Table of Contents

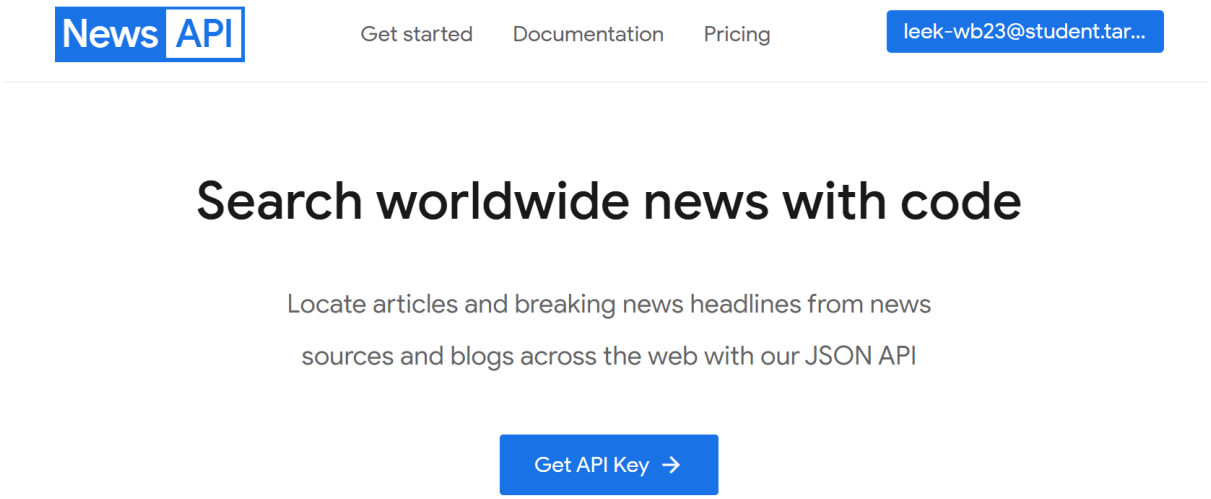
1. Data Ingestion

1.1. Data Source

Data source	1. NewsAPI: Used to fetch a list of relevant news article URLs and metadata based on a query. 2. Individual News Article Web Pages: The actual HTML content of each news article obtained from the URLs provided by NewsAPI.
URL	https://newsapi.org/
Python libraries/API used for scraping/crawling:	1. Requests: Used to perform HTTP GET requests to connect to the News API endpoint and retrieve the JSON-formatted news data. 2. BeautifulSoup (bs4): Employed to parse and clean HTML content. This is particularly used in the ArticleScraper class to extract meaningful text from web pages. 3. Regular Expressions (re): Utilized for text post-processing, including cleaning non-relevant content, removing extraneous characters, and enforcing formatting rules. 4. PySpark: The pipeline uses PySpark to parallelize the scraping and content-cleaning tasks. Spark's distributed processing capabilities are harnessed to process multiple article URLs concurrently. 5. Kafka:

	Kafka is integrated for message production. It facilitates the real-time streaming of both basic article information and cleaned content to designated topics, thereby decoupling data ingestion from downstream processing.

Brief description



1.2. Kafka Streaming

Use case	Track Public Sentiment: This use case involves analyzing news articles to assess the overall

	public sentiment towards Petronas. It aims to determine whether the discourse around the company is generally positive, negative, or neutral based on sentiment analysis. Monitor Reputation: This use case is focused on evaluating the effects of Petronas' recent announcements or events on its public reputation. It intends to capture immediate changes in public perception, highlighting any notable shifts following significant company actions. Observe Sentiment Trends: This use case tracks changes in sentiment over time to detect emerging trends and potential future shifts. It examines historical data to identify patterns in public reception, supporting predictive analysis of Petronas' brand perception.
Objectives	The primary objective is to measure overall public opinion towards Petronas by analyzing the sentiment expressed in news articles and other media sources. The system aims to quantify and monitor sentiment dynamics, providing actionable insights into public reception. Furthermore, these insights will be used to inform strategic decisions and forecast potential impacts on the company's reputation.
Kafka Topic	news-articles-content
End user(s)	The end users of this system are our research group and any interested parties who seek data-driven insights into media influence and public sentiment regarding major corporations. We require detailed analytics on how Petronas is portrayed and perceived in the news to inform our research and strategic decisions. Our findings will support academic studies, industry analyses, and advisory roles for anyone interested in understanding public opinion dynamics.

Screenshot of Kafka message content:

```
Topic 'news-articles' already exists.
Topic 'news-articles-content' already exists.

--- Step 1: Fetch News & Produce Basic Info to Kafka ('news-articles') ---
Fetched 97 articles, 97 have URLs.
Displaying first 3 fetched articles (Title, Desc, URL, PublishedAt, Source):
1. Title: Gas pipeline leak sparks Malaysian inferno
  Description: Pipeline belonging to state-run Petronas sends fire spreading to villages during Eid public holiday.
  URL: https://www.aljazeera.com/news/2025/4/1/gas-pipeline-leak-sparks-inferno-in-malaysia
  PublishedAt: 2025-04-01T08:09:13Z
  Source Name: Al Jazeera English

2. Title: Will Formula 1 Ditch Hybrids? The V10 Engines Revival Rumors Explained
  Description: The Formula 1 season may be just two races old, but 2026 is already dominating the conversation amid speculations the sport may welcome
 back the V10 engines.
  URL: https://www.forbes.com/sites/dancancian/2025/03/27/will-formula-1-ditch-hybrids-the-v10-engines-revival-rumors-explained/
  PublishedAt: 2025-03-27T15:11:05Z
  Source Name: Forbes

3. Title: Lando Norris Claims Australian GP Win, But Lewis Hamilton Struggles
  Description: The Briton took pole-position and then held his nerve during a chaotic race to capitalize on his car’s superiority and take out the fir
st win of the season.
  URL: https://www.forbes.com/sites/dancancian/2025/03/17/lando-norris-claims-australian-gp-win-but-lewis-hamilton-struggles/
  PublishedAt: 2025-03-17T09:29:02Z
  Source Name: Forbes

--- Step 2: Scrape & Clean Article Content (Parallel Spark Job) ---
Extracted 97 URLs for scraping.
Collecting cleaned content from Spark workers...
-
Step 2 Finished: Collected scraped content for 93 articles.


--- Merging Original Article Metadata with Scraped Content ---
Successfully merged metadata with scraped content for 93 articles.

--- Step 3: Produce Merged Content to Kafka ('news-articles-content') ---
KafkaProducer connected to ['localhost:9092'].
Producing merged data for 93 articles to Kafka topic 'news-articles-content'...
Flushing Kafka producer...
Closing Kafka producer.
Step 3 Finished: Produced 93 messages to 'news-articles-content'.

--- Kafka Consumer Started for topic 'news-articles-content' (Timeout: 5000 ms) ---
Received message: Partition=0, Offset=829
Key=None
Value=Gas pipeline leak sparks Malaysian inferno
Processing consumed message...
--------------------
Received message: Partition=0, Offset=830
Key=None
Value=Will Formula 1 Ditch Hybrids? The V10 Engines Revival Rumors Explained
Processing consumed message...
--------------------
Received message: Partition=0, Offset=831
Key=None
Value=Lando Norris Claims Australian GP Win, But Lewis Hamilton Struggles
Processing consumed message...
--------------------
Kafka Consumer: Received a total of 93 articles.
Kafka consumer closed.
```

1.3. List of Python classes:

Name of Python classes	Author
CONFIG	Lee Kevin
ArticlesScraper	Lee Kevin
KafkaHandler	Lee Kevin
NewsAPIFetcher	Lee Kevin
NewsPipeline	Lee Kevin

1.4. Code for Python Classes

Class: CONFIG
Description: This code defines a configuration dictionary for a news processing pipeline.

```
import os

CONFIG = {
    "spark": {
        "appName": "NewsPipeline_OOP_Distributed",
        "logLevel": "WARN",
        "showMaxFields": 100,
    },
    "newsApi": {
        "apiKey": os.environ.get('NEWSAPI_KEY',
'a38a8cf3d941413997ab8b6fdd5d1cc4'),
        "query": "Petronas",
        "url_template":
'https://newsapi.org/v2/everything?q={query}&sortBy=popularity&language=en&apiKey
={apiKey}',
    },
    "kafka": {
        "bootstrap_servers": ['localhost:9092'],
        "topic_articles": 'news-articles',
        "topic_content": 'news-articles-content',
        "producer_retries": 3,
        "topic_partitions": 1,
        "topic_replication": 1,
    },
    "scraping": {
        "timeout": 30,
        "user_agent": 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36',
        "min_content_words": 15,
        "faq_keywords": [
            "faq", "frequently asked questions", "how to", "questions", "help",
"contact us", "support", "terms and conditions",
            "privacy policy", "cookie policy", "all rights reserved",
"disclaimer", "sitemap", "legal", "copyright"
        ],
        "non_article_keywords": [
             "subscription", "subscribe", "comment", "comments", "create a
display name", "Follow Al Jazeera English",
             "Sponsored", "edited by", "Sign up", "name", "email", "website",
"news", "offer",
             "Email address", "Follow", "info", "Your bid", "proceed", "inbox",
"receive", "Thank you for your report!",
             "Your daily digest", "Search", "Review", "Reviews", "Car Launches",
"Driven Communications Sdn. Bhd.", "200801035597 (836938-P)",
             "Follow", "Email address", "Sign up", "For more of the latest",
"subscribing", "2025 Hearst Magazines, Inc. .",
             "Connect", "enjoy", "love", "Best", "The Associated Press",
"NBCUniversal Media, LLC",
             "Reporting by", "Contact", "ResearchAndMarkets.com",
"Advertisement", "thank you", "Your daily digest of everything happening on the
site. 2025 Bring a Trailer Media, LLC. .",
```

```
            "The materials provided on this Web site are for informational",
"Cookies", "Connect With Us", "Back to top",
            "Comments have to be in English", "We have migrated to a new
commenting platform", "Vuukle",
            "Patriots membership", "Become a Daily Caller Patriot today",
"KicksOnFire.com", "F1technical",
            "Facebook", "Account", "Mediacorp 2025", "TouringPlans.com",
"copyright", "Robb Report tote bag", "All below"
        ],
        "non_article_statements": [
            'Search the news', 'Personalise the news', 'stay in the know',
      'Emergency', 'Backstory', 'Newsletters', '中文新闻',
            'BERITA BAHASA INDONESIA', 'TOK PISIN'
        ],
    },
    "output": {
        "parquet_path":
"hdfs:///user/student/filtered_articles_spark_oop_parquet",
        "output_coalesce": None,
        "output_mode": "overwrite",
    },
}
```

Class: ArticlesScraper

Description: This class is responsible for scraping and cleaning the content of a single article from a given URL.

```
import requests
import json
import re
import sys
import os

from typing import List, Dict, Optional, Any
from bs4 import BeautifulSoup
from kafka import KafkaProducer, KafkaAdminClient, KafkaConsumer
from kafka.admin import NewTopic
from kafka.errors import TopicAlreadyExistsError, NoBrokersAvailable
from pyspark.sql import SparkSession, DataFrame
from pyspark.sql.types import StructType, StructField, StringType
from pyspark import SparkContext
from langdetect import detect, LangDetectException

class ArticleScraper:
    """
    Handles scraping and cleaning logic for a single article URL.
    Static method design allows usage within Spark transformations without
    serializing the entire class instance.
    """
    @staticmethod
    def scrape_and_clean(url: str, config: Dict[str, Any]) -> List[Dict[str,
str]]:
        """
        Fetches, parses, and cleans content from a single URL.
        Returns a list containing zero or one dictionary: [{'title': ..., 'url':
..., 'content': ...}]
        Args:
            url: The URL to scrape.
            config: Dictionary containing scraping parameters ('timeout',
'user_agent', keyword lists, etc.).
        """
        timeout = config.get('timeout', 30)
        user_agent = config.get('user_agent', 'Mozilla/5.0')
        faq_keywords = config.get('faq_keywords', [])
        non_article_keywords = config.get('non_article_keywords', [])
        non_article_statements = config.get('non_article_statements', [])
        min_content_words = config.get('min_content_words', 10)

        try:
            response = requests.get(url, timeout=timeout, headers={'User-Agent':
user_agent})

            if response.status_code in [403, 404, 429, 401, 405] or
response.status_code >= 500:
                return []

            response.raise_for_status()

            if 'text/html' not in response.headers.get('Content-Type',
'').lower():
                return []
```

```
            soup = BeautifulSoup(response.text, 'html.parser')
            title = soup.title.get_text(strip=True) if soup.title else "No Title
Found"

            # --- HTML Cleaning ---
            selectors_to_remove = [
                'footer', 'aside', 'nav', 'form', 'header', 'script', 'style',
'noscript',
                'div.subscription', 'div.newsletter', 'div.comments',
'div.related-articles', 'div.advertisement',
                'div.popup', 'div.banner', 'div.sponsored', 'div.social-media',
'div.more-articles', 'div.alerts',
                'section.subscription', 'section.newsletter', 'section.comments',
'section.related-articles', 'section.advertisement',
                'section.popup', 'section.banner', 'section.sponsored',
'section.social-media', 'section.more-articles', 'section.alerts',
                'span.subscription', 'span.newsletter', 'span.comments',
'span.related-articles', 'span.advertisement',
                'span.popup', 'span.banner', 'span.sponsored',
'span.social-media', 'span.more-articles', 'span.alerts',
                'div.topic', 'div.acknowledgment', 'div.external-source',
'div.time-zone', 'div.multilingual', 'div.search',
                'div.manage-alerts', 'div.article-commenting',
'div.breaking-news', 'div.article-comments', 'div.affiliate-links',
            ]
            for selector in selectors_to_remove:
                for element in soup.select(selector):
                    element.decompose()

            tags_to_remove = [
                # Multimedia tags
                'img', 'picture', 'figure', 'figcaption',
                'video', 'audio', 'source', 'track',
                'iframe', 'embed', 'object',
                # Other non-paragraph content or structural elements often
irrelevant for text
                'h1', 'h2', 'h3', 'h4', 'h5', 'h6',
                'a',
                'ul', 'ol', 'li',
                'table', 'thead', 'tbody', 'tr', 'th', 'td',
                'button', 'svg', 'canvas'
            ]
            for tag_name in tags_to_remove:
                for tag in soup.find_all(tag_name):
                    tag.decompose()

            # --- Text Extraction & Cleaning ---
            paragraphs = soup.find_all('p')
            page_text = "\n".join([para.get_text(separator=' ', strip=True) for
para in paragraphs if para.get_text(strip=True)])

            if not page_text: return []

            try:
                detected_language = detect(page_text)
                if detected_language != 'en':
                    return []
            except LangDetectException:
                return []
```

```
            for keyword in non_article_keywords:
                page_text = re.sub(r'[^.?!]*?\b' + re.escape(keyword) +
r'\b[^.?!]*?[.?!]', '', page_text, flags=re.IGNORECASE | re.DOTALL)
                page_text = re.sub(r'^\s*\b' + re.escape(keyword) +
r'\b[^.?!]*?[.?!]', '', page_text, flags=re.IGNORECASE | re.DOTALL |
re.MULTILINE)
            for statement in non_article_statements: page_text =
page_text.replace(statement, '')
            for faq in faq_keywords: page_text = re.sub(r'\b' + re.escape(faq) +
r'\b', '', page_text, flags=re.IGNORECASE)
            page_text = re.sub(r'(\b(©|All Rights Reserved|Privacy Policy|Terms
and Conditions|Cookie Policy|Disclaimer)\b)', '', page_text, flags=re.IGNORECASE)
            page_text = re.sub(r'[^\x00-\x7F]+', '', page_text)
            page_text = re.sub(r'\s+', ' ', page_text).strip()

            # Remove non-English words (keeping only ASCII characters and spaces)
            page_text = re.sub(r'[^\x00-\x7F\s]+', '', page_text).strip()
            page_text = re.sub(r'\s+', ' ', page_text).strip()

            # --- Final Validation ---
            if not page_text or len(page_text.split()) < min_content_words:
                return []

            return [{'title': title, 'url': url, 'content': page_text}]

        except requests.exceptions.Timeout: return []
        except requests.exceptions.RequestException: return []
        except Exception as e:
            return []

def consume_messages(topic_name: str, bootstrap_servers: str, group_id: str =
'my-group'):
    try:
        consumer = KafkaConsumer(
            topic_name,
            bootstrap_servers=bootstrap_servers,
            auto_offset_reset='earliest',
            enable_auto_commit=True,
            group_id=group_id,
            value_deserializer=lambda x: json.loads(x.decode('utf-8')))

        print(f"Consumer started for topic '{topic_name}'...")
        for message in consumer:
            print(f"Received message: Partition={message.partition},
Offset={message.offset}")
            print(f"Key={message.key}")
            print(f"Value={message.value}")

    except NoBrokersAvailable:
        print(f"Error: Could not connect to Kafka brokers at
{bootstrap_servers}")
    except json.JSONDecodeError:
        print("Error: Could not decode JSON message from Kafka.")
    except KeyboardInterrupt:
        print("Consumer stopped.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
    finally:
        if 'consumer' in locals() and consumer:
```

```
            consumer.close()
            print("Consumer closed.")
```

Class: KafkaHandler

Description: This class is responsible for managing Kafka interactions, including creating topics and producing messages.

```
import requests
import json
import re
import sys
import os

from bs4 import BeautifulSoup
from kafka import KafkaProducer, KafkaAdminClient
from kafka.admin import NewTopic
from kafka.errors import TopicAlreadyExistsError, NoBrokersAvailable
from pyspark.sql import SparkSession, DataFrame
from pyspark.sql.types import StructType, StructField, StringType
from pyspark import SparkContext
from typing import List, Dict, Optional, Any
from typing import List, Dict, Optional, Any

class KafkaHandler:
    """
    Handles interactions with Kafka, including topic creation and message
production.
    Encapsulates Kafka connection details and producer logic.
    """
    def __init__(self, bootstrap_servers: List[str], retries: int = 3):
        self.bootstrap_servers = bootstrap_servers
        self.retries = retries
        print(f"KafkaHandler initialized for servers: {self.bootstrap_servers}")

    def create_topic_if_not_exists(self, topic_name: str, num_partitions: int =
1, replication_factor: int = 1) -> None:
        """Checks if a Kafka topic exists and creates it if not."""
        admin_client = None
        try:
            admin_client =
KafkaAdminClient(bootstrap_servers=self.bootstrap_servers)
            existing_topics = admin_client.list_topics()
            if topic_name not in existing_topics:
                print(f"Topic '{topic_name}' not found. Attempting to create...")
                topic = NewTopic(name=topic_name, num_partitions=num_partitions,
replication_factor=replication_factor)
                admin_client.create_topics(new_topics=[topic],
validate_only=False)
                print(f"Topic '{topic_name}' created successfully.")
            else:
                print(f"Topic '{topic_name}' already exists.")
        except NoBrokersAvailable:
            print(f"ERROR: Cannot connect to Kafka brokers at
{self.bootstrap_servers}. Please ensure Kafka is running.", file=sys.stderr)
            raise
        except Exception as e:
            print(f"ERROR during Kafka topic check/creation for '{topic_name}':
{e}", file=sys.stderr)
        finally:
            if admin_client:
                admin_client.close()
```

```
    def get_producer(self) -> Optional[KafkaProducer]:
        """Creates and returns a KafkaProducer instance."""
        try:
            producer = KafkaProducer(
                bootstrap_servers=self.bootstrap_servers,
                value_serializer=lambda v: json.dumps(v).encode('utf-8'),
                retries=self.retries
            )
            print(f"KafkaProducer connected to {self.bootstrap_servers}.")
            return producer
        except NoBrokersAvailable:
            print(f"ERROR: Kafka connection failed for producer. Cannot produce
messages.", file=sys.stderr)
            return None
        except Exception as e:
            print(f"ERROR: Failed to initialize KafkaProducer: {e}",
file=sys.stderr)
            return None

    @staticmethod
    def send_message(producer: KafkaProducer, topic: str, value: Dict[str, Any])
-> bool:
        """Sends a single message using the provided producer."""
        if not producer: return False
        try:
            producer.send(topic, value=value)
            return True
        except Exception as e:
            print(f"ERROR sending message to topic '{topic}': {e}",
file=sys.stderr)
            return False

    @staticmethod
    def flush_producer(producer: Optional[KafkaProducer]) -> None:
        """Flushes the producer to ensure messages are sent."""
        if producer:
            try:
                print("Flushing Kafka producer...")
                producer.flush()
            except Exception as e:
                 print(f"ERROR flushing Kafka producer: {e}", file=sys.stderr)

    @staticmethod
    def close_producer(producer: Optional[KafkaProducer]) -> None:
        """Closes the Kafka producer."""
        if producer:
            try:
                print("Closing Kafka producer.")
                producer.close()
            except Exception as e:
                print(f"ERROR closing Kafka producer: {e}", file=sys.stderr)
```

Class: NewsAPIFetcher
Description: This class is responsible for fetching news articles from the NewsAPI based on a query and sending them to a Kafka topic.

```
import json
import time
import threading
import logging
import requests


from text_cleaner.processor.processor import RegexProcessor


from kafka import KafkaProducer
from bs4 import BeautifulSoup

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

NEWS_API_KEY = "41db74da891e480c9384a475decd3206"
NEWS_API_URL = "https://newsapi.org/v2/everything"
SEARCH_QUERY = "Petronas"
LANGUAGE = "en"
SORT_BY = "popularity"

class NewsAPIFetcher:
    """
    Class to fetch articles from NewsAPI and send them to Kafka.
    """


    def __init__(self, api_key, query, kafka_bootstrap_servers, kafka_topic):
        self.api_key = api_key
        self.query = query
        self.kafka_producer = KafkaProducer(
            bootstrap_servers=kafka_bootstrap_servers,
            value_serializer=lambda v: json.dumps(v).encode('utf-8')
        )
        self.kafka_topic = kafka_topic
        self.last_fetch_time = None
        self.fetch_interval = 300  # 5 minutes in seconds
        self.running = False
        self.thread = None

    def fetch_articles(self):
        """
        Fetches articles from NewsAPI based on the query.
        """
        try:
            # Construct the API URL
            url = (f'{NEWS_API_URL}?'
```

```
                f'q={self.query}&'
                f'sortBy={SORT_BY}&'
                f'language={LANGUAGE}&'
                f'apiKey={self.api_key}')

        # Add from parameter if we've fetched before to avoid duplicates
        if self.last_fetch_time:
            # Format the time as ISO 8601
            from_time = self.last_fetch_time.strftime('%Y-%m-%dT%H:%M:%S')
            url += f'&from={from_time}'

        # Send the GET request and parse the JSON response
        response = requests.get(url)
        data = response.json()

        # Update the last fetch time
        self.last_fetch_time = time.time()

        # Process the articles
        if "articles" in data:
            return data["articles"]
        else:
            logger.warning(f"No articles found in API response: {data}")
            return []
    except Exception as e:
        logger.error(f"Error fetching articles from NewsAPI: {str(e)}")
        return []

def process_and_send_articles(self, articles):
    """
    Processes articles and sends them to Kafka.
    """
    sent_count = 0
    for article in articles:
        try:
            # Extract article information
            title = article.get("title", "")
            description = article.get("description", "")
            url = article.get("url", "")

            # Skip if title or URL is missing
            if not title or not url:
                continue

            # Fetch and clean the article content
            content = RegexProcessor.fetch_and_clean_article_content(url)

            # Prepare the article information
            article_info = {
                "title": title,
                "description": description,
                "url": url,
                "content": content
            }

            # Send the article data to Kafka topic
            self.kafka_producer.send(self.kafka_topic, value=article_info)

            logger.info(f"Sent article to Kafka: {title}")
            sent_count += 1
```

```
            except Exception as e:
                logger.error(f"Error processing article: {str(e)}")

        # Flush to ensure all messages are sent
        self.kafka_producer.flush()
        logger.info(f"Total articles sent to Kafka: {sent_count}")
        return sent_count

    def fetch_and_send(self):
        """
        Fetches articles and sends them to Kafka.
        """
        articles = self.fetch_articles()
        return self.process_and_send_articles(articles)

    def run_continuously(self):
        """
        Runs the fetcher continuously at the specified interval.
        """
        while self.running:
            try:
                self.fetch_and_send()
            except Exception as e:
                logger.error(f"Error in continuous fetching: {str(e)}")

            # Sleep for the specified interval
            time.sleep(self.fetch_interval)

    def start(self):
        """
        Starts the fetcher in a separate thread.
        """
        if not self.running:
            self.running = True
            self.thread = threading.Thread(target=self.run_continuously)
            self.thread.daemon = True
            self.thread.start()
            logger.info("NewsAPI fetcher started")

    def stop(self):
        """
        Stops the fetcher.
        """
        if self.running:
            self.running = False
            if self.thread:
                self.thread.join(timeout=10)
            logger.info("NewsAPI fetcher stopped")
```

Class: NewsPipeline
Description: This class runs the entire process of getting news, cleaning it, and saving it.

```
import sys
import news_fetcher
import json
import logging

from typing import List, Dict, Optional, Any
from pyspark.sql import SparkSession, DataFrame
from pyspark.sql.types import StructType, StructField, StringType
from pyspark import SparkContext
from kafka_handler_Task1 import KafkaHandler
from news_fetcher_Task1 import NewsAPIFetcher
from article_scraper_Task1 import ArticleScraper

from config_Task1 import CONFIG
from kafka import KafkaConsumer
from kafka.errors import NoBrokersAvailable, KafkaError

logging.getLogger('kafka.coordinator').setLevel(logging.CRITICAL)

class NewsPipeline:
    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.spark: Optional[SparkSession] = None
        self.sc: Optional[SparkContext] = None
        self.kafka_handler = KafkaHandler(
            bootstrap_servers=config['kafka']['bootstrap_servers'],
            retries=config['kafka']['producer_retries']
        )
        self.news_fetcher = NewsAPIFetcher(
            api_key=config['newsApi']['apiKey'],
            query=config['newsApi']['query'],
            kafka_bootstrap_servers=config['kafka']['bootstrap_servers'],
            kafka_topic=config['kafka']['topic_articles']
        )
        print("NewsPipeline initialized.")

    def _start_spark(self) -> None:
        if self.spark:
            print("SparkSession already started.")
            return
```

```
        try:
            self.spark = SparkSession.builder \
                .appName(self.config['spark']['appName']) \
                .config("spark.sql.debug.maxToStringFields",
self.config['spark']['showMaxFields']) \
                .getOrCreate()
            self.sc = self.spark.sparkContext
            self.sc.setLogLevel(self.config['spark']['logLevel'])
            print(f"SparkSession '{self.config['spark']['appName']}' started.")
        except Exception as e:
            print(f"FATAL: Error initializing SparkSession: {e}",
file=sys.stderr)
            raise

    def _stop_spark(self) -> None:
        if self.spark:
            print("Stopping SparkSession...")
            self.spark.stop()
            self.spark = None
            self.sc = None
            print("SparkSession stopped.")

    def _prepare_kafka_topics(self) -> None:
        print("\n--- Preparing Kafka Topics ---")
        try:
            self.kafka_handler.create_topic_if_not_exists(
                topic_name=self.config['kafka']['topic_articles'],
                num_partitions=self.config['kafka']['topic_partitions'],
                replication_factor=self.config['kafka']['topic_replication']
            )
            self.kafka_handler.create_topic_if_not_exists(
                topic_name=self.config['kafka']['topic_content'],
                num_partitions=self.config['kafka']['topic_partitions'],
                replication_factor=self.config['kafka']['topic_replication']
            )
        except Exception as e:
            print(f"WARNING: Could not ensure Kafka topics exist (may require
manual creation or permissions): {e}", file=sys.stderr)

    def _run_step1_fetch_and_produce(self) -> Optional[List[Dict[str, Any]]]:
        print(f"\n--- Step 1: Fetch News & Produce Basic Info to Kafka
('{self.config['kafka']['topic_articles']}') ---")
        articles = self.news_fetcher.fetch_articles()
        produced_count = 0

        if articles is None:
            print("Step 1 failed: Could not fetch articles from NewsAPI.")
            return None
        if not articles:
            print("Step 1 completed: No articles found by NewsAPI.")
            return []

        valid_articles = [a for a in articles if a.get("url")]
        print(f"Fetched {len(articles)} articles, {len(valid_articles)} have
URLs.")
        articles = valid_articles

        if not articles:
            print("Step 1 completed: No articles with valid URLs found.")
            return []
```

```
        print("Displaying first 3 fetched articles (Title, Desc, URL,
PublishedAt, Source):")
        for i, article in enumerate(articles[:3], start=1):
            source_info = article.get("source", {})
            source_name = source_info.get("name", "N/A") if
isinstance(source_info, dict) else "N/A"
            print(f"{i}. Title: {article.get('title', 'N/A')}")
            print(f"  Description: {article.get('description', 'N/A')}")
            print(f"  URL: {article.get('url', 'N/A')}")
            print(f"  PublishedAt: {article.get('publishedAt', 'N/A')}")
            print(f"  Source Name: {source_name}\n")
        if len(articles) > 3:
            print("... (only first 3 articles displayed)")

        producer = self.kafka_handler.get_producer()
        if not producer:
            print("Step 1 WARNING: Could not get Kafka producer. Skipping Kafka
production for Step 1.")
        else:
            print(f"Producing basic info for {len(articles)} articles to Kafka
topic '{self.config['kafka']['topic_articles']}'...")
            for article in articles:
                source_info = article.get("source", {})
                source_name = source_info.get("name", "N/A") if
isinstance(source_info, dict) else "N/A"

                article_info = {
                    "title": article.get("title", "N/A"),
                    "description": article.get("description", "N/A"),
                    "url": article.get("url"),
                    "publishedAt": article.get("publishedAt"),
                    "source_name": source_name
                }
                if self.kafka_handler.send_message(producer,
self.config['kafka']['topic_articles'], article_info):
                    produced_count += 1

            self.kafka_handler.flush_producer(producer)
            self.kafka_handler.close_producer(producer)
            print(f"Step 1 Kafka Finished: Produced {produced_count} messages to
'{self.config['kafka']['topic_articles']}'.")

        print(f"Step 1 Finished. Returning {len(articles)} fetched articles with
valid URLs.\n")
        return articles

    def _run_step2_scrape_parallel(self, articles_data: List[Dict[str, Any]]) ->
List[Dict[str, Any]]:
        print("\n--- Step 2: Scrape & Clean Article Content (Parallel Spark Job)
---")
        if not articles_data:
            print("Step 2 Skipped: No valid articles received from Step 1.")
            return []
        if not self.sc:
            print("Step 2 Failed: SparkContext not available.")
            raise RuntimeError("SparkContext not initialized before running Step
2.")

        urls_to_scrape = [article["url"] for article in articles_data]
```

```
        print(f"Extracted {len(urls_to_scrape)} URLs for scraping.")
        if not urls_to_scrape:
            print("Step 2 Finished: No URLs to scrape.")
            return []

        broadcast_config = self.sc.broadcast(self.config['scraping'])

        url_rdd = self.sc.parallelize(urls_to_scrape,
numSlices=len(urls_to_scrape))
        cleaned_content_rdd = url_rdd.flatMap(
            lambda url: ArticleScraper.scrape_and_clean(url,
broadcast_config.value)
        )

        print("Collecting cleaned content from Spark workers...")
        try:
            cleaned_content_list = cleaned_content_rdd.collect()
            print(f"Step 2 Finished: Collected scraped content for
{len(cleaned_content_list)} articles.\n")
            return cleaned_content_list
        except Exception as e:
            print(f"ERROR during Spark collect operation in Step 2: {e}",
file=sys.stderr)
            return []

    def _run_step3_produce_cleaned(self, merged_data: List[Dict[str, Any]]) ->
None:
        print(f"\n--- Step 3: Produce Merged Content to Kafka
('{self.config['kafka']['topic_content']}') ---")
        produced_count = 0
        if not merged_data:
            print("Step 3 Skipped: No merged data available (perhaps scraping
failed or no initial articles).")
            return

        producer = self.kafka_handler.get_producer()
        if not producer:
            print("Step 3 Failed: Could not get Kafka producer.")
            return

        print(f"Producing merged data for {len(merged_data)} articles to Kafka
topic '{self.config['kafka']['topic_content']}'...")
        for item in merged_data:
            if self.kafka_handler.send_message(producer,
self.config['kafka']['topic_content'], item):
                produced_count += 1

        self.kafka_handler.flush_producer(producer)
        self.kafka_handler.close_producer(producer)
        print(f"Step 3 Finished: Produced {produced_count} messages to
'{self.config['kafka']['topic_content']}'.\n")

    def _run_step4_create_dataframe(self, merged_data: List[Dict[str, Any]]) ->
None:
        print("\n--- Step 4: Create Spark DataFrame & Save Output ---")
        if not merged_data:
            print("Step 4 Skipped: No merged data available.")
            return
        if not self.spark:
            print("Step 4 Failed: SparkSession not available.")
```

```
            raise RuntimeError("SparkSession not initialized before running Step
4.")

        try:
            schema = StructType([
                StructField("title", StringType(), True),
                StructField("url", StringType(), True),
                StructField("content", StringType(), True),
                StructField("publishedAt", StringType(), True),
                StructField("source_name", StringType(), True)
            ])

            print("Creating Spark DataFrame from merged data...")
            df: DataFrame = self.spark.createDataFrame(merged_data,
schema=schema)

            print("DataFrame created successfully.")

            output_path = self.config['output']['parquet_path']
            coalesce_partitions = self.config['output']['output_coalesce']
            write_mode = self.config['output']['output_mode']
            print(f"Attempting to save DataFrame to: {output_path} (Mode:
{write_mode}, Format: Parquet)")

            df_writer = df.write.mode(write_mode)

            if coalesce_partitions and isinstance(coalesce_partitions, int) and
coalesce_partitions > 0:
                print(f"Coalescing DataFrame to {coalesce_partitions}
partition(s) before writing.")
                df = df.coalesce(coalesce_partitions)
                df_writer = df.write.mode(write_mode)

            df_writer.parquet(output_path)

            print(f"DataFrame successfully saved as Parquet to: {output_path}")
            print(f"Step 4 Finished.\n")

        except Exception as e:
            print(f"ERROR during DataFrame creation or saving in Step 4: {e}",
file=sys.stderr)

    def _consume_kafka_messages(self) -> None:
        topic_name = self.config['kafka']['topic_content']
        bootstrap_servers = self.config['kafka']['bootstrap_servers']
        group_id = self.config.get('kafka', {}).get('consumer_group_id',
'news_pipeline_consumer')
        consumer_timeout_ms = 5000
        received_count = 0

        try:
            consumer = KafkaConsumer(
                topic_name,
                bootstrap_servers=bootstrap_servers,
                auto_offset_reset='earliest',
                enable_auto_commit=True,
                group_id=group_id,
                value_deserializer=lambda x: json.loads(x.decode('utf-8')),
                consumer_timeout_ms=consumer_timeout_ms
            )
```

```
            print(f"\n--- Kafka Consumer Started for topic '{topic_name}'
(Timeout: {consumer_timeout_ms} ms) ---")
            messages_received = False
            displayed_count = 0
            for record in consumer:
                if displayed_count < 3:
                    print(f"Received message: Partition={record.partition},
Offset={record.offset}")
                    print(f"Key={record.key}")
                    print(f"Value={record.value.get('title', 'N/A')}")
                    print("Processing consumed message...")
                    print("-" * 20)
                    displayed_count += 1
                received_count += 1

            if received_count > 0:
                print(f"Kafka Consumer: Received a total of {received_count}
articles.")
            else:
                print("Kafka Consumer: No messages were available in the topic.")

        except NoBrokersAvailable:
            print(f"Error: Could not connect to Kafka brokers at
{bootstrap_servers}")
        except json.JSONDecodeError:
            print("Error: Could not decode JSON message from Kafka.")
        except KafkaError as e:
            print(f"Kafka Consumer Error: {e}")
        except KeyboardInterrupt:
            print("Kafka consumer stopped by user.")
        finally:
            if 'consumer' in locals() and consumer:
                consumer.close()
                print("Kafka consumer closed.")

    def run(self) -> None:
        print("Starting News Processing Pipeline...")
        try:
            self._start_spark()
            self._prepare_kafka_topics()
            step1_articles = self._run_step1_fetch_and_produce()
            articles_for_step2 = step1_articles if isinstance(step1_articles,
list) else []
            step2_cleaned_data =
self._run_step2_scrape_parallel(articles_for_step2)

            print("\n--- Merging Original Article Metadata with Scraped Content
---")
            if not articles_for_step2 or not step2_cleaned_data:
                print("Merge Skipped: Missing original articles or scraped
data.")
                merged_data = []
            else:
                original_articles_map = {a['url']: a for a in articles_for_step2
if a.get('url')}
                scraped_content_map = {c['url']: c for c in step2_cleaned_data if
c.get('url')}

                merged_data = []
```

```
                processed_urls = set()

                for url, scraped_info in scraped_content_map.items():
                    if url in original_articles_map and url not in
processed_urls:
                        original_info = original_articles_map[url]
                        source_info = original_info.get("source", {})
                        source_name = source_info.get("name", "N/A") if
isinstance(source_info, dict) else "N/A"

                        merged_item = {
                            "title": original_info.get("title",
scraped_info.get("title", "N/A")),
                            "url": url,
                            "content": scraped_info.get("content"),
                            "publishedAt": original_info.get("publishedAt"),
                            "source_name": source_name
                        }

                        if merged_item["content"]:
                            merged_data.append(merged_item)
                            processed_urls.add(url)
                        else:
                            print(f"INFO: Skipping article (URL: {url}) due to
missing scraped content after merge.")
                print(f"Successfully merged metadata with scraped content for
{len(merged_data)} articles.")

            self._run_step3_produce_cleaned(merged_data)
            self._consume_kafka_messages()
            self._run_step4_create_dataframe(merged_data)

            print("\nNews Processing Pipeline Finished Successfully.")

        except Exception as e:
            print(f"FATAL ERROR: An error occurred during pipeline execution:
{e}", file=sys.stderr)

        finally:
            self._stop_spark()
```

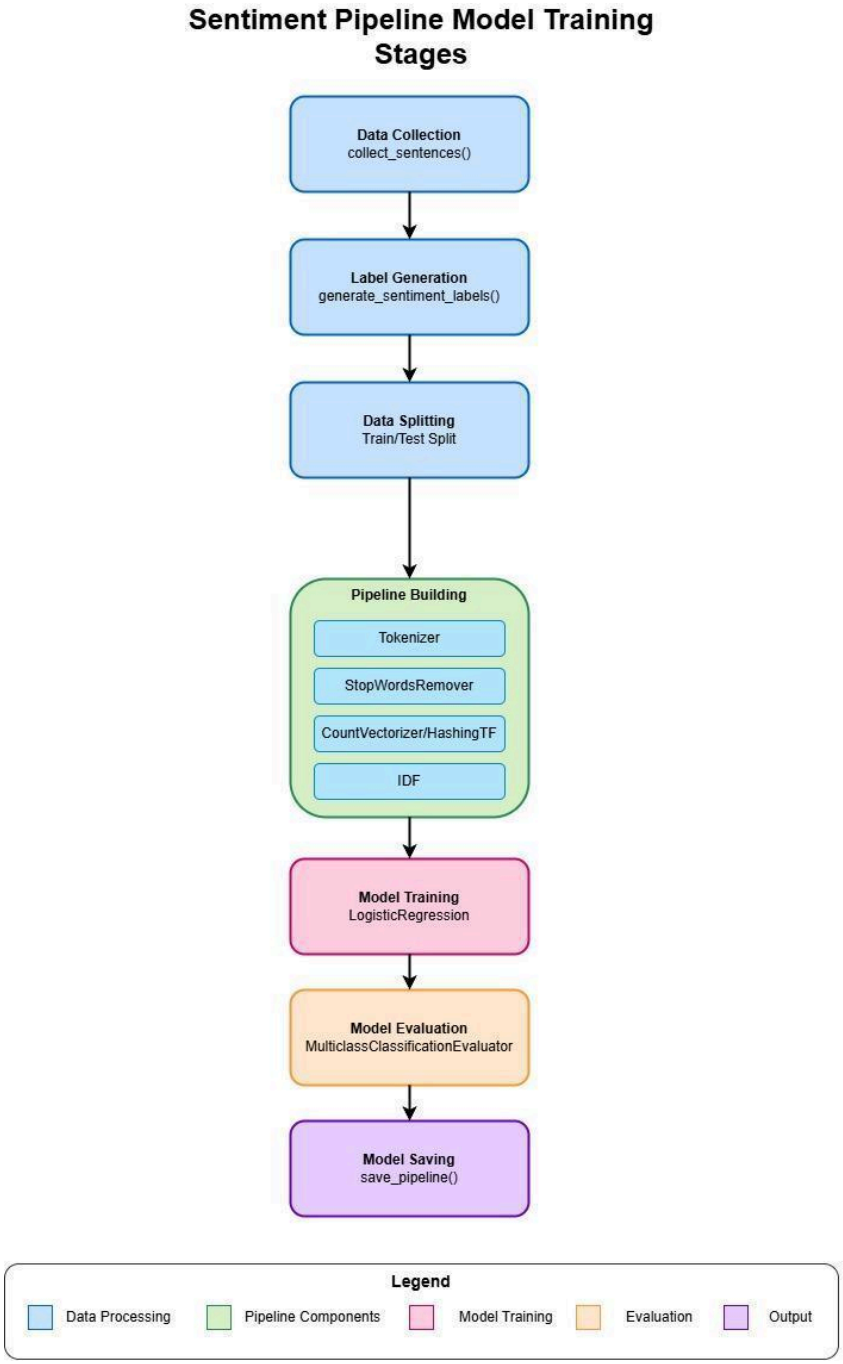
1.5. Data Quantity / EDA

Total raw records	97
Total records after data cleaning	93

```
KafkaProducer connected to ['localhost:9092'].
Producing basic info for 97 articles to Kafka topic 'news-articles'...
Flushing Kafka producer...
Closing Kafka producer.
Step 1 Kafka Finished: Produced 97 messages to 'news-articles'.
Step 1 Finished. Returning 97 fetched articles with valid URLs.


--- Step 2: Scrape & Clean Article Content (Parallel Spark Job) ---
Extracted 97 URLs for scraping.
Collecting cleaned content from Spark workers...
Step 2 Finished: Collected scraped content for 93 articles.


--- Merging Original Article Metadata with Scraped Content ---
Successfully merged metadata with scraped content for 93 articles.
```

2. Model Building

2.1. Model Training Stages Diagram



2.2. List of Python classes:

Name of Python classes	Author
DataValidator_Task2	Wong Kiong Wei
DataEnricher_Task2	Wong Kiong Wei
SentimentPipeline_Task2	Wong Kiong Wei

2.3. Code for Python Classes

Class: DataValidator_Task2
Description: This class validates and performs basic cleaning on a Spark DataFrame containing news article data, checking for missing columns, null/empty values, URL format, content length, and duplicate URLs.

```
# Core Spark Imports
from pyspark.sql import SparkSession, DataFrame
from pyspark.sql.functions import (
    col, when, sum, length, count, regexp_extract, udf, lit, split,
array_contains,
    size, to_date, coalesce, format_string, lpad, explode, isnull
)
from pyspark.sql.types import (
    IntegerType, StringType, ArrayType, StructType, StructField, FloatType,
    DateType, TimestampType
)

# ML and NLP Imports
from pyspark.ml.feature import Tokenizer, StopWordsRemover, CountVectorizer
from pyspark.ml.clustering import LDA
import spacy
from pyspark.sql import functions as F

# Standard Python Libraries
import re
from urllib.parse import urlparse
from datetime import datetime
from collections import OrderedDict
import os


# ================================================
# Class Definition: DataValidator
# ================================================
class DataValidator_Task2:
    """
    Validates the input news articles DataFrame based on predefined rules.
    """
    def __init__(self, spark: SparkSession):
        self.spark = spark
        self.df = None
        self.validation_summary = {}
        print("DataValidator initialized.")

    def load_data(self, path: str) -> 'DataValidator':
        """Loads data from a Parquet file."""
        print(f"--- [Validator] Loading data from: {path} ---")
        try:
            self.df = self.spark.read.parquet(path)
            print(f"Successfully loaded data. Row count: {self.df.count()}")
            self.df.printSchema()
        except Exception as e:
            print(f"ERROR: Failed to load Parquet file from {path}: {e}")
            self.df = None # Ensure df is None if loading fails
```

```
            raise # Re-raise the exception to halt execution if loading fails
        return self

    def run_all_validations(self,
                            required_columns=["url", "content", "title"],
                            potential_date_columns=["publish_date", "date",
"published_at", "timestamp", "creation_date"],
                            url_column="url",
                            content_column="content",
                            min_content_length=20) -> 'DataValidator':
        """Runs all defined validation checks."""
        if self.df is None:
            print("ERROR: DataFrame not loaded. Cannot run validations.")
            return self

        print("\n--- [Validator] Running Data Validation Checks ---")
        self._check_required_columns(required_columns)
        self._check_nulls_or_empty(required_columns + potential_date_columns)
        self._validate_url_format(url_column)
        self._check_content_length(content_column, min_content_length)
        self._check_duplicate_urls(url_column)
        print("--- [Validator] Validation checks completed ---")
        self._print_summary()
        return self

    def _check_required_columns(self, required_columns: list):
        """Checks if essential columns exist in the DataFrame."""
        print("Checking for required columns...")
        missing_cols = [c for c in required_columns if c not in self.df.columns]
        if missing_cols:
            self.validation_summary['missing_required_columns'] = missing_cols
            print(f"WARNING: Missing required columns: {missing_cols}")
        else:
            self.validation_summary['missing_required_columns'] = []
            print("All required columns are present.")

    def _check_nulls_or_empty(self, columns_to_check: list):
        """Checks for null or empty string values in specified columns."""
        print("Checking for null or empty values...")
        null_counts = {}
        existing_cols = [c for c in columns_to_check if c in self.df.columns]
        if existing_cols:
            results = self.df.select([
                sum(when(isnull(c) | (col(c) == ""),
1).otherwise(0)).alias(f"null_empty_{c}")
                for c in existing_cols
            ]).first().asDict()
            null_counts = {col_alias.replace("null_empty_", ""): count for
col_alias, count in results.items()}
            self.validation_summary['null_empty_counts'] = null_counts
            print("Null/Empty counts:", null_counts)
        else:
            print("No columns found to check for nulls/emptiness.")
            self.validation_summary['null_empty_counts'] = {}

    def _validate_url_format(self, url_column: str):
        """Validates the URL format (basic http/https check)."""
        print(f"Validating URL format in column '{url_column}'...")
        if url_column in self.df.columns:
            url_pattern = r"^https?://.+"
```

```
            invalid_count =
self.df.filter(~col(url_column).rlike(url_pattern)).count()
            self.validation_summary['invalid_url_format_count'] = invalid_count
            print(f"Rows with potentially invalid URL format: {invalid_count}")
        else:
            print(f"Skipping URL format validation: Column '{url_column}' not
found.")
            self.validation_summary['invalid_url_format_count'] = 'N/A'

    def _check_content_length(self, content_column: str, min_length: int):
        """Checks if the content length meets a minimum requirement."""
        print(f"Checking content length in column '{content_column}' (min:
{min_length})...")
        if content_column in self.df.columns:
            short_content_count = self.df.filter(
                col(content_column).isNotNull() & (length(col(content_column)) <
min_length)
            ).count()
            self.validation_summary['short_content_count'] = short_content_count
            print(f"Rows with content shorter than {min_length} characters:
{short_content_count}")
        else:
            print(f"Skipping content length check: Column '{content_column}' not
found.")
            self.validation_summary['short_content_count'] = 'N/A'

    def _check_duplicate_urls(self, url_column: str):
        """Checks for duplicate values in the URL column."""
        print(f"Checking for duplicate URLs in column '{url_column}'...")
        if url_column in self.df.columns:
            total_rows = self.df.count()
            distinct_non_null_urls =
self.df.filter(col(url_column).isNotNull()).select(url_column).distinct().count()
            non_null_rows = self.df.filter(col(url_column).isNotNull()).count()
            duplicate_count = non_null_rows - distinct_non_null_urls
            self.validation_summary['duplicate_url_count'] = duplicate_count
            self.validation_summary['distinct_url_count'] =
distinct_non_null_urls
            print(f"Total non-null rows with URL: {non_null_rows}")
            print(f"Distinct non-null URLs: {distinct_non_null_urls}")
            print(f"Duplicate non-null URLs found: {duplicate_count}")
        else:
            print(f"Skipping duplicate URL check: Column '{url_column}' not
found.")
            self.validation_summary['duplicate_url_count'] = 'N/A'
            self.validation_summary['distinct_url_count'] = 'N/A'

    def _print_summary(self):
        """Prints the collected validation summary."""
        print("\n--- Validation Summary ---")
        for key, value in self.validation_summary.items():
            print(f"{key}: {value}")
        print("----------------------------")

    def get_summary(self) -> dict:
        """Returns the validation summary dictionary."""
        return self.validation_summary

    def get_dataframe(self) -> DataFrame:
        """Returns the loaded DataFrame."""
```

```
        return self.df

    def get_clean_dataframe(self, drop_duplicates_url=True,
filter_short_content=True, min_length=20) -> DataFrame:
        """Returns a DataFrame filtered based on validation checks."""
        if self.df is None:
            print("ERROR: Cannot get clean DataFrame, data not loaded.")
            return None

        print("\n--- [Validator] Applying basic cleaning ---")
        clean_df = self.df

        # Filter null/invalid URLs
        if "url" in clean_df.columns:
             url_pattern = r"^https?://.+"
             clean_df = clean_df.filter(col("url").isNotNull() &
col("url").rlike(url_pattern))
             print(f"Rows after filtering null/invalid URLs: {clean_df.count()}")
             if drop_duplicates_url:
                 clean_df = clean_df.dropDuplicates(["url"])
                 print(f"Rows after dropping duplicate URLs: {clean_df.count()}")

        # Filter short content
        if filter_short_content and "content" in clean_df.columns:
             clean_df = clean_df.filter(col("content").isNotNull() &
(length(col("content")) >= min_length))
             print(f"Rows after filtering short content (<{min_length}):
{clean_df.count()}")

        print("--- [Validator] Cleaning finished ---")
        return clean_df
```

Class: DataEnricher_Task2
Description: This class enhances a PySpark DataFrame of news articles by extracting metadata, identifying entities, finding project names, categorizing topics, and splitting the content into clean sentences.

```
from pyspark.sql import SparkSession, DataFrame
from pyspark.sql.functions import (
    col, when, sum, length, count, regexp_extract, udf, lit, split,
array_contains,
    size, to_date, coalesce, format_string, lpad, explode, isnull
)
from pyspark.sql.types import (
    IntegerType, StringType, ArrayType, StructType, StructField, FloatType,
    DateType, TimestampType
)

# ML and NLP Imports
from pyspark.ml.feature import Tokenizer, StopWordsRemover, CountVectorizer
from pyspark.ml.clustering import LDA
import spacy
from pyspark.sql import functions as F

# Standard Python Libraries
import re
from urllib.parse import urlparse
from datetime import datetime
from collections import OrderedDict
import os

# ================================================
# Class Definition: DataEnricher
# ================================================
class DataEnricher_Task2:
    """
    Enriches the news articles DataFrame with metadata, entities, topics, etc.
    """
    def __init__(self, spark: SparkSession):
        self.spark = spark
        self.df = None
        self.nlp = None # To hold the loaded spaCy model
        self._initialize_spacy() # Attempt to load spaCy on init
        print("DataEnricher initialized.")

    def set_dataframe(self, df: DataFrame) -> 'DataEnricher_Task2':
        """Sets the DataFrame to be enriched."""
        self.df = df
        if self.df:
            print(f"--- [Enricher] DataFrame set. Row count: {self.df.count()}
---")
        else:
            print("WARNING: [Enricher] Received an empty DataFrame.")
        return self

    def _initialize_spacy(self):
        """Loads the spaCy model."""
        if self.nlp is None:
            try:
                # Make sure you have the model downloaded:
```

```
                # python -m spacy download en_core_web_sm
                self.nlp = spacy.load("en_core_web_sm")
                print("spaCy model 'en_core_web_sm' loaded successfully.")
            except ImportError:
                print("WARNING: spaCy library not found. Cannot perform spaCy
entity extraction. Install with: pip install spacy")
                self.nlp = None
            except OSError:
                 print("WARNING: spaCy model 'en_core_web_sm' not found. Download
it: python -m spacy download en_core_web_sm")
                 self.nlp = None
            except Exception as e:
                print(f"WARNING: An unexpected error occurred loading spaCy:
{e}")
                self.nlp = None

    def run_all_enrichments(self,
                            url_col="url", content_col="content",
                            num_topics=5, lda_max_iter=15, lda_vocab_size=5000,
                            final_columns_order=None) -> DataFrame:
        """Runs all enrichment steps sequentially."""
        if self.df is None:
            print("ERROR: DataFrame not set in Enricher. Cannot run
enrichments.")
            return None

        print("\n--- [Enricher] Running Data Enrichment Steps ---")
        self.enrich_metadata(url_col=url_col, content_col=content_col)
        self.enrich_entities(content_col=content_col)
        self.enrich_project_names(content_col=content_col)
        self.enrich_topics(content_col=content_col, num_topics=num_topics,
max_iter=lda_max_iter, vocab_size=lda_vocab_size)
        self.enrich_sentence_processing(content_col=content_col)

        print("--- [Enricher] Enrichment steps completed ---")
        # Select final columns if order is specified
        if final_columns_order:
            self.df = self._select_final_columns(self.df, final_columns_order)

        return self.df

    def enrich_metadata(self, url_col="url", content_col="content",
                        potential_date_cols=["publish_date", "date",
"published_at", "timestamp", "creation_date"],
                        target_date_col="publication_date"):
        """Adds metadata columns: source, domain, date, category, word count."""
        print("Enriching metadata...")
        if self.df is None: return


        # Add Source Type based on URL domain pattern
        if url_col in self.df.columns:
            # Define the UDF to extract source_type from URL
            source_type_udf = udf(self._extract_source_type_from_url,
StringType())

            # Apply the UDF to create source_type column
            self.df = self.df.withColumn("source", source_type_udf(col(url_col)))

            # Add Domain for reference
```

```
            domain_udf = self._get_domain_udf()
            self.df = self.df.withColumn("source_domain",
domain_udf(col(url_col)))
        else:
            self.df = self.df.withColumn("source", lit("Unknown"))
            self.df = self.df.withColumn("source_domain",
lit(None).cast(StringType()))

        # Standardize Date
        self.df = self._standardize_date(url_col, potential_date_cols,
target_date_col)

        # Add Category based on content
        if content_col in self.df.columns:
            self.df = self.df.withColumn(
                "category",

when(col(content_col).rlike("(?i)financ|profit|revenue|earnings|dividend|investme
nt|stock|market|shares"), "Financial")

.when(col(content_col).rlike("(?i)sustainab|green|environment|carbon|climate|emis
sion|esg|renewable"), "Sustainability")

.when(col(content_col).rlike("(?i)gas|oil|drilling|exploration|discovery|field|re
serves|lng|upstream|downstream|refinery"), "Exploration & Production")

.when(col(content_col).rlike("(?i)partner|deal|agreement|contract|acquisition|mer
ger|joint venture|mou"), "Business Deals")

.when(col(content_col).rlike("(?i)technolog|digital|innovation|research|developme
nt|ai|platform"), "Technology & Innovation")
                .otherwise("General")
            )
        else:
             self.df = self.df.withColumn("category", lit("Unknown"))

        # Calculate Word Count
        if content_col in self.df.columns:
            self.df = self.df.withColumn(
                "word_count",
                when(col(content_col).isNotNull(), size(split(col(content_col),
"\\s+"))).otherwise(0)
            )
        else:
            self.df = self.df.withColumn("word_count",
lit(0).cast(IntegerType()))
        print("Metadata enrichment done (source, domain, date, category,
word_count).")


    @staticmethod
    def _extract_source_type_from_url(url):
        """Extracts source_type from URL based on domain patterns."""
        if not url or not isinstance(url, str):
            return "Unknown"

        try:
            # Parse the URL to get the domain
            domain = urlparse(url).netloc.lower()
```

```
            # Remove common prefixes and TLDs
            clean_domain =
re.sub(r'www\.|\.com|\.org|\.net|\.co\.[a-z]{2}|\.gov|\.edu', '', domain)

            # Identify source type based on domain keywords
            if re.search(r'reuters|bloomberg|cnbc|wsj|nytimes|ft|forbes',
clean_domain):
                return "Major International News"
            elif re.search(r'thestar|nst|theedge|malaymail|bernama',
clean_domain):
                return "Malaysian News"
            elif re.search(r'finance|investor|market|business|investing|money',
clean_domain):
                return "Financial News"
            elif re.search(r'energy|oil|gas|petroleum|offshore|rigzone|upstream',
clean_domain):
                return "Energy News"
            elif re.search(r'blog|medium', clean_domain):
                return "Blog"
            elif re.search(r'gov|government', clean_domain):
                return "Government"
            elif re.search(r'edu|university|college|school', clean_domain):
                return "Educational"
            elif re.search(r'corp|company|inc|ltd|llc', clean_domain):
                return "Corporate"
            else:
                # If no pattern matches, use the domain name as a fallback
                main_part = clean_domain.split('.')[0]
                if len(main_part) > 2:
                    return f"Other - {main_part.capitalize()}"
                else:
                    return "Other"
        except:
            return "Unknown"


    @staticmethod
    def _get_domain_udf():
        """Creates and returns the UDF for extracting domain from URL."""
        def get_domain(url):
            try:
                if url and isinstance(url, str) and url.startswith(('http://',
'https://')):
                    return urlparse(url).netloc
                return None
            except Exception: return None
        return udf(get_domain, StringType())

    def _standardize_date(self, url_col, potential_date_cols, target_date_col):
        """Internal helper to extract and standardize the publication date."""
        print(f"Attempting to standardize date into '{target_date_col}'...")
        if self.df is None: return None

        current_df = self.df
        found_date_source = False

        # Strategy 1: Check existing potential date columns
        possible_source_cols = [c for c in potential_date_cols if c in
current_df.columns]
        if possible_source_cols:
```

```
            source_col = possible_source_cols[0]
            print(f"Using source column '{source_col}' for date.")
            source_col_type = dict(current_df.dtypes)[source_col]
            if "string" in source_col_type.lower():
                current_df = current_df.withColumn(
                    target_date_col,
                    coalesce( # Try multiple formats
                        to_date(col(source_col), "yyyy-MM-dd HH:mm:ss"),
to_date(col(source_col), "yyyy-MM-dd'T'HH:mm:ss"),
                        to_date(col(source_col), "yyyy-MM-dd"),
to_date(col(source_col), "MM/dd/yyyy"),
                        to_date(col(source_col), "dd-MMM-yyyy"),
to_date(col(source_col), "yyyyMMdd"),
                        to_date(col(source_col), "MM-dd-yyyy"),
to_date(col(source_col), "dd/MM/yyyy"),
                        to_date(col(source_col), "yyyy/MM/dd"),
to_date(col(source_col), "dd.MM.yyyy"),
                        to_date(col(source_col), "MMM dd, yyyy"),
to_date(col(source_col), "dd MMM yyyy")
                    )
                )
                found_date_source = True
            elif "date" in source_col_type.lower():
                current_df = current_df.withColumn(target_date_col,
col(source_col))
                found_date_source = True
            elif "timestamp" in source_col_type.lower():
                current_df = current_df.withColumn(target_date_col,
to_date(col(source_col)))
                found_date_source = True

        # Strategy 2: Extract from URL if not found or invalid from Strategy 1
        if url_col in current_df.columns:
             url_date_pattern = r"[/_-](\d{4})[/_-](\d{1,2})[/_-](\d{1,2})[/_-]"
             url_date_pattern_alt = r"[/_-](\d{4})(\d{2})(\d{2})[/_-]"
             df_temp = current_df.withColumn("url_year",
regexp_extract(col(url_col), url_date_pattern, 1)) \
                                 .withColumn("url_month",
regexp_extract(col(url_col), url_date_pattern, 2)) \
                                 .withColumn("url_day",
regexp_extract(col(url_col), url_date_pattern, 3)) \
                                 .withColumn("url_year_alt",
regexp_extract(col(url_col), url_date_pattern_alt, 1)) \
                                 .withColumn("url_month_alt",
regexp_extract(col(url_col), url_date_pattern_alt, 2)) \
                                 .withColumn("url_day_alt",
regexp_extract(col(url_col), url_date_pattern_alt, 3))

             df_temp = df_temp.withColumn(
                "extracted_date_str",
                when(col("url_year") != "", format_string("%s-%s-%s",
col("url_year"), lpad(col("url_month"), 2, '0'), lpad(col("url_day"), 2, '0')))
                .when(col("url_year_alt") != "", format_string("%s-%s-%s",
col("url_year_alt"), lpad(col("url_month_alt"), 2, '0'), lpad(col("url_day_alt"),
2, '0')))
                .otherwise(None)
             )
             url_extracted_date = to_date(col("extracted_date_str"),
"yyyy-MM-dd")
```

```
            # Add/update the target column: Use URL date if existing date is
null OR if no date col was found initially
            if target_date_col in current_df.columns:
                current_df = df_temp.withColumn(target_date_col,
when(col(target_date_col).isNull(),
url_extracted_date).otherwise(col(target_date_col)))
            elif not found_date_source: # Only add if strategy 1 didn't yield a
column
                 current_df = df_temp.withColumn(target_date_col,
url_extracted_date)
            else: # Keep the date from strategy 1 if it exists
                current_df = df_temp # Keep df_temp structure but don't
overwrite date

            current_df = current_df.drop("url_year", "url_month", "url_day",
"url_year_alt", "url_month_alt", "url_day_alt", "extracted_date_str")
            found_date_source = True # Mark as attempted/found

        # Ensure column exists even if no date was found
        if not found_date_source and target_date_col not in current_df.columns:
             current_df = current_df.withColumn(target_date_col,
lit(None).cast(DateType()))
             print(f"Warning: Could not determine date. Column
'{target_date_col}' created with nulls.")
        elif not found_date_source:
             print(f"Warning: Could not determine date for many rows in
'{target_date_col}'.")

        return current_df


    def enrich_entities(self, content_col="content"):
        """Extracts named entities using spaCy."""
        print("Enriching entities using spaCy...")
        if self.df is None: return
        if content_col not in self.df.columns:
            print(f"Skipping entity extraction: Column '{content_col}' not
found.")
            self._add_empty_entity_columns()
            return
        if self.nlp is None:
            print("Skipping entity extraction: spaCy model not loaded.")
            self._add_empty_entity_columns()
            return

        # Define UDF schema and function
        entity_schema = StructType([
            StructField("people", ArrayType(StringType()), True),
StructField("organizations", ArrayType(StringType()), True),
            StructField("locations", ArrayType(StringType()), True),
StructField("dates", ArrayType(StringType()), True),
            StructField("money", ArrayType(StringType()), True)
        ])
        extract_entities_udf = self._get_extract_entities_udf(self.nlp,
entity_schema) # Pass loaded model

        # Apply UDF
        self.df = self.df.withColumn("extracted_entities",
extract_entities_udf(col(content_col)))
```

```
        # Flatten struct into columns
        self.df = self.df.withColumn("people_mentioned",
col("extracted_entities.people")) \
                         .withColumn("organizations_mentioned",
col("extracted_entities.organizations")) \
                         .withColumn("locations_mentioned",
col("extracted_entities.locations")) \
                         .withColumn("dates_mentioned",
col("extracted_entities.dates")) \
                         .withColumn("financial_figures",
col("extracted_entities.money")) \
                         .drop("extracted_entities")
        print("Entity extraction done.")


    @staticmethod
    def _get_extract_entities_udf(nlp_model, schema):
         """Creates the UDF for spaCy entity extraction."""
         if nlp_model is None: # Return a dummy UDF if spacy failed
             def dummy_extract(text):
                 return {"people": [], "organizations": [], "locations": [],
"dates": [], "money": []}
             return udf(dummy_extract, schema)

         # --- Actual UDF function using the passed nlp_model ---
         def extract_entities(text):
             # This function now closes over the nlp_model variable passed to
_get_extract_entities_udf
             entities = {"people": [], "organizations": [], "locations": [],
"dates": [], "money": []}
             if not text or not isinstance(text, str): return entities
             try:
                 doc = nlp_model(text[:100000]) # Limit text size
                 for ent in doc.ents:
                     ent_text = ent.text.strip()
                     if len(ent_text) < 3 or ent_text.isspace(): continue
                     label = ent.label_
                     if label == "PERSON" and len(ent_text.split()) <= 4:
entities["people"].append(ent_text)
                     elif label == "ORG" and "petronas" not in ent_text.lower():
entities["organizations"].append(ent_text)
                     elif label in ["GPE", "LOC"]:
entities["locations"].append(ent_text)
                     elif label == "DATE": entities["dates"].append(ent_text)
                     elif label == "MONEY": entities["money"].append(ent_text)
                 for key in entities: entities[key] =
list(OrderedDict.fromkeys(entities[key]))[:10]
             except Exception as e: pass # Log errors if needed, but don't fail
the UDF
             return entities
         # --- End of actual UDF function ---

         return udf(extract_entities, schema)

    def _add_empty_entity_columns(self):
         """Adds empty array columns if entity extraction is skipped."""
         if self.df is None: return
         entity_cols = ["people_mentioned", "organizations_mentioned",
"locations_mentioned", "dates_mentioned", "financial_figures"]
         for col_name in entity_cols:
```

```
             if col_name not in self.df.columns:
                 self.df = self.df.withColumn(col_name,
lit(None).cast(ArrayType(StringType())))

    def enrich_project_names(self, content_col="content"):
        """Extracts project names using regular expressions."""
        print("Enriching project names using regex...")
        if self.df is None: return
        if content_col not in self.df.columns:
            print(f"Skipping project name extraction: Column '{content_col}' not
found.")
            self.df = self.df.withColumn("project_names",
lit(None).cast(ArrayType(StringType())))
            return

        project_pattern =
r"(?i)(?:Project|Basin|Field|Platform|Terminal|Plant|Block)\s+([A-Z][-a-zA-Z0-9\s
]*[a-zA-Z0-9])"
        extract_projects_udf = udf(
            lambda text: list(OrderedDict.fromkeys([match.strip() for match in
re.findall(project_pattern, text)])) if text else [],
            ArrayType(StringType())
        )
        self.df = self.df.withColumn("project_names",
extract_projects_udf(col(content_col)))
        print("Project name extraction done.")


    def enrich_topics(self, content_col="content", num_topics=5, max_iter=15,
vocab_size=5000, min_df=5):
        """Performs LDA topic modeling."""
        print("Enriching topics using LDA...")
        if self.df is None: return
        if content_col not in self.df.columns:
            print(f"Skipping topic modeling: Column '{content_col}' not found.")
            self._add_empty_topic_columns()
            return

        # Prepare data for LDA
        df_for_lda = self.df.select("url", content_col).fillna({content_col: ''})
# Use URL as identifier
        tokenizer = Tokenizer(inputCol=content_col, outputCol="tokens")
        df_tokens = tokenizer.transform(df_for_lda)
        custom_stopwords = ["petronas", "said", "also", "year", "company",
"group", "malaysia", "kuala", "lumpur", "ringgit", "rm", "mln", "bln", "pct",
"news", "report", "update", "inc", "bhd"]
        remover = StopWordsRemover(inputCol="tokens",
outputCol="filtered_tokens")
        remover.setStopWords(StopWordsRemover.loadDefaultStopWords("english") +
custom_stopwords)
        df_no_stop = remover.transform(df_tokens)
        vectorizer = CountVectorizer(inputCol="filtered_tokens",
outputCol="features", vocabSize=vocab_size, minDF=min_df)
        try:
            cv_model = vectorizer.fit(df_no_stop)
            df_features = cv_model.transform(df_no_stop)

            # Train LDA
            lda = LDA(k=num_topics, maxIter=max_iter, featuresCol="features",
seed=42)
```

```
            lda_model = lda.fit(df_features)
            df_with_topics = lda_model.transform(df_features)

            # Process results
            dominant_topic_udf = udf(lambda dist: int(max(enumerate(dist),
key=lambda x: x[1])[0]) if dist else None, IntegerType())
            df_with_topics = df_with_topics.withColumn("dominant_topic",
dominant_topic_udf(col("topicDistribution")))

            # Create topic labels
            vocab = cv_model.vocabulary
            topicDescDF = lda_model.describeTopics(maxTermsPerTopic=5)
            generic_stop_words =
set(StopWordsRemover.loadDefaultStopWords("english") + custom_stopwords)
            def indices_to_words(termIndices):
                keywords = [vocab[i] for i in termIndices if vocab[i].lower() not
in generic_stop_words and len(vocab[i]) > 2]
                return ", ".join(keywords[:3]) if keywords else "Unknown Topic"
            indices_to_words_udf = udf(indices_to_words, StringType())
            topicDescDF = topicDescDF.withColumn("topic_keywords",
indices_to_words_udf(col("termIndices")))
            topic_mapping = {row['topic']: row['topic_keywords'] for row in
topicDescDF.select("topic", "topic_keywords").collect()}
            topic_mapping_bc = self.spark.sparkContext.broadcast(topic_mapping)
            map_topic_label_udf = udf(lambda idx: topic_mapping_bc.value.get(idx,
f"Unknown Topic {idx}") if idx is not None else "N/A", StringType())
            df_with_topics = df_with_topics.withColumn("topic_label",
map_topic_label_udf(col("dominant_topic")))

            # Join results back (selecting only necessary topic columns)
            topic_results = df_with_topics.select("url", "dominant_topic",
"topic_label") # "topicDistribution" removed - too large
            self.df = self.df.join(topic_results, on="url", how="left")
            print("Topic modeling enrichment done.")

        except Exception as e:
            print(f"ERROR during topic modeling: {e}. Skipping topic
enrichment.")
            self._add_empty_topic_columns()


    def _add_empty_topic_columns(self):
        """Adds empty topic columns if LDA fails or is skipped."""
        if self.df is None: return
        if "dominant_topic" not in self.df.columns:
            self.df = self.df.withColumn("dominant_topic",
lit(None).cast(IntegerType()))
        if "topic_label" not in self.df.columns:
            self.df = self.df.withColumn("topic_label",
lit("N/A").cast(StringType()))


    def enrich_sentence_processing(self, content_col="content"):
        """Processes content into cleaned sentences."""
        print("Processing content into sentences...")
        if self.df is None: return
        if content_col not in self.df.columns:
            print(f"Skipping sentence processing: Column '{content_col}' not
found.")
            self.df = self.df.withColumn("processed_sentences",
```

```
lit(None).cast(ArrayType(StringType())))
            return

        # UDF for sentence processing
        process_content_udf = self._get_process_content_udf()
        self.df = self.df.withColumn("processed_sentences",
process_content_udf(col(content_col)))


        # UDF for sentence processing
        process_content_udf = self._get_process_content_udf()

        self.df = self.df.withColumn("processed_sentences",
process_content_udf(col(content_col)))
        self.df.show()
       # Step 4: explode into new rows but keep _all_ other columns
        self.df = self.df.withColumn("processed_sentence",
explode(col("processed_sentences")))
        # (optional) drop the array if you only want the flat column
        self.df = self.df.drop("processed_sentences")




    @staticmethod
    def _get_process_content_udf():
         """Creates the UDF for sentence processing."""
         def process_content_sentences(input_text):
             # ... (Sentence splitting, cleaning, filtering logic from previous
example) ...
             if input_text is None or not isinstance(input_text, str): return []
             sentences = re.split(r'[.?!]\s+|\n+', input_text)
             sentences = [s.strip() for s in sentences if s and not s.isspace()]
             unique_sentences = list(OrderedDict.fromkeys(sentences))
             final_sentences = []
             for sentence in unique_sentences:
                 words = sentence.split()
                 words_no_digits = [word for word in words if not
any(char.isdigit() for char in word)]
                 cleaned_sentence = ' '.join(words_no_digits)
                 if len(cleaned_sentence.split()) >= 3:
                     final_sentences.append(cleaned_sentence)
             return final_sentences
         return udf(process_content_sentences, ArrayType(StringType()))

    @staticmethod
    def _select_final_columns(df, columns_order):
        """Selects and orders columns based on the provided list."""
        print("Selecting and ordering final columns...")
        existing_columns = [c for c in columns_order if c in df.columns]
        missing_columns = [c for c in columns_order if c not in df.columns]
        if missing_columns:
            print(f"Warning: Requested final columns not found in DataFrame:
{missing_columns}")
        print(f"Final selected columns: {existing_columns}")
        return df.select(existing_columns)
```

Class: SentimentPipeline_Task2
Description:This class automates the process of collecting text data from a Spark DataFrame, generating sentiment labels using a Hugging Face model, training a Spark ML sentiment analysis pipeline (TF-IDF or HashingTF with Logistic Regression), evaluating its performance on training and testing data, and saving the trained pipeline.

```
################################################################################
# Sentiment Pipeline Class Using Spark ML with Train/Test Evaluation
################################################################################

import os
import numpy as np
from pyspark.ml import Pipeline
from pyspark.sql import SparkSession
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, IDF, Tokenizer, StopWordsRemover,
CountVectorizer

class SentimentPipeline_Task2:
    """
    Encapsulates the full pipeline process using Spark ML:
      - Collect sentences from a Spark DataFrame.
      - Generate numerical sentiment labels using a Hugging Face pipeline.
      - Split into train/test sets.
      - Build, train, and evaluate a Spark ML pipeline (TF-IDF + Logistic
Regression).
      - Save the trained pipeline model.
    """
    def __init__(self, spark_df=None, column_name="processed_sentence",

pipeline_path="file:///home/student/de-prj/sentiment_pipeline_spark",
                 train_ratio=0.8, seed=42):
        self.spark_df = spark_df
        self.column_name = column_name
        self.pipeline_path = pipeline_path
        self.train_ratio = train_ratio
        self.seed = seed

        self.sentences = []
        self.labels = []
        self.pipeline_model = None

    def collect_sentences(self):
        if self.spark_df is None:
            raise ValueError("No Spark DataFrame provided. Please set spark_df
before collecting sentences.")
        self.sentences =
self.spark_df.select(self.column_name).rdd.flatMap(lambda x: x).collect()
        print(f"Collected {len(self.sentences)} sentences.")
        if not self.sentences:
            raise ValueError("No sentences collected.")
        return self.sentences

    def generate_sentiment_labels(self):
        from transformers import pipeline as hf_pipeline
        sentiment_pipe = hf_pipeline("text-classification",
model="tabularisai/multilingual-sentiment-analysis")
```

```
        self.labels = []
        for sentence in self.sentences:
            try:
                result = sentiment_pipe(sentence)[0]
                mapping = {
                    'very negative': 1,
                    'negative': 2,
                    'neutral': 3,
                    'positive': 4,
                    'very positive': 5
                }
                self.labels.append(mapping.get(result['label'].lower(), 3))
            except Exception:
                self.labels.append(3)
        counts = dict(zip(*np.unique(self.labels, return_counts=True)))
        print("Label distribution:", counts)
        return self.labels

    def build_and_evaluate_pipeline(self):
        if not self.sentences or not self.labels:
            raise ValueError("Sentences or labels missing. Run collect_sentences
and generate_sentiment_labels first.")

        spark = SparkSession.builder.getOrCreate()
        data = list(zip(self.sentences, self.labels))
        df = spark.createDataFrame(data, schema=[self.column_name,
"label"]).cache()

        train_df, test_df = df.randomSplit([self.train_ratio, 1 -
self.train_ratio], seed=self.seed)
        print(f"Train set: {train_df.count()} rows, Test set: {test_df.count()}
rows.")

        tokenizer = Tokenizer(inputCol=self.column_name, outputCol="words")
        remover = StopWordsRemover(inputCol="words", outputCol="filtered_words")
        try:
            cv = CountVectorizer(inputCol="filtered_words", outputCol="tf",
vocabSize=10000, minDF=1.0)
            idf = IDF(inputCol="tf", outputCol="features", minDocFreq=1)
            pipeline_stages = [tokenizer, remover, cv, idf]
        except Exception:
            print("CountVectorizer failed, falling back to HashingTF...")
            hashingTF = HashingTF(inputCol="filtered_words", outputCol="tf",
numFeatures=100)
            idf = IDF(inputCol="tf", outputCol="features", minDocFreq=1)
            pipeline_stages = [tokenizer, remover, hashingTF, idf]

        lr = LogisticRegression(featuresCol="features", labelCol="label",
maxIter=1000)
        complete_pipeline = Pipeline(stages=pipeline_stages + [lr])

        print("Fitting pipeline on training data...")
        self.pipeline_model = complete_pipeline.fit(train_df)

        evaluator = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="accuracy")
        train_acc = evaluator.evaluate(self.pipeline_model.transform(train_df))
        test_acc = evaluator.evaluate(self.pipeline_model.transform(test_df))
        print(f"Training Accuracy: {train_acc:.4f}")
        print(f"Test Accuracy: {test_acc:.4f}")
```

```
        return {"model": self.pipeline_model, "train_accuracy": train_acc,
"test_accuracy": test_acc}

    def save_pipeline(self):
        if self.pipeline_model is None:
            raise ValueError("No trained pipeline to save. Please run
build_and_evaluate_pipeline first.")
        self.pipeline_model.write().overwrite().save(self.pipeline_path)
        print(f"Pipeline saved to '{self.pipeline_path}'")

    def run(self):
        self.collect_sentences()
        self.generate_sentiment_labels()
        results = self.build_and_evaluate_pipeline()
        self.save_pipeline()
        return results
```

3. Real-time Sentiment Analysis

3.1. Workflow Diagram for Real-time Sentiment Analysis

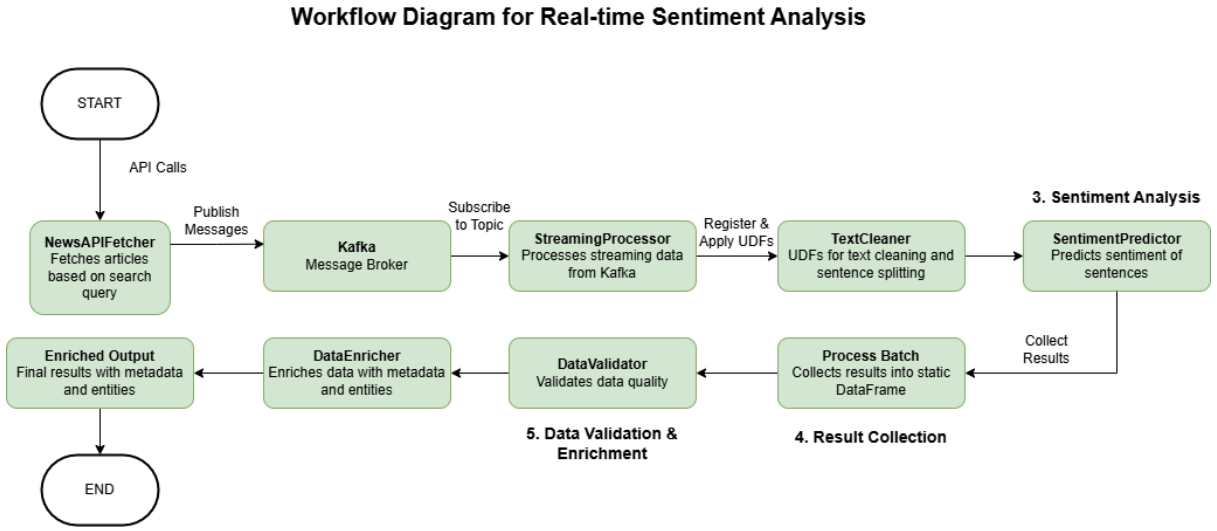
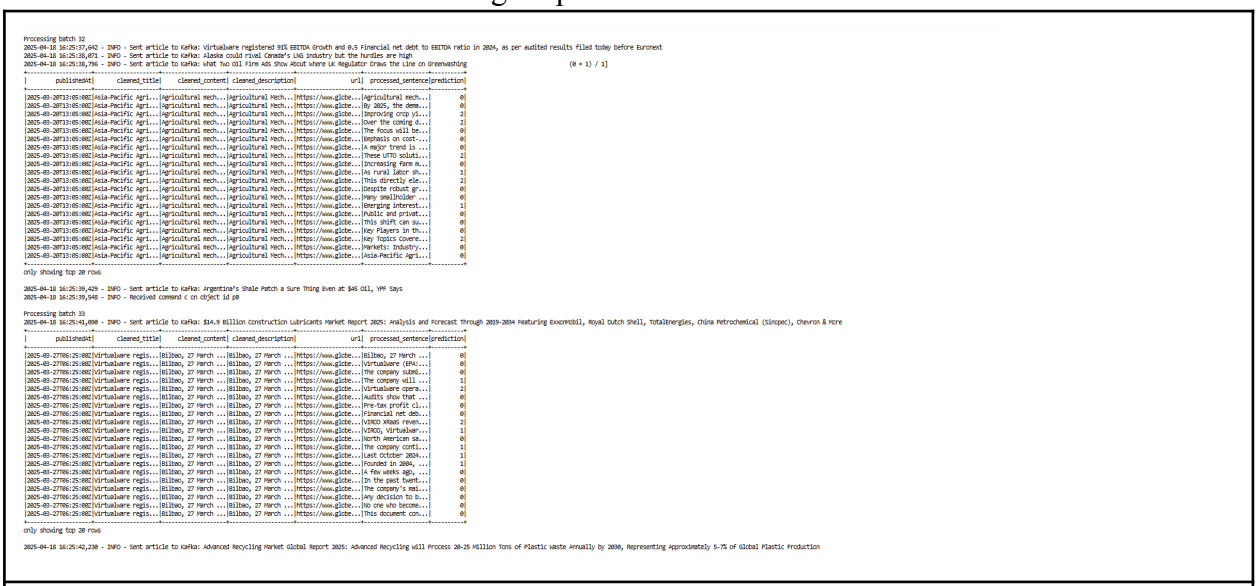


Figure 3.1: Workflow Diagram for Real-time Sentiment Analysis

3.2. Structured Streaming

Use case:	The primary use case for the structured streaming implementation in this project is real-time sentiment analysis of news articles. The system will ingest live data from NewsAPI, stream it through Kafka for real-time processing, and then perform sentiment analysis on each article's content as the data flows in. The sentiment analysis will be applied dynamically, enabling the system to provide real-time insights into public sentiment on various topics covered in the articles.
Source:	The streaming data will be sourced from Kafka topics, where it is called “news-article”, where each message represents a new article or data point to be processed.
Sink:	The processed dataset, including sentiment analysis labels, as well as metadata for the news articles, will be retained in Parquet format. This chosen data format will support fast querying alongside allowing for any future activities, including further analysis, or reporting, to be carried out within Task 4 or Task 5 of the project.
End user(s):	The system is designed to support different types of users: Data analysts will leverage the outputs from real-time sentiment analysis to track public sentiment and identify developing trends within news stories. Researchers will be able to utilize sentiment information for research questions related to public opinion, social processes, and other analytical issues.

Screenshot of Structured Streaming output:



```
Processing batch 32
2025-04-18 16:25:37,642 - INFO - Sent article to Kafka: Virtualware registered 91% EBITDA Growth and 0.5 Financial net debt to EBITDA ratio in 2024, as per audited results filed today before Euronext
2025-04-18 16:25:38,073 - INFO - Sent article to Kafka: Alaska could rival Canada's LNG industry but the hurdles are high
2025-04-18 16:25:38,796 - INFO - Sent article to Kafka: What Two Oil Firm Ads Show About Where UK Regulator Draws the Line on Greenwashing                    (0 + 1) / 1]
+--------------------+--------------+---------------+-------------------+--------------------+--------------------+----------+
|         publishedAt|   cleaned_title|cleaned_content|cleaned_description|                 url|  processed_sentence|prediction|
+--------------------+--------------+---------------+-------------------+--------------------+--------------------+----------+
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Agricultural mech...|         0|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|By 2025, the demo...|         0|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Improving crop yi...|         2|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Over the coming d...|         2|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|The focus will be...|         0|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Emphasis on cost-...|         0|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|A major trend is ...|         0|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|These UTTO soluti...|         2|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Increasing farm m...|         0|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|As rural labor sh...|         1|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|This directly ele...|         2|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Despite robust gr...|         0|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Many smallholder ...|         0|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Emerging interest...|         1|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Public and privat...|         0|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|This shift can su...|         0|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Key Players in th...|         0|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Key Topics Covere...|         1|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Markets: Industry...|         0|
|2025-03-20T13:05:00Z|Asia-Pacific Agri...|Agricultural mech...|Agricultural Mech...|https://www.globe...|Asia-Pacific Agri...|         0|
+--------------------+--------------+---------------+-------------------+--------------------+--------------------+----------+
only showing top 20 rows

2025-04-18 16:25:39,429 - INFO - Sent article to Kafka: Argentina's Shale Patch a Sure Thing Even at $45 Oil, YPF Says
2025-04-18 16:25:39,548 - INFO - Received command c on object id p0

Processing batch 33
2025-04-18 16:25:41,090 - INFO - Sent article to Kafka: $14.9 Billion Construction Lubricants Market Report 2025: Analysis and Forecast Through 2019-2024 Featuring ExxonMobil, Royal Dutch Shell, TotalEnergies, China Petrochemical (Sinopec), Chevron & More
+--------------------+--------------+---------------+-------------------+--------------------+--------------------+----------+
|         publishedAt|   cleaned_title|cleaned_content|cleaned_description|                 url|  processed_sentence|prediction|
+--------------------+--------------+---------------+-------------------+--------------------+--------------------+----------+
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|Bilbao, 27 March ...|         0|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|Virtualware (EPA:...|         0|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|The company submi...|         0|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|The company will ...|         1|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|Virtualware opera...|         2|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|Audits show that ...|         0|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|Pre-tax profit cl...|         0|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|Financial net deb...|         0|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|VIRCO XMAS reven...|         2|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|VIRCO, Virtualwar...|         1|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|North American sa...|         0|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|The company cont...|         1|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|Last October 2024...|         1|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|Founded in 2004, ...|         1|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|A few weeks ago, ...|         0|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|In the past twent...|         0|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|The company's ensi...|         0|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|Any decision to b...|         0|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|No one who become...|         0|
|2025-03-27T06:25:00Z|Virtualware regis...|Bilbao, 27 March ...|Bilbao, 27 March ...|https://www.globe...|This document con...|         0|
+--------------------+--------------+---------------+-------------------+--------------------+--------------------+----------+
only showing top 20 rows

2025-04-18 16:25:42,230 - INFO - Sent article to Kafka: Advanced Recycling Market Global Report 2025: Advanced Recycling will Process 20-25 Million Tons of Plastic Waste Annually by 2030, Representing Approximately 5-7% of Global Plastic Production
```

```
Sentiment_Result          | 0
-RECORD 2-----------------------------------
publishedAt               | 2025-04-01T05:29:00Z
url                       | https://www.chann...
cleaned_title             | 33 injured in hug...
cleaned_description       | The flames report...
source                    | Other
source_domain             | www.channelnewsas...
category                  | Sustainability
word_count                | 865
people_mentioned          | [Dzulkefly Ahmad,...
organizations_mentioned   | [Cyberjaya Hospit...
locations_mentioned       | [Puchong, Putra H...
project_names             | []
financial_figures         | []
dates_mentioned           | [Tuesday, a year,...
sentence                  | A Bernama survey ...
Sentiment_Result          | 0
-RECORD 3-----------------------------------
publishedAt               | 2025-04-02T12:15:09Z
url                       | https://bringatra...
cleaned_title             | 2023 BAC Mono X
cleaned_description       | This 2023 BAC Mon...
source                    | Other
source_domain             | bringatrailer.com
category                  | Sustainability
word_count                | 648
people_mentioned          | [Petronas Green, ...
organizations_mentioned   | [Metallic Liquid ...
locations_mentioned       | [Carfax]
project_names             | []
financial_figures         | []
dates_mentioned           | [August 2023, the...
sentence                  | A California BAR ...
Sentiment_Result          | 0
```

3.3. List of Python classes:

Name of Python classes	Author
config	Lee Yen Han
DataEnricher	Lee Yen Han
DataEnricherWrapper	Lee Yen Han
DataValidator	Lee Yen Han
NewsAPIFetcher	Lee Yen Han
SchemaDefinitions	Lee Yen Han
SentimentPredictor	Lee Yen Han
SparkSessionManager	Lee Yen Han
StreamingProcessor	Lee Yen Han
TextCleaner	Lee Yen Han

3.4. Code for Python Classes

Class: config
Description: This class is responsible for configuring and managing the settings for the real-time sentiment analysis pipeline

```
# API settings
NEWS_API_KEY = "41db74da891e480c9384a475decd3206"
NEWS_API_URL = "https://newsapi.org/v2/everything"
SEARCH_QUERY = "Petronas"
LANGUAGE = "en"
SORT_BY = "popularity"

# Kafka settings
KAFKA_BOOTSTRAP_SERVERS = "localhost:9092"
KAFKA_TOPIC = "news-articles"

# Spark settings
APP_NAME = "Real-time Sentiment Analysis"

# Final column structure for enriched data
FINAL_COLUMN_STRUCTURE = [
    "publishedAt", "url", "cleaned_title", "cleaned_description", "source",
    "source_domain", "category", "word_count", "people_mentioned",
    "organizations_mentioned", "locations_mentioned", "project_names",
```

```
    "financial_figures", "dates_mentioned", "topic_label",
    "sentence", "Sentiment_Result"
]

SENTIMENT_MODEL_PATH = "sentiment_pipeline_spark"
RAW_OUTPUT_PATH = "/user/student/Final_Result/Final5"
ENRICHED_OUTPUT_PATH = "/user/student/Final_Result/Final13"
```

Class: DataEnricher
Description: The DataEnricher class enriches news article data by extracting metadata, named entities, project names, and topics, while performing data transformations.

```
# Core Spark Imports
from pyspark.sql import SparkSession, DataFrame
from pyspark.sql.functions import (
    col, when, sum, length, count, regexp_extract, udf, lit, split, array_contains,
    size, to_date, coalesce, format_string, lpad, explode, isnull
)
from pyspark.sql.types import (
    IntegerType, StringType, ArrayType, StructType, StructField, FloatType,
    DateType, TimestampType
)

# ML and NLP Imports
from pyspark.ml.feature import Tokenizer, StopWordsRemover, CountVectorizer
from pyspark.ml.clustering import LDA
import spacy
from pyspark.sql import functions as F

# Standard Python Libraries
import re
from urllib.parse import urlparse
from datetime import datetime
from collections import OrderedDict
import os

class DataEnricher:
    """
    Enriches the news articles DataFrame with metadata, entities, topics, etc.
    """
    def __init__(self, spark: SparkSession):
        self.spark = spark
        self.df = None
        self.nlp = None # To hold the loaded spaCy model
        self._initialize_spacy() # Attempt to load spaCy on init
        print("DataEnricher initialized.")

    def set_dataframe(self, df: DataFrame) -> 'DataEnricher':
        """Sets the DataFrame to be enriched."""
        self.df = df
        if self.df:
            print(f"--- [Enricher] DataFrame set. Row count: {self.df.count()} ---")
        else:
            print("WARNING: [Enricher] Received an empty DataFrame.")
        return self

    def _initialize_spacy(self):
        """Loads the spaCy model."""
        if self.nlp is None:
```

```
            try:
                # Make sure you have the model downloaded:
                # python -m spacy download en_core_web_sm
                self.nlp = spacy.load("en_core_web_sm")
                print("spaCy model 'en_core_web_sm' loaded successfully.")
            except ImportError:
                print("WARNING: spaCy library not found. Cannot perform spaCy entity
extraction. Install with: pip install spacy")
                self.nlp = None
            except OSError:
                    print("WARNING: spaCy model 'en_core_web_sm' not found. Download
it: python -m spacy download en_core_web_sm")
                 self.nlp = None
            except Exception as e:
                print(f"WARNING: An unexpected error occurred loading spaCy: {e}")
                self.nlp = None

    def run_all_enrichments(self,
                            url_col="url", content_col="cleaned_content",
                            num_topics=5, lda_max_iter=15, lda_vocab_size=5000,
                            final_columns_order=None) -> DataFrame:
        """Runs all enrichment steps sequentially."""
        if self.df is None:
            print("ERROR: DataFrame not set in Enricher. Cannot run enrichments.")
            return None

        print("\n--- [Enricher] Running Data Enrichment Steps ---")
        self.enrich_metadata(url_col=url_col, content_col=content_col)
        self.enrich_entities(content_col=content_col)
        self.enrich_project_names(content_col=content_col)
                #self.enrich_topics(content_col=content_col,  num_topics=num_topics,
max_iter=lda_max_iter, vocab_size=lda_vocab_size)


        print("--- [Enricher] Enrichment steps completed ---")
        # Select final columns if order is specified
        if final_columns_order:
            self.df = self._select_final_columns(self.df, final_columns_order)

        return self.df

    def enrich_metadata(self, url_col="url", content_col="content",
                        potential_date_cols=["publish_date", "date", "published_at",
"timestamp", "creation_date"],
                        target_date_col="publication_date"):
        """Adds metadata columns: source, domain, date, category, word count."""
        print("Enriching metadata...")
        if self.df is None: return

        # Add Source based on URL
        if url_col in self.df.columns:
            self.df = self.df.withColumn(
```

```
                "source",
                when(col(url_col).contains("reuters.com"), "Reuters")
                .when(col(url_col).contains("bloomberg.com"), "Bloomberg")
                .when(col(url_col).contains("nst.com"), "New Straits Times")
                .when(col(url_col).contains("thestar.com"), "The Star")
                       .when(col(url_col).contains("theedgemarkets.com"), "The  Edge
Markets")
                .otherwise("Other")
            )
            # Add Domain
            domain_udf = self._get_domain_udf()
            self.df = self.df.withColumn("source_domain", domain_udf(col(url_col)))
        else:
            self.df = self.df.withColumn("source", lit("Unknown"))
                                    self.df   =   self.df.withColumn("source_domain",
lit(None).cast(StringType()))


        # Add Category based on content
        if content_col in self.df.columns:
            self.df = self.df.withColumn(
                "category",

when(col(content_col).rlike("(?i)financ|profit|revenue|earnings|dividend|investment|
stock|market|shares"), "Financial")

.when(col(content_col).rlike("(?i)sustainab|green|environment|carbon|climate|emissio
n|esg|renewable"), "Sustainability")

.when(col(content_col).rlike("(?i)gas|oil|drilling|exploration|discovery|field|reser
ves|lng|upstream|downstream|refinery"), "Exploration & Production")

.when(col(content_col).rlike("(?i)partner|deal|agreement|contract|acquisition|merger
|joint venture|mou"), "Business Deals")

.when(col(content_col).rlike("(?i)technolog|digital|innovation|research|development|
ai|platform"), "Technology & Innovation")
                .otherwise("General")
            )
        else:
             self.df = self.df.withColumn("category", lit("Unknown"))

        # Calculate Word Count
        if content_col in self.df.columns:
            self.df = self.df.withColumn(
                "word_count",
                     when(col(content_col).isNotNull(), size(split(col(content_col),
"\\s+"))).otherwise(0)
            )
        else:
            self.df = self.df.withColumn("word_count", lit(0).cast(IntegerType()))
```

```
              print("Metadata  enrichment  done  (source,  domain,  date,  category,
word_count).")


    @staticmethod
    def _get_domain_udf():
        """Creates and returns the UDF for extracting domain from URL."""
        def get_domain(url):
            try:
                     if url and isinstance(url, str) and url.startswith(('http://',
'https://')):
                    return urlparse(url).netloc
                return None
            except Exception: return None
        return udf(get_domain, StringType())



    def enrich_entities(self, content_col="content"):
        """Extracts named entities using spaCy."""
        print("Enriching entities using spaCy...")
        if self.df is None: return
        if content_col not in self.df.columns:
            print(f"Skipping entity extraction: Column '{content_col}' not found.")
            self._add_empty_entity_columns()
            return
        if self.nlp is None:
            print("Skipping entity extraction: spaCy model not loaded.")
            self._add_empty_entity_columns()
            return

        # Define UDF schema and function
        entity_schema = StructType([
                            StructField("people",  ArrayType(StringType()),  True),
StructField("organizations", ArrayType(StringType()), True),
                        StructField("locations",  ArrayType(StringType()),  True),
StructField("dates", ArrayType(StringType()), True),
            StructField("money", ArrayType(StringType()), True)
        ])
                  extract_entities_udf  =  self._get_extract_entities_udf(self.nlp,
entity_schema) # Pass loaded model

        # Apply UDF
                             self.df    =    self.df.withColumn("extracted_entities",
extract_entities_udf(col(content_col)))

        # Flatten struct into columns
                              self.df    =    self.df.withColumn("people_mentioned",
col("extracted_entities.people")) \
                                              .withColumn("organizations_mentioned",
col("extracted_entities.organizations")) \
```

```
                                                    .withColumn("locations_mentioned",
col("extracted_entities.locations")) \
                                                        .withColumn("dates_mentioned",
col("extracted_entities.dates")) \
                                                      .withColumn("financial_figures",
col("extracted_entities.money")) \
                            .drop("extracted_entities")
        print("Entity extraction done.")


    @staticmethod
    def _get_extract_entities_udf(nlp_model, schema):
         """Creates the UDF for spaCy entity extraction."""
         if nlp_model is None: # Return a dummy UDF if spacy failed
             def dummy_extract(text):
                       return {"people": [], "organizations": [], "locations": [],
"dates": [], "money": []}
             return udf(dummy_extract, schema)

         # --- Actual UDF function using the passed nlp_model ---
         def extract_entities(text):
                 # This function now closes over the nlp_model variable passed to
_get_extract_entities_udf
                   entities = {"people": [], "organizations": [], "locations": [],
"dates": [], "money": []}
             if not text or not isinstance(text, str): return entities
             try:
                 doc = nlp_model(text[:100000]) # Limit text size
                 for ent in doc.ents:
                     ent_text = ent.text.strip()
                     if len(ent_text) < 3 or ent_text.isspace(): continue
                     label = ent.label_
                             if label == "PERSON" and len(ent_text.split()) <= 4:
entities["people"].append(ent_text)
                         elif label == "ORG" and "petronas" not in ent_text.lower():
entities["organizations"].append(ent_text)
                                                    elif label in ["GPE", "LOC"]:
entities["locations"].append(ent_text)
                     elif label == "DATE": entities["dates"].append(ent_text)
                     elif label == "MONEY": entities["money"].append(ent_text)
                                         for key in entities: entities[key] =
list(OrderedDict.fromkeys(entities[key]))[:10]
              except Exception as e: pass # Log errors if needed, but don't fail the
UDF
             return entities
         # --- End of actual UDF function ---

         return udf(extract_entities, schema)

    def _add_empty_entity_columns(self):
         """Adds empty array columns if entity extraction is skipped."""
         if self.df is None: return
```

```
                    entity_cols  =  ["people_mentioned",  "organizations_mentioned",
"locations_mentioned", "dates_mentioned", "financial_figures"]
         for col_name in entity_cols:
             if col_name not in self.df.columns:
                                             self.df  =  self.df.withColumn(col_name,
lit(None).cast(ArrayType(StringType())))

    def enrich_project_names(self, content_col="content"):
        """Extracts project names using regular expressions."""
        print("Enriching project names using regex...")
        if self.df is None: return
        if content_col not in self.df.columns:
               print(f"Skipping project name extraction: Column '{content_col}' not
found.")
                                       self.df  =  self.df.withColumn("project_names",
lit(None).cast(ArrayType(StringType())))
            return

                                                                  project_pattern       =
r"(?i)(?:Project|Basin|Field|Platform|Terminal|Plant|Block)\s+([A-Z][-a-zA-Z0-9\s]*[
a-zA-Z0-9])"
        extract_projects_udf = udf(
                 lambda text: list(OrderedDict.fromkeys([match.strip() for match in
re.findall(project_pattern, text)])) if text else [],
            ArrayType(StringType())
        )
                                    self.df    =    self.df.withColumn("project_names",
extract_projects_udf(col(content_col)))
        print("Project name extraction done.")


       def  enrich_topics(self,  content_col="content",  num_topics=5,  max_iter=15,
vocab_size=5000, min_df=5):
        """Performs LDA topic modeling."""
        print("Enriching topics using LDA...")
        if self.df is None: return
        if content_col not in self.df.columns:
            print(f"Skipping topic modeling: Column '{content_col}' not found.")
            self._add_empty_topic_columns()
            return

        # Prepare data for LDA
         df_for_lda = self.df.select("url", content_col).fillna({content_col: ''}) #
Use URL as identifier
        tokenizer = Tokenizer(inputCol=content_col, outputCol="tokens")
        df_tokens = tokenizer.transform(df_for_lda)
         custom_stopwords = ["petronas", "said", "also", "year", "company", "group",
"malaysia",  "kuala",  "lumpur",  "ringgit",  "rm",  "mln",  "bln",  "pct",  "news",
"report", "update", "inc", "bhd"]
        remover = StopWordsRemover(inputCol="tokens", outputCol="filtered_tokens")
            remover.setStopWords(StopWordsRemover.loadDefaultStopWords("english")  +
custom_stopwords)
```

```
        df_no_stop = remover.transform(df_tokens)
                        vectorizer   =   CountVectorizer(inputCol="filtered_tokens",
outputCol="features", vocabSize=vocab_size, minDF=min_df)
        try:
            cv_model = vectorizer.fit(df_no_stop)
            df_features = cv_model.transform(df_no_stop)

            # Train LDA
                 lda  =  LDA(k=num_topics,  maxIter=max_iter,  featuresCol="features",
seed=42)
            lda_model = lda.fit(df_features)
            df_with_topics = lda_model.transform(df_features)

            # Process results
                    dominant_topic_udf  =  udf(lambda  dist:  int(max(enumerate(dist),
key=lambda x: x[1])[0]) if dist else None, IntegerType())
                       df_with_topics  =  df_with_topics.withColumn("dominant_topic",
dominant_topic_udf(col("topicDistribution")))

            # Create topic labels
            vocab = cv_model.vocabulary
            topicDescDF = lda_model.describeTopics(maxTermsPerTopic=5)
                                                              generic_stop_words     =
set(StopWordsRemover.loadDefaultStopWords("english") + custom_stopwords)
            def indices_to_words(termIndices):
                keywords = [vocab[i] for i in termIndices if vocab[i].lower() not in
generic_stop_words and len(vocab[i]) > 2]
                return ", ".join(keywords[:3]) if keywords else "Unknown Topic"
            indices_to_words_udf = udf(indices_to_words, StringType())
                             topicDescDF   =   topicDescDF.withColumn("topic_keywords",
indices_to_words_udf(col("termIndices")))
                   topic_mapping  =  {row['topic']:  row['topic_keywords']  for  row  in
topicDescDF.select("topic", "topic_keywords").collect()}
            topic_mapping_bc = self.spark.sparkContext.broadcast(topic_mapping)
              map_topic_label_udf = udf(lambda idx: topic_mapping_bc.value.get(idx,
f"Unknown Topic {idx}") if idx is not None else "N/A", StringType())
                          df_with_topics  =  df_with_topics.withColumn("topic_label",
map_topic_label_udf(col("dominant_topic")))

            # Join results back (selecting only necessary topic columns)
                    topic_results  =  df_with_topics.select("url",  "dominant_topic",
"topic_label") # "topicDistribution" removed - too large
            self.df = self.df.join(topic_results, on="url", how="left")
            print("Topic modeling enrichment done.")

        except Exception as e:
            print(f"ERROR during topic modeling: {e}. Skipping topic enrichment.")
            self._add_empty_topic_columns()


    def _add_empty_topic_columns(self):
        """Adds empty topic columns if LDA fails or is skipped."""
```

```
        if self.df is None: return
        if "dominant_topic" not in self.df.columns:
                                    self.df   =   self.df.withColumn("dominant_topic",
lit(None).cast(IntegerType()))
        if "topic_label" not in self.df.columns:
                                       self.df   =   self.df.withColumn("topic_label",
lit("N/A").cast(StringType()))




    @staticmethod
    def _get_process_content_udf():
         """Creates the UDF for sentence processing."""
         def process_content_sentences(input_text):
                # ... (Sentence splitting, cleaning, filtering logic from previous
example) ...
             if input_text is None or not isinstance(input_text, str): return []
             sentences = re.split(r'[.?!]\s+|\n+', input_text)
             sentences = [s.strip() for s in sentences if s and not s.isspace()]
             unique_sentences = list(OrderedDict.fromkeys(sentences))
             final_sentences = []
             for sentence in unique_sentences:
                 words = sentence.split()
                 words_no_digits = [word for word in words if not any(char.isdigit()
for char in word)]
                 cleaned_sentence = ' '.join(words_no_digits)
                 if len(cleaned_sentence.split()) >= 3:
                     final_sentences.append(cleaned_sentence)
             return final_sentences
         return udf(process_content_sentences, ArrayType(StringType()))

    @staticmethod
    def _select_final_columns(df, columns_order):
        """Selects and orders columns based on the provided list."""
        print("Selecting and ordering final columns...")
        existing_columns = [c for c in columns_order if c in df.columns]
        missing_columns = [c for c in columns_order if c not in df.columns]
        if missing_columns:
                print(f"Warning: Requested final columns not found in DataFrame:
{missing_columns}")
        print(f"Final selected columns: {existing_columns}")
        return df.select(existing_columns)
```

Class: DataEnricherWrapper
Description: The DataEnricherWrapper class introduces data enrichment and validation functionalities by first utilizing the DataValidator class for validation of the input DataFrame, followed by enrichment using the DataEnricher class. The process is concluded by storing the altered dataset as Parquet format at a specific output destination.

```
from pyspark.sql import SparkSession, DataFrame
import logging
from Data_Validation_Task3 import DataValidator
from Data_Enricher_Task3 import DataEnricher
from config_Task3 import FINAL_COLUMN_STRUCTURE

class DataEnricherWrapper:
    """Wrapper for the DataValidator and DataEnricher classes."""

    def __init__(self, spark):
        """
        Initialize the data enricher wrapper.

        Args:
            spark (SparkSession): The Spark session
        """
        self.spark = spark
        self.logger = logging.getLogger(__name__)

    def process_data(self, input_df, output_path):
        """
        Process data through validation and enrichment pipeline.

        Args:
            input_df (DataFrame): Input DataFrame
            output_path (str): Path to save the enriched data
        """
        # Validate data
        validator = DataValidator(self.spark)
        validated_df = validator.set_dataframe(input_df) \
                       .run_all_validations() \
                       .get_clean_dataframe(drop_duplicates_url=False,
                                            drop_duplicates_sentence=True,
                                            filter_short_content=True)

        # Enrich data
        enricher = DataEnricher(self.spark)
        enriched_df = enricher.set_dataframe(validated_df) \

.run_all_enrichments(final_columns_order=FINAL_COLUMN_STRUCTURE)

        if enriched_df:
            print("\n--- Final Enriched Dataset (Sample Data) ---")
            enriched_df.show(5, truncate=True, vertical=True)
```

```
            print(f"\n--- Saving final enriched dataset to: {output_path} ---")



              enriched_df.coalesce(10).write.format("parquet").option("compression",
"snappy").mode("overwrite").save(output_path)
        else:
            print("\nERROR: Enrichment process failed to produce a DataFrame.")
            return None
```

Class: DataValidator
Description: The DataValidator class validates and cleans a DataFrame of news articles by checking required columns, URL formats, null values, content length, and removing duplicates, and it provides a summary of validation results.

```
 # Core Spark Imports
from pyspark.sql import SparkSession, DataFrame
from pyspark.sql.functions import (
    col, when, sum, length, count, regexp_extract, udf, lit, split, array_contains,
    size, to_date, coalesce, format_string, lpad, explode, isnull
)
from pyspark.sql.types import (
    IntegerType, StringType, ArrayType, StructType, StructField, FloatType,
    DateType, TimestampType
)

# ML and NLP Imports
from pyspark.ml.feature import Tokenizer, StopWordsRemover, CountVectorizer
from pyspark.ml.clustering import LDA
import spacy
from pyspark.sql import functions as F

# Standard Python Libraries
import re
from urllib.parse import urlparse
from datetime import datetime
from collections import OrderedDict
import os


class DataValidator:
    """
    Validates the input news articles DataFrame based on predefined rules.
    """
    def __init__(self, spark: SparkSession):
        self.spark = spark
        self.df = None
        self.validation_summary = {}
        print("DataValidator initialized.")

    def load_data(self, path: str) -> 'DataValidator':
        """Loads data from a Parquet file."""
        print(f"--- [Validator] Loading data from: {path} ---")
        try:
            self.df = self.spark.read.parquet(path)
            print(f"Successfully loaded data. Row count: {self.df.count()}")
            self.df.printSchema()
        except Exception as e:
            print(f"ERROR: Failed to load Parquet file from {path}: {e}")
            self.df = None # Ensure df is None if loading fails
            raise # Re-raise the exception to halt execution if loading fails
```

```
        return self


    def set_dataframe(self, df: DataFrame) -> 'DataValidator':
        """Sets the DataFrame to be validated."""
        self.df = df
        if self.df:
                print(f"--- [Validator] DataFrame set. Row count: {self.df.count()}
---")
            self.df.printSchema()
        else:
            print("WARNING: [Validator] Received an empty DataFrame.")
        return self

    def run_all_validations(self,
                                           required_columns=["url", "cleaned_content",
"cleaned_title"],
                                      potential_date_columns=["publish_date", "date",
"published_at", "timestamp", "creation_date"],
                              url_column="url",
                              content_column="content",
                              min_content_length=20) -> 'DataValidator':
        """Runs all defined validation checks."""
        if self.df is None:
            print("ERROR: DataFrame not loaded. Cannot run validations.")
            return self

        print("\n--- [Validator] Running Data Validation Checks ---")
        self._check_required_columns(required_columns)
        self._check_nulls_or_empty(required_columns + potential_date_columns)
        self._validate_url_format(url_column)
        self._check_content_length(content_column, min_content_length)
        #self._check_duplicate_urls(url_column)
        self. _check_duplicate_sentences('sentence')
        print("No Duplication Undergo")
        print("--- [Validator] Validation checks completed ---")
        self._print_summary()
        return self

    def _check_required_columns(self, required_columns: list):
        """Checks if essential columns exist in the DataFrame."""
        print("Checking for required columns...")
        missing_cols = [c for c in required_columns if c not in self.df.columns]
        if missing_cols:
            self.validation_summary['missing_required_columns'] = missing_cols
            print(f"WARNING: Missing required columns: {missing_cols}")
        else:
            self.validation_summary['missing_required_columns'] = []
            print("All required columns are present.")

    def _check_nulls_or_empty(self, columns_to_check: list):
        """Checks for null or empty string values in specified columns."""
```

```
        print("Checking for null or empty values...")
        null_counts = {}
        existing_cols = [c for c in columns_to_check if c in self.df.columns]
        if existing_cols:
            results = self.df.select([
                                        sum(when(isnull(c)   |   (col(c)   ==   ""),
1).otherwise(0)).alias(f"null_empty_{c}")
                for c in existing_cols
            ]).first().asDict()
                    null_counts  =  {col_alias.replace("null_empty_",  ""):  count  for
col_alias, count in results.items()}
            self.validation_summary['null_empty_counts'] = null_counts
            print("Null/Empty counts:", null_counts)
        else:
            print("No columns found to check for nulls/emptiness.")
            self.validation_summary['null_empty_counts'] = {}

    def _validate_url_format(self, url_column: str):
        """Validates the URL format (basic http/https check)."""
        print(f"Validating URL format in column '{url_column}'...")
        if url_column in self.df.columns:
            url_pattern = r"^https?://.+"
                                                                invalid_count     =
self.df.filter(~col(url_column).rlike(url_pattern)).count()
            self.validation_summary['invalid_url_format_count'] = invalid_count
            print(f"Rows with potentially invalid URL format: {invalid_count}")
        else:
                 print(f"Skipping  URL  format  validation:  Column  '{url_column}'  not
found.")
            self.validation_summary['invalid_url_format_count'] = 'N/A'

    def _check_content_length(self, content_column: str, min_length: int):
        """Checks if the content length meets a minimum requirement."""
               print(f"Checking  content  length  in  column  '{content_column}'  (min:
{min_length})...")
        if content_column in self.df.columns:
            short_content_count = self.df.filter(
                    col(content_column).isNotNull() & (length(col(content_column)) <
min_length)
            ).count()
            self.validation_summary['short_content_count'] = short_content_count
                  print(f"Rows  with  content  shorter  than  {min_length}  characters:
{short_content_count}")
        else:
                print(f"Skipping content length check: Column '{content_column}' not
found.")
            self.validation_summary['short_content_count'] = 'N/A'

    def _check_duplicate_urls(self, url_column: str):
        """Checks for duplicate values in the URL column."""
        print(f"Checking for duplicate URLs in column '{url_column}'...")
        if url_column in self.df.columns:
```

```
            total_rows = self.df.count()
                                                                distinct_non_null_urls     =
self.df.filter(col(url_column).isNotNull()).select(url_column).distinct().count()
            non_null_rows = self.df.filter(col(url_column).isNotNull()).count()
            duplicate_count = non_null_rows - distinct_non_null_urls
            self.validation_summary['duplicate_url_count'] = duplicate_count
            self.validation_summary['distinct_url_count'] = distinct_non_null_urls
            print(f"Total non-null rows with URL: {non_null_rows}")
            print(f"Distinct non-null URLs: {distinct_non_null_urls}")
            print(f"Duplicate non-null URLs found: {duplicate_count}")
        else:
            print(f"Skipping duplicate URL check: Column '{url_column}' not found.")
            self.validation_summary['duplicate_url_count'] = 'N/A'
            self.validation_summary['distinct_url_count'] = 'N/A'

    def _check_duplicate_sentences(self, sentence_column: str):
        """Checks for duplicate sentences in the specified column."""
        print(f"Checking for duplicate sentences in column '{sentence_column}'...")
        if sentence_column in self.df.columns:
            total_rows = self.df.count()
                                                        distinct_non_null_sentences    =
self.df.filter(col(sentence_column).isNotNull()).select(sentence_column).distinct().
count()
            non_null_rows = self.df.filter(col(sentence_column).isNotNull()).count()
            duplicate_count = non_null_rows - distinct_non_null_sentences
            self.validation_summary['duplicate_sentences_count'] = duplicate_count
                                self.validation_summary['distinct_sentences_count']   =
distinct_non_null_sentences
            print(f"Total non-null rows with sentences: {non_null_rows}")
            print(f"Distinct non-null sentences: {distinct_non_null_sentences}")
            print(f"Duplicate sentences found: {duplicate_count}")

            # Optionally, show some examples of duplicated sentences
                  if duplicate_count > 0 and duplicate_count < 1000:   # Only for
reasonable numbers
                print("\nExamples of duplicated sentences:")
                duplicate_sentences = (self.df
                    .filter(col(sentence_column).isNotNull())
                    .groupBy(sentence_column)
                    .count()
                    .filter(col("count") > 1)
                    .orderBy(col("count").desc())
                    .limit(5))
                duplicate_sentences.show(truncate=False)
        else:
               print(f"Skipping duplicate sentence check: Column '{sentence_column}'
not found.")
            self.validation_summary['duplicate_sentences_count'] = 'N/A'
            self.validation_summary['distinct_sentences_count'] = 'N/A'

    def _check_duplicate_urls(self, url_column: str):
        """Checks for duplicate values in the URL column."""
```

```
        print(f"Checking for duplicate URLs in column '{url_column}'...")
        if url_column in self.df.columns:
            total_rows = self.df.count()
                                                                distinct_non_null_urls     =
self.df.filter(col(url_column).isNotNull()).select(url_column).distinct().count()
            non_null_rows = self.df.filter(col(url_column).isNotNull()).count()
            duplicate_count = non_null_rows - distinct_non_null_urls
            self.validation_summary['duplicate_url_count'] = duplicate_count
            self.validation_summary['distinct_url_count'] = distinct_non_null_urls
            print(f"Total non-null rows with URL: {non_null_rows}")
            print(f"Distinct non-null URLs: {distinct_non_null_urls}")
            print(f"Duplicate non-null URLs found: {duplicate_count}")
        else:
            print(f"Skipping duplicate URL check: Column '{url_column}' not found.")
            self.validation_summary['duplicate_url_count'] = 'N/A'
            self.validation_summary['distinct_url_count'] = 'N/A'

    def _print_summary(self):
        """Prints the collected validation summary."""
        print("\n--- Validation Summary ---")
        for key, value in self.validation_summary.items():
            print(f"{key}: {value}")
        print("--------------------------")

    def get_summary(self) -> dict:
        """Returns the validation summary dictionary."""
        return self.validation_summary

    def get_dataframe(self) -> DataFrame:
        """Returns the loaded DataFrame."""
        return self.df

                      def     get_clean_dataframe(self,     drop_duplicates_url=False,
drop_duplicates_sentence=False,
                          filter_short_content=True, min_length=20) -> DataFrame:
        """Returns a DataFrame filtered based on validation checks."""
        if self.df is None:
            print("ERROR: Cannot get clean DataFrame, data not loaded.")
            return None

        print("\n--- [Validator] Applying basic cleaning ---")
        clean_df = self.df

        # Filter null/invalid URLs
        if "url" in clean_df.columns:
            url_pattern = r"^https?://.+"
                               clean_df   =   clean_df.filter(col("url").isNotNull()   &
col("url").rlike(url_pattern))
            print(f"Rows after filtering null/invalid URLs: {clean_df.count()}")

        # Drop duplicate URLs if requested
        if drop_duplicates_url and "url" in clean_df.columns:
```

```
            before_count = clean_df.count()
            clean_df = clean_df.dropDuplicates(["url"])
            after_count = clean_df.count()
            print(f"Dropped {before_count - after_count} duplicate URLs")

        # Drop duplicate sentences if requested
        if drop_duplicates_sentence and "sentence" in clean_df.columns:
            before_count = clean_df.count()
            clean_df = clean_df.dropDuplicates(["sentence"])
            after_count = clean_df.count()
            print(f"Dropped {before_count - after_count} duplicate sentences")

        # Filter short content
        if filter_short_content and "content" in clean_df.columns:
                            clean_df  =  clean_df.filter(col("content").isNotNull()  &
(length(col("content")) >= min_length))
                        print(f"Rows  after  filtering  short  content  (<{min_length}):
{clean_df.count()}")

        print("--- [Validator] Cleaning finished ---")
        return clean_df
```

Class: Main Execution
Description: The script serves as the driver for all the modules, setting up a Spark session, fetching news data via NewsAPI, processing it in real-time using Kafka and sentiment analysis, and then enriching and saving the results to the specified output path.

```
import logging
import os
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType, ArrayType

# Import custom modules
from spark_session_Task3 import SparkSessionManager
from streaming_processor_Task3 import StreamingProcessor
from data_enricher_wrapper_Task3 import DataEnricherWrapper
from config_Task3 import (
    KAFKA_BOOTSTRAP_SERVERS,
    KAFKA_TOPIC,
    NEWS_API_KEY,
    SEARCH_QUERY,
    APP_NAME,
    SENTIMENT_MODEL_PATH,
    RAW_OUTPUT_PATH,
    ENRICHED_OUTPUT_PATH
)

# Configure logging
logging.basicConfig(level=logging.INFO,   format='%(asctime)s   -   %(levelname)s   -
%(message)s')
logger = logging.getLogger(__name__)

def main():
    """Main execution function"""
    try:
        # Create Spark session
        spark = SparkSessionManager.create_spark_session(APP_NAME)
        print("Spark session created successfully")

        # Import text processing modules
        from text_cleaner_Task3 import TextCleaner
        from news_fetcher_Task3 import NewsAPIFetcher

        # Register UDFs for text processing
        clean_text_udf = udf(TextCleaner.clean_text, StringType())
                    split_sentences_udf   =   udf(TextCleaner.split_into_sentences,
ArrayType(StringType()))

        print("Text processing functions defined and UDFs registered")

        # Initialize the NewsAPI fetcher
        news_fetcher = NewsAPIFetcher(
            api_key=NEWS_API_KEY,
```

```
            query=SEARCH_QUERY,
            kafka_bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
            kafka_topic=KAFKA_TOPIC
        )

        # Create streaming processor
        processor = StreamingProcessor(
            spark=spark,
            kafka_bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
            kafka_topic=KAFKA_TOPIC,
            sentiment_model_path=SENTIMENT_MODEL_PATH
        )

        # Start news fetcher
        news_fetcher.start()

        # Start streaming and perform sentiment analysis
                    processor.start_streaming(clean_text_udf,   split_sentences_udf,
news_fetcher)

        # Get results DataFrame
        result_df = processor.get_results_dataframe()

        if result_df:
            # Show results
            result_df.show()

            # Enrich and save data
            enricher = DataEnricherWrapper(spark)
            enricher.process_data(result_df, ENRICHED_OUTPUT_PATH)
        else:
            print("No results to process")

    except Exception as e:
        logger.error(f"Error in main execution: {str(e)}")
    finally:
         # Stop SparkSession (optional, as it will be stopped when the application
ends)
        if 'spark' in locals():
            spark.stop()
            logger.info("Spark session stopped")

if __name__ == "__main__":
    main()
```

Class: NewsAPIFetcher
Description: The NewsAPIFecther class fetches news articles from NewsAPI based on a search query, processes the articles by cleaning their content, and sends them to a Kafka topic. It operates continuously, fetching articles at specified intervals in a separate thread, and handles retries and error logging.

```
# Standard library imports
import json
import time
import threading
import logging
import datetime
# Third-party imports
import requests
from kafka import KafkaProducer
from bs4 import BeautifulSoup

# Setup logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


# API settings
NEWS_API_KEY = "41db74da891e480c9384a475decd3206"
NEWS_API_URL = "https://newsapi.org/v2/everything"
SEARCH_QUERY = "Petronas"
LANGUAGE = "en"
SORT_BY = "popularity"

from text_cleaner_Task3 import TextCleaner


class NewsAPIFetcher:
    """
    Class to fetch articles from NewsAPI and send them to Kafka.
    """


    def __init__(self, api_key, query, kafka_bootstrap_servers, kafka_topic):
        self.api_key = api_key
        self.query = query
        self.kafka_producer = KafkaProducer(
            bootstrap_servers=kafka_bootstrap_servers,
            value_serializer=lambda v: json.dumps(v).encode('utf-8')
        )
        self.kafka_topic = kafka_topic
        self.last_fetch_time = None
        self.fetch_interval = 30  # 5 minutes in seconds
        self.running = False
        self.thread = None
```

```
    def fetch_articles(self):
        """
        Fetches articles from NewsAPI based on the query.
        """
        try:
            # Construct the API URL
            url = (f'{NEWS_API_URL}?'
                   f'q={self.query}&'
                   f'sortBy={SORT_BY}&'
                   f'language={LANGUAGE}&'
                   f'apiKey={self.api_key}')

            # Add from parameter if we've fetched before to avoid duplicates
            if self.last_fetch_time:
                # Format the time as ISO 8601
                                                                    from_time    =
datetime.datetime.fromtimestamp(self.last_fetch_time).strftime('%Y-%m-%dT%H:%M:%S')
                url += f'&from={from_time}'

            # Send the GET request and parse the JSON response
            response = requests.get(url)
            data = response.json()

            # Update the last fetch time
            self.last_fetch_time = time.time()

            # Process the articles
            if "articles" in data:
                return data["articles"]
            else:
                logger.warning(f"No articles found in API response: {data}")
                return []
        except Exception as e:
            logger.error(f"Error fetching articles from NewsAPI: {str(e)}")
            return []

    def process_and_send_articles(self, articles):
        """
        Processes articles and sends them to Kafka.
        """
        sent_count = 0
        for article in articles:
            try:
                # Extract article information
                title = article.get("title", "")
                description = article.get("description", "")
                url = article.get("url", "")
                publishedAt = article.get("publishedAt", "")
                # Skip if title or URL is missing
                if not title or not url:
                    continue
```

```
                # Fetch and clean the article content
                content = TextCleaner.fetch_and_clean_article_content(url)

                # Prepare the article information
                article_info = {
                    "publishedAt":publishedAt,
                    "title": title,
                    "description": description,
                    "url": url,
                    "content": content
                }

                # Send the article data to Kafka topic
                self.kafka_producer.send(self.kafka_topic, value=article_info)

                logger.info(f"Sent article to Kafka: {title}")
                sent_count += 1

            except Exception as e:
                logger.error(f"Error processing article: {str(e)}")

        # Flush to ensure all messages are sent
        self.kafka_producer.flush()
        logger.info(f"Total articles sent to Kafka: {sent_count}")
        return sent_count

    def fetch_and_send(self):
        """
        Fetches articles and sends them to Kafka.
        """
        articles = self.fetch_articles()
        return self.process_and_send_articles(articles)

    def run_continuously(self):
        """
        Runs the fetcher continuously at the specified interval.
        """
        while self.running:
            try:
                sent_count=self.fetch_and_send()

                if sent_count==0:
                       logger.info("No articles fetched in this batch; auto-stopping
the fetcher.")
                    self.running = False
                    break
            except Exception as e:
                logger.error(f"Error in continuous fetching: {str(e)}")

            # Sleep for the specified interval
            time.sleep(self.fetch_interval)
```

```
    def start(self):
        """
        Starts the fetcher in a separate thread.
        """
        if not self.running:
            self.running = True
            self.thread = threading.Thread(target=self.run_continuously)
            self.thread.daemon = True
            self.thread.start()
            logger.info("NewsAPI fetcher started")

    def stop(self):
        """
        Stops the fetcher.
        """
        if self.running:
            self.running = False
            if self.thread:
                self.thread.join(timeout=10)
            logger.info("NewsAPI fetcher stopped")
```

Class: SchemaDefinitions
Description: The class SchemaDefinitions defines and provides the different schemas used within the application, including those that are specific to Kafka messages and outputs for sentiment analysis. They help structure information within the application, thus supporting smooth operations.

```
from pyspark.sql.types import StructType, StructField, StringType, TimestampType, FloatType, ArrayType, IntegerType

class SchemaDefinitions:
    """Defines schemas used in the application."""

    @staticmethod
    def get_kafka_schema():
        """
        Returns the schema for Kafka messages.

        Returns:
            StructType: Schema for Kafka messages
        """
        return StructType([
            StructField("publishedAt", StringType(), True),
            StructField("title", StringType(), True),
            StructField("description", StringType(), True),
            StructField("url", StringType(), True),
            StructField("content", StringType(), True)
        ])

    @staticmethod
    def get_result_schema():
        """
        Returns the schema for sentiment analysis results.

        Returns:
            StructType: Schema for sentiment results
        """
        return StructType([
            StructField("publishedAt", StringType(), True),
            StructField("cleaned_title", StringType(), True),
            StructField("cleaned_content", StringType(), True),
            StructField("cleaned_description", StringType(), True),
            StructField("url", StringType(), True),
            StructField("sentence", StringType(), True),
            StructField("Sentiment_Result", IntegerType(), True)
        ])
```

Class: SentimentPredictor
Description: The SentimentPredictor class loads a pre-trained PySpark ML sentiment analysis pipeline and uses it to make sentiment predictions on input text.

```
from pyspark.ml import PipelineModel
from pyspark.sql import SparkSession

class SentimentPredictor:
    """
    A class for loading and using a PySpark ML pipeline for sentiment prediction.
    """

    def __init__(self, spark, pipeline_path="sentiment_pipeline_spark"):
        """
        Initialize the SentimentPredictor with a SparkSession and pipeline path.

        Args:
            spark (SparkSession): The active Spark session
            pipeline_path (str): Path to the saved pipeline model
        """
        self.spark = spark
        self.pipeline_path = pipeline_path
        self.model = None
        self._load_pipeline()

    def _load_pipeline(self):
        """
        Load the pipeline model from the specified path.
        """
        try:
            self.model = PipelineModel.load(self.pipeline_path)
            print(f"Loaded pipeline successfully from {self.pipeline_path}.")
        except Exception as e:
            print(f"Error loading pipeline from '{self.pipeline_path}': {e}")
            self.model = None

    def predict(self, sample_texts):
        """
        Make sentiment predictions on the provided texts.

        Args:
                sample_texts (str or list): A single text string or a list of text
strings

        Returns:
            The first prediction if successful, None if an error occurs
        """
        # Ensure sample_text is a list if provided as a single string
        if isinstance(sample_texts, str):
            sample_texts = [sample_texts]
```

```
        if self.model is None:
            print("Pipeline model not loaded. Cannot make predictions.")
            return None

        # Create a DataFrame from the input sample texts
        # The column name should match the expected input in your pipeline
        try:
            pred_df = self.spark.createDataFrame(
                [(text,) for text in sample_texts],
                ["processed_sentence"]
            )
        except Exception as e:
            print(f"Error creating DataFrame from input texts: {e}")
            return None

        # Use the pipeline's transform method to get predictions
        try:
            predictions_df = self.model.transform(pred_df)
        except Exception as e:
            print(f"Error during transformation: {e}")
            return None

        # Extract the predictions from the DataFrame as a list
        try:
            prediction_list = predictions_df.select("prediction").rdd.map(
                lambda row: row.prediction
            ).collect()
            print("Predicted sentiment:", prediction_list)
            return prediction_list[0]
        except Exception as e:
            print(f"Error collecting predictions: {e}")
            return None
```

Class: SparkSessionManager
Description: The SparkSessionManager class is responsible for creating and configuring a SparkSession with necessary settings for real-time processing.

```
from pyspark.sql import SparkSession
import logging
import warnings

class SparkSessionManager:
    """Creates and manages a SparkSession with required configurations."""
```

```
    @staticmethod
    def create_spark_session(app_name="Real-time Sentiment Analysis"):
        """
        Creates and returns a SparkSession with the required configurations.

        Args:
            app_name (str): Name of the Spark application

        Returns:
            SparkSession: Configured Spark session
        """
        logger = logging.getLogger(__name__)
        logger.info("Initializing Spark session...")

        # Suppress specific Kafka warnings

logging.getLogger("org.apache.spark.sql.kafka.KafkaDataConsumer").setLevel(logging.E
RROR)
        logging.getLogger("org.apache.kafka").setLevel(logging.ERROR)

        # Combine the packages from SPARK_PACKAGES with the Kafka package
        combined_packages = ",".join([
                "org.apache.spark:spark-sql-kafka-0-10_2.13:3.5.1",   # Spark Kafka
connector
            "org.apache.kafka:kafka-clients:2.8.1",             # Kafka client
                 "org.apache.spark:spark-streaming-kafka-0-10_2.13:3.5.1",   # Kafka
Streaming connector
             "org.apache.spark:spark-token-provider-kafka-0-10_2.13:3.5.1",  # Kafka
token provider
            "org.apache.commons:commons-pool2:2.11.0"  # Apache Commons Pool 2
        ])

        # Add configuration to suppress Kafka warnings
        spark = SparkSession.builder \
                .appName(app_name) \
                .config("spark.jars.packages", combined_packages) \
                .config("spark.sql.streaming.metricsEnabled", "false") \
                .config("spark.sql.shuffle.partitions", "10") \
                .getOrCreate()

        # Set log level to reduce verbosity
        spark.sparkContext.setLogLevel("ERROR")  # Changed from WARN to ERROR

        # Additional step to suppress specific Kafka warnings using Java properties
        conf = spark.sparkContext._jsc.hadoopConfiguration()
        conf.set("mapreduce.job.log4j.hierarchy.append", "false")

        # Suppress Python warnings as well
        warnings.filterwarnings("ignore", message=".*KafkaDataConsumer.*")

        logger.info("Spark session initialized successfully")
        return spark
```

Class: StreamingProcessor
Description: The StreamingProcessor class streams data from Kafka, applies sentiment analysis to sentences, and processes each micro-batch in a distributed manner. It transforms, cleans, and splits text, then collects and consolidates the results for further use, running continuously until signaled to stop.

```
from pyspark.sql import SparkSession, DataFrame
from pyspark.sql.functions import col, from_json, lit, explode
import pyspark.sql.functions as F
import logging
import time
from schema_Task3 import SchemaDefinitions
from sentiment_predictor_Task3 import SentimentPredictor
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

class StreamingProcessor:
    """Processes streaming data from Kafka and performs sentiment analysis."""

            def   __init__(self,   spark,   kafka_bootstrap_servers,   kafka_topic,
sentiment_model_path):
        """
        Initialize the streaming processor.

        Args:
            spark (SparkSession): The Spark session
            kafka_bootstrap_servers (str): Kafka bootstrap servers
            kafka_topic (str): Kafka topic to subscribe to
            sentiment_model_path (str): Path to the sentiment model file
        """
        self.spark = spark
        self.kafka_bootstrap_servers = kafka_bootstrap_servers
        self.kafka_topic = kafka_topic
        self.kafka_schema = SchemaDefinitions.get_kafka_schema()
        self.final_results = []
        self.logger = logging.getLogger(__name__)

        # Initialize the sentiment predictor
        self.sentiment_predictor = SentimentPredictor(spark, sentiment_model_path)

    def create_kafka_stream(self):
        """
        Create a streaming DataFrame from Kafka.

        Returns:
            DataFrame: Streaming DataFrame from Kafka
        """
```

```
        return self.spark.readStream \
            .format("kafka") \
            .option("kafka.bootstrap.servers", self.kafka_bootstrap_servers) \
            .option("subscribe", self.kafka_topic) \
            .option("startingOffsets", "latest") \
            .load()

              def    create_streaming_dataframe(self,    df_kafka,    clean_text_udf,
split_sentences_udf):
        """
            Transform raw Kafka stream into a structured DataFrame with exploded
sentences.

        Args:
            df_kafka (DataFrame): Raw Kafka DataFrame
            clean_text_udf: UDF for cleaning text
            split_sentences_udf: UDF for splitting text into sentences

        Returns:
            DataFrame: Transformed DataFrame with exploded sentences
        """
        # Cast the raw Kafka message value to string
              df_raw  =  df_kafka.selectExpr("CAST(value  AS  STRING)  as  raw_value",
"timestamp")

        # Parse the JSON data using the provided schema
                               df_parsed    =    df_raw.select(from_json("raw_value",
self.kafka_schema).alias("data"), "timestamp")

        # Flatten the JSON struct into individual columns
        df_final = df_parsed.select("data.*", "timestamp")

        # Filter out rows where 'title' or 'content' are null
                        df_filtered   =   df_final.filter(col('title').isNotNull()   &
col('content').isNotNull())

        # Clean text fields via the provided UDFs
        cleaned_df = df_filtered \
            .withColumn("cleaned_title", clean_text_udf(col("title"))) \
              .withColumn("cleaned_description", clean_text_udf(col("description")))
\
            .withColumn("cleaned_content", clean_text_udf(col("content")))

        # Combine text fields into one field for analysis.
        combined_df = cleaned_df.withColumn("combined_text",
                                       F.when((col("cleaned_content").isNotNull())  &
(col("cleaned_content") != ""),
                               col("cleaned_content"))
                                       .when((col("cleaned_description").isNotNull())  &
(col("cleaned_description") != ""),
                                            F.concat(col("cleaned_title"),  lit(".  "),
col("cleaned_description")))
```

```
                    .otherwise(col("cleaned_title")))

        # Split the cleaned content into sentences using the provided UDF
                            df_sentences    =    combined_df.withColumn("sentences",
split_sentences_udf(col("cleaned_content")))

        # Explode the sentences array so that each sentence gets its own row
        df_exploded = df_sentences.select(
            "publishedAt",
            "cleaned_title",
            "cleaned_content",
            "cleaned_description",
            "url",
            F.explode(col("sentences")).alias("sentence")
        )

        return df_exploded

    def process_batch(self, df, epoch_id):
        """
        Process each batch of streaming data using distributed operations.

        Args:
            df (DataFrame): The batch DataFrame
            epoch_id (int): The epoch ID
        """
        try:
            print(f"\nProcessing batch {epoch_id}")

            # Rename the column to match model input expectation
            # (Assumes the model expects a column named "processed_sentence")
            df_for_model = df.withColumnRenamed("sentence", "processed_sentence")

            # Apply the sentiment model in one distributed transformation
            predictions_df = self.sentiment_predictor.model.transform(df_for_model)

              map_sentiment_udf = udf(lambda score: int(score) - 3 if score is not
None else None, IntegerType())

             # Apply mapping directly in a single select operation (more efficient)
            Mapped_df = predictions_df.select(
                "publishedAt",
                "cleaned_title",
                "cleaned_content",
                "cleaned_description",
                "url",
                "processed_sentence",
                map_sentiment_udf(col("prediction")).alias("prediction")
            )
            # Optionally cache the DataFrame if reused and then show the results
            Mapped_df.cache()
            Mapped_df.show(truncate=True)
```

```
            # If you need to use the results in the driver for further processing,
            # collect them in a batch rather than row-by-row processing.
            batch_results = Mapped_df.rdd.map(lambda row: (
                row.publishedAt,
                row.cleaned_title,
                row.cleaned_content,
                row.cleaned_description,
                row.url,
                row.processed_sentence,
                int(row.prediction)
            )).collect()

            if batch_results:
                self.final_results.extend(batch_results)

        except Exception as e:
            print(f"Error processing batch {epoch_id}: {str(e)}")

    def start_streaming(self, clean_text_udf, split_sentences_udf, news_fetcher):
        """
        Start the streaming query and continuously process data.

        Args:
            clean_text_udf: UDF for cleaning text.
            split_sentences_udf: UDF for splitting text into sentences.
            news_fetcher: News fetcher instance used for external control.
        """
        try:
            # Create Kafka stream
            df_kafka = self.create_kafka_stream()

               # Transform raw Kafka stream into structured and exploded sentence
DataFrame
             df_exploded = self.create_streaming_dataframe(df_kafka, clean_text_udf,
split_sentences_udf)

            # Use foreachBatch to handle each micro-batch in a distributed manner
            query = df_exploded.writeStream \
                .foreachBatch(self.process_batch) \
                .outputMode("append") \
                .start()

            # Monitor the query periodically; stop if news_fetcher signals to stop
            while query.isActive:
                time.sleep(10)  # Check every 10 seconds
                if not news_fetcher.running:
                    print("Fetcher is not running.")
                          self.logger.info("Fetcher has stopped; stopping streaming
query")
                    time.sleep(30)
                    query.stop()
```

```
                    break

            query.awaitTermination()

        except Exception as e:
            print(f"Error in streaming: {str(e)}")

    def get_results_dataframe(self):
        """
        Get the consolidated DataFrame of processed results.

        Returns:
                DataFrame: A DataFrame with all processed results, coalesced into a
single partition.
        """
        if self.final_results:
            result_schema = SchemaDefinitions.get_result_schema()
                        result_df  =  self.spark.createDataFrame(self.final_results,
schema=result_schema)
            return result_df.coalesce(1)
        return None
```

Class: TextCleaner
Description: The TextCleaner class provides functionalities for improving the quality of the text by removing unnecessary elements, breaking up text into individual sentences, and retrieving and cleaning news content from a given URL. It effectively removes unnecessary parts and reassembles the content into a coherent structure.

```
import re
import requests
from bs4 import BeautifulSoup
import logging

# Setup logging configuration at module level if desired
logging.basicConfig(level=logging.INFO)

class TextCleaner:
    # List of common non-article keywords to filter out (from Task 1)
    NON_ARTICLE_KEYWORDS = [
            "subscription", "subscribe", "comment", "comments", "create a display
name", "Follow Al Jazeera English",
          "Sponsored", "edited by", "Sign up", "name", "email", "website", "news",
"offer",
              "Email address", "Follow", "info", "Your bid", "proceed", "inbox",
"receive", "Thank you for your report!",
         "Your daily digest", "Search", "Review", "Reviews", "Car Launches", "Driven
Communications Sdn. Bhd.", "200801035597 (836938-P)",
               "Follow", "Email address", "Sign up", "For more of the latest",
"subscribing", "2025 Hearst Magazines, Inc. .",
          "Connect", "enjoy", "love", "Best", "The Associated Press", "NBCUniversal
Media, LLC",
             "Reporting by", "Contact", "ResearchAndMarkets.com", "Advertisement",
"thank you"
    ]

    # List of common FAQ-related terms to filter out (from Task 1)
    FAQ_KEYWORDS = [
             "faq", "frequently asked questions", "how to", "questions", "help",
"contact us", "support", "terms and conditions",
           "privacy policy", "cookie policy", "all rights reserved", "disclaimer",
"sitemap", "legal", "copyright"
    ]

    logger = logging.getLogger(__name__)

    """
    Provides methods to clean input text, split it into sentences,
    and fetch and clean article content.
    """

    @staticmethod
    def clean_text(text):
        """
```

```
        Cleans the input text.
        """
        if text is None or not isinstance(text, str):
            return ""

        text = re.sub(r'[^\x00-\x7F]+', '', text)
        text = re.sub(
               r'(©|All Rights Reserved|Privacy Policy|Terms and Conditions|Cookie
Policy|Disclaimer)',
            '', text
        )
        text = re.sub(r'[^a-zA-Z0-9\s.,?!;:()\'"-]', '', text)

        for keyword in TextCleaner.NON_ARTICLE_KEYWORDS:
            text = re.sub(
                r'[^.?!]*\b' + re.escape(keyword) + r'\b[^.?!]*[.?!]',
                '', text, flags=re.IGNORECASE
            )

        for faq in TextCleaner.FAQ_KEYWORDS:
            text = re.sub(
                r'\b' + re.escape(faq) + r'\b',
                '', text, flags=re.IGNORECASE
            )

        text = re.sub(r'\s+', ' ', text).strip()
        return text

    @staticmethod
    def split_into_sentences(text):
        """
        Splits text into sentences.
        """
        if text is None or not isinstance(text, str) or not text.strip():
            return []

        sentences = re.split(r'(?<=[.!?])\s+(?=[A-Z])', text)
         sentences = [s.strip() for s in sentences if s.strip() and len(s.strip()) >
10]
        return sentences

    @staticmethod
    def fetch_and_clean_article_content(url):
        """
        Fetches article content from a URL and cleans it.
        """
        try:
            response = requests.get(url, timeout=30)

            # Skip if status code is 429, 401, or 405
            if response.status_code in [429, 401, 405]:
                return ""
```

```
            # Check if the content type is HTML and response is OK
                  if 'text/html' in response.headers.get('Content-Type', '') and
response.status_code == 200:
                soup = BeautifulSoup(response.text, 'html.parser')

                # Remove non-article sections
                for footer in soup.find_all(
                    ['footer', 'aside', 'nav', 'form', 'section', 'div', 'span'],
                                class_=['subscription', 'newsletter', 'comments',
'related-articles',
                                  'advertisement', 'popup', 'banner', 'sponsored',
'more-articles',
                            'alerts', 'social-media']
                ):
                    footer.decompose()

                # Remove image captions, metadata, or unnecessary content
                for img in soup.find_all(['img', 'figure', 'figcaption']):
                    img.decompose()

                # Remove specific non-article content
                for non_article in soup.find_all(
                    ['div', 'span', 'section'],
                              class_=['topic', 'acknowledgment', 'external-source',
'time-zone',
                           'multilingual', 'search', 'alerts']
                ):
                    non_article.decompose()

                # Remove promotional content
                for promo in soup.find_all(
                    ['div', 'span', 'section'],
                    class_=['manage-alerts', 'article-commenting', 'breaking-news',
                           'article-comments', 'affiliate-links']
                ):
                    promo.decompose()

                # Remove all header tags
                for header in soup.find_all(['h1', 'h2', 'h3', 'h4', 'h5', 'h6']):
                    header.decompose()

                # Remove all <a> tags (hyperlinks)
                for a_tag in soup.find_all('a'):
                    a_tag.decompose()

                # Remove all <li> tags (list items)
                for li_tag in soup.find_all('li'):
                    li_tag.decompose()

                # Find the first <p> tag and the last <p> tag
                first_paragraph = soup.find('p')
```

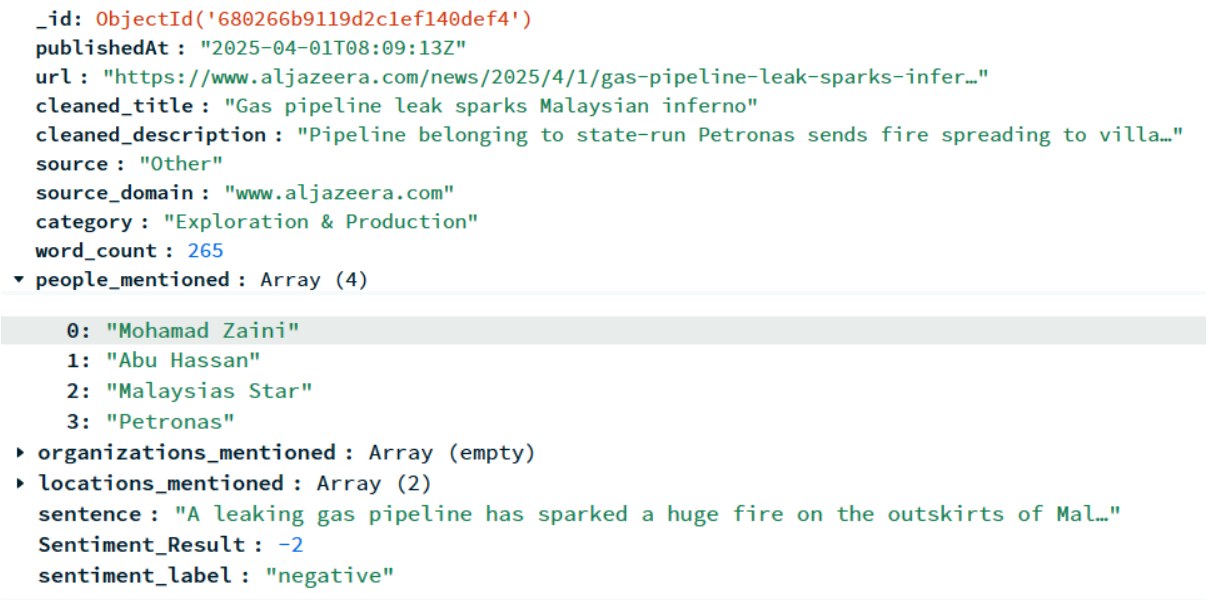
```
                all_paragraphs = soup.find_all('p')
                last_paragraph = all_paragraphs[-1] if all_paragraphs else None

                if first_paragraph and last_paragraph:
                    page_text = "\n".join(
                        [para.get_text(separator=' ', strip=True)
                         for para in first_paragraph.find_all_next('p')
                         if para != last_paragraph and para.get_text(strip=True)]
                    )
                         page_text += "\n" + last_paragraph.get_text(separator=' ',
strip=True)
                else:
                    page_text = ""

                # Use the clean_text static method via the class
                return TextCleaner.clean_text(page_text)
        except Exception as e:
              TextCleaner.logger.error(f"Error fetching article content from {url}:
{str(e)}")

        return ""
```

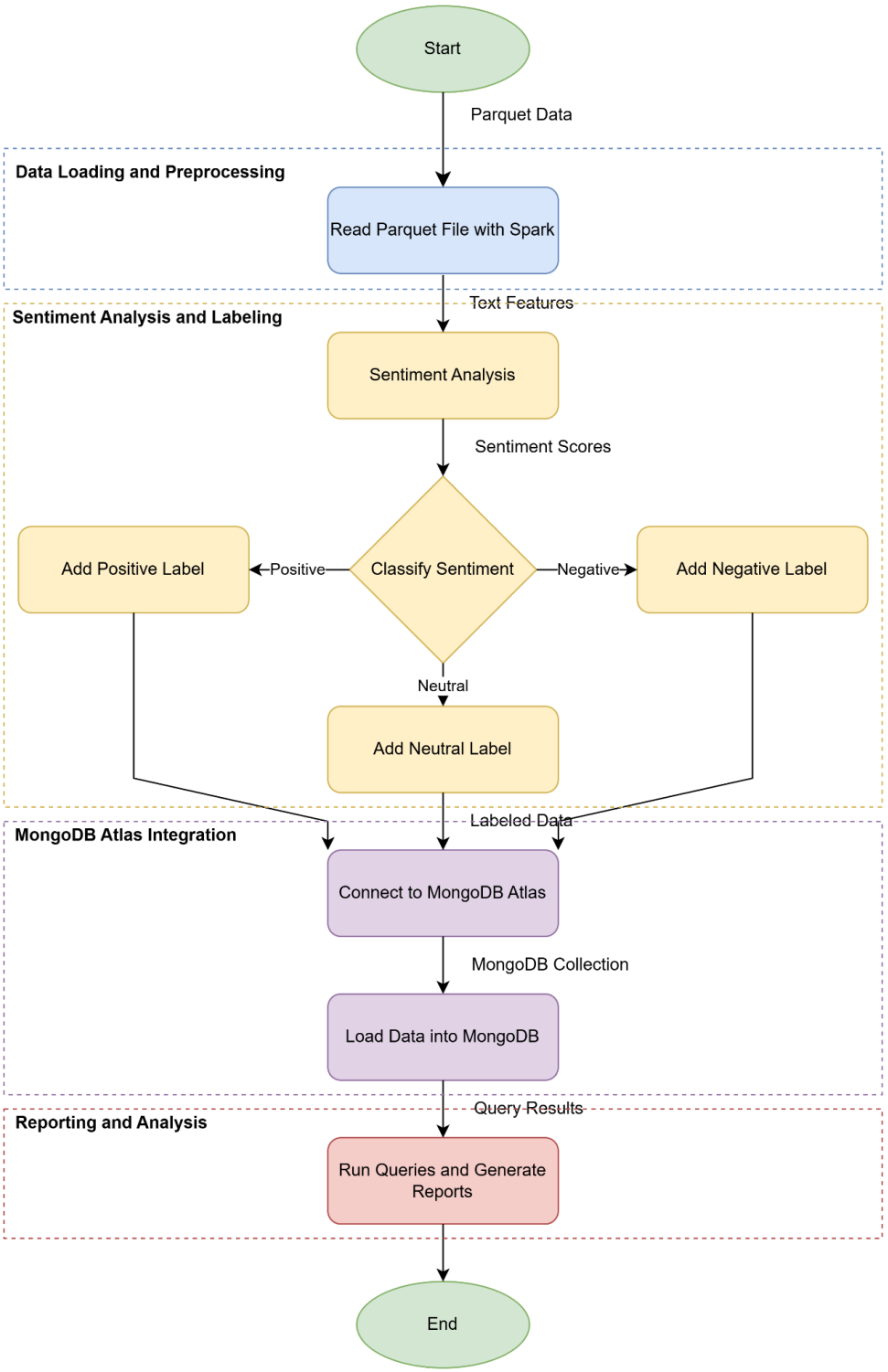
4. Querying and Reporting

4.1. MongoDB Data Model Design

```
publishedAt: string (nullable = true)
url: string (nullable = true)
cleaned_title: string (nullable = true)
cleaned_description: string (nullable = true)
source: string (nullable = true)
source_domain: string (nullable = true)
category: string (nullable = true)
word_count: integer (nullable = true)
people_mentioned: array (nullable = true)
 |-- element: string (containsNull = true)
organizations_mentioned: array (nullable = true)
 |-- element: string (containsNull = true)
locations_mentioned: array (nullable = true)
 |-- element: string (containsNull = true)
project_names: array (nullable = true)
 |-- element: string (containsNull = true)
financial_figures: array (nullable = true)
 |-- element: string (containsNull = true)
dates_mentioned: array (nullable = true)
 |-- element: string (containsNull = true)
sentence: string (nullable = true)
Sentiment_Result: integer (nullable = true)
```

4.2. Diagram with Example of Values in MongoDB




Start
Parquet Data
Data Loading and Preprocessing
Read Parquet File with Spark
Text Features
Sentiment Analysis and Labeling
Sentiment Analysis
Sentiment Scores
Add Positive Label
Positive
Classify Sentiment
Negative
Add Negative Label
Neutral
Add Neutral Label
Labeled Data
MongoDB Atlas Integration
Connect to MongoDB Atlas
MongoDB Collection
Load Data into MongoDB
Query Results
Reporting and Analysis
Run Queries and Generate Reports
End

4.3. List of Python classes:

Name of Python classes	Author
StoreMongoDBData	Soo Hong Lik
Query_and_Report	Soo Hong Lik
StoreData	Soo Hong Lik

4.4. Code for Python Classes:

Class: StoreMongoDBData
Description: This class, StoreMongoDBData, facilitates the storage of data, particularly from Parquet files with optional sentiment labeling via PySpark, into a designated MongoDB collection and offers a function to clear the collection.

```
from pyspark.sql import SparkSession
from pymongo import MongoClient
from pyspark.sql.functions import col, when, lit
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
import json
import time
import pprint  # Import pprint for pretty-printing


class StoreMongoDBData:
    def __init__(self, MONGO_URI, DATABASE_NAME, COLLECTION_NAME):
        """
        Initializes the StoreMongoDBData class.
        """
        self.mongo_uri = MONGO_URI
        self.database_name = DATABASE_NAME
        self.collection_name = COLLECTION_NAME

        # Initialize MongoDB client
        self.client = MongoClient(MONGO_URI)
        self.db = self.client[DATABASE_NAME]
        self.collection = self.db[COLLECTION_NAME]

        self.ping() # Ping MongoDB on initialization

    # Ping MongoDB to check the connection and output current database and
collections
    def ping(self):
        """
        Ping MongoDB to check if the connection is active.
        """
        try:
            # Perform the ping to check connection
            self.client.admin.command('ping')
            print("MongoDB is connected!")

            # Print current database name
            print("Current database: " + self.db.name)

            # Print collections in the current database
            print(f"Collections in {self.db.name}: ")
```

```
            pprint.pprint(self.db.list_collection_names())  # Pretty print
collection names

        except ConnectionFailure as e:
            print(f"Error: Unable to connect to MongoDB - {e}")


    # # Step 2: Read the Parquet file into a DataFrame
    # def read_parquet(self, spark, parquet_file_path):
    #     """
    #     Reads the Parquet file into a Spark DataFrame.
    #     """
    #     try:
    #         print(f"Reading Parquet file from: {parquet_file_path}")
    #         df = self.spark.read.parquet(parquet_file_path)
    #         print(f"Successfully read the Parquet file: {parquet_file_path}")

    #         # Print the schema of the DataFrame to check structure
    #         print("DataFrame schema:")
    #         df.printSchema()

    #         # Print the DataFrame to inspect the data (top 20 rows)
    #         print("DataFrame preview (top 20 rows):")
    #         df.show(5, truncate=False)  # Display the first 20 rows without
truncation
    #         return df

    #     except Exception as e:
    #         print(f"Error reading Parquet file: {e}")
    #         return None

    # Step 3: Optional transformation - Adding sentiment label
(positive/negative/neutral) based on sentiment score
    def _add_sentiment_label(self, df):
        """
        Adds a sentiment label based on the sentiment score:
        - Positive (Sentiment_Result >= 1)
        - Neutral (Sentiment_Result == 0)
        - Negative (Sentiment_Result < 0)
        """
        if "Sentiment_Result" not in df.columns:
            print("Warning: 'Sentiment_Result' column not found. Cannot add
sentiment label.")
            return df

        df_transformed = df.withColumn("sentiment_label",
                                       when(col("Sentiment_Result") >= 1,
"positive")
                                       .when(col("Sentiment_Result") == 0,
"neutral")
                                       .otherwise("negative"))
        return df_transformed
```

```
    def load_parquet_and_store_to_mongodb(self, spark: SparkSession,
parquet_file_path):
        """
        Reads a Parquet file using the provided SparkSession,
        adds a sentiment label, and loads the data into MongoDB.
        """
        try:
            print(f"Reading Parquet file from: {parquet_file_path}")
            df = spark.read.parquet(parquet_file_path)
            print(f"Successfully read the Parquet file: {parquet_file_path}")
            print("DataFrame schema:")
            df.printSchema()
            df.show(5, truncate=False)

            df_transformed = self._add_sentiment_label(df)

            if df_transformed is not None and df_transformed.count() > 0:
                record_count = df_transformed.count()
                print(f"Writing {record_count} records to MongoDB collection:
{self.database_name}.{self.collection_name}...")

                write_options = {
                    "uri": self.mongo_uri,
                    "database": self.database_name,
                    "collection": self.collection_name,
                    "mode": "append"
                }
                df_transformed.write.format("mongodb") \
                    .options(**write_options) \
                    .mode(write_options["mode"]) \
                    .save()
                print(f"Successfully loaded {record_count} records into MongoDB.")
                return record_count
            else:
                print("No data to load into MongoDB after transformation.")
                return 0

        except Exception as e:
            print(f"An error occurred during the load process: {e}")
            return 0

    # Delete all documents in the collection
    def delete_all_documents(self):
        """
        Delete all documents in the collection.
        """
        try:
            result = self.collection.delete_many({})  # Empty filter matches all
documents
            print(f"Documents deleted: {result.deleted_count}")
```

```
        except Exception as e:
            print(f"Error deleting documents: {e}")
```

Class: Query_and_Report
Description: This class, Query_and_Report, provides methods for querying and generating reports on sentiment-annotated news articles stored in MongoDB, enabling analysis based on various criteria such as sentiment, location, category, keywords, and time.

```
from pymongo import MongoClient
from datetime import datetime
import pandas as pd
from collections import Counter

class Query_and_Report:
    def __init__(self, MONGO_URI, DATABASE_NAME, COLLECTION_NAME):
        """
        Initialize the MongoDB client and connection to the specific database and
collection.
        """
        # Initialize MongoDB connection
        self.client = MongoClient(MONGO_URI)
        self.database_name = self.client[DATABASE_NAME]
        self.collection_name = self.database_name[COLLECTION_NAME]

    def get_positive_sentiment_sentences_by_articles(self):
        # MongoDB Aggregation Pipeline
        pipeline = [
            # Step 1: Match documents where sentiment_label is positive and
Sentiment_Result > 0
            {"$match": {
                "sentiment_label": "positive",
                "Sentiment_Result": {"$gt": 0}  # You can adjust the condition as
per requirement
            }},

            # Step 2: Group by title, collect sentences, calculate positive
sentence count, and calculate average sentiment
            {"$group": {
                "_id": "$cleaned_title",  # Group by article title
                "sentences": {"$push": "$sentence"},  # Collect all sentences for
each article
                "positive_count": {"$sum": 1},  # Count of positive sentences
                "average_sentiment": {"$avg": "$Sentiment_Result"},  # Calculate
the average sentiment score
                "sentiment_results": {"$push": "$Sentiment_Result"}  # Collect all
sentiment results for each sentence
            }},

            # Step 3: Sort by the count of positive sentences, descending
            {"$sort": {"positive_count": -1}}  # Sort by the number of positive
sentences
```

```
        ]

        # Step 4: Execute the aggregation pipeline
        positive_sentiment_sentences_by_articles =
self.collection_name.aggregate(pipeline)

        # Display results with complexity
        print("Positive Sentiment Articles (sorted by positive sentence count):")
        for doc in positive_sentiment_sentences_by_articles:
            # Display title, total sentences, sentiment label, and sentiment
result (average)
            print(f"\nTitle: {doc['_id']}")
            print(f"Total Sentences: {len(doc['sentences'])}")
            print(f"Sentiment Label: All Positive")

            # Display total counts for sentiment results 1 and 2
            sentiment_result_1_count = sum(1 for res in doc['sentiment_results']
if res == 1)
            sentiment_result_2_count = sum(1 for res in doc['sentiment_results']
if res == 2)
            print(f"Total Sentiment Result 1: {sentiment_result_1_count}")
            print(f"Total Sentiment Result 2: {sentiment_result_2_count}")

            # Print the table of sentences with sentiment result for each sentence
            print(f"\n{'#':<5} {'Sentence':<100} {'Sentiment Result':<20}")
            print("="*120)  # Separator for table rows

            # Enumerate through the sentences and print each with its sentiment
result
            for idx, (sentence, sentiment_result) in
enumerate(zip(doc['sentences'], doc['sentiment_results']), 1):
                print(f"{idx:<5} {sentence:<100} {sentiment_result:<20}")

            # Print the average sentiment result
            print(f"\nAverage Sentiment Result: {doc['average_sentiment']}")

            # Print a separator after each document for better readability
            print("=" * 120)
        return positive_sentiment_sentences_by_articles

    def get_negative_sentiment_sentences_by_articles(self):
        # MongoDB Aggregation Pipeline
        pipeline = [
            # Step 1: Match documents where sentiment_label is negative and
Sentiment_Result < 0
            {"$match": {
                "sentiment_label": "negative",
                "Sentiment_Result": {"$lt": 0}  # Adjust condition to negative
sentiment
            }},
```

```
            # Step 2: Group by title, collect sentences, calculate negative
sentence count, and calculate average sentiment
            {"$group": {
                "_id": "$cleaned_title",  # Group by article title
                "sentences": {"$push": "$sentence"},  # Collect all sentences for
each article
                "negative_count": {"$sum": 1},  # Count of negative sentences
                "average_sentiment": {"$avg": "$Sentiment_Result"},  # Calculate
the average sentiment score
                "sentiment_results": {"$push": "$Sentiment_Result"}  # Collect all
sentiment results for each sentence
            }},

            # Step 3: Sort by the count of negative sentences, descending
            {"$sort": {"negative_count": -1}}  # Sort by the number of negative
sentences
        ]

        # Step 4: Execute the aggregation pipeline
        negative_sentiment_sentences_by_articles =
self.collection_name.aggregate(pipeline)

        # Display results with complexity
        print("Negative Sentiment Articles (sorted by negative sentence count):")
        for doc in negative_sentiment_sentences_by_articles:
            # Display title, total sentences, sentiment label, and sentiment
result (average)
            print(f"\nTitle: {doc['_id']}")
            print(f"Total Sentences: {len(doc['sentences'])}")
            print(f"Sentiment Label: All Negative")

            # Display total counts for sentiment results -1 and -2
            sentiment_result_neg1_count = sum(1 for res in
doc['sentiment_results'] if res == -1)
            sentiment_result_neg2_count = sum(1 for res in
doc['sentiment_results'] if res == -2)
            print(f"Total Sentiment Result -1: {sentiment_result_neg1_count}")
            print(f"Total Sentiment Result -2: {sentiment_result_neg2_count}")

            # Print the table of sentences with sentiment result for each sentence
            print(f"\n{'#':<5} {'Sentence':<100} {'Sentiment Result':<20}")
            print("="*120)  # Separator for table rows

            # Enumerate through the sentences and print each with its sentiment
result
            for idx, (sentence, sentiment_result) in
enumerate(zip(doc['sentences'], doc['sentiment_results']), 1):
                print(f"{idx:<5} {sentence:<100} {sentiment_result:<20}")

            # Print the average sentiment result
            print(f"\nAverage Sentiment Result: {doc['average_sentiment']}")
```

```
            # Print a separator after each document for better readability
            print("=" * 120)

        return negative_sentiment_sentences_by_articles


    ### Q3. Find articles with neutral or positive sentiment label and specific
locations like 'Kuala Lumpur' or 'Malaysia
    def find_neutral_or_positive_sentiment_articles_with_location(self,
location_1, location_2, n):
        """
        Find articles with neutral or positive sentiment label and specific
locations like 'Kuala Lumpur' or 'Malaysia'.
        """
        pipeline = [
            # Step 1: Match articles where sentiment_label is either "neutral" or
"positive" and location is either "Kuala Lumpur" or "Malaysia"
            {"$match": {
                "sentiment_label": {"$in": ["neutral", "positive"]},  # Filter
articles with neutral or positive sentiment
                "locations_mentioned": {"$in": [location_1, location_2]}  # Filter
by the specified locations (either "Kuala Lumpur" or "Malaysia")
            }},

            # Step 2: Sort articles by published date (most recent first)
            {"$sort": {"publishedAt": -1}},  # Sort by publishedAt in descending
order

            # Step 3: Limit to the top n most recent articles
            {"$limit": n},

            # Step 4: Project the necessary fields (optional, you can adjust based
on your needs)
            {"$project": {
                "_id": 0,  # Do not include the MongoDB _id field
                "title": "$cleaned_title",
                "cleaned_description": 1,
                "publishedAt": 1,
                "sentiment_label": 1,
                "locations_mentioned": 1,
                "sentence": 1
            }}
        ]

        # Execute the aggregation pipeline
        neutral_or_positive_sentiment_articles_with_location =
list(self.collection_name.aggregate(pipeline))

        # Display results
        print(f"Found {len(neutral_or_positive_sentiment_articles_with_location)}
neutral or positive sentiment articles with location '{location_1}' or
'{location_2}'")
```

```
        for idx, article in
enumerate(neutral_or_positive_sentiment_articles_with_location, 1):
            print("=" * 50)  # Separator for readability
            print(f"\nArticle {idx}:")
            print(f"Title: {article['title']}")
            print(f"Description: {article['cleaned_description']}")
            print(f"Published At: {article['publishedAt']}")
            print(f"Sentiment Label: {article['sentiment_label']}")
            print(f"Locations Mentioned: {',
'.join(article['locations_mentioned'])}")
            print(f"Sentence: {article['sentence']}\n")

        return neutral_or_positive_sentiment_articles_with_location


    ### Q4. Find the most common categories associated with a given sentiment
label
    def get_most_common_categories(self, sentiment_label, top_n):
        # Aggregate query to get category count based on sentiment label
        pipeline = [
            {"$match": {"sentiment_label": sentiment_label}},
            {"$group": {
                "_id": "$category",
                "count": {"$sum": 1}
            }},
            {"$sort": {"count": -1}},  # Sort by category count in descending
order
            {"$limit": top_n}  # Limit to top_n categories
        ]

        most_common_categories = list(self.collection_name.aggregate(pipeline))

        # Display the most common categories in a neat tabular format
        print(f"\nTop {top_n} categories associated with {sentiment_label}
sentiment:")

        # Convert the result into a pandas DataFrame for better readability
        df = pd.DataFrame(most_common_categories)
        df.columns = ['Category', 'Count']  # Rename columns

        # Display the table
        print(df.to_string(index=False))

        return most_common_categories


    ### Q5. Latest Sentiment Trend: retrieve articles sorted by the published date
(from current to old)
    def get_articles_by_published_date(self, start_date, end_date, limit):
        # Build the match conditions based on the provided dates (if any)
        match_conditions = {}
        if start_date and end_date:
```

```
            match_conditions = {
                "publishedAt": {"$gte": start_date, "$lte": end_date}
            }

        # MongoDB aggregation pipeline
        pipeline = [
            # Step 1: Match articles by date range and other criteria if necessary
            {"$match": match_conditions},  # Only articles within the date range

            # Step 2: Add additional fields if needed (e.g., formatted date,
sentiment group, etc.)
            {"$project": {
                "publishedAt": 1,
                "cleaned_title": 1,
                "cleaned_description": 1,
                "sentiment_label": 1,
                "source": 1,
                "category": 1,
                "sentence": 1,
                "url": 1,
                "Sentiment_Result": 1  # Add the Sentiment_Result for each
sentence
            }},

            # Step 3: Group by title and description
            {"$group": {
                "_id": {"title": "$cleaned_title", "description":
"$cleaned_description"},  # Group by title and description
                "sentiment_labels": {"$addToSet": "$sentiment_label"},  # Collect
all sentiment labels for the article
                "sentences": {"$push": "$sentence"},  # Collect all sentences for
each article
                "sentiment_results": {"$push": "$Sentiment_Result"},  # Collect
all sentiment results for each sentence
                "sources": {"$addToSet": "$source"},  # Collect all sources for
the title
                "categories": {"$addToSet": "$category"},  # Collect all
categories for the title
                "publishedAt": {"$first": "$publishedAt"},  # Get the first
published date
                "url": {"$first": "$url"}  # Get the first URL
            }},

            # Step 4: Add a field for the total sentence count and calculate the
average sentiment result
            {"$addFields": {
                "sentence_count": {"$size": "$sentences"},  # Calculate total
sentence count
                "average_sentiment_result": {
                    "$avg": "$sentiment_results"  # Calculate average sentiment
result based on Sentiment_Result
                }
```

```
            }},

            # Step 5: Sort by the most recent published date first (from current
to old)
            {"$sort": {"publishedAt": -1}},  # Sort by published date in
descending order

            # Step 6: Limit the number of articles
            {"$limit": limit}
        ]

        # Execute the aggregation pipeline
        articles_by_published_date =
list(self.collection_name.aggregate(pipeline))

        # Display the results
        if articles_by_published_date:
            print(f"Articles sorted by Published Date:")
            for article in articles_by_published_date:
                print("=" * 50)
                print(f"Published At: {article['publishedAt']}")
                print(f"Title: {article['_id']['title']}")
                print(f"Description: {article['_id']['description']}")
                print(f"Sentiment Labels: {',
'.join(article['sentiment_labels'])}")
                print(f"Total Sentences: {article['sentence_count']}")
                print(f"Average Sentiment Result:
{article['average_sentiment_result']}")
                print(f"Sources: {', '.join(article['sources'])}")
                print(f"Categories: {', '.join(article['categories'])}")
                print(f"URL: {article['url']}")
                print("=" * 50)
        else:
            print("No articles found based on the given criteria.")

        return articles_by_published_date


    ### Q6. Mixed Sentiment Articles with Percentiles
    def find_mixed_sentiment_articles_with_percentiles(self):
        # MongoDB aggregation pipeline to find articles with mixed sentiment and
calculate percentiles manually
        pipeline = [
            # Step 1: Group by title and collect the sentiments and count
            {"$group": {
                "_id": "$cleaned_title",  # Group by article title
                "sentiments": {"$addToSet": "$sentiment_label"},  # Collect all
unique sentiments for the article
                "count": {"$sum": 1},  # Count how many sentences belong to each
title
                "positive_count": {"$sum": {"$cond": [{"$eq": ["$sentiment_label",
"positive"]}, 1, 0]}},  # Count of positive sentiments
```

```
                "neutral_count": {"$sum": {"$cond": [{"$eq": ["$sentiment_label",
"neutral"]}, 1, 0]}},  # Count of neutral sentiments
                "negative_count": {"$sum": {"$cond": [{"$eq": ["$sentiment_label",
"negative"]}, 1, 0]}},  # Count of negative sentiments
                "sources": {"$addToSet": "$source"},  # Collect all sources for
the title
                "urls": {"$addToSet": "$url"},  # Collect all urls for the title
                "cleaned_description": {"$first": "$cleaned_description"},  # Get
the cleaned description of the article
                "publishedAt": {"$first": "$publishedAt"}  # Get the published
date of the article
            }},
            # Step 2: Match articles where there are multiple sentiments
            {"$match": {
                "$expr": {"$gt": [{"$size": "$sentiments"}, 1]}  # Ensure there
are multiple sentiments (mixed sentiment)
            }},
            # Step 3: Sort by the number of sentences in the article
            {"$sort": {"count": -1}}  # Sort by the count of sentences in
descending order
        ]

        # Execute the aggregation pipeline
        mixed_sentiment_articles = list(self.collection_name.aggregate(pipeline))

        # Display results
        print(f"Found {len(mixed_sentiment_articles)} articles with mixed
sentiment and percentiles calculated")
        for article in mixed_sentiment_articles:
            # Manually calculate percentiles for sentiment labels
            positive_count = article['positive_count']
            neutral_count = article['neutral_count']
            negative_count = article['negative_count']

            total_count = positive_count + neutral_count + negative_count

            # Calculate percentage for each sentiment label
            if total_count > 0:
                positive_percentage = (positive_count / total_count) * 100
                neutral_percentage = (neutral_count / total_count) * 100
                negative_percentage = (negative_count / total_count) * 100
            else:
                positive_percentage = neutral_percentage = negative_percentage = 0

            # Manually calculate percentiles for positive, neutral, and negative
sentiments
            # Positive Sentiment Percentile
            positive_percentile = round(positive_percentage, 2)
            # Neutral Sentiment Percentile
            neutral_percentile = round(neutral_percentage, 2)
            # Negative Sentiment Percentile
            negative_percentile = round(negative_percentage, 2)
```

```
            print(f"\nTitle: {article['_id']}")
            print(f"Sentiments: {', '.join(article['sentiments'])}")
            print(f"Sentence count: {article['count']}")
            print(f"Positive Sentiment Count: {article['positive_count']}
({positive_percentile}%)")
            print(f"Neutral Sentiment Count: {article['neutral_count']}
({neutral_percentile}%)")
            print(f"Negative Sentiment Count: {article['negative_count']}
({negative_percentile}%)")
            print(f"Description: {article['cleaned_description']}")
            print(f"Sources: {', '.join(article['sources'])}")
            print(f"URLs: {', '.join(article['urls'])}")
            print(f"Published At: {article['publishedAt']}\n")
            print("=" * 50)  # Separator after each record

        return mixed_sentiment_articles


    ### Q7.1 Articles with Strong Positive Sentiment
    def find_top_positive_sentiment_articles(self):
        """
        Find articles with the most positive sentiment and highest number of
sentences.
        """
        pipeline = [
            {"$match": {"sentiment_label": "positive"}},  # Match only positive
sentiment
            {"$group": {
                "_id": "$cleaned_title",  # Group by article title
                "positive_count": {"$sum": 1},  # Count of positive sentences
                "sentences": {"$push": "$sentence"}  # Collect all sentences
            }},
            {"$sort": {"positive_count": -1}},  # Sort by the number of positive
sentences
            {"$limit": 10}  # Limit to top 10 articles
        ]

        # Execute the aggregation pipeline
        top_positive_sentiment_articles =
list(self.collection_name.aggregate(pipeline))

        # Display results
        print(f"Top Positive Sentiment Articles:")
        for article in top_positive_sentiment_articles:
            print("=" * 50)
            print(f"\nTitle: {article['_id']}\n")
            print(f"Positive Sentiment Count: {article['positive_count']}\n")
            for idx, sentence in enumerate(article['sentences'], 1):
                print(f"Sentence {idx}: {sentence}")

        return top_positive_sentiment_articles
```

```
    ### Q7.2 Articles with Strong Negative Sentiment
    def find_top_negative_sentiment_articles(self):
        """
        Find articles with the most negative sentiment (Sentiment_Result <= -2)
and highest number of sentences.
        """
        pipeline = [
            {"$match": {"Sentiment_Result": {"$lte": -2}}},  # Match articles with
strong negative sentiment
            {"$group": {
                "_id": "$cleaned_title",  # Group by article title
                "negative_count": {"$sum": 1},  # Count of negative sentences
                "sentences": {"$push": "$sentence"}  # Collect all sentences
            }},
            {"$sort": {"negative_count": -1}},  # Sort by the number of negative
sentences
            {"$limit": 10}  # Limit to top 10 articles
        ]

        # Execute the aggregation pipeline
        top_negative_sentiment_articles =
list(self.collection_name.aggregate(pipeline))

        # Display results
        print(f"Top Negative Sentiment Articles:")
        for article in top_negative_sentiment_articles:
            print("=" * 50)
            print(f"\nTitle: {article['_id']}\n")
            print(f"Negative Sentiment Count: {article['negative_count']}\n")
            for idx, sentence in enumerate(article['sentences'], 1):
                print(f"Sentence {idx}: {sentence}")

        return top_negative_sentiment_articles


    ### Q8. Keyword Alerts in Title or Description
    def search_articles_with_sentences_by_keyword(self, keyword):
        """
        Search for articles where the title or description contains the specific
keyword.
        """
        pipeline = [
            # Step 1: Match articles containing the keyword in the title or
description
            {"$match": {
                "$or": [
                    {"cleaned_title": {"$regex": keyword, "$options": "i"}},  #
Match keyword in title, case-insensitive
                    {"cleaned_description": {"$regex": keyword, "$options": "i"}}
# Match keyword in description, case-insensitive
```

```
                ]
            }},

            # Step 2: Group by title and aggregate sentences, sentiment labels,
and counts
            {"$group": {
                "_id": "$cleaned_title",  # Group by article title
                "sentences": {"$push": "$sentence"},  # Collect all sentences
                "sentiment_labels": {"$addToSet": "$sentiment_label"},  # Collect
all unique sentiment labels
                "positive_count": {"$sum": {"$cond": [{"$eq": ["$sentiment_label",
"positive"]}, 1, 0]}},  # Count of positive sentiments
                "neutral_count": {"$sum": {"$cond": [{"$eq": ["$sentiment_label",
"neutral"]}, 1, 0]}},  # Count of neutral sentiments
                "negative_count": {"$sum": {"$cond": [{"$eq": ["$sentiment_label",
"negative"]}, 1, 0]}},  # Count of negative sentiments
                "sources": {"$addToSet": "$source"},  # Collect all sources for
the title
                "urls": {"$addToSet": "$url"}  # Collect all URLs for the title
            }},

            # Step 3: Sort by the count of sentences (can also sort by title or
sentiment count)
            {"$sort": {"positive_count": -1}}  # Sort by the number of positive
sentences first
        ]

        # Execute the aggregation pipeline
        articles_with_sentences_by_keyword =
list(self.collection_name.aggregate(pipeline))

        # Display results
        print(f"Articles containing the keyword '{keyword}':")
        for article in articles_with_sentences_by_keyword:
            print("=" * 50)
            print(f"\nTitle: {article['_id']}")
            print(f"Sentiment Labels: {', '.join(article['sentiment_labels'])}")
            print(f"Positive Sentiment Count: {article['positive_count']}")
            print(f"Neutral Sentiment Count: {article['neutral_count']}")
            print(f"Negative Sentiment Count: {article['negative_count']}")
            print(f"Sources: {', '.join(article['sources'])}")
            print(f"URLs: {', '.join(article['urls'])}")

            print("\nSentences:")
            for idx, sentence in enumerate(article['sentences'], 1):
                print(f"Sentence {idx}: {sentence}")


            print("=" * 50)  # Separator after each record


    ### Q9. Trending Topics (using content field)
```

```
    def extract_articles_by_keywords(self, keywords, start_date=None,
end_date=None, limit=10):
        """
        Extract articles containing specific keywords in their content. Allows for
filtering by date range.
        """
        # Build the match conditions for the keywords and optional date range
        match_conditions = {
            "sentence": {"$regex": "|".join(keywords), "$options": "i"}  # Match
any of the keywords in sentence (case-insensitive)
        }

        if start_date and end_date:
            match_conditions["publishedAt"] = {"$gte": start_date, "$lte":
end_date}  # Filter by date range

        # MongoDB aggregation pipeline
        pipeline = [
            # Step 1: Match articles that contain the specified keywords in the
sentence
            {"$match": match_conditions},

            # Step 2: Group by title and collect the sentences that match the
keywords
            {"$group": {
                "_id": "$cleaned_title",  # Group by article title
                "sentences": {"$push": "$sentence"},  # Collect sentences matching
the keywords
                "sentiment_label": {"$first": "$sentiment_label"},  # Take the
sentiment label of the article
                "publishedAt": {"$first": "$publishedAt"},  # Take the first
published date of the article
                "source": {"$first": "$source"},  # Collect the source of the
article
                "url": {"$first": "$url"}  # Collect the URL of the article
            }},

            # Step 3: Sort by the published date to show the most recent articles
first
            {"$sort": {"publishedAt": -1}},  # Sort by published date descending

            # Step 4: Limit the number of results
            {"$limit": limit}
        ]

        # Execute the aggregation pipeline
        articles_by_keywords = list(self.collection_name.aggregate(pipeline))

        # Display the results
        if articles_by_keywords:
            print(f"Articles containing the keywords {', '.join(keywords)}:")
            for doc in articles_by_keywords:
```

```
                print("=" * 50)
                print(f"\nTitle: {doc['_id']}")
                print(f"Published At: {doc['publishedAt']}")
                print(f"Sentiment Label: {doc['sentiment_label']}")
                print(f"Source: {doc['source']}")
                print(f"URL: {doc['url']}")

                # Show the sentences with the keywords
                print("\nSentences with the keywords:")
                for idx, sentence in enumerate(doc['sentences'], 1):
                    print(f"Sentence {idx}: {sentence}")

                print("=" * 50)  # Separator after each record
        else:
            print("No articles found for the given keywords.")

        return articles_by_keywords


    ### R1: Generate Daily Sentiment Report
    def get_daily_sentiment_report_with_percentages(self):
        """
        Generates a daily sentiment report for articles, calculates the average
sentiment score,
        and provides the percentage distribution of each sentiment label
(positive, neutral, negative).
        """
        # MongoDB aggregation pipeline to group articles by date and title, and
calculate sentiment counts
        pipeline = [
            # Step 1: Group by date and title and aggregate the sentiment counts
and average sentiment
            {"$group": {
                "_id": {
                    "date": {"$substr": ["$publishedAt", 0, 10]},  # Group by date
(YYYY-MM-DD)
                    "title": "$cleaned_title"  # Group by article title
                },
                "count": {"$sum": 1},  # Count of sentences
                "average_sentiment": {"$avg": "$Sentiment_Result"},  # Average
sentiment score
                "positive_count": {"$sum": {"$cond": [{"$eq": ["$sentiment_label",
"positive"]}, 1, 0]}},
                "neutral_count": {"$sum": {"$cond": [{"$eq": ["$sentiment_label",
"neutral"]}, 1, 0]}},
                "negative_count": {"$sum": {"$cond": [{"$eq": ["$sentiment_label",
"negative"]}, 1, 0]}}
            }},

            # Step 2: Sort by date and title
            {"$sort": {"_id.date": 1, "_id.title": 1}}  # Sort by date and then by
title
```

```
        ]

        # Execute the aggregation pipeline
        daily_sentiment_report = self.collection_name.aggregate(pipeline)

        # Print the daily sentiment report with percentage calculations
        for doc in daily_sentiment_report:
            date = doc['_id']['date']
            title = doc['_id']['title']
            total_sentences = doc['count']
            positive_count = doc['positive_count']
            neutral_count = doc['neutral_count']
            negative_count = doc['negative_count']

            # Calculate percentages for each sentiment label
            positive_percentage = (positive_count / total_sentences) * 100 if
total_sentences > 0 else 0
            neutral_percentage = (neutral_count / total_sentences) * 100 if
total_sentences > 0 else 0
            negative_percentage = (negative_count / total_sentences) * 100 if
total_sentences > 0 else 0

            # Print the results
            print(f"Date: {date}, Title: {title}")
            print(f"Sentences: {total_sentences}, Average Sentiment:
{doc['average_sentiment']}")
            print(f"Positive: {positive_count} ({positive_percentage:.2f}%),
Neutral: {neutral_count} ({neutral_percentage:.2f}%), Negative: {negative_count}
({negative_percentage:.2f}%)")
            print("=" * 50)  # Separator between each record for better
readability


    ### R2. Generates Source Summary Report
    def generate_source_summary(self):
        """
        Generates a summary report for each source with sentiment counts,
percentages, and top categories.

        Returns:
        source_summary (list): A summary report for each source.
        """
        pipeline = [
            {
                "$group": {
                    "_id": "$source",
                    "total_sentences": {"$sum": 1},
                    "distinct_titles": {"$addToSet": "$cleaned_title"},
                    "all_categories": {"$push": "$category"},

                    "avg_sentiment": {"$avg": "$Sentiment_Result"},
                    "positive_count": {
```

```
                    "$sum": {
                        "$cond": [
                            {"$eq": ["$sentiment_label", "positive"]}, 1, 0
                        ]
                    }
                },
                "neutral_count": {
                    "$sum": {
                        "$cond": [
                            {"$eq": ["$sentiment_label", "neutral"]}, 1, 0
                        ]
                    }
                },
                "negative_count": {
                    "$sum": {
                        "$cond": [
                            {"$eq": ["$sentiment_label", "negative"]}, 1, 0
                        ]
                    }
                }
            }
        },
        { "$sort": {"avg_sentiment": -1} }
    ]

    source_summary = list(self.collection_name.aggregate(pipeline))

    print("Source Summary Report")
    print("="*80)
    for doc in source_summary:
        source = doc["_id"]
        total_sentences = doc["total_sentences"]
        total_articles  = len(doc["distinct_titles"])
        avg_sent        = doc["avg_sentiment"]
        pos_cnt         = doc["positive_count"]
        neu_cnt         = doc["neutral_count"]
        neg_cnt         = doc["negative_count"]

        # percentages
        pos_pct = pos_cnt / total_sentences * 100 if total_sentences else 0
        neu_pct = neu_cnt / total_sentences * 100 if total_sentences else 0
        neg_pct = neg_cnt / total_sentences * 100 if total_sentences else 0

        # top category
        cat_counts = Counter(doc["all_categories"])
        top_cat, top_cat_cnt = cat_counts.most_common(1)[0]

        # print
        print(f"Source            : {source}")
        print(f"  Total Articles       : {total_articles}")
        print(f"  Total Sentences      : {total_sentences}")
        print(f"  Avg Sentiment        : {avg_sent:.2f}")
```

```
            print(f"  Positive Count       : {pos_cnt} , Positive Percentage :
{pos_pct:.2f}%")
            print(f"  Neutral Count        : {neu_cnt} , Neutral Percentage  :
{neu_pct:.2f}%")
            print(f"  Negative Count       : {neg_cnt} , Negative Percentage :
{neg_pct:.2f}%")
            print(f"  Top Category         : {top_cat} ({top_cat_cnt}) sentences
occurrences")
            print("-"*80)


    def get_total_sentences(self):
        """
        Calculates the total number of sentences across all articles in the
collection.

        Returns:
        result (list): A list with the total sentence count.
        """
        pipeline = [
            {
                # Grouping by nothing to count all sentences
                "$group": {
                    "_id": None,  # No grouping by title
                    "total_sentences": {"$sum": 1}  # Count the total number of
sentences
                }
            }
        ]

        # Execute the aggregation pipeline
        total_sentences = self.collection_name.aggregate(pipeline)

        # Output the result
        for record in total_sentences:
            print(f"Total Sentences in the Collection:
{record['total_sentences']}")


    def get_unique_titles_count(self):
        """
        Calculates the total number of unique article titles in the collection.

        Returns:
        result (list): A list with the unique title count.
        """
        pipeline = [
            {
                # Group by title to count unique titles
                "$group": {
                    "_id": "$cleaned_title"  # Group by the cleaned title field
                }
```

```
            },
            {
                # Count the number of unique titles
                "$count": "unique_titles"
            }
        ]

        # Execute the aggregation pipeline
        unique_titles_count = self.collection_name.aggregate(pipeline)

        # Output the result
        for record in unique_titles_count:
            print(f"Total Unique Titles in the Collection:
{record['unique_titles']}")


    ### R3. Generates monthly distribution of articles, sentences, sentiment
labels
    def monthly_sentiment_distribution_by_date_range(self, start_date, end_date):
        """
        Generates monthly distribution of articles with counts per sentiment label
(positive, neutral, negative)
        for the given date range.

        Parameters:
        start_date (str): The start date for filtering articles in ISO 8601
format.
        end_date (str): The end date for filtering articles in ISO 8601 format.

        Returns:
        monthly_sentiment_distribution (list): A list of monthly sentiment
distribution results.
        """

        def convert_to_datetime(date_str):
            return datetime.strptime(date_str, "%Y-%m-%dT%H:%M:%SZ")

        start_datetime = convert_to_datetime(start_date)
        end_datetime = convert_to_datetime(end_date)

        pipeline = [
            # Step 1: Convert 'publishedAt' string to date and add year and month
based on the 'publishedAt' field
            {"$addFields": {
                "publishedAt": {"$dateFromString": {"dateString": "$publishedAt",
"format": "%Y-%m-%dT%H:%M:%SZ"}}}},
            # Step 2: Match articles by date range
            {"$match": {"publishedAt": {"$gte": start_datetime, "$lte":
end_datetime}}},
            # Step 3: Group by year, month, and sentiment label
            {"$group": {
```

```
                "_id": {"year": {"$year": "$publishedAt"}, "month": {"$month":
"$publishedAt"}, "sentiment": "$sentiment_label"},
                "count": {"$sum": 1}}},
            # Step 4: Sort by year, month, and sentiment
            {"$sort": {"_id.year": 1, "_id.month": 1, "_id.sentiment": 1}}
        ]

        # Execute the aggregation pipeline
        monthly_sentiment_distribution =
list(self.collection_name.aggregate(pipeline))

        # Display the results
        if monthly_sentiment_distribution:
            print(f"Monthly Sentiment Distribution from {start_date} to
{end_date}:")
            for doc in monthly_sentiment_distribution:
                year_month = f"{doc['_id']['year']}-{doc['_id']['month']:02d}"
                print(f"\nYear-Month: {year_month}, Sentiment:
{doc['_id']['sentiment']}, Count: {doc['count']}")

        else:
            print("No data found for the given date range.")

        return monthly_sentiment_distribution
```

Class: StoreData
Description: This code segment initializes a Spark session, reads data from a Parquet file, adds a sentiment label based on the 'Sentiment_Result' column, converts the resulting DataFrame into a list of dictionaries, and then inserts these records into a specified MongoDB collection.

```
from pyspark.sql import SparkSession
from pyspark import SparkConf
from pyspark.sql.functions import col, lit, current_timestamp, struct
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
from pymongo import MongoClient
from kafka import KafkaConsumer
import json
import time

# Step 1: Initialize Spark Session and configure MongoDB
spark = SparkSession.builder \
    .appName("Real-Time Sentiment Analysis with MongoDB") \
    .config("spark.mongodb.input.uri", uri) \
    .config("spark.mongodb.output.uri", uri) \
    .config("spark.mongodb.output.database", "sentiment_analysis_database") \
    .config("spark.mongodb.output.collection", "sentiment_dataa") \
    .getOrCreate()

# The 'collection' variable is already a MongoDB collection object from the
earlier section.
# DO NOT reassign it here.
# collection = "sentiment_dataa"

# Step 2: Read the Parquet file into a DataFrame
# Path to your uploaded Parquet file
parquet_file_path2 = 'Enriched_With_Date1.parquet'

df = spark.read.parquet(parquet_file_path2)

# Show the DataFrame (Optional) to check the structure of the data
df.show(truncate=True,vertical=True)

# Step 3: Optional transformation - Adding sentiment label
(positive/negative/neutral) based on sentiment score
from pyspark.sql.functions import when

df_transformed = df.withColumn("sentiment_label",
                               when(col("Sentiment_Result") >= 1, "positive")
                               .when(col("Sentiment_Result") == 0, "neutral")
                               .otherwise("negative"))

df_transformed.show(truncate=True)

# Step 5: Convert to a list of dictionaries
```

```
records = df_transformed.select("publishedAt", "url", "cleaned_title",
"cleaned_description", "source", "source_domain",
                                "category", "word_count", "people_mentioned",
"organizations_mentioned",
                                "locations_mentioned", "sentence",
"Sentiment_Result", "sentiment_label") \
    .rdd.map(lambda row: row.asDict()).collect()

# Example of how one document might look like with separated columns and better
readability
for record in records:  # Print the first 20 records (you can adjust the range as
needed)
    print("========================================")
    for key, value in record.items():
        # Check if the value is a list or other complex type and format
accordingly
        if isinstance(value, list):
            print(f"{key}: {', '.join(map(str, value))}")  # For lists, print the
values as a comma-separated string
        else:
            print(f"{key}: {value}")

    # Add a separator after each record for clarity
    print("\n========================================\n")

print(f"{len(records)} records.")

# Insert the records into MongoDB collection
if records:
    collection.insert_many(records)
    print(f"{len(records)} records inserted into MongoDB.")

# # Delete all documents in the collection
# result = collection.delete_many({})  # Empty filter matches all documents

 # # Check how many documents were deleted
 # print(f"Documents deleted: {result.deleted_count}")
```

4.5. Query Output

Q1. Finds sentences based on articles with positive sentiment

```
Positive Sentiment Articles (sorted by positive sentence count):

Title: REACTIONS What did teams say after the season-openingAustralian Grand Prix?
Total Sentences: 60
Sentiment Label: All Positive
Total Sentiment Result 1: 57
Total Sentiment Result 2: 3

#     Sentence                                                                                                    Sentiment Result
=========================================================================================================================================
1     Lando Norris won the Australian Grand Prix starting from pole position, but it was far from straightforward, coming at the end o
what was a thrilling race, particularly in the closing stages. 1
2     Its an incredible start to the year and the team have given me an amazing car from the get-go.       2
3     Whilst it was a tough race due to the unpredictable conditions and the pressure from Oscar and Max, I felt comfortable and confi
ent out there in the package the team have given us. 1
4     Its been a great weekend.                                                                           1
5     We now need to go to China, do it again, and continue from there.                                   1
6     I had the speed and I felt in very good shape to fight for the win.                                 1
7     Thanks to the team for all their efforts and thanks to all the Australian fans who have given me so much good energy and over th
last few days. 1
8     It means a lot.                                                                                     2
```

Q2. Finds sentences based on articles with negative sentiment

```
Negative Sentiment Articles (sorted by negative sentence count):

Title: REACTIONS What did teams say after the season-openingAustralian Grand Prix?
Total Sentences: 37
Sentiment Label: All Negative
Total Sentiment Result -1: 35
Total Sentiment Result -2: 2

#     Sentence                                                                                                  Sentiment Result
=======================================================================================================================================
1     A disappointing result after a really positive weekend.                                                   -1
2     Unfortunately, at the end, I just lost it in the sudden rain.                                             -1
3     Again, it is the first race of the season and we went for it and learnt a lot, so this is a positive start." This whole weekend w
as pretty terrible. -1
4     Today we were too slow at the start and then we gambled.                                                  -1
5     Starting from the pitlane was tough and we just didnt really have the speed in the first stint on the inter. -1
6     We took a chance staying out on the medium and hoping half the track would stay dry.                      -1
7     We knew that sector three was bad but we thought sector one would stay a little bit drier, so we risked it but it was bucketing d
```

Q3. Find articles with neutral or positive sentiment label and specific locations like 'Kuala Lumpur' or 'Malaysia'

```
Found 10 neutral or positive sentiment articles with location 'Kuala Lumpur' or 'Malaysia'
==================================================

Article 1:
Title: Putra Heights inferno: 179 witness statements recorded, 769 reports filed, say cops
Description: KUALA LUMPUR: Authorities have recorded 179 witness statements as part of investigations into the gas pipeline fire in P
ra Heights, Subang Jaya on April 1, says Selangor police chief Datuk Hussein Omar Khan. Read full story
Published At: 2025-04-12T11:06:00Z
Sentiment Label: neutral
Locations Mentioned: Malaysia, Kusel
Sentence: One temporary relief centre remains open at Masjid Putra Heights.

==================================================
```

Q4. Find the most common categories associated with a given sentiment label

```
Top 5 categories associated with positive sentiment:
               Category  Count
              Financial    268
Exploration & Production     51
          Business Deals     35
          Sustainability     32
 Technology & Innovation     19
[{'_id': 'Financial', 'count': 268},
 {'_id': 'Exploration & Production', 'count': 51},
 {'_id': 'Business Deals', 'count': 35},
 {'_id': 'Sustainability', 'count': 32},
 {'_id': 'Technology & Innovation', 'count': 19}]

Top 3 categories associated with negative sentiment:
               Category  Count
Exploration & Production    101
              Financial     76
          Sustainability     39
[{'_id': 'Exploration & Production', 'count': 101},
 {'_id': 'Financial', 'count': 76},
 {'_id': 'Sustainability', 'count': 39}]


Top 3 categories associated with neutral sentiment:
               Category  Count
              Financial    579
Exploration & Production    270
          Sustainability    235
[{'_id': 'Financial', 'count': 579},
 {'_id': 'Exploration & Production', 'count': 270},
 {'_id': 'Sustainability', 'count': 235}]
```

Q5. Latest Sentiment Trend: retrieve articles sorted by the published date (from current to old)

```
Articles sorted by Published Date:
==================================================
Published At: 2025-04-13T09:09:30Z
Title: "Much better than we expected," claims Wolff as he reflects on Mercedes' brilliant
...
Description: Following a strong performance on Saturday, Mercedes team boss Toto Wolff has
claimed that the Brackley-based outfit performed much better than they had hoped for ahead
of the qualifying session.
Sentiment Labels: neutral, positive, negative
Total Sentences: 20
Average Sentiment Result: 0.25
Sources: Other
Categories: Financial
URL: https://www.f1technical.net/news/26609
==================================================
==================================================
Published At: 2025-04-13T03:59:00Z
Title: Putra Heights inferno: 21-year-old ICU patient's condition improving, says Dzulkefly
Description: PUTRAJAYA, April 13: The condition of a man who was seriously injured in the gas pipeline
fire in Putra Heights, Subang Jaya, on April 1 is improving, says Datuk Seri Dr Dzulkefly Ahmad. Read
full story
Sentiment Labels: positive, negative, neutral
Total Sentences: 7
Average Sentiment Result: -0.42857142857142855
Sources: The Star
Categories: Exploration & Production
URL: https://www.thestar.com.my/news/nation/2025/04/13/putra-heights-inferno-21-year-old-icu-patient03
9s-condition-improving-says-dzulkefly
==================================================
```

```
==================================================
Published At: 2025-04-12T11:06:00Z
Title: Putra Heights inferno: 179 witness statements recorded, 769 reports filed, say cops
Description: KUALA LUMPUR: Authorities have recorded 179 witness statements as part of investigations
into the gas pipeline fire in Putra Heights, Subang Jaya on April 1, says Selangor police chief Datuk
Hussein Omar Khan. Read full story
Sentiment Labels: neutral, negative
Total Sentences: 10
Average Sentiment Result: -0.2
Sources: The Star
Categories: Sustainability
URL: https://www.thestar.com.my/news/nation/2025/04/12/putra-heights-inferno-179-witness-statements-re
corded-769-reports-filed-say-cops
==================================================
```

Q6. Mixed Sentiment Articles with Percentiles

```
Found 65 articles with mixed sentiment and percentiles calculated

Title: REACTIONS What did teams say after the season-openingAustralian Grand Prix?
Sentiments: negative, positive, neutral
Sentence count: 130
Positive Sentiment Count: 60   (46.15%)
Neutral Sentiment Count: 33    (25.38%)
Negative Sentiment Count: 37   (28.46%)
Description: Lando Norris won the Australian Grand Prix starting from pole position, but it was far fr
om straightforward, coming at the end of what was a thrilling race, particularly in the closing stage
s. F1Technical's senior writer Balazs Szabo delivers what teams had t
Sources: Other
URLs: https://www.f1technical.net/news/26443
Published At: 2025-03-16T18:00:00Z

==================================================


Title: Recycled Polyethylene Terephthalate (rPET) Market to Reach 21.3 Billion by 2034, Growing at a
7.2 CAGR Exactitude Consultancy
Sentiments: neutral, negative, positive
Sentence count: 121
Positive Sentiment Count: 58   (47.93%)
Neutral Sentiment Count: 62    (51.24%)
Negative Sentiment Count: 1   (0.83%)
Description: Recycled Polyethylene Terephthalate (rPET) Market Outlook 20252034 Recycled Polyethylene
Terephthalate (rPET) Market Outlook 20252034
Sources: Other
URLs: https://www.globenewswire.com/news-release/2025/03/28/3051150/0/en/Recycled-Polyethylene-Terepht
halate-rPET-Market-to-Reach-21-3-Billion-by-2034-Growing-at-a-7-2-CAGR-Exactitude-Consultancy.html
Published At: 2025-03-28T07:00:00Z

==================================================
--------------------------------------------------

Title: REACTIONS - What did teams have to say after the opening day in Bahrain?
Sentiments: neutral, positive, negative
Sentence count: 82
Positive Sentiment Count: 26   (31.71%)
Neutral Sentiment Count: 45    (54.88%)
Negative Sentiment Count: 11   (13.41%)
Description: McLaren topped the time sheet at the end of the first day of free practice for the Bahrai
n Grand Prix, with Lando Norris and Oscar Piastri having set the benchmark in both one-hour practice s
essions.
Sources: Other
URLs: https://www.f1technical.net/news/26583
Published At: 2025-04-12T08:25:00Z

==================================================
```

Q7. Articles with Strong Positive Sentiment

```
Top Positive Sentiment Articles:
==================================================

Title: REACTIONS What did teams say after the season-openingAustralian Grand P
ix?

Positive Sentiment Count: 60

Sentence 1: Lando Norris won the Australian Grand Prix starting from pole posi
ion, but it was far from straightforward, coming at the end of what was a thri
ling race, particularly in the closing stages.
Sentence 2: Its an incredible start to the year and the team have given me an
mazing car from the get-go.
Sentence 3: Whilst it was a tough race due to the unpredictable conditions and
the pressure from Oscar and Max, I felt comfortable and confident out there in
the package the team have given us.
Sentence 4: Its been a great weekend.
Sentence 5: We now need to go to China, do it again, and continue from there.
```

Q8. Articles with Strong Negative Sentiment

```
Top Negative Sentiment Articles:
==================================================

Title: Residents recall morning blast and monster fire

Negative Sentiment Count: 11

Sentence 1: The 70-year-old man was shocked as a towering mushroom cloud of fi
e shot into the sky with a deafening explosion.
Sentence 2: The towering inferno at Putra Heights was visible from kilometres
way.
Sentence 3: I was shocked because it was a monster fire, he added.
Sentence 4: Some house windows and car windows were cracked, and even broken.M
ny people fled from their houses as they were afraid, said Harban.
Sentence 5: Retiree Tan Sri Syed Zainal Abidin Syed Mohamed Tahir said he hear
the explosions, after which residents immediately evacuated from their houses
s the heat was too intense.
Sentence 6: The explosion happened at 8.12am and after that, people were just
```

Q9. Keyword Alerts in Title or Description

```
Articles containing the keyword 'Petronas':
==================================================

Title: PETRONAS to continue to put safety as priority
Sentiment Labels: neutral, negative, positive
Positive Sentiment Count: 1
Neutral Sentiment Count: 6
Negative Sentiment Count: 1
Sources: The Star
URLs: https://www.thestar.com.my/news/nation/2025/04/02/petronas-to-continue-to
-put-safety-as-priority

Sentences:
Sentence 1: KUALA LUMPUR: PETRONAS has assured the public that it will continue
to put safety as a priority following the blaze that happened after an explosio
n at its gas pipeline near Putra Heights, Subang Jaya.
Sentence 2: The safety of the surrounding community and environment and the sec
urity of the gas supply to the nation continue to be our utmost priority, the c
```

Q10. Trending Topics (using content field)

```
Articles containing the keywords fire, leak:
==================================================

Title: Putra Heights inferno: 21-year-old ICU patient's condition improving, sa
ys Dzulkefly
Published At: 2025-04-13T03:59:00Z
Sentiment Label: negative
Source: The Star
URL: https://www.thestar.com.my/news/nation/2025/04/13/putra-heights-inferno-21
-year-old-icu-patient039s-condition-improving-says-dzulkefly

Sentences with the keywords:
Sentence 1: The PETRONAS gas pipeline fire and explosioncaused flames to shoot
over 30 metres high, with temperatures reaching up to 1,000 degrees Celsius.
Sentence 2: Firemen tooknearly eight hours to extinguish the blaze.
Sentence 3: Previously, Selangor Mentri Besar Datuk Seri Amirudin Shari said th
e full report on the fire is expected to be completed by mid-May.
==================================================
```

```
total_sentences = qnr.get_total_sentences()
```

```
Total Sentences in the Collection: 1843
```

```
unique_titles_count = qnr.get_unique_titles_count()
```

```
Total Unique Titles in the Collection: 72
```

4.6. Report

R1: Generate Daily Sentiment Report

```
Date: 2025-03-13, Title: Lewis Hamilton Drove a Ferrari Convertible in a Tribute to Ferris Buellers Da
y Off
Sentences: 17, Average Sentiment: 0.23529411764705882
Positive: 4 (23.53%), Neutral: 13 (76.47%), Negative: 0 (0.00%)
================================================
Date: 2025-03-15, Title: Lintas firefighters swap helmets for chefs hats in bubur lambuk giveaway
Sentences: 10, Average Sentiment: 0.1
Positive: 2 (20.00%), Neutral: 7 (70.00%), Negative: 1 (10.00%)
================================================
Date: 2025-03-15, Title: Petronas says still in talks with Sarawak over gas aggregator role
Sentences: 16, Average Sentiment: 0.0
Positive: 0 (0.00%), Neutral: 16 (100.00%), Negative: 0 (0.00%)
================================================
Date: 2025-03-16, Title: How F1s Tech Innovations Are Changing Medicine, Athletics, Aerospace, and Mor
e
Sentences: 33, Average Sentiment: 0.5454545454545454
Positive: 13 (39.39%), Neutral: 19 (57.58%), Negative: 1 (3.03%)
================================================
```

R2. Generates Source Summary Report

```
Source Summary Report
================================================================================
Source            : Other
  Total Articles       : 43
  Total Sentences      : 1524
  Avg Sentiment        : 0.19
  Positive Count       : 379 , Positive Percentage : 24.87%
  Neutral Count        : 997 , Neutral Percentage  : 65.42%
  Negative Count       : 148 , Negative Percentage : 9.71%
  Top Category         : Financial (877) sentences occurrences
--------------------------------------------------------------------------------
Source            : The Star
  Total Articles       : 29
  Total Sentences      : 319
  Avg Sentiment        : -0.31
  Positive Count       : 26 , Positive Percentage : 8.15%
  Neutral Count        : 225 , Neutral Percentage  : 70.53%
  Negative Count       : 68 , Negative Percentage : 21.32%
```

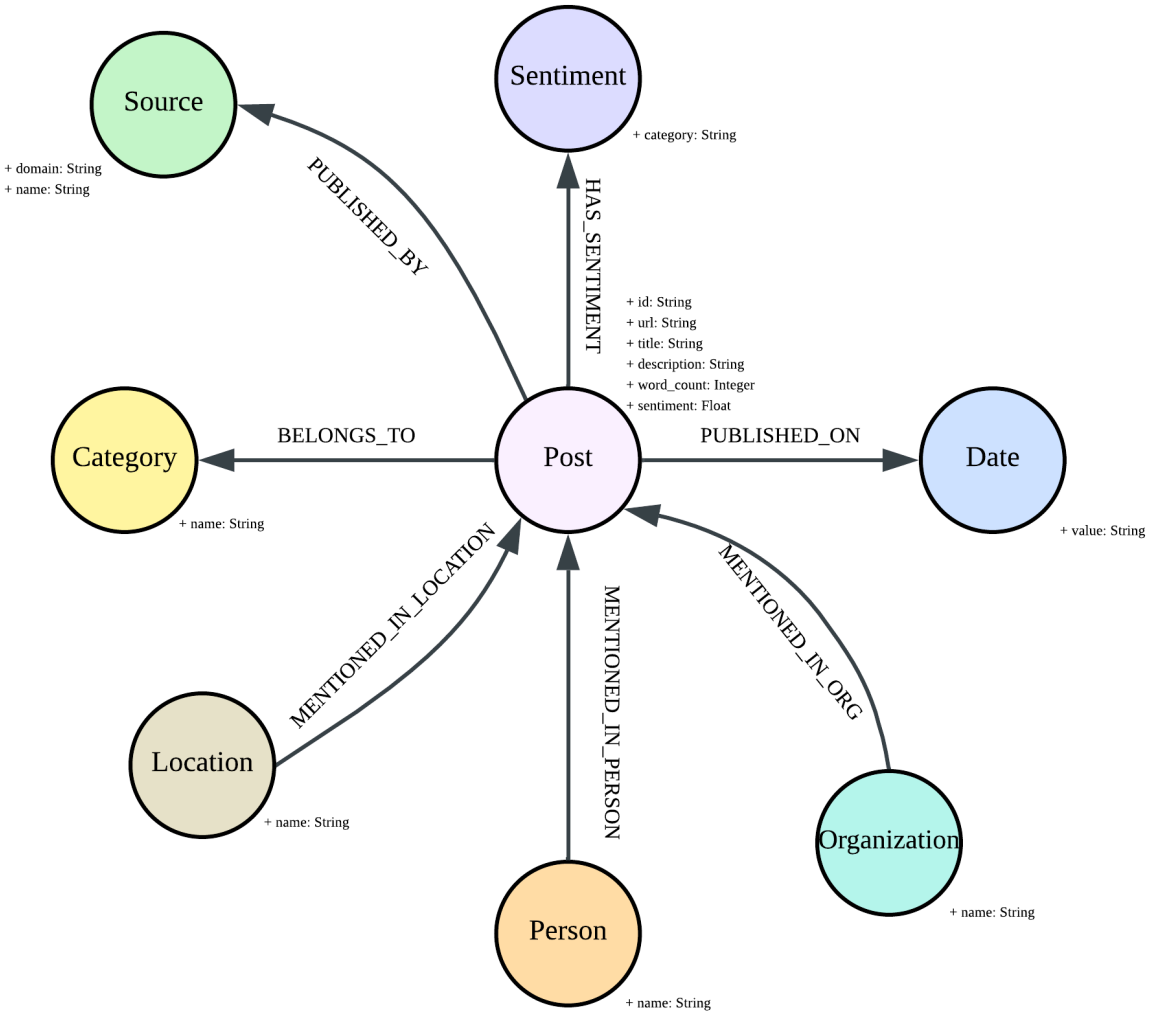
R3. Generates monthly distribution of articles, sentences, sentiment labels

```
Monthly Sentiment Distribution from 2025-03-01T00:00:00Z to 2025-04-30T23:59:59Z:

Year-Month: 2025-03, Sentiment: negative, Count: 59

Year-Month: 2025-03, Sentiment: neutral, Count: 562

Year-Month: 2025-03, Sentiment: positive, Count: 278

Year-Month: 2025-04, Sentiment: negative, Count: 157

Year-Month: 2025-04, Sentiment: neutral, Count: 660

Year-Month: 2025-04, Sentiment: positive, Count: 127
[{'_id': {'year': 2025, 'month': 3, 'sentiment': 'negative'}, 'count': 59},
 {'_id': {'year': 2025, 'month': 3, 'sentiment': 'neutral'}, 'count': 562},
 {'_id': {'year': 2025, 'month': 3, 'sentiment': 'positive'}, 'count': 278},
 {'_id': {'year': 2025, 'month': 4, 'sentiment': 'negative'}, 'count': 157},
 {'_id': {'year': 2025, 'month': 4, 'sentiment': 'neutral'}, 'count': 660},
 {'_id': {'year': 2025, 'month': 4, 'sentiment': 'positive'}, 'count': 127}]
```

5. Relationship Analysis

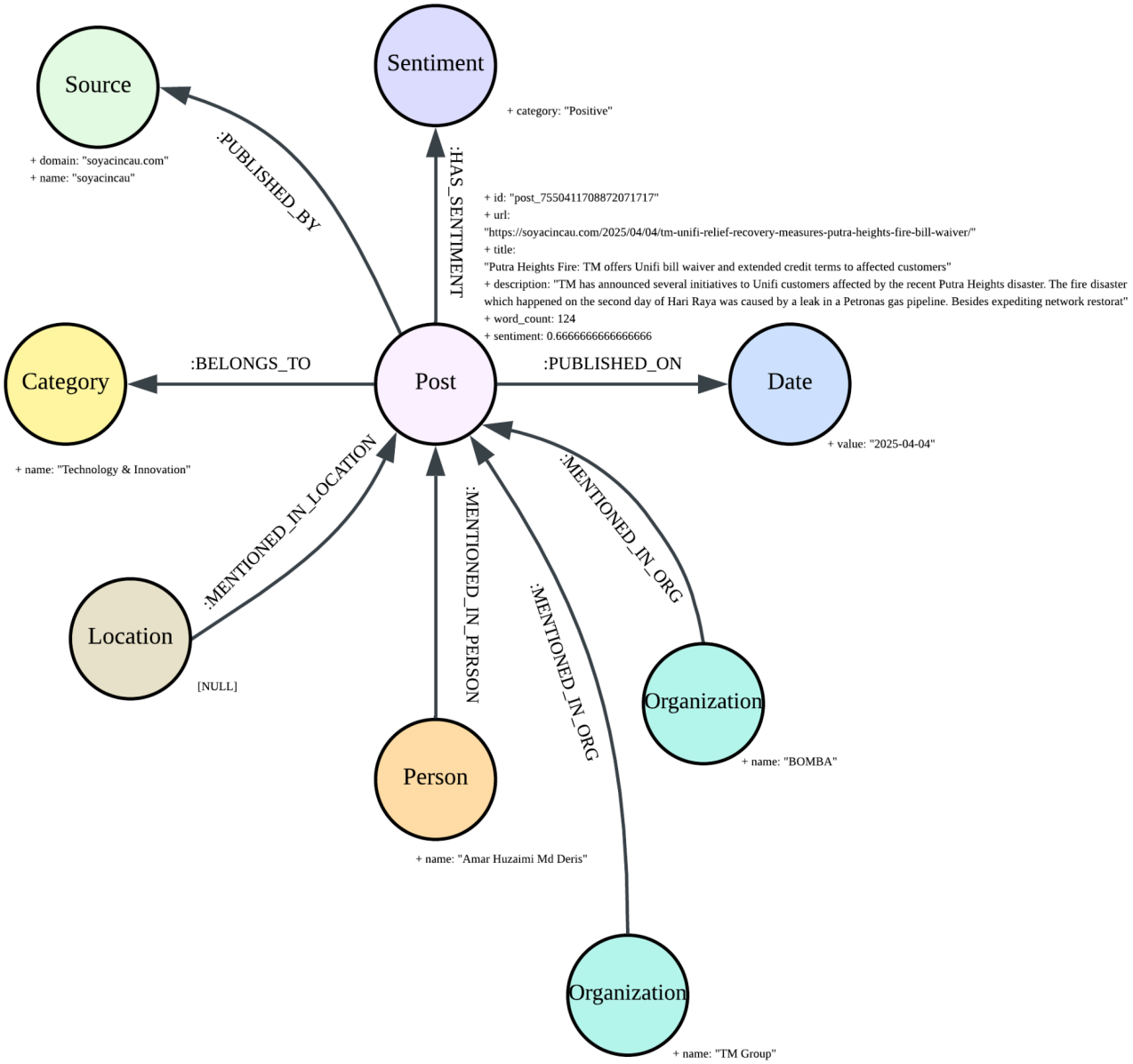
5.1. Neo4j Data Model Design

Click here to view full-resolution Neo4j Data Model Design



5.2. Diagram with Example of Values in Neo4j

Click here to view full-resolution Diagram with Example of Values in Neo4j



5.3. List of Python classes:

Name of Python classes	Author
neo4j_connector_Task5	Heng Leng Ning
processor_Task5	Heng Leng Ning
queries_Task5	Heng Leng Ning
query_runner_Task5	Heng Leng Ning

5.4. Code for Python Classes

Class: neo4j_connector_Task5
Description: neo4j_connector_Task5 class provides a utility class for managing all interactions with the Neo4j database. It encapsulates functions for creating nodes (such as Post, Person, Organization, etc.), establishing relationships between them, and enforcing uniqueness constraints. By centralising all Cypher operations, it ensures a clean separation between business logic and database communication.

```
from neo4j import GraphDatabase

class Neo4jConnector:
    def __init__(self, uri, user, password):
        self.uri = uri
        self.user = user
        self.password = password
        self.driver = GraphDatabase.driver(uri, auth=(user, password))

    def close(self):
        self.driver.close()

    @staticmethod
    def ensure_constraints(uri, user, password):
        """
        Create uniqueness constraints for all relevant labels.
        Call once at startup to avoid duplicate nodes under concurrency.
        """
        constraints = {
            "Sentiment": "category",
            "Person": "name",
            "Organization": "name",
            "Location": "name",
            "Category": "name",
            "Date": "value",
            "Source": "domain"
        }

        driver = GraphDatabase.driver(uri, auth=(user, password))
        with driver.session() as session:
            for label, property in constraints.items():
                query = f"""
                CREATE CONSTRAINT IF NOT EXISTS
                FOR (n:{label}) REQUIRE (n.{property}) IS UNIQUE
                """
                session.run(query)
        driver.close()

    # Internal helper for normalization
    @staticmethod
    def _normalize(raw: str) -> str:
        return raw.strip()
```

```
    def create_post_node(self, post_id, publishedAt, url, cleaned_title,
cleaned_description, word_count, sentiment):
        query = """
        MERGE (p:Post {id: $post_id})
        SET p.url = $url,
            p.title = $cleaned_title,
            p.description = $cleaned_description,
            p.word_count = $word_count,
            p.sentiment = $sentiment
        """
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query,
                                                    post_id=post_id,
                                                    url=url,

cleaned_title=cleaned_title,

cleaned_description=cleaned_description,

word_count=word_count,
                                                    sentiment=sentiment))

    def create_person_node(self, person: str):
        name = self._normalize(person)
        query = "MERGE (p:Person {name: $name})"
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query, name=name))

    def link_post_to_person(self, post_id: str, person: str):
        name = self._normalize(person)
        query = (
            "MATCH (p:Post {id: $post_id}), (pe:Person {name: $name}) \n"
            "MERGE (pe)-[:MENTIONED_IN_PERSON]->(p)"
        )
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query, post_id=post_id,
name=name))

    def create_organization_node(self, org: str):
        name = self._normalize(org)
        query = "MERGE (o:Organization {name: $name})"
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query, name=name))

    def link_post_to_organization(self, post_id: str, org: str):
        name = self._normalize(org)
        query = (
            "MATCH (p:Post {id: $post_id}), (o:Organization {name: $name})
\n"
            "MERGE (o)-[:MENTIONED_IN_ORG]->(p)"
        )
```

```
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query, post_id=post_id,
name=name))

    def create_location_node(self, loc: str):
        name = self._normalize(loc)
        query = "MERGE (l:Location {name: $name})"
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query, name=name))

    def link_post_to_location(self, post_id: str, loc: str):
        name = self._normalize(loc)
        query = (
            "MATCH (p:Post {id: $post_id}), (l:Location {name: $name}) \n"
            "MERGE (l)-[:MENTIONED_IN_LOCATION]->(p)"
        )
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query, post_id=post_id,
name=name))

    # NEW METHODS FOR CATEGORY

    def create_category_node(self, category: str):
        name = self._normalize(category)
        query = "MERGE (c:Category {name: $name})"
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query, name=name))

    def link_post_to_category(self, post_id: str, category: str):
        name = self._normalize(category)
        query = (
            "MATCH (p:Post {id: $post_id}), (c:Category {name: $name}) \n"
            "MERGE (p)-[:BELONGS_TO]->(c)"
        )
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query, post_id=post_id,
name=name))

    def create_sentiment_node(self, sentiment_category: str):
        """MERGE a single Sentiment node by category."""
        cat = self._normalize(sentiment_category)
        query = "MERGE (s:Sentiment {category: $cat})"
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query, cat=cat))

    def link_post_to_sentiment(self, post_id: str, sentiment_category:
str):
        """MERGE relationship between Post and Sentiment."""
        cat = self._normalize(sentiment_category)
        query = (
            "MATCH (p:Post {id: $post_id}), (s:Sentiment {category: $cat})
\n"
```

```
            "MERGE (p)-[:HAS_SENTIMENT]->(s)"
        )
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query, post_id=post_id,
cat=cat))

    def create_date_node(self, date_value: str):
        val = self._normalize(date_value)
        query = "MERGE (d:Date {value: $val})"
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query, val=val))

    def link_post_to_date(self, post_id: str, date_value: str,
relationship_type: str = "PUBLISHED_ON"):
        val = self._normalize(date_value)
        # MERGE dynamic relationship type
        query = (
            f"MATCH (p:Post {{id: $post_id}}), (d:Date {{value: $val}}) \n"
            f"MERGE (p)-[:{relationship_type}]->(d)"
        )
        with self.driver.session() as session:
            session.execute_write(lambda tx: tx.run(query, post_id=post_id,
val=val))

    def create_source_node(self, domain: str, name: str):
        query = """
        MERGE (s:Source {domain: $domain})
        SET s.name = $name
        """
        with self.driver.session() as session:
            session.write_transaction(lambda tx: tx.run(query,
domain=domain, name=name))

    def link_post_to_source(self, post_id: str, domain: str):
        query = """
        MATCH (p:Post {id: $post_id})
        MATCH (s:Source {domain: $domain})
        MERGE (p)-[:PUBLISHED_BY]->(s)
        """
        with self.driver.session() as session:
            session.write_transaction(lambda tx: tx.run(query,
post_id=post_id, domain=domain))


    def normalize_source_names(self):
        query = """
        MATCH (s:Source)
        WHERE s.name = 'Other'
        WITH s, replace(s.domain, 'www.', '') AS stripped
        WITH s, split(stripped, '.')[0] AS shortName
        SET s.name = shortName
        """
```

```
        with self.driver.session() as session:
            session.run(query)
```

Class: processor_Task5

Description: processor_Task5 class defines the `Task5Processor` class, which handles the full ETL (Extract, Transform, Load) flow using PySpark. It loads enriched news data from a Parquet file, computes article-level sentiment, and delegates node and relationship creation to the Neo4j connector. Each process function targets a specific entity type (e.g., people, locations, sentiment) and pushes the corresponding data into Neo4j in a structured format.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, explode, avg
from neo4j_connector_Task5 import Neo4jConnector

class Task5Processor:
    def __init__(self, parquet_path, neo4j_uri, neo4j_user,
neo4j_password):
        # init Spark
        self.spark =
SparkSession.builder.appName("Task5Processor").getOrCreate()
        self.parquet_path = parquet_path

        # init ONE Neo4jConnector instance
        self.connector = Neo4jConnector(neo4j_uri, neo4j_user,
neo4j_password)
        self.df = None

    def load_data(self):
        self.df = self.spark.read.parquet(self.parquet_path)
        self.df.printSchema()
        print(f"Total rows in DataFrame: {self.df.count()}")

    def compute_article_sentiment(self):
        df_valid = self.df.filter(col("Sentiment_Result").isNotNull())
        df_avg = (
            df_valid
            .groupBy("url")
            .agg(avg("Sentiment_Result").alias("article_sentiment"))
        )
        # join back so every row carries the article_sentiment
        self.df = self.df.join(df_avg, on="url", how="left")
        print("✅ Article-level sentiment computed.")

    def process_posts(self):
        posts = (
            self.df
            .select(
                "url",
```

```
                col("publishedAt"),
                col("cleaned_title"),
                col("cleaned_description"),
                col("word_count"),
                col("article_sentiment")
            )
            .dropDuplicates(["url"])
            .rdd
            .map(lambda r: (
                "post_"+str(abs(hash(r.url))),
                r.publishedAt or "",
                r.url or "",
                r.cleaned_title or "",
                r.cleaned_description or "",
                int(r.word_count or 0),
                float(r.article_sentiment) if r.article_sentiment is not
None else None
            ))
            .collect()
        )

        for post_id, pub, url, title, desc, wc, sent in posts:
            self.connector.create_post_node(
                post_id, pub, url, title, desc, wc, sent
            )
        print(f"✅ Created {len(posts)} Post nodes.")

    def process_person_nodes(self):
        people = (
            self.df
            .select(explode(col("people_mentioned")).alias("name"))
            .filter(col("name").isNotNull())
            .select(col("name"))
            .distinct()
            .rdd
            .map(lambda r: r.name.strip())
            .filter(lambda s: s != "")
            .collect()
        )
        for name in people:
            self.connector.create_person_node(name)
        print(f"✅ Created {len(people)} Person nodes.")

    def process_link_person_post(self):
        pairs = (
            self.df
            .select("url",
explode(col("people_mentioned")).alias("person"))
            .filter(col("url").isNotNull() & col("person").isNotNull())
            .rdd
            .map(lambda r: (
                "post_"+str(abs(hash(r.url))),
```

```
                r.person.strip()
            ))
            .distinct()
            .filter(lambda x: x[1] != "")
            .collect()
        )
        for pid, person in pairs:
            self.connector.link_post_to_person(pid, person)
        print(f"✅ Linked {len(pairs)} Person→Post relationships.")

    def process_organizations(self):
        orgs = (
            self.df
            .select(explode(col("organizations_mentioned")).alias("name"))
            .filter(col("name").isNotNull())
            .distinct()
            .rdd.map(lambda r: r.name.strip())
            .filter(lambda s: s != "")
            .collect()
        )
        for name in orgs:
            self.connector.create_organization_node(name)
        print(f"✅ Created {len(orgs)} Organization nodes.")

    def process_link_organization_post(self):
        pairs = (
            self.df
            .select("url",
explode(col("organizations_mentioned")).alias("org"))
            .filter(col("url").isNotNull() & col("org").isNotNull())
            .rdd.map(lambda r: (
                "post_"+str(abs(hash(r.url))),
                r.org.strip()
            ))
            .distinct()
            .filter(lambda x: x[1] != "")
            .collect()
        )
        for pid, org in pairs:
            self.connector.link_post_to_organization(pid, org)
        print(f"✅ Linked {len(pairs)} Organization→Post relationships.")

    def process_locations(self):
        locs = (
            self.df
            .select(explode(col("locations_mentioned")).alias("name"))
            .filter(col("name").isNotNull())
            .distinct()
            .rdd.map(lambda r: r.name.strip())
            .filter(lambda s: s != "")
            .collect()
        )
```

```
        for name in locs:
            self.connector.create_location_node(name)
        print(f"✅ Created {len(locs)} Location nodes.")

    def process_link_location_post(self):
        pairs = (
            self.df
            .select("url",
explode(col("locations_mentioned")).alias("loc"))
            .filter(col("url").isNotNull() & col("loc").isNotNull())
            .rdd.map(lambda r: (
                "post_"+str(abs(hash(r.url))),
                r.loc.strip()
            ))
            .distinct()
            .filter(lambda x: x[1] != "")
            .collect()
        )
        for pid, loc in pairs:
            self.connector.link_post_to_location(pid, loc)
        print(f"✅ Linked {len(pairs)} Location→Post relationships.")

    def process_categories(self):
        cats = (
            self.df
            .select(col("category").alias("name"))
            .filter(col("name").isNotNull())
            .distinct()
            .rdd.map(lambda r: r.name.strip())
            .filter(lambda s: s != "")
            .collect()
        )
        for name in cats:
            self.connector.create_category_node(name)
        print(f"✅ Created {len(cats)} Category nodes.")

    def process_link_category_post(self):
        pairs = (
            self.df
            .select("url", col("category").alias("cat"))
            .filter(col("url").isNotNull() & col("cat").isNotNull())
            .rdd.map(lambda r: (
                "post_"+str(abs(hash(r.url))),
                r.cat.strip()
            ))
            .distinct()
            .filter(lambda x: x[1] != "")
            .collect()
        )
        for pid, cat in pairs:
            self.connector.link_post_to_category(pid, cat)
        print(f"✅ Linked {len(pairs)} Category→Post relationships.")
```

```
    def process_sentiment(self):
        def classify(v):
            if v is None: return "Unknown"
            if v > 0.1:   return "Positive"
            if v < -0.1:  return "Negative"
            return "Neutral"

        pairs = (
            self.df
            .select("url", col("article_sentiment"))
            .rdd.map(lambda r: (
                "post_"+str(abs(hash(r.url))),
                classify(r.article_sentiment)
            ))
            .distinct()
            .collect()
        )
        for _, sentiment in pairs:
            self.connector.create_sentiment_node(sentiment)
        for pid, sentiment in pairs:
            self.connector.link_post_to_sentiment(pid, sentiment)
        print(f"✅ Created & linked {len(pairs)} Sentiment→Post
relationships.")

    def process_dates(self):
        pairs = (
            self.df
            .select("url", col("publishedAt").alias("pub"))
            .filter(col("url").isNotNull() & col("pub").isNotNull())
            .rdd
            .map(lambda r: (
                "post_" + str(abs(hash(r.url))),
                # strip off the 'T' and everything after
                (r.pub.strip().split("T", 1)[0] if "T" in r.pub else
r.pub.strip())
            ))
            .distinct()
            .filter(lambda x: x[1] != "")
            .collect()
        )

        for _, pub in pairs:
            self.connector.create_date_node(pub)
        for pid, pub in pairs:
            self.connector.link_post_to_date(pid, pub, "PUBLISHED_ON")
        print(f"✅ Created & linked {len(pairs)} Date→Post relationships.")

    def process_sources(self):
        pairs = (
            self.df
            .select(
```

```
                "url",
                col("source").alias("name"),
                col("source_domain").alias("domain")
            )
            .filter(
                col("url").isNotNull() &
                col("name").isNotNull() &
                col("domain").isNotNull()
            )
            .distinct()
            .rdd.map(lambda r: (
                "post_" + str(abs(hash(r.url))),
                r.domain.strip(),
                r.name.strip()
            ))
            .filter(lambda x: x[1] != "" and x[2] != "")
            .collect()
        )

        for _, domain, name in pairs:
            self.connector.create_source_node(domain, name)
        for post_id, domain, _ in pairs:
            self.connector.link_post_to_source(post_id, domain)

        print(f"✅ Created & linked {len(pairs)} Source nodes.")


    def close(self):
        """Close Neo4j connection and stop Spark."""
        self.connector.close()
        self.spark.stop()
        print("🛑 Connector & Spark session closed.")
```

Class: queries_Task5

Description: queries_Task5 class contains the `QueryHelper` class, which defines and executes Cypher queries for graph analytics. It includes methods to retrieve sentiment distribution, top people mentions, category-based trends, and co-mention patterns, offering deep insights into the graph structure. The class is designed for reuse and modular analysis across different reporting needs.

```
# queries.py
# -------------
from neo4j import GraphDatabase
from typing import List, Dict, Any


class QueryHelper:
    def __init__(self, uri: str, user: str, password: str):
        self.driver = GraphDatabase.driver(uri, auth=(user, password))

    # -------------  generic helpers -------------
    def _run(self, cypher: str, **params) -> List[Dict[str, Any]]:
        with self.driver.session() as sess:
            recs = sess.run(cypher, **params)
            return [dict(r) for r in recs]

    def close(self):
        self.driver.close()

    # Sentiment distribution
    def post_count_by_sentiment(self) -> List[Dict[str, Any]]:
        cypher = """
        MATCH (p:Post)-[:HAS_SENTIMENT]->(s:Sentiment)
        RETURN s.category AS sentiment, count(*) AS posts
        ORDER BY posts DESC
        """
        return self._run(cypher)

    # Top 15 most mentioned people
    def top_people(self, top_n: int = 15) -> List[Dict[str, Any]]:
        cypher = """
        MATCH (pe:Person)-[:MENTIONED_IN_PERSON]->(p:Post)
        RETURN pe.name AS person, count(*) AS mentions
        ORDER BY mentions DESC
        LIMIT $n
        """
        return self._run(cypher, n=top_n)

    # Top 10 sources domain by number of articles
    def top_sources(self, top_n: int = 10) -> List[Dict[str, Any]]:
        cypher = """
        MATCH (so:Source)<-[:PUBLISHED_BY]-(p:Post)
        RETURN so.domain AS source , count(*) AS posts
```

```
        ORDER BY posts DESC
        LIMIT $n
        """
        return self._run(cypher, n=top_n)

    # Article count by date
    def daily_post_count(self) -> List[Dict[str, Any]]:
        cypher = """
        MATCH (p:Post)-[:PUBLISHED_ON]->(d:Date)
        WITH d.value AS date , count(*) AS posts
        RETURN date , posts
        ORDER BY date
        """
        return self._run(cypher)

    # Top categories by post count
    def top_categories(self, top_n: int = 10) -> List[Dict[str, Any]]:
        cypher = """
        MATCH (c:Category)<-[:BELONGS_TO]-(p:Post)
        RETURN c.name AS category, COUNT(p) AS post_count
        ORDER BY post_count DESC
        LIMIT $n
        """
        return self._run(cypher, n=top_n)

    # Most mentioned locations
    def top_locations(self, top_n: int = 10) -> List[Dict[str, Any]]:
        cypher = """
        MATCH (l:Location)-[:MENTIONED_IN_LOCATION]->(p:Post)
        RETURN l.name AS location, COUNT(p) AS mentions
        ORDER BY mentions DESC
        LIMIT $n
        """
        return self._run(cypher, n=top_n)

    # Sentiment breakdown per category
    def sentiment_by_category(self) -> List[Dict[str, Any]]:
        cypher = """
        MATCH (p:Post)-[:BELONGS_TO]->(c:Category),
              (p)-[:HAS_SENTIMENT]->(s:Sentiment)
        RETURN c.name AS category, s.category AS sentiment, COUNT(*) AS
count
        ORDER BY category, sentiment
        """
        return self._run(cypher)

    # Number of articles per source domain
    def posts_by_source(self) -> List[Dict[str, Any]]:
        cypher = """
        MATCH (p:Post)-[:PUBLISHED_BY]->(s:Source)
        RETURN s.domain AS source, COUNT(*) AS posts
        ORDER BY posts DESC
```

```
        """
        return self._run(cypher)

    # Top Organizations by Person Mentions
    def orgs_by_top_people(self) -> List[Dict[str, Any]]:
        cypher = """
        MATCH
(p:Person)-[:MENTIONED_IN_PERSON]->(post:Post)<-[:MENTIONED_IN_ORG]-(o:Orga
nization)
        RETURN p.name AS person, o.name AS organization, COUNT(*) AS
co_mentions
        ORDER BY person, co_mentions DESC
        """
        return self._run(cypher)

    # Top Categories per Sentiment
    def top_categories_by_sentiment(self) -> List[Dict[str, Any]]:
        cypher = """
        MATCH (p:Post)-[:BELONGS_TO]->(c:Category),
              (p)-[:HAS_SENTIMENT]->(s:Sentiment)
        RETURN s.category AS sentiment, c.name AS category, COUNT(*) AS
count
        ORDER BY sentiment, count DESC
        """
        return self._run(cypher)

    # Top Sources per Organization
    def top_sources_by_organization(self) -> List[Dict[str, Any]]:
        cypher = """
        MATCH
(o:Organization)-[:MENTIONED_IN_ORG]->(p:Post)-[:FROM_SOURCE]->(s:Source)
        RETURN o.name AS organization, s.domain AS source_domain, COUNT(*)
AS mentions
        ORDER BY organization, mentions DESC
        """
        return self._run(cypher)

    # Daily Sentiment Summary
    def daily_sentiment_summary(self) -> List[Dict[str, Any]]:
        cypher = """
        MATCH (p:Post)-[:PUBLISHED_ON]->(d:Date),
              (p)-[:HAS_SENTIMENT]->(s:Sentiment)
        RETURN d.value AS date, s.category AS sentiment, COUNT(*) AS count
        ORDER BY date ASC
        """
        return self._run(cypher)
```

Class: query_runner_Task5

Description: query_runner_Task5 class serves as the interactive runner for all analytical queries defined in `queries_Task5` class. It formats and prints query results in a clean, readable layout using tables and labels, making it ideal for presenting findings from the Neo4j graph. It focuses purely on querying and does not modify or ingest any data.

```
from queries_Task5 import QueryHelper

def run_all_queries(neo4j_uri, neo4j_user, neo4j_password):
    qh = QueryHelper(neo4j_uri, neo4j_user, neo4j_password)

    print("\n📊 SENTIMENT DISTRIBUTION")
    print("-" * 30)
    print(f"{'Sentiment':<15} | {'Post(s)':>5}")
    print("-" * 30)
    for row in qh.post_count_by_sentiment():
        print(f"{row['sentiment']:<15} | {row['posts']:>5}")

    print("\n🧑 TOP 15 PEOPLE MENTIONED")
    print("-" * 45)
    print(f"{'Person':<30} | {'Mentions':>8}")
    print("-" * 45)
    for row in qh.top_people():
        print(f"{row['person']:<30} | {row['mentions']:>8}")

    print("\n🌐 TOP 10 SOURCES BY DOMAIN")
    print("-" * 45)
    print(f"{'Source':<30} | {'Post(s)':>8}")
    print("-" * 45)
    for row in qh.top_sources(10):
        print(f"{row['source']:<30} | {row['posts']:>8}")

    print("\n📅 DAILY POST COUNT")
    print("-" * 30)
    print(f"{'Date':<15} | {'Post(s)':>8}")
    print("-" * 30)
    for row in qh.daily_post_count():
        print(f"{row['date']:<15} | {row['posts']:>8}")

    print("\n📚 TOP 10 CATEGORIES BY POST COUNT")
    print("-" * 50)
    print(f"{'Category':<30} | {'Post(s)':>8}")
    print("-" * 50)
    for row in qh.top_categories(10):
        print(f"{row['category']:<30} | {row['post_count']:>8}")

    print("\n📍 TOP 10 LOCATIONS MENTIONED")
    print("-" * 50)
    print(f"{'Location':<30} | {'Mentions':>8}")
```

```
    print("-" * 50)
    for row in qh.top_locations(10):
        print(f"{row['location']:<30} | {row['mentions']:>8}")

    print("\n📘 SENTIMENT BREAKDOWN BY CATEGORY")
    print("-" * 70)
    print(f"{'Category':<35} | {'Sentiment':<10} | {'Post(s)':>8}")
    print("-" * 70)
    for row in qh.sentiment_by_category():
        print(f"{row['category']:<35} | {row['sentiment']:<10} |
{row['count']:>8}")

    print("\n🔗 CO-MENTIONS: TOP PEOPLE & ORGANIZATIONS")
    print("-" * 100)
    print(f"{'Person':<25} ↔ {'Organization':<60} | {'Times':>6}")
    print("-" * 100)
    for row in qh.orgs_by_top_people()[:15]:
        print(f"{row['person']:<25} ↔ {row['organization']:<60} |
{row['co_mentions']:>6}")

    print("\n📈 SENTIMENT SUMMARY BY CATEGORY")
    print("-" * 65)
    print(f"{'Sentiment':<12} | {'Category':<35} | {'Post(s)':>8}")
    print("-" * 65)
    for row in qh.top_categories_by_sentiment()[:15]:
        print(f"{row['sentiment']:<12} | {row['category']:<35} |
{row['count']:>8}")

    print("\n📆 DAILY SENTIMENT SUMMARY")
    print("-" * 40)
    print(f"{'Date':<15} | {'Sentiment':<10} | {'Post(s)':>8}")
    print("-" * 40)
    for row in qh.daily_sentiment_summary():
        print(f"{row['date']:<15} | {row['sentiment']:<10} |
{row['count']:>8}")


    qh.close()
    print("\n✅ All queries finished.")
```

5.5. Query Output

1. SENTIMENT DISTRIBUTION

```
📊SENTIMENT DISTRIBUTION
------------------------------
Sentiment       | Post(s)
------------------------------
Positive        |    29
Negative        |    26
Neutral         |    19
```

A summary showing how post articles are distributed across sentiment categories: Positive, Negative, and Neutral. It provides an overall view of the emotional tone found within the media coverage.

2. TOP 15 PEOPLE MENTIONED

```
👤TOP 15 PEOPLE MENTIONED
-------------------------------------------
Person                      | Mentions
-------------------------------------------
Petronas                    |       22
Anwar Ibrahim               |        7
Subang Jaya                 |        6
McLaren                     |        6
Datuk Seri                  |        5
Anwar                       |        5
Bernama                     |        5
Seri Amirudin Shari         |        4
Andrea Kimi Antonelli       |        4
Oscar Piastri               |        4
George Russell              |        4
Russell                     |        3
Hamilton                    |        3
Lewis Hamilton              |        3
Hussein                     |        3
```

A ranked list of the most frequently mentioned individuals in the dataset. Useful for identifying key figures that dominate public attention or appear in trending topics.

3. TOP 10 SOURCES BY DOMAIN

```
🌐TOP 10 SOURCES BY DOMAIN
-------------------------------------------
Source                      | Post(s)
-------------------------------------------
www.thestar.com.my          |       29
www.globenewswire.com       |        7
soyacincau.com              |        4
www.fitechnical.net         |        4
www.channelnewsasia.com     |        3
www.forbes.com              |        3
robbreport.com              |        3
bringatrailer.com           |        2
paultan.org                 |        2
economictimes.indiatimes.com |       1
```

An overview of the top news sources based on how many articles each domain contributed. This helps identify which media outlets are most active or influential within the coverage.

4. DAILY POST COUNT

```
📅DAILY POST COUNT
--------------------------------
Date            |  Post(s)
--------------------------------
2025-03-13      |        1
2025-03-15      |        3
2025-03-16      |        3
2025-03-17      |        5
2025-03-18      |        1
2025-03-19      |        1
2025-03-20      |        2
2025-03-21      |        1
2025-03-24      |        1
2025-03-25      |        2
2025-03-26      |        1
2025-03-27      |        3
2025-03-28      |        2
2025-04-01      |       15
2025-04-02      |       11
2025-04-03      |        1
2025-04-04      |        4
2025-04-05      |        3
2025-04-06      |        2
2025-04-07      |        2
2025-04-08      |        3
2025-04-10      |        2
2025-04-11      |        1
2025-04-12      |        2
2025-04-13      |        2
```

A breakdown of how many articles were published each day. Spikes in this chart may correspond to major news events or press cycles.

5. TOP 10 CATEGORIES BY POST COUNT

TOP 10 CATEGORIES BY POST COUNT

Category	Post(s)
Exploration & Production	25
Financial	24
Sustainability	13
Technology & Innovation	7
General	3
Business Deals	2

A ranking of the most discussed topic categories, such as “Financial” or “Sustainability”. Offers insight into which areas receive the most media focus.

6. TOP 10 LOCATIONS MENTIONED

TOP 10 LOCATIONS MENTIONED

Location	Mentions
Putra Heights	20
Malaysia	14
Kuala Lumpur	10
China	8
Puchong	7
Melbourne	5
Ferrari	5
Selangor	5
Australia	4
Europe	4

A list of geographic places most frequently referenced in the articles. It reveals locations of interest or hotspots relevant to current affairs.

7. SENTIMENT BREAKDOWN BY CATEGORY

SENTIMENT BREAKDOWN BY CATEGORY

Category	Sentiment	Post(s)
Business Deals	Neutral	1
Business Deals	Positive	1
Exploration & Production	Negative	16
Exploration & Production	Neutral	7
Exploration & Production	Positive	2
Financial	Negative	3
Financial	Neutral	4
Financial	Positive	17
General	Neutral	3
Sustainability	Negative	7
Sustainability	Neutral	2
Sustainability	Positive	4
Technology & Innovation	Neutral	2
Technology & Innovation	Positive	5

A detailed view showing how sentiment varies within each article category. Enables detection of sentiment bias or tone differences across topics.

8. CO-MENTIONS: TOP PEOPLE & ORGANIZATIONS

CO-MENTIONS: TOP PEOPLE & ORGANIZATIONS

Person	↔ Organization	Times
3.4.3 Bio-	↔ Application	1
3.4.3 Bio-	↔ Value Chain Analysis	1
3.4.3 Bio-	↔ Key Topics Covered	1
3.4.3 Bio-	↔ Universal Tractor Transmission Oil	1
3.4.3 Bio-	↔ UTTO	1
3.4.3 Bio-	↔ the Market Segmentation	1
3.4.3 Bio-	↔ APAC	1
3.4.3 Bio-	↔ 1.2.2.8 Australia	1
3.4.3 Bio-	↔ Based Lubricants 3.4.2 Synthetic-	1
3.4.3 Bio-	↔ Asia-Pacific Agriculture Lubricants Market	1
ACP Wan Azlan	↔ ACP Wan Azlan	1
ACP Wan Azlan	↔ Tenaga Nasional Bhd	1
ACP Wan Azlan	↔ the Incident Control Post	1
ADIF	↔ GE Vernova	1
ADIF	↔ Kessler Foundation	1

Highlights which individuals and organisations co-occur within the same articles. This sheds light on potential relationships or associations in the news narrative.

9. SENTIMENT SUMMARY BY CATEGORY

SENTIMENT SUMMARY BY CATEGORY

Sentiment	Category	Post(s)
Negative	Exploration & Production	16
Negative	Sustainability	7
Negative	Financial	3
Neutral	Exploration & Production	7
Neutral	Financial	4
Neutral	General	3
Neutral	Technology & Innovation	2
Neutral	Sustainability	2
Neutral	Business Deals	1
Positive	Financial	17
Positive	Technology & Innovation	5
Positive	Sustainability	4
Positive	Exploration & Production	2
Positive	Business Deals	1

Combines sentiment and category into a compact summary, displaying how sentiment is distributed across different content types.

10. DAILY SENTIMENT SUMMARY

DAILY SENTIMENT SUMMARY

Date	Sentiment	Post(s)
2025-03-13	Positive	1
2025-03-15	Neutral	3
2025-03-16	Positive	3
2025-03-17	Neutral	3
2025-03-17	Positive	2
2025-03-18	Positive	1
2025-03-19	Positive	1
2025-03-20	Neutral	1
2025-03-20	Positive	1
2025-03-21	Positive	1
2025-03-24	Neutral	1
2025-03-25	Positive	2
2025-03-26	Neutral	1
2025-03-27	Positive	3
2025-03-28	Positive	2
2025-04-01	Negative	11
2025-04-01	Neutral	3
2025-04-01	Positive	1
2025-04-02	Negative	5
2025-04-02	Neutral	1
2025-04-02	Positive	5
2025-04-03	Negative	1
2025-04-04	Negative	1
2025-04-04	Neutral	1
2025-04-04	Positive	2
2025-04-05	Negative	2
2025-04-05	Positive	1
2025-04-06	Negative	1
2025-04-06	Neutral	1
2025-04-07	Negative	2
2025-04-08	Negative	1
2025-04-08	Neutral	2
2025-04-10	Neutral	1
2025-04-10	Positive	1
2025-04-11	Neutral	1
2025-04-12	Negative	1
2025-04-12	Positive	1
2025-04-13	Negative	1
2025-04-13	Positive	1

Tracks how sentiment changes on a day-to-day basis. Useful for observing fluctuations in public mood or media tone over time.