Homework – 3

K-mer Counting Program using Spark Streaming

Introduction:

Spark Streaming is a powerful extension of Apache Spark that enables real-time data processing. It allows users to process and analyze streaming data in a scalable and fault-tolerant manner, making it suitable for applications that require immediate insights from live data feeds. In this report, we explore the implementation of a k-mer counting program using Spark Streaming, where data is streamed from a TCP socket using Netcat (ncat). Netcat acts as a simple utility that facilitates the simulation of a live data stream by sending text data to a specified port.

Additionally, to simulate continuous data streaming, I implemented a custom TCP server that sends data from a text file (sentences.txt) to the Spark Streaming application. The server operates on localhost:9999, sending each line of the file with a 10-second delay to simulate real-time data. The Spark program processes the incoming sentences, extracts 3-mers (substrings of length 3) from the text, and counts their occurrences in real-time. This setup is discussed in detail along with the steps for implementing the program, handling data streams, and updating k-mer counts across multiple batches.

Input Data Generation

The provided sentence_generator.py script was executed to generate the sentences.txt file. For continuous streaming, sentences.txt was used as the input stream, sent via Netcat on TCP port 9999. This setup allowed the Spark Streaming application to process real-time data.

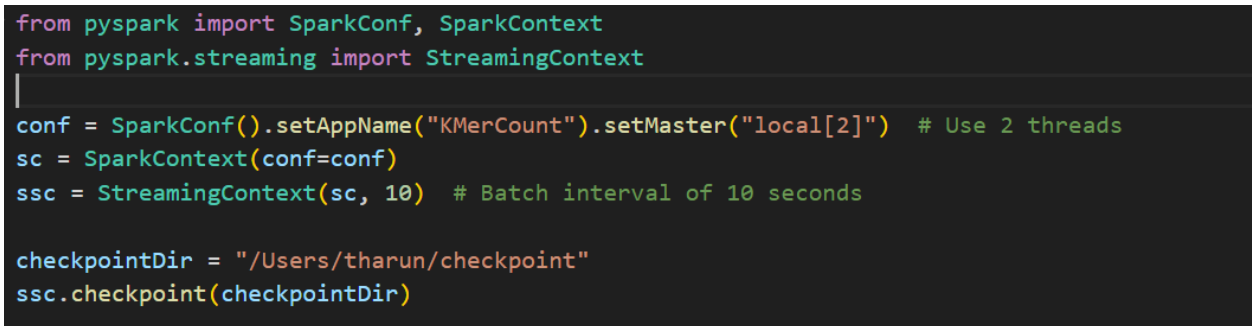
Approach to the Problem

To tackle the problem of counting 3-mers in a real-time data stream, I utilized Spark Streaming's micro-batch processing capabilities. My approach began with setting up a Spark Streaming environment to listen to data from a TCP socket. I then implemented a method to extract 3-mers from each incoming line of text, map them to key-value pairs for counting, and use Spark's stateful operations to maintain and update the k-mer counts across micro-batches. The solution also involved sorting and displaying the top 10 most frequent k-mers in real time, providing insights into the text stream as it was processed.

Detailed Implementation of K-Mer Count Program

1. **Initializing SparkConf, SparkContext and Creating the StreamingContext**

To begin the implementation, we first configure and initialize Spark's core components. The SparkConf object is used to configure the Spark application, setting the application name to "KMerCount" for easy identification in Spark's UI. The SparkContext object establishes a connection to the Spark cluster and serves as the main entry point for Spark functionality. It is responsible for creating a StreamingContext, which coordinates the execution of the streaming job. In this case, the StreamingContext is created with a batch interval of 10 seconds, meaning the program processes incoming data in micro-batches of 10 seconds, making real-time processing feasible in a distributed environment. Additionally, checkpointing is enabled by specifying a directory to save the state of the DStream, allowing Spark to recover from failures and resume processing without losing the state of the application. This ensures fault tolerance and reliability in the streaming job.

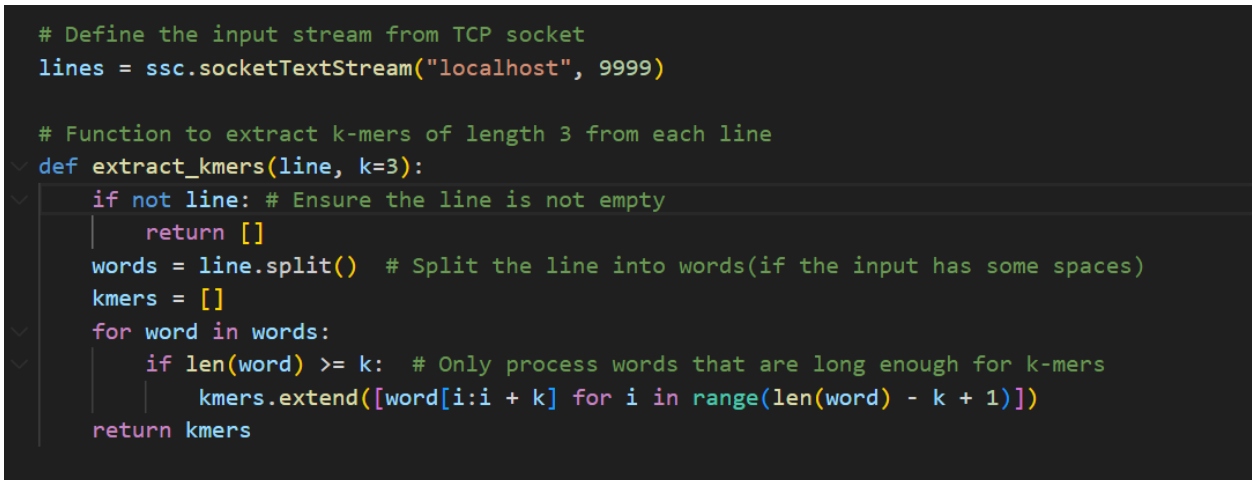


```python
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext

conf = SparkConf().setAppName("KMerCount").setMaster("local[2]")  # Use 2 threads
sc = SparkContext(conf=conf)
ssc = StreamingContext(sc, 10)  # Batch interval of 10 seconds

checkpointDir = "/Users/tharun/checkpoint"
ssc.checkpoint(checkpointDir)
```

2. Defining the Input Stream and Extracting k-mers

The program defines the input data stream from a TCP socket using ssc.socketTextStream. This allows Spark Streaming to read data from the specified TCP port (9999) on the localhost. The data is continuously streamed into the socket using Netcat (ncat), which simulates real-time text input. Each incoming line of text is treated as a unit of data for the Spark Streaming job to process.

To extract the k-mers (substrings of length 3) from the incoming text, the extract_kmers() function is used. It first checks if the line is non-empty, and then splits the line into words using the split() function. Each word is processed to generate all possible substrings of length 3 (k-mers). For example, the word "hello" will generate the k-mers "hel", "ell", and "llo". The function also ensures that only words with a length of 3 or more characters are considered, as shorter words cannot form valid k-mers. This approach efficiently handles text input and prepares it for further processing in the Spark Streaming pipeline.

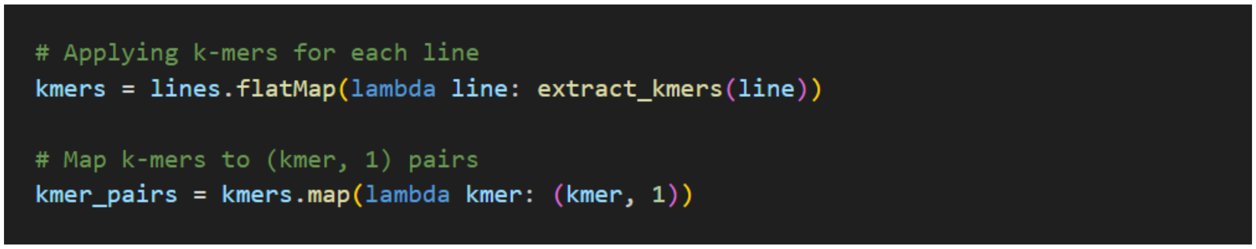


```python
# Define the input stream from TCP socket
lines = ssc.socketTextStream("localhost", 9999)

# Function to extract k-mers of length 3 from each line
def extract_kmers(line, k=3):
    if not line: # Ensure the line is not empty
        return []
    words = line.split()  # Split the line into words(if the input has some spaces)
    kmers = []
    for word in words:
        if len(word) >= k:  # Only process words that are long enough for k-mers
            kmers.extend([word[i:i + k] for i in range(len(word) - k + 1)])
    return kmers
```

3. Flattening the K-mers and Mapping to Key-Value Pairs

After extracting the k-mers, the program uses the flatMap() operation to transform each line into a list of k-mers. The flatMap() function flattens the resulting list of k-mers into a single stream, which is then processed further. This step is necessary because each line of text can generate multiple k-mers, and flatMap() ensures that all of them are included in the final stream for further analysis.

Next, the program maps each k-mer to a tuple in the form of (kmer, 1), where 1 represents the initial count for that particular k-mer. The map() operation is applied to each k-mer in the stream, converting it into a key-value pair. The key is the k-mer itself, and the value is the initial count, which is set to 1 for each occurrence. This step prepares the k-mers for counting their occurrences in subsequent stages of the streaming pipeline.

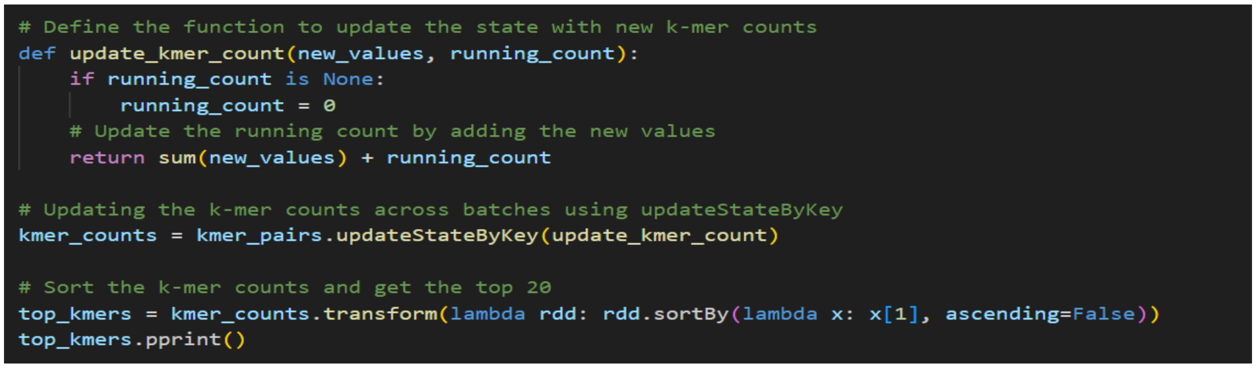


```python
# Applying k-mers for each line
kmers = lines.flatMap(lambda line: extract_kmers(line))

# Map k-mers to (kmer, 1) pairs
kmer_pairs = kmers.map(lambda kmer: (kmer, 1))
```

4. Updating K-mer Counts and Sorting Top K-mers

To maintain a running count of each k-mer across multiple batches, the program uses the updateStateByKey() function. This stateful transformation ensures that the k-mer counts are updated continuously as new data arrives. The update_kmer_count() function takes two arguments: new_values, which represents the k-mer counts from the current batch, and running_count, which stores the cumulative count of each k-mer from previous batches. If no previous count exists (i.e., it's the first time encountering the k-mer), the count is initialized to zero. The new count is calculated by summing the new values and adding them to the running count, which ensures the correct accumulation of counts over time.

Once the k-mer counts are updated, the program sorts them in descending order by their frequency to highlight the most frequent k-mers. The transform() operation is used to sort the k-mer counts, with x[1] being the count of the k-mer. This sorting ensures that the top k-mers are displayed first. Finally, the pprint() function is employed to print the top 20 k-mers to the console, allowing the user to monitor the most frequent k-mers in each batch as the streaming job progresses.

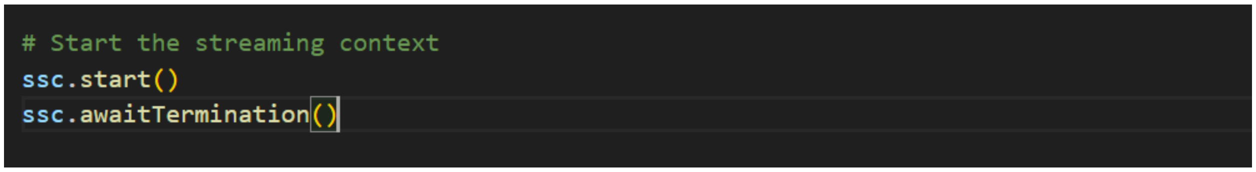


```python
# Define the function to update the state with new k-mer counts
def update_kmer_count(new_values, running_count):
    if running_count is None:
        running_count = 0
    # Update the running count by adding the new values
    return sum(new_values) + running_count

# Updating the k-mer counts across batches using updateStateByKey
kmer_counts = kmer_pairs.updateStateByKey(update_kmer_count)

# Sort the k-mer counts and get the top 20
top_kmers = kmer_counts.transform(lambda rdd: rdd.sortBy(lambda x: x[1], ascending=False))
top_kmers.pprint()
```

Starting the Streaming Context

Finally, the program starts the Spark Streaming context with the ssc.start() function. This begins the execution of the streaming job, and Spark starts processing the incoming data in batches as per the defined batch interval of 10 seconds. The data is continuously ingested, processed, and transformed into k-mers and their counts.

To keep the application running and processing the data stream, the ssc.awaitTermination() function is called. This function causes the program to wait until it is manually stopped, allowing the streaming job to continue running indefinitely or until the job is terminated. This step ensures continuous processing of data in real-time, enabling the application to handle dynamic incoming streams effectively.



```
# Start the streaming context
ssc.start()
ssc.awaitTermination()
```

Netcat (ncat) for Streaming

Netcat (ncat) is a versatile networking utility used to create TCP or UDP connections and listen to or send data across network ports. In this project, ncat is used to simulate a continuous data stream into the Spark Streaming application via a TCP socket. Here's a brief overview of how to use ncat for streaming:

1. **Listening for Incoming Data**: To start streaming data, the first step is to run ncat in "listening" mode on a specific port. In this case, we use port 9999. The command to start listening is: ncat -lk 9999. The -l flag tells ncat to listen on the specified port, and -k allows the connection to remain open and accept multiple data streams. Once the command is executed, ncat is ready to accept and send data over port 9999.

2. **Sending Test Data**: After starting the listener, you can begin sending test data into the socket. Any text you type into the terminal will be sent as a stream of data through the open TCP connection. This data will be picked up by the Spark Streaming application for processing.

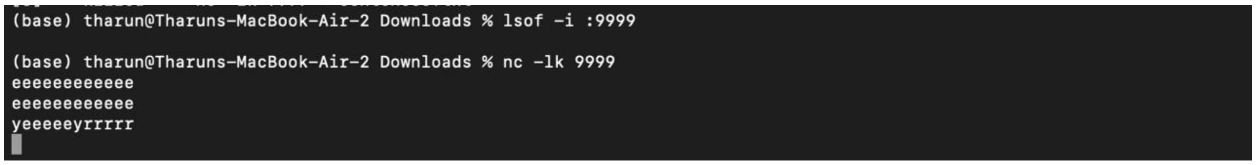
3. **Verifying Data Reception**: On the Spark Streaming side, the program continuously listens to the incoming data stream from localhost:9999. As ncat sends text data into the port, Spark processes the data in real time. You can verify the receipt of data by checking the output in Spark’s console or by observing the results of the k-mer counting operation.

Manual Input Test Results:

For the initial test, I manually provided input data into the Netcat listener to verify the functionality of the k-mer counting program. This allowed me to check whether the application correctly processed the data and maintained running counts across batches. Below is an explanation of the screenshots capturing the test results:

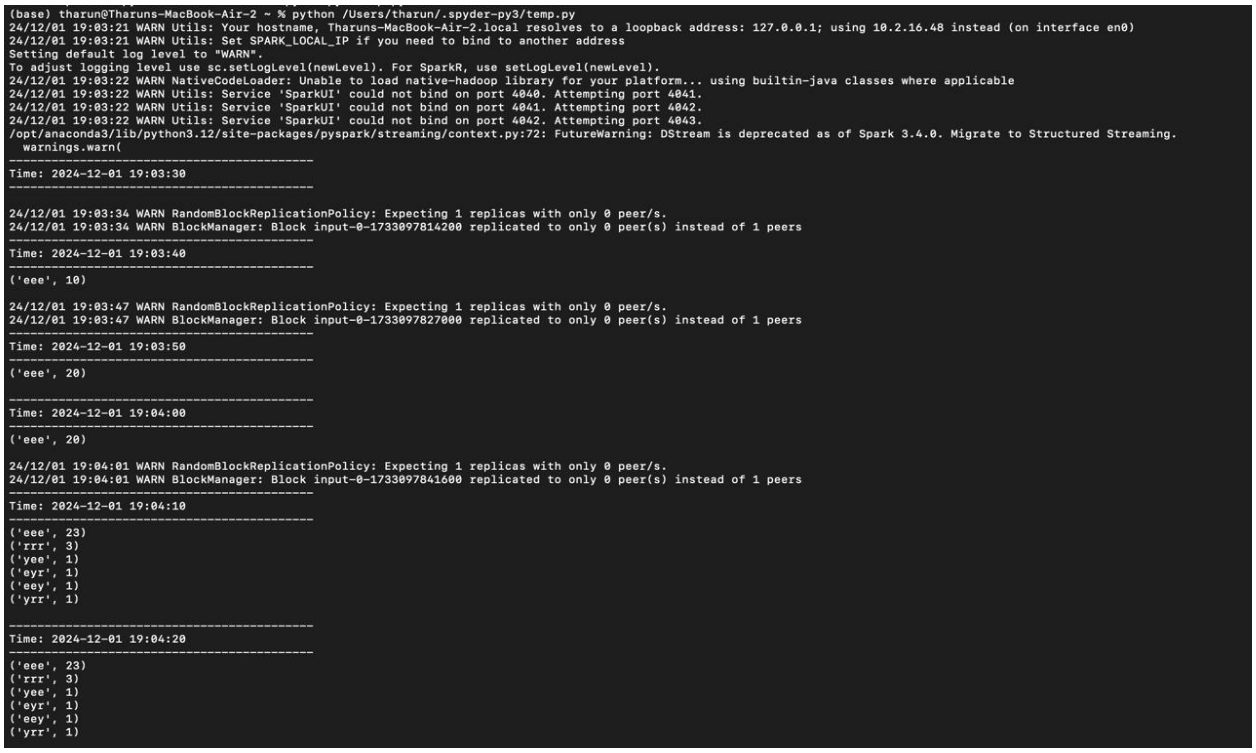
Input Data and Batch Processing In the below screenshot, you can see the data manually entered into the Netcat listener, followed by the output from Spark Streaming. The input data was streamed into the application in real time. As each batch of data was processed (with a batch interval of 10 seconds), the corresponding k-mer counts were calculated. The program correctly identified and counted the 3-mers from the input data, as shown in the output.

This screenshot demonstrates the input data being streamed into the Spark application via the Netcat listener.



```
(base) tharun@Tharuns-MacBook-Air-2 Downloads % lsof -i :9999

(base) tharun@Tharuns-MacBook-Air-2 Downloads % nc -lk 9999
eeeeeeeeeeee
eeeeeeeeeeee
yeeeeeyrrrrr
```

K-mer Counts and Update State: In the below screenshot, you can see the output after multiple batches have been processed. The program aggregates the counts of k-mers across batches, maintaining a running total for each k-mer.



```
(base) tharun@Tharuns-MacBook-Air-2 ~ % python /Users/tharun/.spyder-py3/temp.py
24/12/01 19:03:21 WARN Utils: Your hostname, Tharuns-MacBook-Air-2.local resolves to a loopback address: 127.0.0.1; using 10.2.16.48 instead (on interface en0)
24/12/01 19:03:21 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
24/12/01 19:03:22 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
24/12/01 19:03:22 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
24/12/01 19:03:22 WARN Utils: Service 'SparkUI' could not bind on port 4041. Attempting port 4042.
24/12/01 19:03:22 WARN Utils: Service 'SparkUI' could not bind on port 4042. Attempting port 4043.
/opt/anaconda3/lib/python3.12/site-packages/pyspark/streaming/context.py:72: FutureWarning: DStream is deprecated as of Spark 3.4.0. Migrate to Structured Streaming.
  warnings.warn(
-------------------------------------------
Time: 2024-12-01 19:03:30
-------------------------------------------

24/12/01 19:03:34 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 19:03:34 WARN BlockManager: Block input-0-1733097814200 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2024-12-01 19:03:40
-------------------------------------------
('eee', 10)

24/12/01 19:03:47 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 19:03:47 WARN BlockManager: Block input-0-1733097827000 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2024-12-01 19:03:50
-------------------------------------------
('eee', 20)

-------------------------------------------
Time: 2024-12-01 19:04:00
-------------------------------------------
('eee', 20)

24/12/01 19:04:01 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 19:04:01 WARN BlockManager: Block input-0-1733097841600 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2024-12-01 19:04:10
-------------------------------------------
('eee', 23)
('rrr', 3)
('yee', 1)
('eyr', 1)
('eey', 1)
('yrr', 1)

-------------------------------------------
Time: 2024-12-01 19:04:20
-------------------------------------------
('eee', 23)
('rrr', 3)
('yee', 1)
('eyr', 1)
('eey', 1)
('yrr', 1)
```

This screenshot highlights the k-mer counts printed to the console. The output shows how each k-mer's count is updated after each batch, and it demonstrates the program's ability to maintain a cumulative count across multiple micro-batches.

Execution Steps

To execute the Spark Streaming application and simulate the input stream

1. **Simulate Input Stream Using Netcat**: Open a terminal and run the "nc -lk 9999 < sentences.txt" Netcat command to stream data from the sentences.txt file into the Spark application. This command listens on port 9999 and continuously sends the contents of paragraph.txt to the Spark Streaming application.
2. **Start the Spark Streaming Application:** Run the Python script to initialize the Spark Streaming job. This will set up the necessary configurations and start the SparkContext, initiating the real-time data processing.
3. **Monitor Output:** The output from the Spark Streaming application will be displayed in the terminal where the application is running. The console will show the k-mer counts calculated for each batch of data, updating in real-time.
4. **Re-running the Application**: To restart the application, you must first kill any existing processes that are holding the port 9999. This can be done by using the following commands:

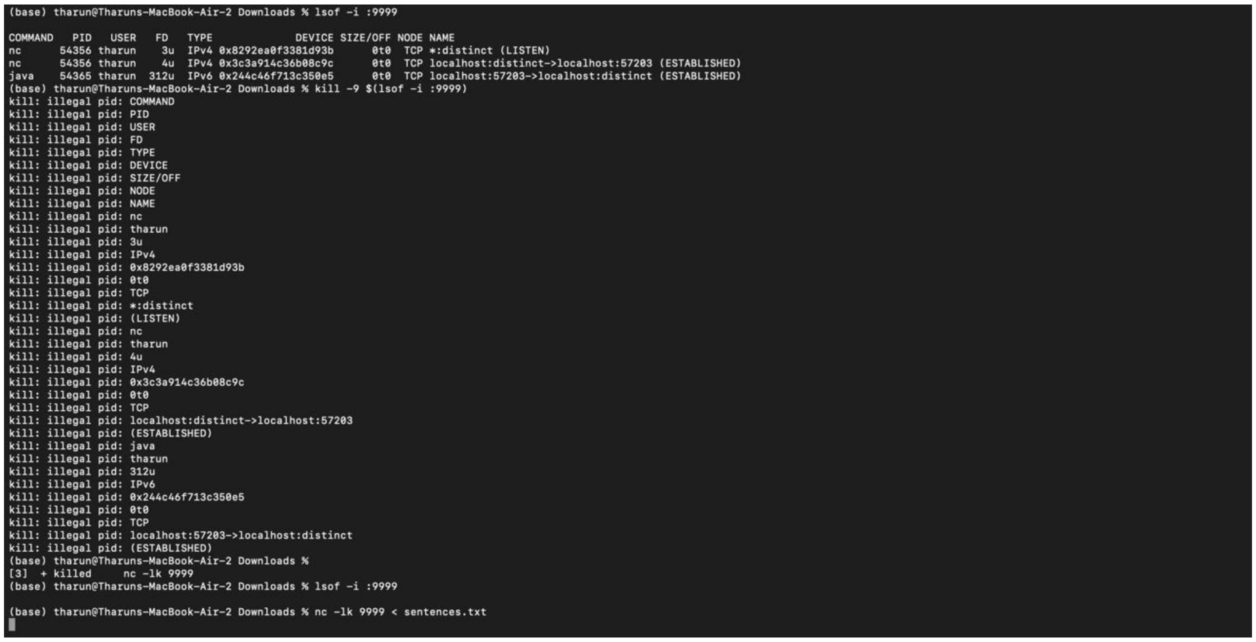
   - Find the process ID using: lsof -i :9999
   - Kill the process using: kill -9 <PID>

Using NetCat Output and Results:

The input data used for the test was randomly generated using the code provided in the homework, and the output from the Spark application was captured.

Input Data and Batch Processing

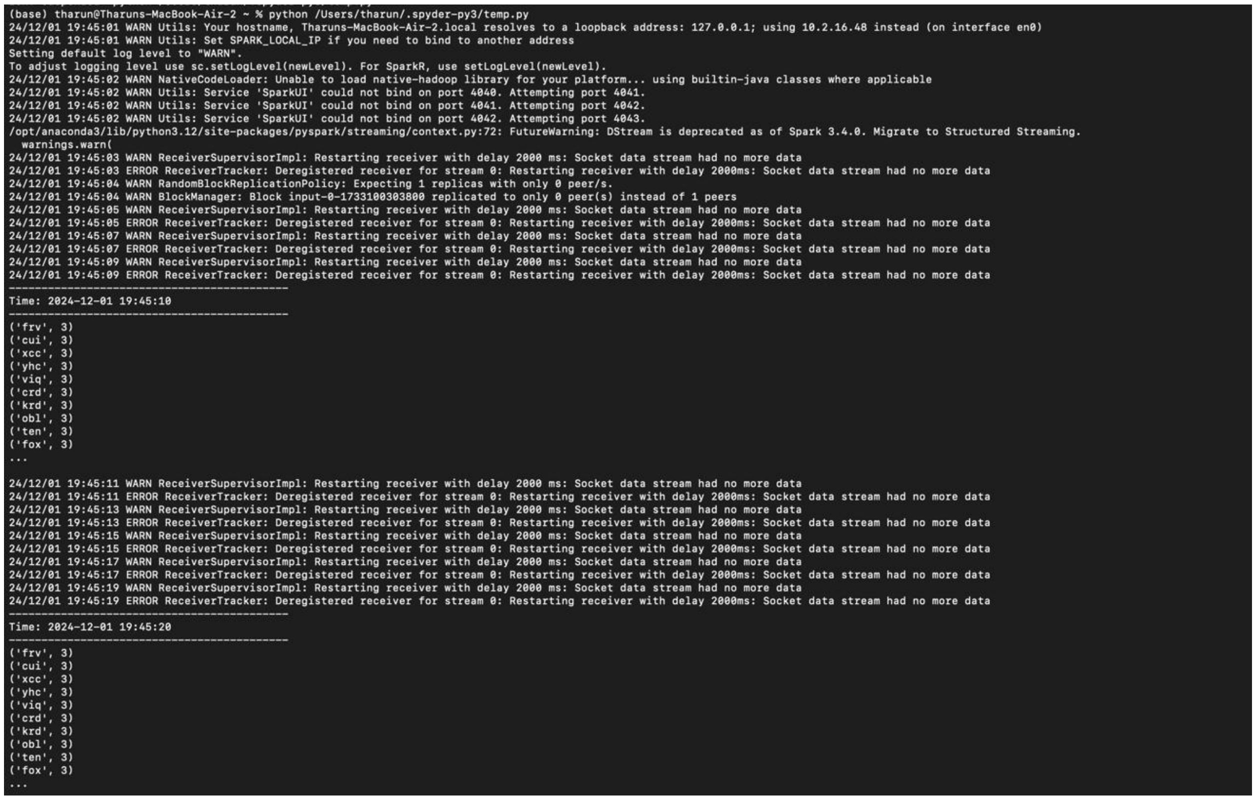
The input data comes from the sentences.txt file, which contains randomly generated text. To simulate the input stream, the Netcat listener was used with the command "**nc -lk 9999 < sentences.txt**", which sends the text into the Spark Streaming application via TCP port 9999. The input text consists of a combination of various words and phrases, and the Spark Streaming application processes in 10-second micro-batches. For each batch, the program extracts 3-mers from the text, counts their occurrences, and maintains a running total of counts for each k-mer across batches. The results are printed after each batch is processed, reflecting the real-time data streaming and processing capabilities of Spark Streaming. In below screenshot, you can observe the raw input data being streamed into the application, with the program continuously updating the k-mer frequencies as each batch is processed.



```
(base) tharun@Tharuns-MacBook-Air-2 Downloads % lsof -i :9999

COMMAND   PID   USER   FD   TYPE             DEVICE SIZE/OFF NODE NAME
nc      54356 tharun    3u  IPv4 0x8292ea0f3381d93b      0t0  TCP *:distinct (LISTEN)
nc      54356 tharun    4u  IPv4 0x3c3a914c36b08c9c      0t0  TCP localhost:distinct->localhost:57203 (ESTABLISHED)
java    54365 tharun  312u  IPv6 0x244c46f713c350e5      0t0  TCP localhost:57203->localhost:distinct (ESTABLISHED)
(base) tharun@Tharuns-MacBook-Air-2 Downloads % kill -9 $(lsof -i :9999)
kill: illegal pid: COMMAND
kill: illegal pid: PID
kill: illegal pid: USER
kill: illegal pid: FD
kill: illegal pid: TYPE
kill: illegal pid: DEVICE
kill: illegal pid: SIZE/OFF
kill: illegal pid: NODE
kill: illegal pid: NAME
kill: illegal pid: nc
kill: illegal pid: tharun
kill: illegal pid: 3u
kill: illegal pid: IPv4
kill: illegal pid: 0x8292ea0f3381d93b
kill: illegal pid: 0t0
kill: illegal pid: TCP
kill: illegal pid: *:distinct
kill: illegal pid: (LISTEN)
kill: illegal pid: nc
kill: illegal pid: tharun
kill: illegal pid: 4u
kill: illegal pid: IPv4
kill: illegal pid: 0x3c3a914c36b08c9c
kill: illegal pid: 0t0
kill: illegal pid: TCP
kill: illegal pid: localhost:distinct->localhost:57203
kill: illegal pid: (ESTABLISHED)
kill: illegal pid: java
kill: illegal pid: tharun
kill: illegal pid: 312u
kill: illegal pid: IPv6
kill: illegal pid: 0x244c46f713c350e5
kill: illegal pid: 0t0
kill: illegal pid: TCP
kill: illegal pid: localhost:57203->localhost:distinct
kill: illegal pid: (ESTABLISHED)
(base) tharun@Tharuns-MacBook-Air-2 Downloads %
[3]  + killed     nc -lk 9999
(base) tharun@Tharuns-MacBook-Air-2 Downloads % lsof -i :9999

(base) tharun@Tharuns-MacBook-Air-2 Downloads % nc -lk 9999 < sentences.txt
```

K-mer Counts and Aggregated Results:

The output from the Spark Streaming application shows how the k-mer counts are calculated and aggregated as the data is processed. In this test, the sentences.txt file was sent all at once in one batch, allowing the program to process the entire input in a single cycle. The program uses the updateStateByKey() function to maintain a running total of the k-mer counts, ensuring that the occurrences of each 3-mer are accurately tracked. The program processes the data in real time, and the k-mer counts are continuously updated as the input is streamed. The top k-mers are displayed in order of frequency, with the most frequent k-mers appearing first, showing the program's ability to handle real-time data aggregation effectively.



```
(base) tharun@Tharuns-MacBook-Air-2 ~ % python /Users/tharun/.spyder-py3/temp.py
24/12/01 19:45:01 WARN Utils: Your hostname, Tharuns-MacBook-Air-2.local resolves to a loopback address: 127.0.0.1; using 10.2.16.48 instead (on interface en0)
24/12/01 19:45:01 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
24/12/01 19:45:02 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
24/12/01 19:45:02 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
24/12/01 19:45:02 WARN Utils: Service 'SparkUI' could not bind on port 4041. Attempting port 4042.
24/12/01 19:45:02 WARN Utils: Service 'SparkUI' could not bind on port 4042. Attempting port 4043.
/opt/anaconda3/lib/python3.12/site-packages/pyspark/streaming/context.py:72: FutureWarning: DStream is deprecated as of Spark 3.4.0. Migrate to Structured Streaming.
  warnings.warn(
24/12/01 19:45:03 WARN ReceiverSupervisorImpl: Restarting receiver with delay 2000 ms: Socket data stream had no more data
24/12/01 19:45:03 ERROR ReceiverTracker: Deregistered receiver for stream 0: Restarting receiver with delay 2000ms: Socket data stream had no more data
24/12/01 19:45:04 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 19:45:04 WARN BlockManager: Block input-0-1733100303800 replicated to only 0 peer(s) instead of 1 peers
24/12/01 19:45:05 WARN ReceiverSupervisorImpl: Restarting receiver with delay 2000 ms: Socket data stream had no more data
24/12/01 19:45:05 ERROR ReceiverTracker: Deregistered receiver for stream 0: Restarting receiver with delay 2000ms: Socket data stream had no more data
24/12/01 19:45:07 WARN ReceiverSupervisorImpl: Restarting receiver with delay 2000 ms: Socket data stream had no more data
24/12/01 19:45:07 ERROR ReceiverTracker: Deregistered receiver for stream 0: Restarting receiver with delay 2000ms: Socket data stream had no more data
24/12/01 19:45:09 WARN ReceiverSupervisorImpl: Restarting receiver with delay 2000 ms: Socket data stream had no more data
24/12/01 19:45:09 ERROR ReceiverTracker: Deregistered receiver for stream 0: Restarting receiver with delay 2000ms: Socket data stream had no more data
-------------------------------------------
Time: 2024-12-01 19:45:10
-------------------------------------------
('frv', 3)
('cui', 3)
('xcc', 3)
('yhc', 3)
('viq', 3)
('crd', 3)
('krd', 3)
('obl', 3)
('ten', 3)
('fox', 3)
...

24/12/01 19:45:11 WARN ReceiverSupervisorImpl: Restarting receiver with delay 2000 ms: Socket data stream had no more data
24/12/01 19:45:11 ERROR ReceiverTracker: Deregistered receiver for stream 0: Restarting receiver with delay 2000ms: Socket data stream had no more data
24/12/01 19:45:13 WARN ReceiverSupervisorImpl: Restarting receiver with delay 2000 ms: Socket data stream had no more data
24/12/01 19:45:13 ERROR ReceiverTracker: Deregistered receiver for stream 0: Restarting receiver with delay 2000ms: Socket data stream had no more data
24/12/01 19:45:15 WARN ReceiverSupervisorImpl: Restarting receiver with delay 2000 ms: Socket data stream had no more data
24/12/01 19:45:15 ERROR ReceiverTracker: Deregistered receiver for stream 0: Restarting receiver with delay 2000ms: Socket data stream had no more data
24/12/01 19:45:17 WARN ReceiverSupervisorImpl: Restarting receiver with delay 2000 ms: Socket data stream had no more data
24/12/01 19:45:17 ERROR ReceiverTracker: Deregistered receiver for stream 0: Restarting receiver with delay 2000ms: Socket data stream had no more data
24/12/01 19:45:19 WARN ReceiverSupervisorImpl: Restarting receiver with delay 2000 ms: Socket data stream had no more data
24/12/01 19:45:19 ERROR ReceiverTracker: Deregistered receiver for stream 0: Restarting receiver with delay 2000ms: Socket data stream had no more data
-------------------------------------------
Time: 2024-12-01 19:45:20
-------------------------------------------
('frv', 3)
('cui', 3)
('xcc', 3)
('yhc', 3)
('viq', 3)
('crd', 3)
('krd', 3)
('obl', 3)
('ten', 3)
('fox', 3)
...
```

Limitation of Using Netcat for Continuous Data Streaming

In the previous test, the entire sentences.txt file was sent all at once using Netcat, which simulated a single batch of input data. Since Netcat sends the whole file in one go, the Spark Streaming application processes the entire batch at once. This results in all the k-mer counts being calculated and displayed in a single batch, without any real-time updating of counts as new data is received.

The main disadvantage of using Netcat in this way is that it does not allow for continuous data streaming. Since the input file is sent in one batch, the program does not process incoming data incrementally or update the k-mer counts continuously. As a result, the counts for each k-mer are not updated in real-time, and the k-mer frequencies are calculated only once after the whole file is processed.

To address this limitation, we shifted to using a custom TCP server. The server enables the continuous stream of data, where each line from the sentences.txt file is sent with a delay, simulating a real-time data stream. This approach allows the Spark Streaming application to process the data in smaller, incremental batches, updating the k-mer counts as new data arrives, ensuring continuous and real-time processing of the input stream.

Using a Custom TCP Server for Continuous Data Streaming

In order to simulate continuous data streaming and overcome the limitations of using Netcat (ncat), we implemented a custom TCP server to send the contents of the sentences.txt file to the Spark Streaming application in real-time. Unlike Netcat, which sends the entire file at once, the TCP server sends one line of the file at a time with a 10-second delay between each line, simulating a real-time data stream.

The custom TCP server operates on localhost:9999, listening for incoming connections from the Spark Streaming application. Once the connection is established, the server reads each line from the sentences.txt file and sends it to the Spark application through the open TCP connection. After sending each line, the server waits for 10 seconds (simulating the streaming delay) before sending the next line. This process continues until the entire file is streamed to the Spark application.

By using this approach, we ensure that the data is sent continuously in small increments, allowing the Spark Streaming application to process the data in real-time and update the k-mer counts incrementally. This is a significant improvement over the Netcat-based approach, where the entire file was processed in a single batch. With the TCP server, the program can now handle incoming data in smaller chunks, updating the k-mer counts as new data arrives, and providing real-time insights into the stream.

This method is much more aligned with the real-time nature of Spark Streaming, as it allows the application to process each line of data as it is received, maintaining an accurate running total of k-mer counts over time.

Implementation

This code implements a custom TCP server that sends data continuously to a Spark Streaming application.

Setting Up the Server: The server is created using Python's socket library. It listens on localhost:9999 and waits for a connection from the Spark Streaming application.

Accepting the Connection: Once the client (Spark Streaming program) connects, the server establishes the connection and prints the client's address.

Sending Data:

- The server reads each line from the sentences.txt file.
- Each line is sent to the client using the sendall() function, and the server waits for 10 seconds before sending the next line, simulating real-time streaming.

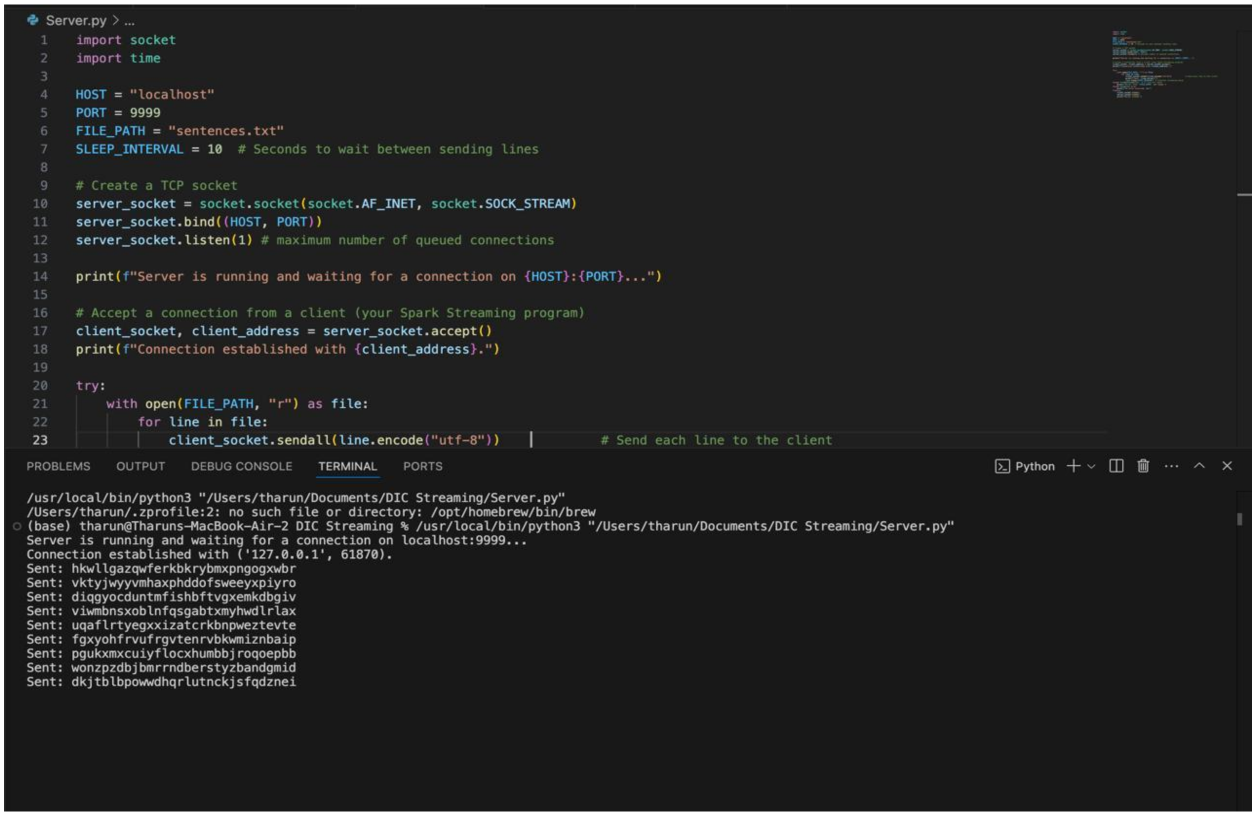
Closing the Connection: After sending all the lines, the server closes both the client and server sockets to properly terminate the connection.

This server simulates a real-time data stream by sending data in small chunks (one line at a time) with a delay, enabling continuous processing in the Spark Streaming application.

Output and Results:

The results of the Spark Streaming application processing continuous data through the custom TCP server are demonstrated through the screenshots provided. As the input data is streamed line by line from the sentences.txt file, the application reads each line, extracts 3-mers (substrings of length 3), and updates the k-mer counts accordingly. This approach allows for real-time k-mer frequency analysis, with the k-mer counts being continuously updated as new data arrives.

Input Streaming



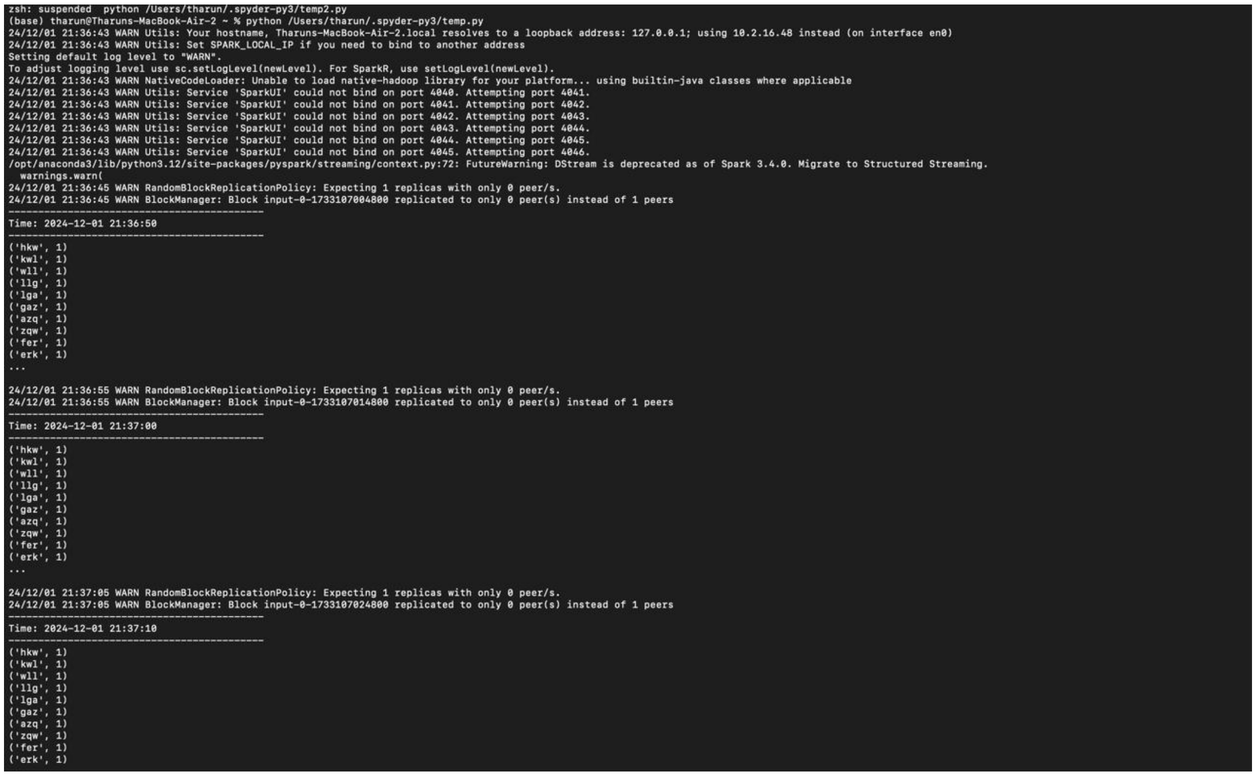
```python
import socket
import time

HOST = "localhost"
PORT = 9999
FILE_PATH = "sentences.txt"
SLEEP_INTERVAL = 10  # Seconds to wait between sending lines

# Create a TCP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((HOST, PORT))
server_socket.listen(1) # maximum number of queued connections

print(f"Server is running and waiting for a connection on {HOST}:{PORT}...")

# Accept a connection from a client (your Spark Streaming program)
client_socket, client_address = server_socket.accept()
print(f"Connection established with {client_address}.")

try:
    with open(FILE_PATH, "r") as file:
        for line in file:
            client_socket.sendall(line.encode("utf-8"))          # Send each line to the client
```

```
/usr/local/bin/python3 "/Users/tharun/Documents/DIC Streaming/Server.py"
/Users/tharun/.zprofile:2: no such file or directory: /opt/homebrew/bin/brew
(base) tharun@Tharuns-MacBook-Air-2 DIC Streaming % /usr/local/bin/python3 "/Users/tharun/Documents/DIC Streaming/Server.py"
Server is running and waiting for a connection on localhost:9999...
Connection established with ('127.0.0.1', 61870).
Sent: hkwllgazqwferkbkrybmxpngogxwbr
Sent: vktyjwyyvmhaxphddofsweeyxpiyro
Sent: diqgyocduntmfishbftvgxemkdbgiv
Sent: viwmbnsxoblnfqsgabtxmyhwdlrlax
Sent: uqaflrtyegxxizatcrkbnpweztevte
Sent: fgxyohfrvufrgvtenrvbkwmiznbaip
Sent: pgukxmxcuiyflocxhumbbjroqoepbb
Sent: wonzpzdbjbmrrndberstyzbandgmid
Sent: dkjtblbpowwdhqrlutnckjsfqdznei
```

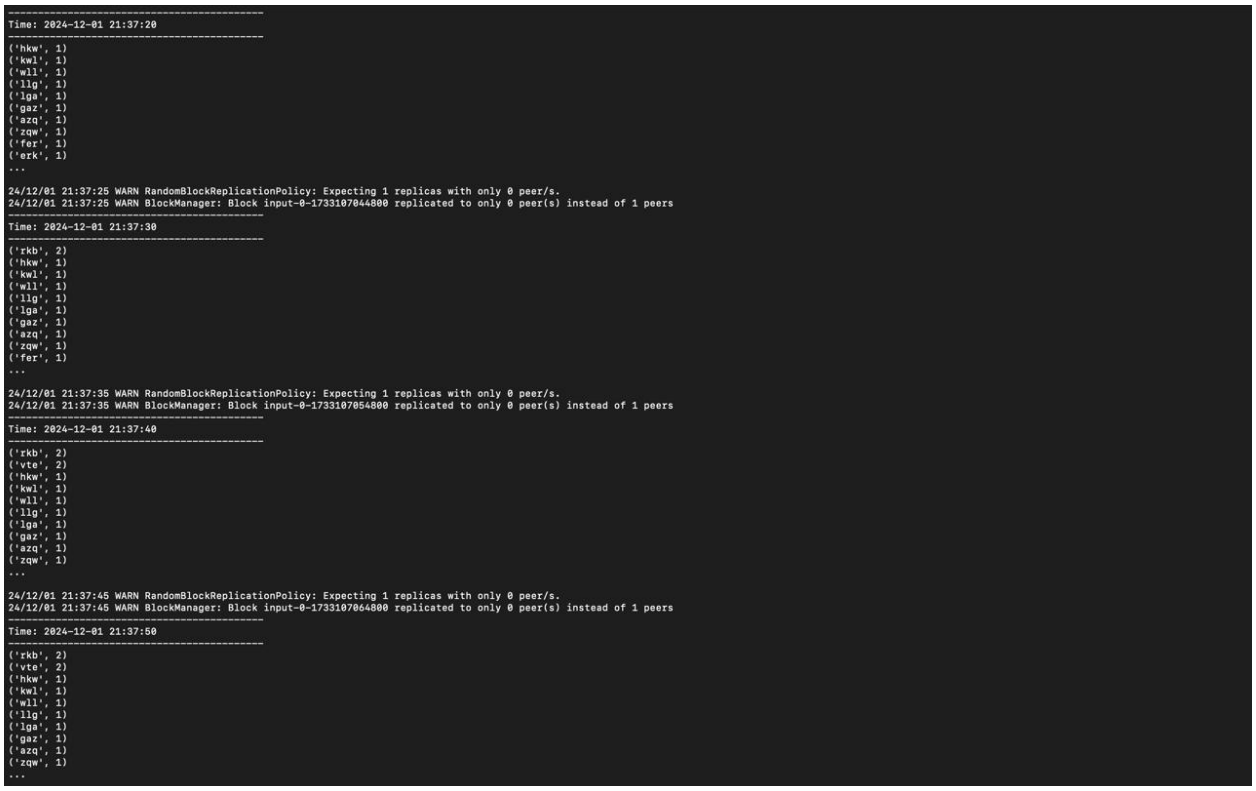
The above screenshot shows the input data being streamed from the TCP server to the Spark application. Each line of the sentences.txt file is sent with a 10-second delay, simulating real-time data. This continuous stream of data is processed by the Spark Streaming application in real time, with the program extracting k-mers from each incoming line.

K-mer Counting and Batch Processing

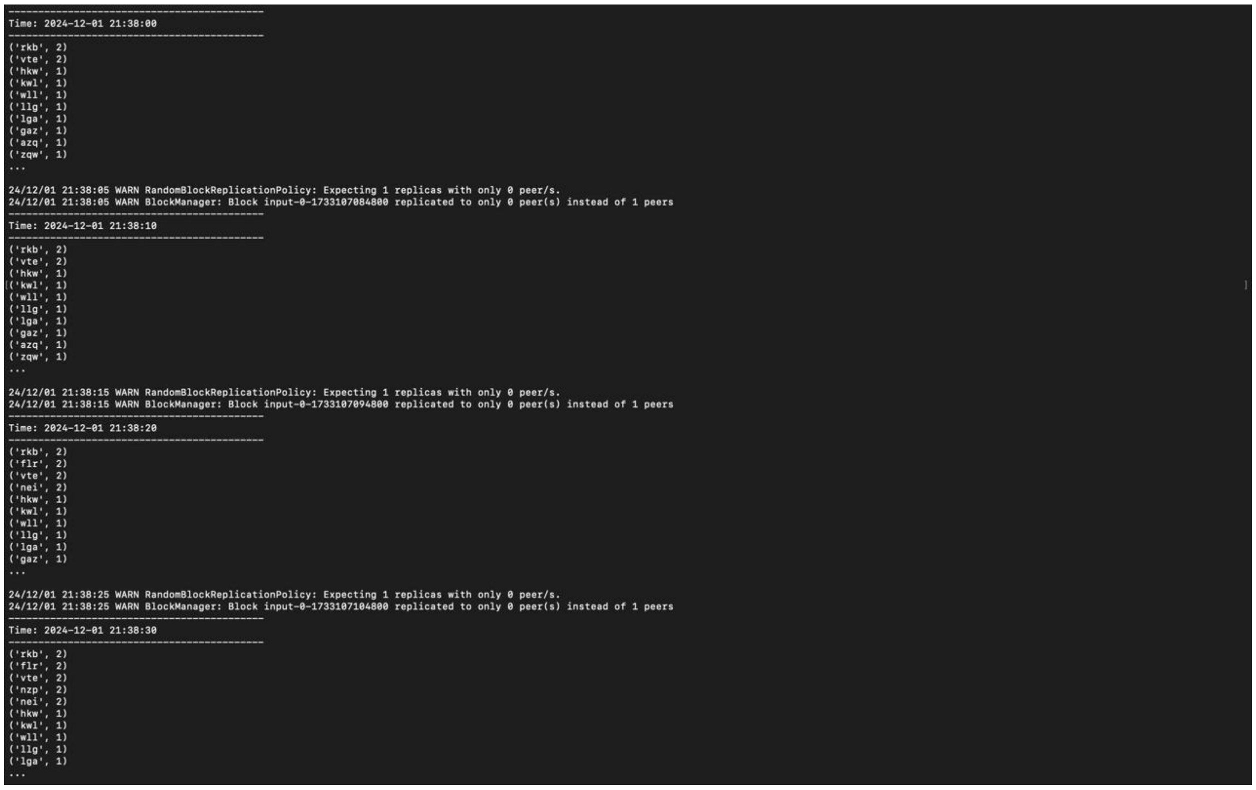
The following screenshots display the results of processing the input data. Each line of text triggers the extraction of k-mers, which are then counted and updated in the k-mer frequency map. The updateStateByKey() function ensures that the k-mer counts are aggregated across multiple batches, maintaining a running total for each k-mer. The k-mer counts are printed to the console, showing how the frequency of each k-mer evolves as the data flows through the system.



```
zsh: suspended  python /Users/tharun/.spyder-py3/temp2.py
(base) tharun@Tharuns-MacBook-Air-2 ~ % python /Users/tharun/.spyder-py3/temp.py
24/12/01 21:36:43 WARN Utils: Your hostname, Tharuns-MacBook-Air-2.local resolves to a loopback address: 127.0.0.1; using 10.2.16.48 instead (on interface en0)
24/12/01 21:36:43 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
24/12/01 21:36:43 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
24/12/01 21:36:43 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
24/12/01 21:36:43 WARN Utils: Service 'SparkUI' could not bind on port 4041. Attempting port 4042.
24/12/01 21:36:43 WARN Utils: Service 'SparkUI' could not bind on port 4042. Attempting port 4043.
24/12/01 21:36:43 WARN Utils: Service 'SparkUI' could not bind on port 4043. Attempting port 4044.
24/12/01 21:36:43 WARN Utils: Service 'SparkUI' could not bind on port 4044. Attempting port 4045.
24/12/01 21:36:43 WARN Utils: Service 'SparkUI' could not bind on port 4045. Attempting port 4046.
/opt/anaconda3/lib/python3.12/site-packages/pyspark/streaming/context.py:72: FutureWarning: DStream is deprecated as of Spark 3.4.0. Migrate to Structured Streaming.
  warnings.warn(
24/12/01 21:36:45 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 21:36:45 WARN BlockManager: Block input-0-1733107004800 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2024-12-01 21:36:50
-------------------------------------------
('hkw', 1)
('kwl', 1)
('wll', 1)
('llg', 1)
('lga', 1)
('gaz', 1)
('azq', 1)
('zqw', 1)
('fer', 1)
('erk', 1)
...

24/12/01 21:36:55 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 21:36:55 WARN BlockManager: Block input-0-1733107014800 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2024-12-01 21:37:00
-------------------------------------------
('hkw', 1)
('kwl', 1)
('wll', 1)
('llg', 1)
('lga', 1)
('gaz', 1)
('azq', 1)
('zqw', 1)
('fer', 1)
('erk', 1)
...

24/12/01 21:37:05 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 21:37:05 WARN BlockManager: Block input-0-1733107024800 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2024-12-01 21:37:10
-------------------------------------------
('hkw', 1)
('kwl', 1)
('wll', 1)
('llg', 1)
('lga', 1)
('gaz', 1)
('azq', 1)
('zqw', 1)
('fer', 1)
('erk', 1)
```



```
-------------------------------------------
Time: 2024-12-01 21:37:20
-------------------------------------------
('hkw', 1)
('kwl', 1)
('wll', 1)
('llg', 1)
('lga', 1)
('gaz', 1)
('azq', 1)
('zqw', 1)
('fer', 1)
('erk', 1)
...

24/12/01 21:37:25 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 21:37:25 WARN BlockManager: Block input-0-1733107044800 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2024-12-01 21:37:30
-------------------------------------------
('rkb', 2)
('hkw', 1)
('kwl', 1)
('wll', 1)
('llg', 1)
('lga', 1)
('gaz', 1)
('azq', 1)
('zqw', 1)
('fer', 1)
...

24/12/01 21:37:35 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 21:37:35 WARN BlockManager: Block input-0-1733107054800 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2024-12-01 21:37:40
-------------------------------------------
('rkb', 2)
('vte', 2)
('hkw', 1)
('kwl', 1)
('wll', 1)
('llg', 1)
('lga', 1)
('gaz', 1)
('azq', 1)
('zqw', 1)
...

24/12/01 21:37:45 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 21:37:45 WARN BlockManager: Block input-0-1733107064800 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2024-12-01 21:37:50
-------------------------------------------
('rkb', 2)
('vte', 2)
('hkw', 1)
('kwl', 1)
('wll', 1)
('llg', 1)
('lga', 1)
('gaz', 1)
('azq', 1)
('zqw', 1)
...
```



```
-------------------------------------------
Time: 2024-12-01 21:38:00
-------------------------------------------
('rkb', 2)
('vte', 2)
('hkw', 1)
('kwl', 1)
('wll', 1)
('llg', 1)
('lga', 1)
('gaz', 1)
('azq', 1)
('zqw', 1)
...

24/12/01 21:38:05 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 21:38:05 WARN BlockManager: Block input-0-1733107084800 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2024-12-01 21:38:10
-------------------------------------------
('rkb', 2)
('vte', 2)
('hkw', 1)
('kwl', 1)
('wll', 1)
('llg', 1)
('lga', 1)
('gaz', 1)
('azq', 1)
('zqw', 1)
...

24/12/01 21:38:15 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 21:38:15 WARN BlockManager: Block input-0-1733107094800 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2024-12-01 21:38:20
-------------------------------------------
('rkb', 2)
('flr', 2)
('vte', 2)
('nei', 2)
('hkw', 1)
('kwl', 1)
('wll', 1)
('llg', 1)
('lga', 1)
('gaz', 1)
...

24/12/01 21:38:25 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/12/01 21:38:25 WARN BlockManager: Block input-0-1733107104800 replicated to only 0 peer(s) instead of 1 peers
-------------------------------------------
Time: 2024-12-01 21:38:30
-------------------------------------------
('rkb', 2)
('flr', 2)
('vte', 2)
('nzp', 2)
('nei', 2)
('hkw', 1)
('kwl', 1)
('wll', 1)
('llg', 1)
('lga', 1)
...
```

Real-Time Processing and Updates

As illustrated in the screenshots, the k-mer counts are updated in real-time with each incoming line. The program continuously aggregates counts for each k-mer, and the most frequent k-mers appear at the top of the list. This process ensures that the program is handling data as it arrives, updating counts and providing a real-time overview of the k-mer frequencies.

The results confirm that the use of the custom TCP server allows the Spark Streaming application to process and update k-mer counts efficiently, in contrast to the static batch processing previously used with Netcat. The continuous data stream processed by the TCP server ensures that the k-mer counts are updated incrementally, providing a more accurate and dynamic reflection of the input data.

Conclusion

In this report, we successfully implemented a k-mer counting program using Spark Streaming to process continuous data streams. Initially, we used Netcat to simulate the data stream, but this approach had limitations, as it processed the entire file in one batch. To overcome this, we implemented a custom TCP server that sends data line by line with a delay, allowing Spark Streaming to process the data incrementally in real-time. This approach enabled continuous k-mer counting, where the counts were updated with each new line of text, providing real-time insights into the data. The results demonstrated the effectiveness of Spark Streaming in handling real-time data streams and updating k-mer frequencies dynamically, showing how the system can efficiently process data and maintain accurate counts over time.