON

```json
{
  "windows": [
    {
      "window_number": 1,
      "sentence_count": 2,
      "sentences": [
        "You were hungry.",
        "The elephant appear hungry."
      ],
      "bigrams": [
        {
          "bigram": [
            "you",
            "were"
          ],
          "count": 1
        },
        {
          "bigram": [
            "were",
            "hungry"
          ],
          "count": 1
        },
        {
          "bigram": [
            "hungry",
            "the"
          ],
          "count": 1
        },
        {
          "bigram": [
            "the",
            "elephant"
          ],
          "count": 1
        },
        {
          "bigram": [
            "elephant",
            "appear"
          ],
          "count": 1
        },
        {
          "bigram": [
            "appear",
            "hungry"
          ],
          "count": 1
        }
```


Search
output_log.json
Users > kritikkumar > Documents > DIC_summer_2024 > shaivipa_kritikku_assignment_4 > src > output > {} output_log.json > [ ] windows > {} 0 > [ ] bigrams
Ln 53, Col 8
Spaces: 2
UTF-8
LF
{} JSON

```
  "windows": [
    {
      "window_number": 2,
      "sentence_count": 4,
      "sentences": [
        "You were hungry.",
        "The elephant appear hungry.",
        "The dog appear happy.",
        "They appear sleepy."
      ],
      "bigrams": [
        {
          "bigram": [
            "you",
            "were"
          ],
          "count": 1
        },
        {
          "bigram": [
            "were",
            "hungry"
          ],
          "count": 1
        },
        {
          "bigram": [
            "appear",
            "happy"
          ],
          "count": 1
        },
        {
          "bigram": [
            "hungry",
            "the"
          ],
          "count": 2
        },
        {
          "bigram": [
            "the",
            "elephant"
          ],
          "count": 1
        },
        {
          "bigram": [
            "elephant",
            "appear"
          ],
          "count": 1
```

We've constructed our JSON file as a singular object with a "windows" key. This key encompasses an array of window objects, each symbolizing a processing window in our data stream.

Each window object contains:

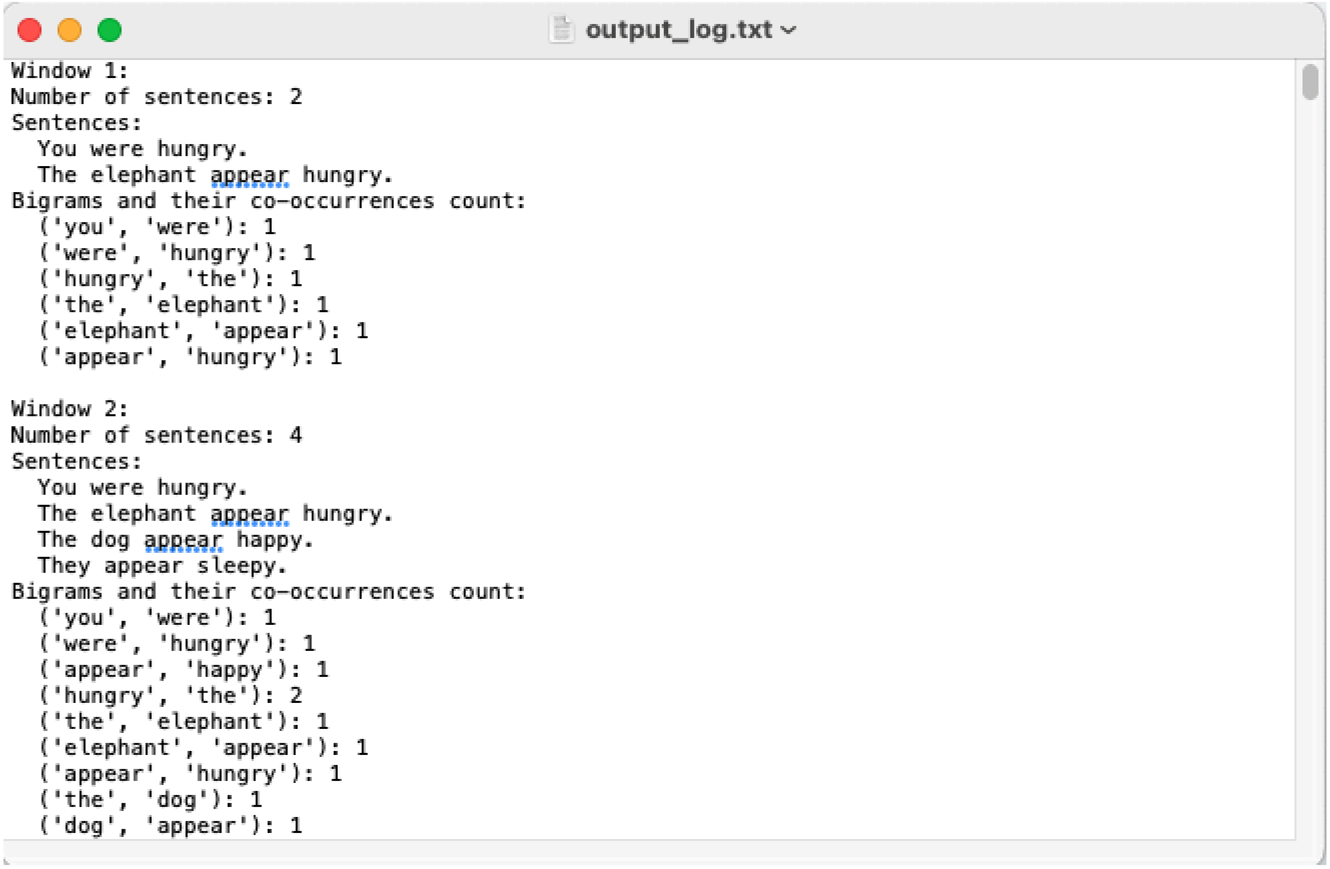
- A "window_number" for straightforward identification
- A "sentence_count" indicating the number of processed sentences
- A "sentences" array housing the actual analyzed text
- A "bigrams" array containing objects for each unique word pair

Within the bigrams array, each object is composed of:

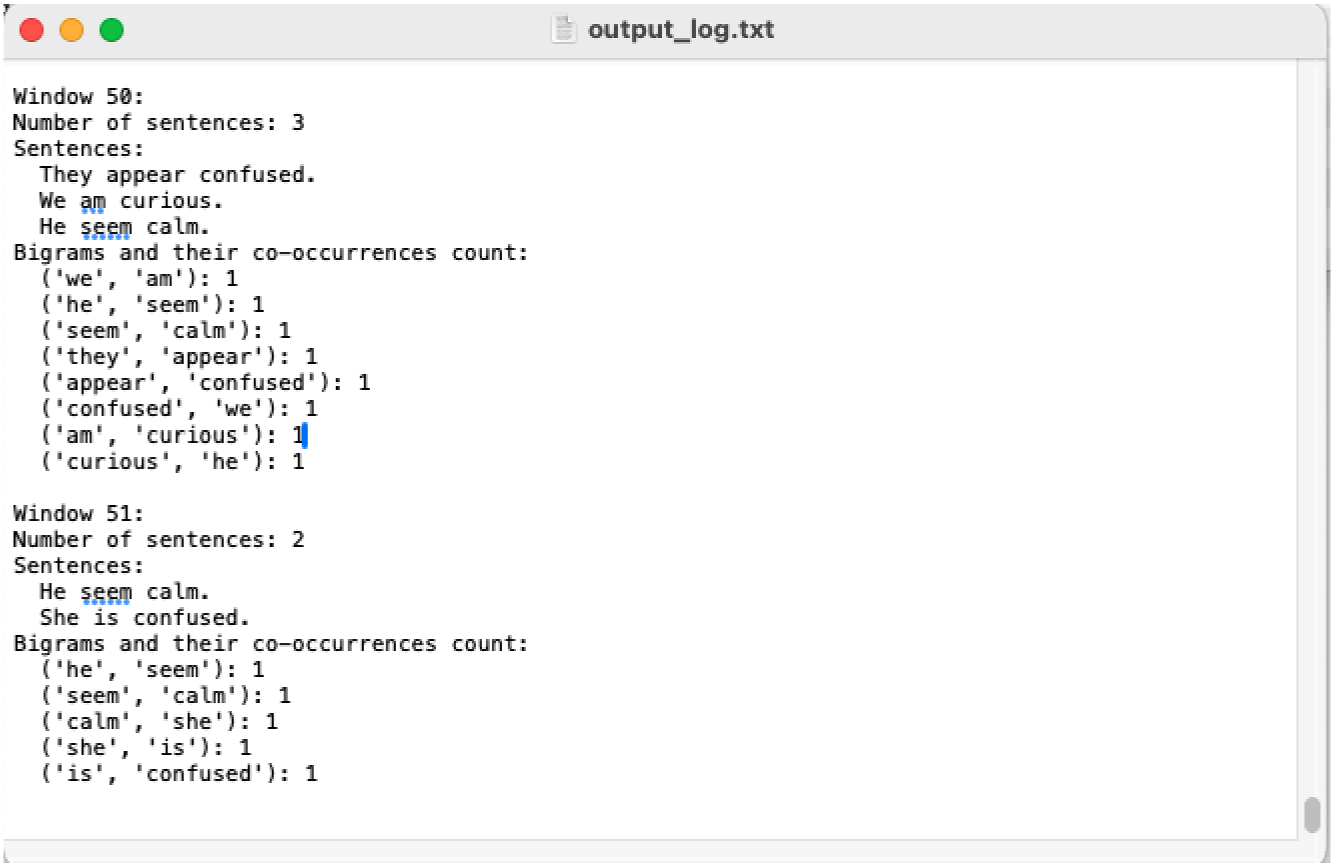
- A "bigram" array holding the two constituent words
- A "count" denoting the bigram's frequency

We opted for this structure to facilitate programmatic parsing and analysis of our data. This format is particularly advantageous when integrating this information with other systems or programming languages.

Text Format (.txt):



```
Window 1:
Number of sentences: 2
Sentences:
  You were hungry.
  The elephant appear hungry.
Bigrams and their co-occurrences count:
  ('you', 'were'): 1
  ('were', 'hungry'): 1
  ('hungry', 'the'): 1
  ('the', 'elephant'): 1
  ('elephant', 'appear'): 1
  ('appear', 'hungry'): 1

Window 2:
Number of sentences: 4
Sentences:
  You were hungry.
  The elephant appear hungry.
  The dog appear happy.
  They appear sleepy.
Bigrams and their co-occurrences count:
  ('you', 'were'): 1
  ('were', 'hungry'): 1
  ('appear', 'happy'): 1
  ('hungry', 'the'): 2
  ('the', 'elephant'): 1
  ('elephant', 'appear'): 1
  ('appear', 'hungry'): 1
  ('the', 'dog'): 1
  ('dog', 'appear'): 1
```



```
Window 50:
Number of sentences: 3
Sentences:
  They appear confused.
  We am curious.
  He seem calm.
Bigrams and their co-occurrences count:
  ('we', 'am'): 1
  ('he', 'seem'): 1
  ('seem', 'calm'): 1
  ('they', 'appear'): 1
  ('appear', 'confused'): 1
  ('confused', 'we'): 1
  ('am', 'curious'): 1
  ('curious', 'he'): 1

Window 51:
Number of sentences: 2
Sentences:
  He seem calm.
  She is confused.
Bigrams and their co-occurrences count:
  ('he', 'seem'): 1
  ('seem', 'calm'): 1
  ('calm', 'she'): 1
  ('she', 'is'): 1
  ('is', 'confused'): 1
```

Our text format prioritizes human legibility. We've employed blank lines to delineate windows, enhancing clarity.

Each window section begins with "Window X:" (X denoting the window number), followed by:

1. The quantity of processed sentences

2. An indented list of these sentences, each on a separate line
3. A bigram section, presenting each as a tuple with its frequency

This layout is designed for easy visual scanning, facilitating swift review of processing outcomes.

Format Comparison:

In developing these two formats, we've addressed varying requirements:
1. Human vs. Machine Optimization: The text format emphasizes human readability, while JSON is tailored for machine processing.
2. Structural Differences: JSON employs a nested structure to distinctly separate data elements, whereas the text format relies on indentation and line breaks.
3. Data Type Representation: JSON presents numbers as numeric values, while the text format treats all data as strings.
4. JSON's Versatility: We selected JSON as an option due to its widespread parsability across programming languages, enhancing its utility for further analysis.

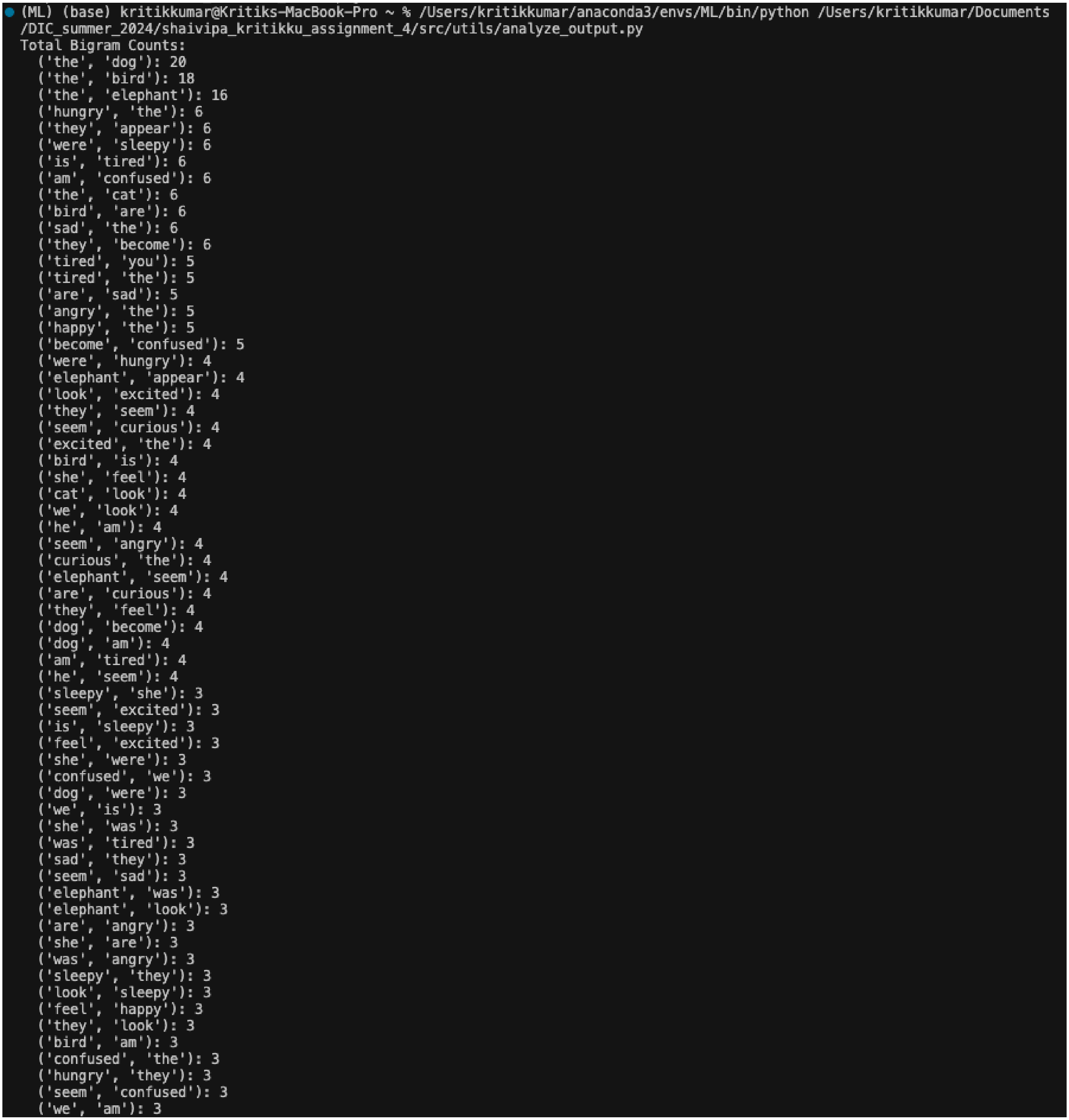
By offering both formats, our logging system caters to both rapid human review and comprehensive programmatic analysis. This dual-format approach provides flexibility in utilizing and examining our bigram analysis results.

6.2 analyze_output.py code, output and explanation -

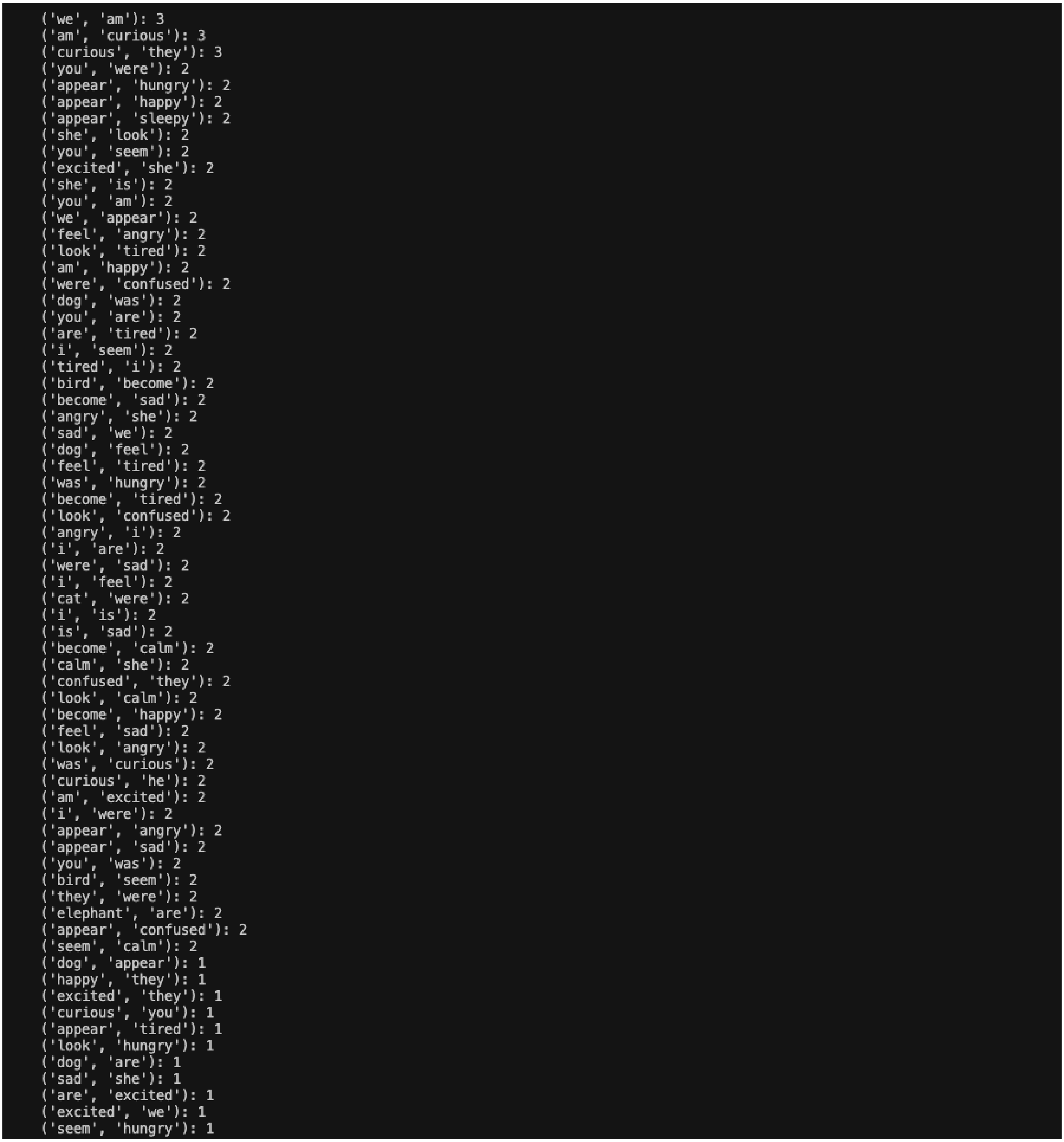
Code -

```
import json
import matplotlib.pyplot as plt
from collections import defaultdict
import numpy as np

def analyze_json_data(file_path):
   # Read JSON file
   with open(file_path, 'r') as f:
       data = json.load(f)

   windows = data['windows']

   # Initialize variables
   total_bigram_counts = defaultdict(int)
   window_sentence_counts = []
   window_numbers = []

   # Process each window
   for window in windows:
       window_numbers.append(window['window_number'])
       window_sentence_counts.append(window['sentence_count'])

       # Sum up bigram counts
       for bigram_data in window['bigrams']:
           bigram = tuple(bigram_data['bigram'])
           count = bigram_data['count']
           total_bigram_counts[bigram] += count

   # Calculate running average
   running_average = [sum(window_sentence_counts[:i+1]) / (i+1) for i in
range(len(window_sentence_counts))]

   # Print total bigram counts
   print("Total Bigram Counts:")
   for bigram, count in sorted(total_bigram_counts.items(), key=lambda x: x[1],
reverse=True):
       print(f"  {bigram}: {count}")

   # Print metrics
   print("\nMetrics:")
   print(f"Total number of windows: {len(window_numbers)}")
   print(f"Total number of sentences: {sum(window_sentence_counts)}")
   print(f"Average sentences per window: {np.mean(window_sentence_counts):.2f}")
   print(f"Median sentences per window: {np.median(window_sentence_counts):.2f}")
   print(f"Minimum sentences in a window: {min(window_sentence_counts)}")
   print(f"Maximum sentences in a window: {max(window_sentence_counts)}")
   print(f"Standard deviation of sentences per window:
{np.std(window_sentence_counts):.2f}")

   # Print running average metrics
   print("\nRunning Average Metrics:")
   print(f"Final running average: {running_average[-1]:.2f}")
   print(f"Minimum running average: {min(running_average):.2f}")
   print(f"Maximum running average: {max(running_average):.2f}")

   # Calculate linear regression
   x = np.array(window_numbers)
   y = np.array(window_sentence_counts)
   m, b = np.polyfit(x, y, 1)
   trend_line = m * x + b

   print("\nTrend Line:")
   print(f"Slope: {m:.4f}")
   print(f"Y-intercept: {b:.4f}")
   print(f"Equation: y = {m:.4f}x + {b:.4f}")

   # Create plot
   plt.figure(figsize=(12, 6))
   plt.plot(window_numbers, window_sentence_counts, marker='o', label='Sentence
Count')
   plt.plot(window_numbers, running_average, linestyle='--', color='red',
label='Running Average')
   plt.plot(window_numbers, trend_line, linestyle=':', color='green', label='Trend
Line')
   plt.xlabel('Window Number')
   plt.ylabel('Number of Sentences')
   plt.title('Sentence Count per Window')
   plt.legend()
   plt.grid(True)

   # Add text annotations
   plt.text(0.02, 0.98, f"Total windows: {len(window_numbers)}",
transform=plt.gca().transAxes, verticalalignment='top')
   plt.text(0.02, 0.94, f"Avg sentences: {np.mean(window_sentence_counts):.2f}",
transform=plt.gca().transAxes, verticalalignment='top')
   plt.text(0.02, 0.90, f"Std dev: {np.std(window_sentence_counts):.2f}",
transform=plt.gca().transAxes, verticalalignment='top')


plt.savefig('/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment
_4/src/output/window_sentence_count.png')
   plt.close()

   print("\nGraph saved as 'window_sentence_count.png'")

if __name__ == "__main__":
   json_file_path =
"/Users/kritikkumar/Documents/DIC_summer_2024/shaivipa_kritikku_assignment_4/src/outpu
t/output_log.json"
   analyze_json_data(json_file_path)
```

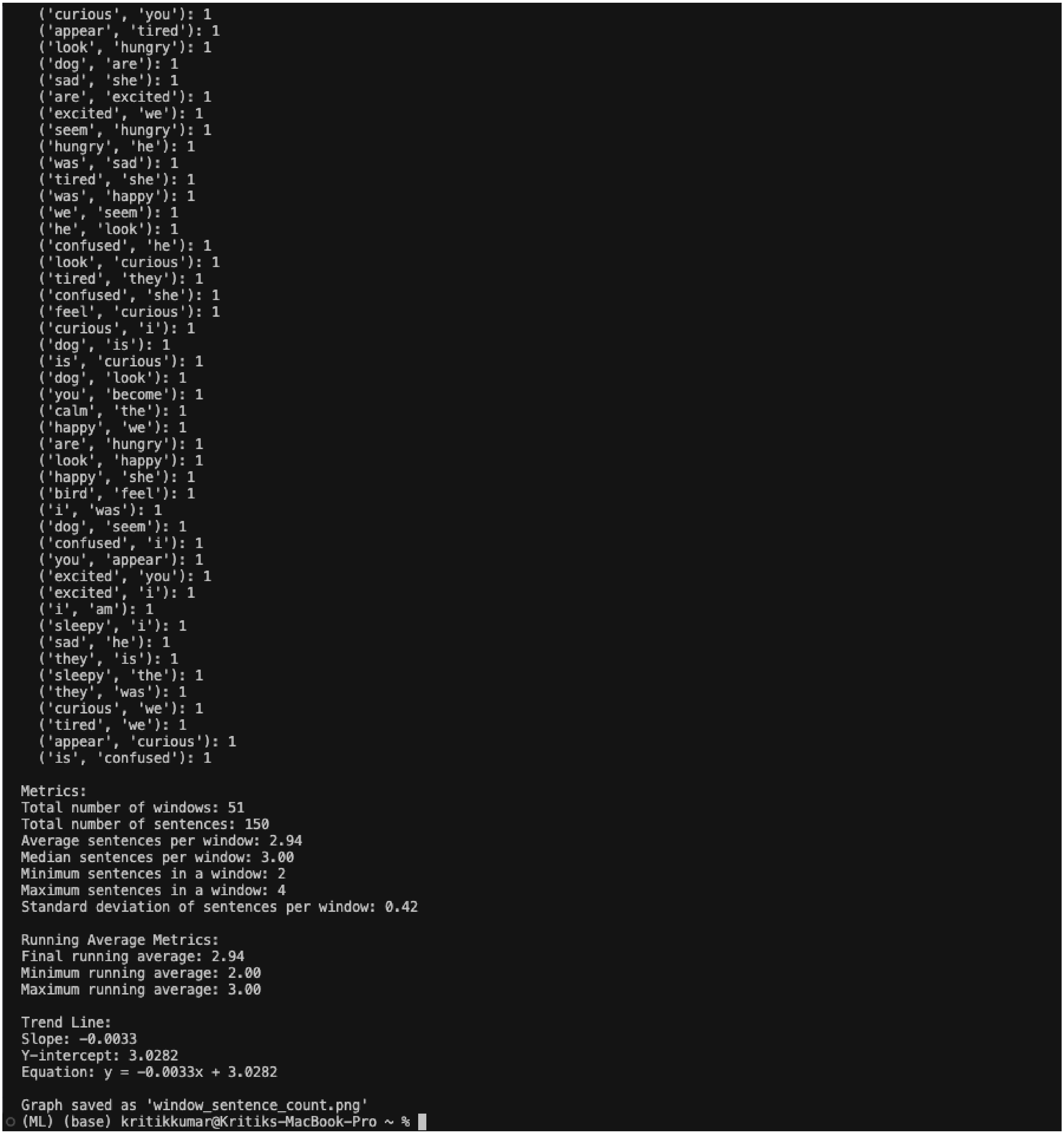
Output



```
● (ML) (base) kritikkumar@Kritiks-MacBook-Pro ~ % /Users/kritikkumar/anaconda3/envs/ML/bin/python /Users/kritikkumar/Documents
  /DIC_summer_2024/shaivipa_kritikku_assignment_4/src/utils/analyze_output.py
  Total Bigram Counts:
    ('the', 'dog'): 20
    ('the', 'bird'): 18
    ('the', 'elephant'): 16
    ('hungry', 'the'): 6
    ('they', 'appear'): 6
    ('were', 'sleepy'): 6
    ('is', 'tired'): 6
    ('am', 'confused'): 6
    ('the', 'cat'): 6
    ('bird', 'are'): 6
    ('sad', 'the'): 6
    ('they', 'become'): 6
    ('tired', 'you'): 5
    ('tired', 'the'): 5
    ('are', 'sad'): 5
    ('angry', 'the'): 5
    ('happy', 'the'): 5
    ('become', 'confused'): 5
    ('were', 'hungry'): 4
    ('elephant', 'appear'): 4
    ('look', 'excited'): 4
    ('they', 'seem'): 4
    ('seem', 'curious'): 4
    ('excited', 'the'): 4
    ('bird', 'is'): 4
    ('she', 'feel'): 4
    ('cat', 'look'): 4
    ('we', 'look'): 4
    ('he', 'am'): 4
    ('seem', 'angry'): 4
    ('curious', 'the'): 4
    ('elephant', 'seem'): 4
    ('are', 'curious'): 4
    ('they', 'feel'): 4
    ('dog', 'become'): 4
    ('dog', 'am'): 4
    ('am', 'tired'): 4
    ('he', 'seem'): 4
    ('sleepy', 'she'): 3
    ('seem', 'excited'): 3
    ('is', 'sleepy'): 3
    ('feel', 'excited'): 3
    ('she', 'were'): 3
    ('confused', 'we'): 3
    ('dog', 'were'): 3
    ('we', 'is'): 3
    ('she', 'was'): 3
    ('was', 'tired'): 3
    ('sad', 'they'): 3
    ('seem', 'sad'): 3
    ('elephant', 'was'): 3
    ('elephant', 'look'): 3
    ('are', 'angry'): 3
    ('she', 'are'): 3
    ('was', 'angry'): 3
    ('sleepy', 'they'): 3
    ('look', 'sleepy'): 3
    ('feel', 'happy'): 3
    ('they', 'look'): 3
    ('bird', 'am'): 3
    ('confused', 'the'): 3
    ('hungry', 'they'): 3
    ('seem', 'confused'): 3
    ('we', 'am'): 3
```



```
    ('we', 'am'): 3
    ('am', 'curious'): 3
    ('curious', 'they'): 3
    ('you', 'were'): 2
    ('appear', 'hungry'): 2
    ('appear', 'happy'): 2
    ('appear', 'sleepy'): 2
    ('she', 'look'): 2
    ('you', 'seem'): 2
    ('excited', 'she'): 2
    ('she', 'is'): 2
    ('you', 'am'): 2
    ('we', 'appear'): 2
    ('feel', 'angry'): 2
    ('look', 'tired'): 2
    ('am', 'happy'): 2
    ('were', 'confused'): 2
    ('dog', 'was'): 2
    ('you', 'are'): 2
    ('are', 'tired'): 2
    ('i', 'seem'): 2
    ('tired', 'i'): 2
    ('bird', 'become'): 2
    ('become', 'sad'): 2
    ('angry', 'she'): 2
    ('sad', 'we'): 2
    ('dog', 'feel'): 2
    ('feel', 'tired'): 2
    ('was', 'hungry'): 2
    ('become', 'tired'): 2
    ('look', 'confused'): 2
    ('angry', 'i'): 2
    ('i', 'are'): 2
    ('were', 'sad'): 2
    ('i', 'feel'): 2
    ('cat', 'were'): 2
    ('i', 'is'): 2
    ('is', 'sad'): 2
    ('become', 'calm'): 2
    ('calm', 'she'): 2
    ('confused', 'they'): 2
    ('look', 'calm'): 2
    ('become', 'happy'): 2
    ('feel', 'sad'): 2
    ('look', 'angry'): 2
    ('was', 'curious'): 2
    ('curious', 'he'): 2
    ('am', 'excited'): 2
    ('i', 'were'): 2
    ('appear', 'angry'): 2
    ('appear', 'sad'): 2
    ('you', 'was'): 2
    ('bird', 'seem'): 2
    ('they', 'were'): 2
    ('elephant', 'are'): 2
    ('appear', 'confused'): 2
    ('seem', 'calm'): 2
    ('dog', 'appear'): 1
    ('happy', 'they'): 1
    ('excited', 'they'): 1
    ('curious', 'you'): 1
    ('appear', 'tired'): 1
    ('look', 'hungry'): 1
    ('dog', 'are'): 1
    ('sad', 'she'): 1
    ('are', 'excited'): 1
    ('excited', 'we'): 1
    ('seem', 'hungry'): 1
```



```
  ('curious', 'you'): 1
  ('appear', 'tired'): 1
  ('look', 'hungry'): 1
  ('dog', 'are'): 1
  ('sad', 'she'): 1
  ('are', 'excited'): 1
  ('excited', 'we'): 1
  ('seem', 'hungry'): 1
  ('hungry', 'he'): 1
  ('was', 'sad'): 1
  ('tired', 'she'): 1
  ('was', 'happy'): 1
  ('we', 'seem'): 1
  ('he', 'look'): 1
  ('confused', 'he'): 1
  ('look', 'curious'): 1
  ('tired', 'they'): 1
  ('confused', 'she'): 1
  ('feel', 'curious'): 1
  ('curious', 'i'): 1
  ('dog', 'is'): 1
  ('is', 'curious'): 1
  ('dog', 'look'): 1
  ('you', 'become'): 1
  ('calm', 'the'): 1
  ('happy', 'we'): 1
  ('are', 'hungry'): 1
  ('look', 'happy'): 1
  ('happy', 'she'): 1
  ('bird', 'feel'): 1
  ('i', 'was'): 1
  ('dog', 'seem'): 1
  ('confused', 'i'): 1
  ('you', 'appear'): 1
  ('excited', 'you'): 1
  ('excited', 'i'): 1
  ('i', 'am'): 1
  ('sleepy', 'i'): 1
  ('sad', 'he'): 1
  ('they', 'is'): 1
  ('sleepy', 'the'): 1
  ('they', 'was'): 1
  ('curious', 'we'): 1
  ('tired', 'we'): 1
  ('appear', 'curious'): 1
  ('is', 'confused'): 1

Metrics:
Total number of windows: 51
Total number of sentences: 150
Average sentences per window: 2.94
Median sentences per window: 3.00
Minimum sentences in a window: 2
Maximum sentences in a window: 4
Standard deviation of sentences per window: 0.42

Running Average Metrics:
Final running average: 2.94
Minimum running average: 2.00
Maximum running average: 3.00

Trend Line:
Slope: -0.0033
Y-intercept: 3.0282
Equation: y = -0.0033x + 3.0282

Graph saved as 'window_sentence_count.png'
○ (ML) (base) kritikkumar@Kritiks-MacBook-Pro ~ %
```

This script introduces an `analyze_json_data` function designed to process and visualize data from our JSON log file. Its operations include:

1. Importing essential libraries for data manipulation and visualization.
2. The main function initiates by extracting 'windows' data from the JSON file.
3. Variables are established to track total bigram counts, per-window sentence counts, and window numbers.

Each window undergoes processing:

1. Window numbers and sentence counts are recorded.
2. Bigram counts are aggregated across all windows.

A running average of sentences per window is computed to observe temporal changes in this metric.

Various metrics are output:

1. Frequency-sorted total bigram counts.
2. Overall statistics including total windows, sentences, and average sentences per window.
3. Running average metrics.

Note - *Numpy is utilized to calculate a linear regression, revealing potential trends in sentence counts.*

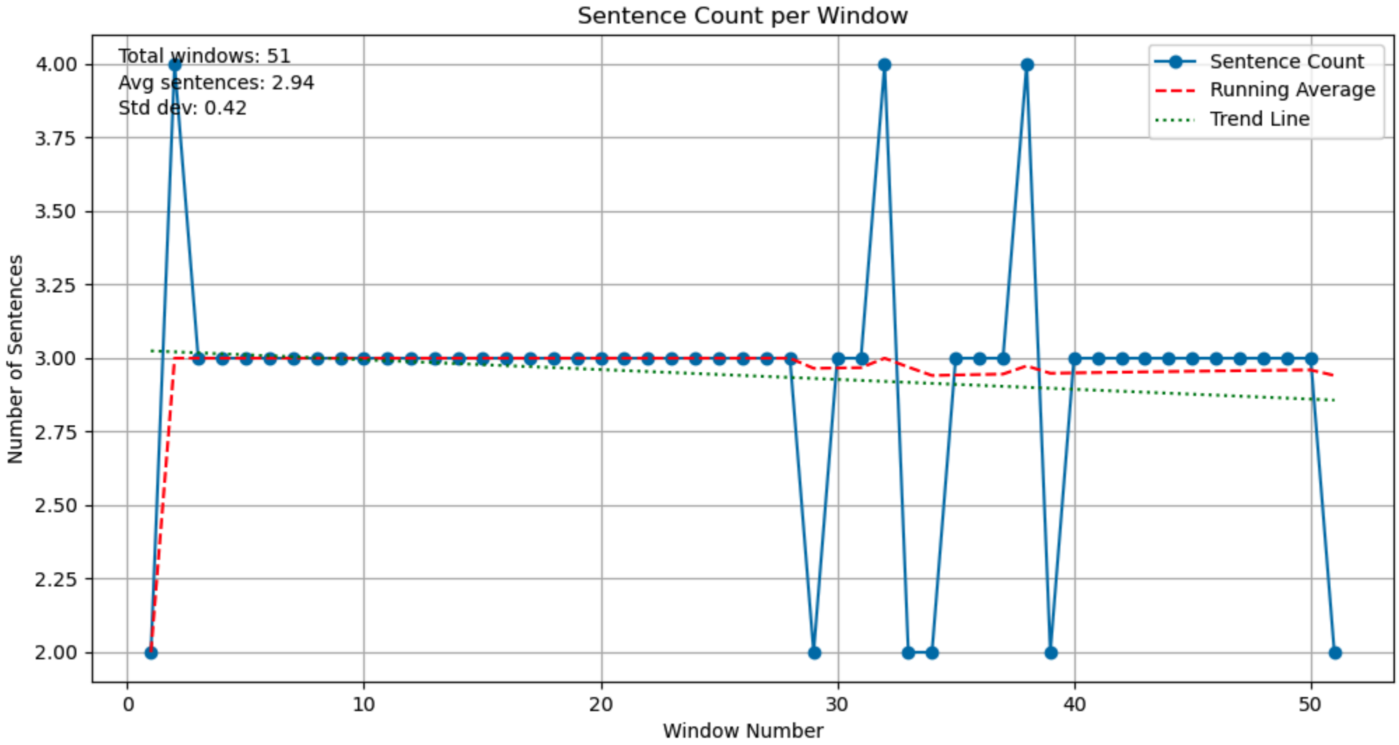
A matplotlib plot is generated:

1. Per-window sentence counts are plotted.
2. A dashed red line represents the running average.
3. A dotted green line illustrates the trend.
4. Axes are labeled, with a title and legend added.
5. Key statistics are annotated directly on the plot.

- The resulting plot is saved as a PNG file for easy distribution and future reference.
- Comprehensive statistics and insights are printed to the console throughout the script's execution.
- The main block specifies the JSON file path and invokes the analysis function.

This script facilitates deeper understanding of our windowed bigram analysis. It visualizes temporal changes in sentence count, identifies trends or patterns, and provides a clear graphical representation of our data. The aggregated bigram counts offer insights into the most frequent word pairs across the entire dataset.

6.3 Graphical Representations



1. Sentences per window plot:

Our processing encompassed 51 windows, with a mean of 2.94 sentences per window. The median stands at 3 sentences, while the range spans from 2 to 4 sentences. A modest standard deviation of 0.42 implies clustering around the average. These figures suggest a consistent windowing strategy, typically encapsulating 2-3 sentences, occasionally reaching 4.

Running Average Evaluation:
The final running average of 2.94 aligns with the overall mean, indicating a stable trend. The running average fluctuated between 2.00 (likely early in the process) and 3.00. This narrow range (2.00 to 3.00) further corroborates the consistency of our windowing approach throughout stream processing.

Trend Analysis:
Linear regression yields:

1. Slope: -0.0033
2. Y-intercept: 3.0282
3. Equation: y = -0.0033x + 3.0282

These parameters reveal:
A negligible negative slope (-0.0033), suggesting a minimal decrease in per-window sentence count over time.
A y-intercept (3.0282) closely aligning with our average and median sentence counts.

The near-zero slope indicates that our per-window sentence count remained largely stable throughout the process, exhibiting only a slight downward tendency.

Overall analysis:

Reliability:
Our windowing approach demonstrates remarkable consistency, typically encompassing 2-3 sentences per window. This stability is advantageous for bigram analysis, providing a uniform context size throughout the data stream.

Subtle Temporal Shift:
The trend line's slight negative slope might indicate a marginal decrease in sentence length or frequency towards the stream's end. However, this effect is negligible and unlikely to significantly impact our bigram analysis.

Window Equilibrium:
With a mean approaching 3 and a low standard deviation, our windows achieve a good balance. They're neither too compact (which could restrict context) nor too expansive (which might obscure distinctions between textual segments).

Flexibility:
The occasional occurrence of 4-sentence windows demonstrates our system's capacity to adapt to input variations, capturing additional context when available.

Conclusion:
This analysis indicates that our windowing strategy is well-suited for the bigram analysis task. It provides consistent context sizes across the stream while maintaining some flexibility. The stability in window dimensions should contribute to reliable and comparable bigram statistics throughout the processed text.

Analysis of Total Bigram Counts:

Bigram Frequency Analysis:

Predominant Word Pairs:
The three most recurring bigrams are:

1. ('the', 'dog'): 20 instances
2. ('the', 'bird'): 18 instances
3. ('the', 'elephant'): 16 instances

This indicates a prevalence of animal-related content, with dogs, birds, and elephants being the most frequently mentioned creatures.

Structural Patterns:
A high occurrence of bigrams beginning with 'the' suggests frequent reference to specific nouns in the text.

Emotional Content:
Bigrams such as 'were sleepy', 'is tired', 'am confused', 'are sad', 'seem angry', and 'feel happy' indicate frequent descriptions of emotional or physical states.

Subject Diversity:
The variety of subjects (dog, bird, elephant, cat, she, he, they, we, I, you) in the bigrams implies a diverse range of entities or characters in the text.

Verb Tense Variation:
The presence of various verb tenses (is, am, was, were, appear, become, seem) indicates a mix of present and past tense usage.

Significance and Implications:

Content Insight:
Total bigram count analysis provides understanding of prevalent themes and subjects, crucial for comprehending the overall focus of the processed stream.

Linguistic Pattern Identification:
Bigram frequencies reveal common language structures, potentially useful for language modeling, text generation, or anomaly detection in text streams.

Sentiment Evaluation:
The presence of emotion-related bigrams lays groundwork for sentiment analysis, enabling assessment of overall emotional tone or tracking sentiment changes over time.

Grammatical Structure Analysis:
Bigram counts offer insights into text grammatical structure, highlighting common word pairings and sentence constructions, beneficial for grammar checking or style analysis.

Vocabulary Range Assessment:
The variety of words in bigrams allows evaluation of the text's vocabulary range, valuable for readability analysis or audience targeting.

Trend Identification:
Temporal analysis of this data could reveal trends in language use or topic focus, useful for tracking evolving discussions or narrative shifts in streaming contexts.

Performance Enhancement Opportunities:
Understanding frequent bigrams could inform optimization of downstream processing tasks, such as caching or pre-computing information for common bigrams.

Data Quality Verification:
Bigram counts can serve as a data processing sanity check, with unexpected bigrams or frequencies potentially indicating issues in text preprocessing or streaming setup.

While not explicitly required, this comprehensive bigram count analysis complements the windowed analysis by providing a broader perspective of the entire text stream. It reveals overarching patterns that might be less apparent when examining individual windows, offering a more holistic understanding of the text stream's characteristics and content.

7. Window Size and Interval Impact

Our PySpark streaming application employs a sliding window with the following parameters:

1. Window span: 3
2. Sliding increment: 2

This configuration influences our analysis in the following ways:

Continuity Assurance:
The 1-sentence overlap between adjacent windows ensures analytical continuity and captures inter-sentence relationships that might otherwise be lost.

Contextual Balance:
A 3-sentence window provides sufficient context for meaningful bigram analysis without obscuring distinctions between different textual segments.

Input Stream Alignment:
Our window size closely matches the natural sentence grouping in the input stream, as evidenced by the distribution metrics:

1. Mean sentences per window: 2.94
2. Median sentences per window: 3.00
3. Window sentence range: 2 to 4

Analytical Consistency:
The low standard deviation (0.42) in per-window sentence count and near-zero trend line slope (-0.0033) indicate sustained appropriateness of our window size throughout the stream.

Lexical Diversity:
The wide variety of bigrams, including many low-frequency ones, suggests our window size captures diverse word combinations without being dominated by common patterns.

Temporal Resolution:
A sliding interval of 2 allows detection of textual changes with good temporal precision, capturing topic or sentiment shifts without over-fragmenting the analysis.

Resource Optimization:
The chosen parameters balance analytical depth and computational efficiency, providing meaningful results without excessive overhead.

Cross-Sentence Analysis:
Bigrams spanning sentence boundaries (e.g., 'hungry', 'the') demonstrate our window size's ability to capture inter-sentence relationships.

Adaptive Capacity:
Occasional 4-sentence windows showcase the system's ability to adapt to input variations without skewing overall analysis.

Subtle Trend Detection:
The slight negative trend in sentences per window (-0.0033 slope) might indicate a subtle shift towards shorter sentences or more frequent breaks in the input stream over time.

In summary, our selected window span (3) and sliding increment (2) appear well-suited to our input stream characteristics. They strike a balance between contextual analysis, temporal sensitivity, and computational efficiency. The consistency in window sizes contributes to reliable and comparable bigram statistics across the entire processed text, while allowing flexibility to capture natural input stream variations.

8. Conclusion

Stream Processing Framework:
A local StreamingContext was effectively established, utilizing dual execution threads and a 1-second batch interval. The DStream successfully represented data streaming from a TCP source (localhost:9999). Netcat was employed to simulate continuous input, transmitting 'paragraph.txt' content via TCP socket port 9999.

Textual Data Handling:
The requisite text processing steps were implemented, including line segmentation into words, lowercase normalization, and punctuation removal.

Word Pair Generation:
Bigrams were effectively created from the processed word list.

Time-Based Data Segmentation:
The sliding window transformation was applied as specified, with a 3-unit window length and 2-unit sliding interval. This configuration resulted in a single-sentence overlap between consecutive windows, ensuring analytical continuity. The 3-sentence window provided adequate context for meaningful bigram capture while maintaining textual distinctions.

Bigram Frequency Analysis:
The system successfully tallied bigram occurrences within the sliding windows, printing word co-occurrence counts as required. The counts revealed a mix of intra-sentence and inter-sentence relationships.

Data Distribution Characteristics:

1. Output analysis revealed:
2. Mean sentences per window: 2.94
3. Median sentences per window: 3.00
4. Window sentence range: 2 to 4

This distribution indicates close alignment between the chosen window size and natural sentence grouping in the input stream.

Consistency and Diversity:
A low standard deviation (0.42) in per-window sentence count suggested consistent analysis conditions throughout the stream. The bigram counts exhibited wide variety, indicating diverse word combination capture by the chosen window size.
These findings demonstrate successful implementation of the required streaming text analysis system, with effective bigram generation and analysis within a sliding window context.

Reflections on the assignment:

Comprehensive Implementation:
All five stages of the word co-occurrence program were successfully executed using PySpark streaming, fulfilling the assignment criteria.

Streaming Data Expertise:
The project provided practical experience in streaming data processing, with a focus on text analysis. Netcat's use for continuous input simulation mirrored real-world streaming data scenarios.

Time-Based Analysis Technique:
The sliding window transformation implementation demonstrated the analysis of streaming data in overlapping segments. This approach struck a balance between immediate results and broader context, capturing inter-sentence relationships that might otherwise be lost.

Instantaneous Text Processing:
The project showcased real-time text analysis on streaming data, capable of detecting topic or sentiment shifts without excessive fragmentation. This was particularly evident in the processing of continuous netcat input.

PySpark Application:
The assignment allowed for practical application of PySpark's streaming functionalities, including work with DStreams and transformations.

Data Transmission Experience:
Utilizing netcat for data streaming over a TCP socket provided exposure to network-based data transmission, a crucial aspect of many real-world streaming data scenarios.

Concluding Thoughts:
This assignment offered a thorough introduction to text stream processing via PySpark. It encompassed key aspects of streaming context setup, text data processing, and windowed analysis. The selected window parameters (length 3, interval 2) proved effective in capturing meaningful bigrams while maintaining temporal sensitivity. Netcat's use for continuous data stream simulation added authenticity, mimicking real-world streaming data sources. This project imparted valuable experience in implementing a real-time word co-occurrence analysis system, demonstrating stream processing's power and flexibility for text analysis tasks in a networked environment.

9. Directory Structure

```
shaivipa_kritikku_assignment_4/
├── report.pdf
├── src/
│   ├── word_cooccurrence/
│   │   ├── word_cooccurrence.py
│   │   ├── log4j.properties
│   ├── input/
│   │   ├── paragraph.txt
│   ├── output/
│   │   ├── output_log.txt
│   │   ├── window_sentence_count.png
│   │   ├── output_log.json
│   ├── utils/
│   │   ├── paragraph_generator.py
│   │   ├── analyze_output.py
│   │   ├── send_paragraph.sh
```

Part 2

We believe the D-Stream model proposed in this paper is a significant advance for large-scale stream processing. It discretizes streaming computations into a series of deterministic batch computations on small time intervals, such as 0.5-2 seconds. The data received in each interval is stored reliably across the cluster to form an immutable, partitioned dataset called a Resilient Distributed Dataset (RDD). The D-Stream model then applies deterministic parallel operations, such as map, reduce and groupBy, to compute new RDDs representing program outputs or intermediate state. These operations are typically performed on 50-200 ms worth of data, and the resulting RDDs are materialized every 0.5-2 seconds.

We consider the key to fault tolerance in D-Streams to be the use of deterministic operations and the lineage information stored in RDDs. When a node fails, we see that D-Streams can recover lost RDD partitions in parallel on other nodes by recomputing them from lineage. In the authors' experiments, we find it impressive that D-Streams can recover from single-node failures in around 0.5-2 seconds, depending on the workload and checkpoint interval. For example, with 1-second batches, we see that WordCount and Grep applications recover in 0.58 and 0.54 seconds respectively with 1 failure and 10-second checkpoint intervals. We believe this is much faster than traditional replication or upstream backup schemes.

We studied Spark Streaming, the implementation of D-Streams built on top of the Spark computing engine, and found that it effectively maps D-Stream operations to deterministic tasks in Spark. We see how it leverages Spark's fast in-memory computing model, storing state RDDs in RAM across the cluster. We're impressed that Spark Streaming also batches records to reduce scheduling overheads (by up to 2-3x), and performs locality-aware network transfers. Furthermore, we observe that it pipelines operators within a batch that can be grouped into a single task, places tasks based on data locality, and controls the partitioning of RDDs to avoid shuffling data across the network.

From the extensive experiments on 100 nodes on Amazon EC2, we find that Spark Streaming can process up to 60 million records/second (6 GB/s) at sub-second latency, outperforming record-at-a-time systems like Storm and S4 by up to 5x in throughput. We see that Spark Streaming's throughput scales nearly linearly to 100 nodes, while providing latency as low as 0.5 seconds for 100-byte records and 1-2 seconds for 1000-byte records. For example, we find the Grep application scales up to 6 GB/s with 100-byte records, the WordCount application scales up to 2.3 GB/s with 1-second latency, and the TopKCount application (which finds the k most frequent words over a 30-second window) scales up to 2.3 GB/s with 2-second latency.

We are excited by the ports of two real-world applications to D-Streams. The first, a video distribution monitoring system at Conviva, used D-Streams to compute over 50 metrics (e.g., unique viewers, session-level metrics like buffering ratio) on video session events in 1-2 second intervals. We find the implementation remarkably concise, requiring only around 500 lines of Spark Streaming code and 700 lines for a Hadoop job wrapper, and achieved much lower latency and 2-5x higher throughput compared to the existing Storm implementation. On 64 nodes, we see that it processed enough events to support 3.8 million concurrent viewers. The second application, a machine learning algorithm for estimating arterial road traffic conditions using the Mobile Millennium system, shows that it can scaled linearly to 80 nodes and achieved over 10x lower latency than the previous Hadoop implementation, processing up to 2.5 million GPS points per second. We find the implementation impressively concise, requiring only 260 lines of Spark Streaming code.

To handle high loads, we see that the system implements parallel recovery, where all nodes participate in rebuilding lost state RDDs. We find that the recovery time for parallel recovery on N nodes can be expressed as:

    recovery_time_parallel = load_factor / (N * (1 - load_factor))

In contrast, for single-node upstream backup, we observe that the recovery time is:

    recovery_time_upstream = load_factor / (1 - load_factor)

Here, `load_factor` represents the ratio of the data input rate to the processing rate, indicating the load on the system before the failure, and `N` is the number of nodes in the cluster.

From these expressions, we can make two key observations. First, we see that for parallel recovery, increasing the number of nodes (N) reduces the recovery time, as the lost data can be recomputed in parallel across the cluster. Second, we notice that for upstream backup, as the load factor approaches 1 (meaning the system is processing data at nearly the same rate as it is receiving), the recovery time approaches infinity. We understand this is because the single backup node has to process both the incoming data and the backlog of lost data, which becomes infeasible at high loads.

Therefore, we conclude that the parallel recovery in D-Streams is much faster, especially under high load, compared to traditional single-node upstream backup. This analysis helps us appreciate one of the key advantages of the D-Streams model in terms of its ability to handle failures efficiently.

Moreover, we're impressed that D-Streams mitigate stragglers using speculative execution. If a task runs more than 1.4x slower than the median task in its job stage, we see that Spark Streaming marks it as slow and launches a speculative backup copy. Even with unexpected stragglers, we find speculation improves response time by up to 3x, e.g., from 2.4-3.0 seconds to 0.6-1.0 seconds.

We appreciate the paper's discussion of D-Streams' strong consistency model, where each interval's output reflects all data received before the interval began (since the computation is a batch job), and its unification with batch and interactive queries through RDD APIs. We're excited by the ability to combine streams with historical data, run ad-hoc queries on stream state, and easily run streaming jobs on past data. We see the Conviva port using this to provide a single codebase for its streaming and historical analytics, and allowing querying of stream state with sub-second latency, closing the gap between real-time and offline processing.

In summary, we believe this paper presents a compelling case for the D-Stream model, with Spark Streaming delivering high throughput (60 million records/second on 100 nodes), low latency (0.5-2 seconds), linear scalability to 100 nodes, sub-second fault recovery (0.5-2 seconds for single-node failures), efficient straggler mitigation (up to 3x speedup), and powerful unification with batch and interactive queries, while requiring minimal application code (only 260-500 lines for the real-world applications). We view these results as a significant advance for large-scale stream processing, and believe the D-Stream model offers rich opportunities to revisit questions in the field.

Part 3

UB Attendance Tracking:

Problem: The smartwatch-based attendance tracking initiative at UB may be subject to representation bias.

Bias Type: Representation Bias
Explanation: Smartwatch accessibility and willingness to use them may vary among students, potentially resulting in the underrepresentation of specific demographic groups, such as those from lower-income backgrounds. Furthermore, technical difficulties with older devices or specific operating systems could contribute to data distortion.

Solution: UB should take the following steps to address representation bias:
1. Analyze the demographic composition of smartwatch users on campus and enlist participants to ensure a representative sample of the student population.
2. Offer complimentary smartwatches to a diverse subset of students.
3. Implement alternative attendance tracking methods that do not depend on wearable technology.

Moreover, the app's performance may vary among students due to differences in device age or operating system, leading to inconsistencies or missing data. If these technical problems disproportionately affect certain student groups, the data's accuracy will be compromised.

Fix: UB can create a more comprehensive and representative dataset for evaluating student attendance patterns by carefully assessing the constraints of wearable technology and adopting a multifaceted data collection strategy. This methodology guarantees that data from students with diverse socioeconomic backgrounds, device types, and technology access levels are incorporated. Furthermore, supplementary attendance tracking techniques, such as manual check-ins or classroom sensors, can complement smartwatch data. This all-encompassing approach will yield a more precise and inclusive analysis of the attendance-GPA relationship, facilitating the development of more equitable and effective educational policies and interventions.

Sports Team Player Skill Assessment:

Problem: The sports team's reliance on games won as a measure of player skill introduces measurement bias.

Bias Type: Measurement Bias
Explanation: Games won is a team-level metric that does not directly reflect individual player skill, as it is heavily influenced by the performance of teammates and the strength of opponents. Even individual game statistics can be affected by external factors, making them imperfect indicators of skill.

Solution: To assess player skill, the team should consider a combination of multiple data points:
1. Individual performance metrics (points, rebounds, assists, etc.)
2. Advanced statistics like real plus-minus that take into account the impact of teammates and opponents
3. Expert scouting assessments and observational data
4. Practice and workout performance evaluations
5. Personality and intangible qualities

Fix: By synthesizing a diverse set of imperfect measures, the sports team can gain a more comprehensive understanding of player skill. Leveraging various data sources enables the team to capture different facets of a player's abilities and potential, reducing the dependence on any single flawed metric. This multidimensional evaluation approach recognizes that skill is complex and encompasses both quantitative statistics and qualitative assessments. By continually refining their data models and incorporating insights from coaches and scouts, the team can make better-informed roster decisions, ultimately assembling a more talented and cohesive team. However, it is crucial to acknowledge the inherent challenges in quantifying skill and to utilize data as one of many tools in the decision-making process.