Real Time Big Data Analytics and AI: Methodological Approach

Submitted June 12, 2023, in partial fulfilment of the condition for the award of the MSc Computer Science degree

Kafayat Adeoye
Student ID: 20385832
Supervised by Peer-Olaf Siebers

I declare that this dissertation is all my own work, expect as indicated with citation and reference

Acknowledgements

I would like express my appreciation to GOD almighty and my parents and siblings for their love and support and lots of gratitude to my amazing supervisor Peer-Olaf Siebers for his mentorship, encouragement and support throughout the duration of this research, also many thanks to the module coordinator Tim Muller, and my friends. It has been a great joy working on this project as it is in the field of data science and artificial intelligence that I am really passion about. I have gained useful knowledge, developed independent research and skills that will definitely give me a strong stand in pursuing this career.

Abstract

Many enterprise and social platforms are generating large volume of data at high velocity at their disposal. It is becoming important to be able to gain rapid actionable insights at the right time through real time big data analysis while applying statistical and sophisticated analytics machine learning algorithms for prediction or finding event trends. Therefore, identifying the essence of having appropriate mechanisms for streaming data which will also address the challenges of unbounded batched analytics which is no longer time constraint. This research develops a framework that combines processes of streaming data engineering, streaming data analytics and end to end machine learning approach. It involves building a scalable stack for real time analytics, with the integration phase through Apache Kafka, processing of streaming data pipeline with Apache Spark structured streaming were events updates incrementally and enables storage of heterogenous data from various sources like IoT device, social media, clickstream, logs, etc. with Data Lake. To evaluate the extent of the framework effectiveness, it is tested on a structured streaming weather data that employs Random Forest Regressor model to predict air quality in a location and unstructured news data with a topic based on climate change using natural language processing to transform text and perform sentimental analysis and K-means clustering. Presented in this dissertation is the framework design, data pipeline methodology, and evaluation test on the two case study datasets which involves transformation, exploration, visualization and modelling of the streaming data.

Contents

List of Figures

List of Tables

List of Abbreviation

Air Quality Index (aqi): Good – 1, Fair – 2, Moderate – 3, Poor – 4, Very Poor – 5

1. Concentration of CO (Carbon monoxide), µg/m3
2. Concentration of NO (Nitrogen monoxide), µg/m3
3. Concentration of NO2 (Nitrogen dioxide), µg/m3
4. Concentration of O3 (Ozone), µg/m3
5. Concentration of SO2 (Sulphur dioxide), µg/m3
6. Concentration of PM2.5 (Fine particles matter), µg/m3
7. Concentration of PM10 (Coarse particulate matter), µg/m3
8. Concentration of NH3 (Ammonia), µg/m3

1. Introduction

The enormous, velocity, heterogeneity, and dynamic nature of real time data generation due to advancement in information technology by enterprise and social platforms have exposed the limitation of conventional Extract, Load and Transform (ETL) big data analytics as streaming data are time bounded and requires quick processing capacity. Deriving valuable understanding and ideas from data are beneficial in every accept of life and business applications.

1.1 Motivation

The motivation behind this study is to address the importance of having a robust and cost-effective analytics framework stack to gain rapid actionable insights at the right time for prediction or finding event trends in streaming data putting into consideration issues of event processing, data compliance and management, streaming query, visualization, and machine learning modeling.

1.2 Aims

- Develop a Lambda methodology to improve the scalability, efficiency, quality, and fault-tolerance of vast real time big data analytics that aligns streaming integration and introduce data lake to achieve a responsible streaming data management and quality from multiple sources and apply statistical and machine learning techniques leveraging Apache open-source technologies.

1.3 Objectives

- Build a generic framework to handle streaming data ingestion to perform analytics workflow.
- Test the streaming data analytic framework with both structured (air quality weather prediction) and unstructured (news feed sentiment) real time dataset in JSON format to evaluate a range of analytics methodologies.
- Transformation with vector assemble and natural language processing (NPL)
- Performing window aggregations and querying on streaming datasets
- Deploy data lake storage to manage streaming data.
- Evaluate machine learning regression and clustering algorithms on streaming datasets.
- Visualize a streaming data using varieties of visualization tools like matplotlib, plotly, word cloud, sentimental analysis polarity plot.

2. LITREATURE REVIEW

The literature review explains the background of real time data analytic, the justification and application of different Apache stack framework and the integration of machine learning in various industries.

2.1 Overview of Real-Time Big Data Analytics

The volume of digital data is expanding rapidly, and data is regarded as a company's most valuable asset. Due to technological improvements, data today arrives in massive amounts and in a variety of unstructured or semi-structured forms, necessitating the use of a variety of methods and tools to handle and analyse it (Qureshi & Gupta, 2014). Big Data may contribute significantly to the expansion of the global economy by boosting the efficiency and competitiveness of businesses and government agencies (Philip Chen & Zhang, 2014).

Aside volatility, variability of big data, the main characteristics which possess analysis challenges are i) volume, the attribute of volume pertains to the substantial amount of data that is produced on a continuous basis. ii) Velocity refers to the rate at which data is produced and subsequently processed. The exponential increase in data generated by digital devices, including sensors and smartphones, has presented significant challenges in managing streaming data and conducting real-time analysis. iii) Variety refers to the numerous data types. Relational databases are a common example of structured data, which makes up the lowest fraction of all current data. Semi-structured pertains to data that lacks adherence to rigid standards such as Extensible Markup Language (XML) and Unstructured data are classified as multimedia content encompasses auditory, textual, visual, and graphic elements and iv) Value is an essential characteristic of big data applications, as it facilitates the production of valuable business insights. (Ait Hammou et al., 2020).

Big data handles enormous datasets in offline batch mode, however, real-time big data stream processing operates on the most recent collection of data; hence, we work in the dimension of now or the recent past. (Gupta,& Sexana, 2016) Examples include the identification of credit card fraud and security. “Real-time big data is not just a method of storing petabytes or exabytes of data in a data warehouse” (Mike Barlow, 2013), It involves integrating and analysing data so that the appropriate action may be taken at the appropriate moment. Real-time refers to the capacity to process data as it enters, rather than storing it and retrieving it at a later time as done with batch processing which has a large delay. In real time analyses, latency and velocity are are the two variables involved (Gupta,& Sexana, 2016). Figure 1 is a mind map diagram of the background of the real time big data analytics.

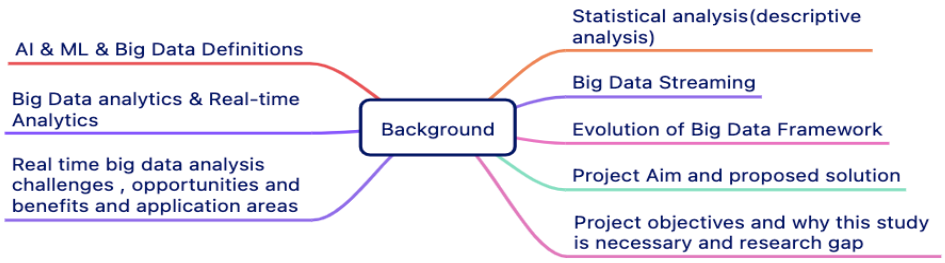


Figure 1 Overview of Real Time Analytics Mind Map

Data analytics is a field concerned with gleaning insights from data. It encompasses the methods, tools, and techniques of data analysis and management, such as data gathering (Olavsrud, Framingham, 2022) while Real-time big data analytics is a process comprising various tools and systems that is iterative. (Mike Barlow, 2013). The primary objective of data analytics or mining is to apply statistical analysis and technologies to data to identify trends and solve problems within vast datasets through the application of diverse algorithms, including K-means and Neural Networksand (Rao et al., 2019) visualisation, which is to see the intuitive insights of hidden patterns and draw some conclusions that are useful for making quicker decisions.

(Rao et al., 2019) In Stream processing, queries may be executed within a progressive time frame or on newly incoming data records, depending on the nature of the data, and the kind of application or domain. Analytics has three common types namely, Descriptive analytics represents the fundamental level of big data analysis, furnishing an overview or illustration of raw data. The application of statistical modelling techniques in order to anticipate future probabilities and patterns is a fundamental aspect of predictive analytics. Prescriptive analytics offers suggestions for one or more plausible courses of action, while also demonstrating the potential consequences linked with each choice. The utilization of this analytical methodology empowers executives in the business realm to arrive at decisions grounded on an ideal array of actions and limitations. Prescriptive analytics is the ultimate stage of big data analytics, providing guidance on potential outcomes and illustrating potential outcomes associated with each decision. It employs a variety of methodologies and instruments, but can be challenging to implement in both commercial and scientific contexts (Sahar Hussain Jambi, 2016).

2.2 Real Time Data Analytics Justification

Real-time large data system necessitates data synchronisation (Zheng et al., 2015) during the processing of data integration, data conversion must be performed due to the substantial heterogeneity of big data (Singh & Reddy, 2015). Important considerations such as data quantity, speed or throughput optimization, and model development often influence the selection of a specific platform for a certain application. Choosing the proper technology among a selection of contemporary instruments may also be tricky. In addition to being difficult to comprehend, real-time use cases must also be accurately identified in order to decrease infrastructure complexity and provide effective data pipelines (Mohamed & Al-Jaroodi, 2014).

Significantly reducing access latency, in-memory processing will play a critical role when real-time analytics are performed (Yadranjiaghdam et al., 2017). Large-scale datasets need an architecture that maintains high computing performance and scalable storage systems (Philip Chen & Zhang, 2014). The creation of large-scale data presents three significant obstacles with respect to the data volume, data velocity, and a wider range of data types. Volume denotes the magnitude of massive datasets. Velocity is the pace at which data are acquired and then refined for analytical purposes (Philip Chen & Zhang, 2014). Acquiring, storing, analysing, and backing up information costs money. A retention strategy that goes beyond the conventional "keep everything forever" approach is required. The age of real-time, massive data introduces a new and more stringent necessity for the processing of data in a timely manner. (Qureshi & Gupta, 2014)

Data are being gathered more and more commonly for purposes other than the simple testing of human-generated hypotheses or necessary record-keeping, and frequently for the prospect of testing hypotheses that have not yet been imagined at the time of collection. (Agarwal & Dhar, 2014) Real-time applications depend on immediate input and quick analysis to reach a decision or take an action within a condensed and highly precise time frame. Real-time large data applications face a variety of difficulties during design, implementation, and operation. (Mohamed & Al-Jaroodi, 2014). Analytics in real time (RT) entails using real-time analytical tools to identify the underlying causes of business and operational issues and exceptions while it is challenging to integrate Machine Learning (ML) due to the dynamic nature of RT. (Nti et al., 2022)

2.3 Real-Time Big Data Apache Framework and AI Techniques Applications

Due to the unavailability of real-time stream mining solutions in IoT (Rehman et al., 2018) developed an innovative concentric computing model (CCM) for deployment of real time big data analytics applications in IoT. Sensors, machine log files, event streams from IoT devices, human actions, industrial robots, and CPS are just a few of the sources that create the immense amount of raw data in IoT. The authors suggested architecture facilitates the processing, integration, and examination of industrial data, that massive data in IoT may impede QoS compliant service delivery. (Syafrudin et al., 2018) also explored IoT sensor data on a Korean assembly line for automobile manufacture. The authors method involves IoT-based sensor devices transmitting the sensor data to Apache Kafka, then Apache Storm processed the data and sent the sensor data and its fault prediction results directly to the monitoring system in real-time, followed by the storage of the sensor data and its prediction result in MongoDB and the dataset was evaluated using the hybrid prediction model of Density-Based Spatial Clustering and Random forest classifier to forecast the condition of faults in the assembly.

One main accepts that real time analytics has been productive in the financial industry is frauds detection in real-time which requires the design and implementation of scalable learning systems, (Carcillo et al., 2017) describe a Real-time Fraud Finder that combines machine learning technique that handles imbalance, non-stationarity, and feedback latency for streaming data demonstrating that Kafka, Spark, and Cassandra may offer simple scalability and fault tolerance for receiving, aggregating, and classifying high-volume transactions, however they encounter many issues while querying a Cassandra table via Spark.

In order to address this querying issues with the an SQL database (Liu et al., 2020) proposed the development of an efficient and scalable data analytics and management framework, which can facilitate complex multilevel predictive analytics for real-time streaming data using data lake architecture a single repository for future analysis without a predefined schema structure to storage enormous volumes at varying velocities, and varieties of either structured and unstructured data, and outline the use of Hadoop Distributed File System (HDFS) on the Hadoop big data platform. However, (Khine & Wang, 2018) has previously highlighted the pro and con of data lake which states that there will be no need for complex preparation and transformation when importing data into data lake compared to traditional data warehouses. Additionally, the initial expenditures of data input may be decreased.

Further exploration on big data analytics on social networks was proposed by (Angskun et al., 2022) for real-time depression identification the author presents a unique model based on the analysis of Twitter users' demographic characteristics and text sentiment ANOVA and SVM-RFE feature extraction were used to enhance the model's performance. Also (Rodrigues et al., 2021) study developed a Twitter trend analysis using latent Dirichlet allocation, cosine similarity, K means clustering, and Jaccard similarity approaches and compared the findings to the Big Data Apache Spark tool implementation findings shown that real-time tweets are evaluated somewhat more quickly with Apache Spark libraries than the conventional execution environment other strategies range from brute force counting techniques to topic modelling and machine learning clustering techniques.

(Hassaan & Elghandour, 2016) proposed initiative involves the creation of a system named DAMB, designed to perform instantaneous processing of streamed data on a diverse cluster comprising of both central processing units (CPUs) and graphics processing units (GPUs) a functionality of called SparkGPU that manage a heterogeneous cluster. The authors investigation centers on a meteorological utilisation that entails the examination of lightning discharges. Minimizing the time lapse between data acquisition and analysis or prediction tasks was of utmost importance to maintain the significance of the resulting conclusions. The literature reports on the integration of Kafka with Spark Streaming, which involves the utilisation of discretized streams or micro-batches as a technique. The central responsibility of the DAMB system's SparkGPU component is to execute applications that manipulate data streams.

(Dinesh Jackson et al., 2019) study real-time violence detection system that is capable of processing large volumes of streaming data and identifying instances of violence through the integration of big data analytics and deep learning techniques for the identification of human actions. The system receives a substantial volume of video streams in real-time from various sources, which undergoes processing within the Spark framework. Segregating of frames and extracting distinctive characteristics was accomplished through the HOG (Histogram of Oriented Gradients) algorithm. The frames are categorized according to distinct characteristics such as the violence model, human part model, and negative model these are employed in the training of the Bidirectional Long Short-Term Memory (BDLSTM) network, which facilitates the identification of scenes depicting violent behavior. The model's performance has been validated, demonstrating the robustness of the system with a recognition accuracy of 94.5% for violent actions.

2.4 Non-Apache Streaming Framework

There are other framework that does not implement Apache technologies for real time analysis, (Z. Zheng et al., 2015) constructs a real-time big data processing (RTDP) architecture based on cloud computing technology and batch-based MapReduce, and other processing modes using Field-programmable gate array (FPGA), Graphics processing unit (GPU), and ASIC technologies to process data with the primary arguments of how to establish the proper mode of computation and achieve unity between batch calculation mode and streaming processing.

The Typhoon framework by (Cho et al., 2017) The Typhoon framework is an integration of Software-Defined Networking (SDN) functionality into a real-time stream framework, aimed at augmenting its capabilities. The SDN control plane of the network updates the routing functions executed by individual nodes in a flexible manner, by transferring the required routing state to the application-level routing functions. The network is programmed on the data plane to transmit data tuples between nodes according to routing decisions made at the application level. The typhoon presents supplementary and distinctive prospects for the framework.

(Ait Hammou et al., 2020) present study suggests a methodology for conducting sentiment analysis that involves the utilisation of fastText in combination with Recurrent Neural Network (RNN) variations to effectively represent textual data. The proposal outlines a distributed intelligent system designed for real-time social big data analytics and employs distributed machine learning and a proposed method to improve decision-making processes. The results of the comprehensive experimentation on two standard data sets indicate that the proposition exhibits superior performance in comparison to established distributed recurrent neural network variations, both in terms of classification accuracy and the ability to manage large-scale data.

(Demirbaga et al., 2022) Big data processing systems, such as Hadoop and Spark, typically operate within extensive, heavily-concurrent, and multi-tenant environments that may result in hardware and software malfunctions or failures, ultimately resulting in performance degradation. study presents AutoDiagn designed as a microservice framework, providing the capability to seamlessly integrate new modules for detecting and analyzing root causes across diverse big data platforms to address the underlying reasons for performance deterioration in Hadoop systems caused by various faults, including but not limited to data locality, heterogeneity of cluster hardware, and network issues, prolonged task execution times, in Hadoop. The presence of outliers poses a significant challenge in large-scale data systems, leading to system overload and a significant decrease in performance.

3. METHODOLOGY

Dealing with Big data in real-time presents several obstacles to prevent infrastructure complexity of streaming data collection, processing, and analysis, as well as framework and architectural selection for a reliable application development (Abdullah & Mohammed, 2021). The proposed simplified framework dived into the research question to explore the strategies require to improve the scalability, efficiency, quality and fault-tolerance of real big data analytics, and coordination between the streaming process and storage units and the integration of various machine learning models based on the data type utilizing Apache stack. The study includes streaming data sources from Open Weather and NewsAPI website to extract data from the stream API. Data on weather air quality in Nottingham city and online news articles and blog on climate change were used to test the framework which is run a local device CPU. To fulfill the needs for the streaming analytical data pipeline and algorithms, an efficient framework is implemented, the Table 1 shows the brief description of the methodological requirements.

Functional Requirements	***Description***
Data Stream Source	*Dataset collected from streaming API source*
Interactive Environment	*Supports Python language, Apache Streaming Modules, Data Lake Storage and Visualization tools*
Data Acquisition	*Streaming Data ingestion connector for data collection*
Database	*Distributed filesystem and a non-relational (NoSQL) and allow raw or multi-structured data and enables data management practices*
Dynamic Data Visualisation	*Provide intuitive insights (live dashboard or plots)*
Machine Learning Libraries	*For prediction and statistical evaluation*

Table 1 Methodology Requirements

3.1 Methodology Procedures

The methodology involves an interactive development environment like Jupyter notebook that allows data ingestion, visualization, and advanced analytics which supports Apache streaming module and Python language API. The initial stage in every analytics is the acquisition of data from the sources preferable in JSON format represented as key/value pairs with a connector before it can be examined. Apache Spark is an open-source platform for local and cluster computing modes in data analytics. Spark provides cluster computation in memory and claims faster computational execution than Hadoop. Hadoop is also an open-source system for

distributed batch processing of enormous amounts of data using the MapReduce programming. Apache Spark is the core of the framework which streaming process is through Structured Streaming module to structure the incoming data as data frame API, and querying streaming data and keep track of event updates. Spark uses the baseline of MapReduce a paradigm computational framework for the manipulation and creation of extensive data sets through the utilisation of map and reduce functions, during the Map stage, the tasks are distributed among multiple computers and the reduce stage involves the logical shuffling of all keys and subsequent reduction for the purpose of computing significant aggregations or performing data transformations (Kakarla et al., 2021). Subsequently, the runtime system distributes the computation across extensive clusters of machines in a parallel distributed manner for big data analytics exploration, visualization and transformation and data locality which is bringing computation to where data reside (Bahga Vijay Madisetti, 2019).

The analytics framework adapts the publish-subscribe mode which enables asynchronous transmission of events in different topic zone between producers and consumers for the streaming data ingestion like Apache Kafka. The real-time analytics Spark Structured Streaming is an Apache Spark component that facilitates the study of streaming data and offers scalable, high-throughput, fault-tolerant stream processing (Bahga Vijay Madisetti, 2019), If a timestamp is included in static data, window operations can be implemented, resulting in a flexible approach to manage late-arriving data (Chambers & Zaharia, 2018).

The streaming data storage requires a distributed filesystems and non-relational (NoSQL) databases, which store the data collected from the raw data sources adopting a schema on read approach using the data access connector (Bahga Vijay Madisetti, 2019), in this case a Data Lake which competes with conventional, dependable data warehouses for storing diverse, complicated data and overcomes issues of data silos. A data lake may include raw, unstructured, or multi-structured data, the lake keeps data in its original format (Khine & Wang, 2018). It also eliminates the outgrow of cluster memory due to the unbounded size of the input stream data. Data visualization is another aspect of analytics process to provide intuitive insights to derive conclusions, since our analysis is real time and requires regular updates, it will require a dynamic visualization like live widget or plots (Rao et al., 2019).

Following, is an interactive querying to facilitate data exploration utilizing interactive querying frameworks such as Spark SQL. Spark Machine learning library module is for machine learning predictions and application deployment. Spark has the capacity to execute large-scale machine learning using its built-in libraries of machine learning algorithms called MLlib for prediction and statistical evaluation to assessment of validity and reliability throughout the experimental phase (Chambers & Zaharia, 2018).

This methodology describes approaches for developing the proposed framework to create a general-purpose streaming analytics process for big streaming data and the significant of addressing the issues posed with Stream processing where computation of a result is continually absorbing fresh input. In stream processing, the incoming data has no beginning or finish that is preset.

4. CONCEPTUAL FRAMEWORK DESIGN

The uniqueness of the nature, collection and transformation of real time data drives the adoption of streaming technologies infrastructure to create computational resources. The conceptual design for the framework adopts the Lambda architecture that serves as an interface for every stage of the analytic blocks leveraging Apache Spark streaming engine that enables end users to analysis or compute on historical or batched data and data arriving in near real-time in a single entity. It provides accurate and comprehensive views while balancing non-functional requirements such as latency, throughput, fault tolerance, data quality and compliance. This architecture creates the ability to ingest streaming data into a big data stack while assessing the data in both batch and real-time modes. The figure 2 diagram demonstrates the proposed framework which has three phases consisting of the following.

1. Phase 1: Streaming Data Ingestion
2. Phase 2: Streaming Analytics and Data Storage and Quality
3. Phase 3: Machine Learning Modelling

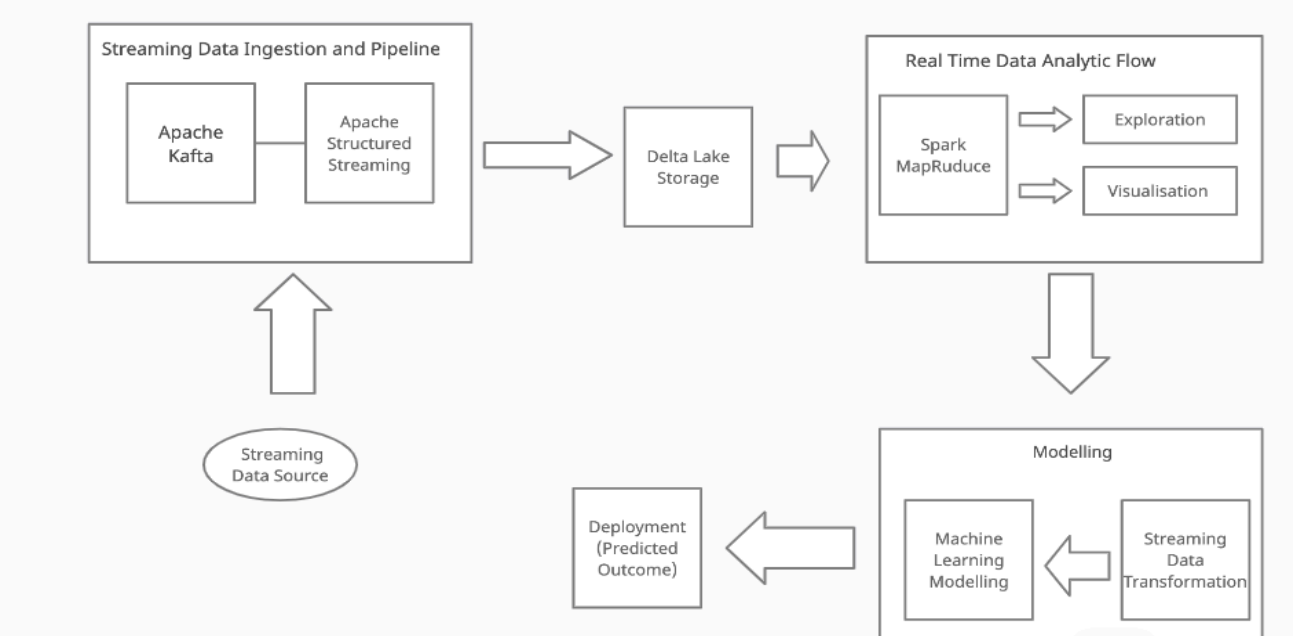


Figure 2 Proposed Real time Analytics Framework

4.1 Streaming Data Ingestion

Streaming data are referred to as data received as a flow of events over time compared to batched data that bounded by size. This initial phase of the framework illustrated in figure 3 where raw streaming data from various sources such as IoT sensor reading from IoT devices, credit card financial transactions, fleet trackers, billing system, social media, clickstream **etc.** are collected and ingested for further analysis using the publish-subscribe connector which serves as a puller for the stream.

Figure 3 Data Ingestion

4.1.1 Apache Kafka

Kafka is an open source and highly scalable publish-subscribe messaging system for large data that is paving the way for the transition from extract, transform, and load and batch processes of big data. Subscribe possess the capability to indicate preference for a particular occurrence or a series of occurrences and informed of any event that is produced by a publisher and aligns with their registered preference. The occurrence is disseminated asynchronously to every subscriber who has expressed interest in the specific event. The event-based interaction style's efficacy lies in the complete separation of time, space, and synchronization between publishers and subscribers (Eugster et al., 2003).

Kafka is a distributed and replicated service to balance incoming streaming load. It serves as a connector of the data stream source to Apache clusters It utilizes message brokers that act as Kafka server for data management and Zookeeper cluster for configuration to arrange data into consumer groups and subjects called Topics (Scott et al., 2022). A Kafka producer is a client type that transmits various data streams to the Kafka brokers which published into topics that have been defined and each topic are partitioned to accepts data in various format. Kafka also offers consumers like NoSQL database e.g., data lake or Cassandra that read data from the brokers. The producers and consumers are entirely decoupled, allowing each customer to function freely. Frequently, memory storage of data in RAM is insufficient to secure data; if the server crashes, the messages are not maintained after a reboot. Kafka was designed with high availability and persistent storage from the outset to deal with these issues (Bahga Vijay Madisetti, 2019).

Some of the characteristics of Kafka are persistent messaging to collect the real value of the data and offers a minimal network overhead, high throughput, supports partitioning in a distributed parallel mechanism, has multiple API integration such as with Python as used for this study and processes data to the consumer in real time. The design of Kafka offers cache of incoming message and stores on filesystem which gives long term retention of data regardless of if it was consumed or not and it address message failure and multiple consumption of the message (Nishant Garg, 2013).

4.2 Streaming Analytics and Data Storage

At this phase, the ingested streaming data is then further processed into a structured data frame API which is consumed by a selected non-relational data lake or memory table to carry out preprocessing operations like interactive querying, visualization, or statistical analysis. Figure 4, illustrated a linear transition from one block of operation to another to assess the functional requirements to process data stream.

4.2.1 Apache Spark

Spark is a computational engine that provides many high-level data analysis tools, the architectural structure includes Spark core, the lower layers, and upper-level libraries: Spark Streaming for streaming operations, Advanced Analytics for analyzing structured data, Spark SQL, the machine learning package, and GraphX for graph processing. Spark offers APIs for the languages Scala, Java, and Python and supports real-time, batch, and interactive queries. (Gupta & Saxena, 2016). Apache Spark reduces reliance on the underlying distributed file system by using the memory of a computer cluster, resulting in considerable speed benefits compared to Hadoop's MapReduce API that involves application of a function to a collection of elements in a distributed fashion and performs aggregation operations though it has the same linear scalability and fault tolerance as MapReduce. Apache spark is based on the Resilient Distributed Datasets (RDDs) abstraction, which facilitates the effective sharing of data across computations. RDD are immutable JVM (Java Virtual Machine) objects that are distributed in nature for performing high-speed computations and serve as the fundamental building blocks of Apache Spark. The dataset is characterised by its distributed nature, whereby it is partitioned into discrete units based on a designated criterion and subsequently allocated to executor nodes (Drabas et al., 2016). Apache Spark also enables the processing of data through a broader directed acyclic graph (DAG) of operators using a diverse collection of transformations and actions(Kienzler et al., 2018). The submission of code to the Spark console results in the generation of a Directed Acyclic Graph (DAG), which serves as an operator graph. Upon the occurrence of an action, such as aggregation, Spark proceeds to submit the graph to the DAG scheduler. The scheduler then partitions the operator graphs into distinct stages. The DAG scheduler consolidates multiple operator graphs into a cohesive unit, as each step may comprise multiple data partitions (Kakarla et al., 2021). Simply put, it automatically parallelizes the needed processes and distributes the data throughout the cluster (Salloum et al., 2016). Table 2 listed the definitions of the required Apache spark entities features and properties (Maas & Garillot, 2019)

Features	
Cluster	*Manages spark workload on a Hadoop YARN (Yet another resource negotiator) manager*
Drivers	*Holds the knowledge of where data are resided in the cluster and the initial stage of a spark job directed to the execution nodes*
Jobs and Tasks	*Jobs is the internal scheduling of incoming load from the cluster while task the distributed jobs into units for local computation*
Properties	
Scalability	*ability to add on computational resources for a stable performance*
Fault tolerance	*Resilient recovery from unplanned cluster failure through replication*

Persistence	*For In memory storage and offers caching a means to temporarily save data in memory, this saves a lot time from repeating computation to enable accessing data multiple times. It saves cost of I/O data operation*

Table 2 Features and properties of Apache Spark

4.2.2 Structured Streaming: Windowing and Event Time

Structured Streaming is a Spark streaming API that supports event time data natively and allows streaming mode the same structured data frame API-based abstraction operations that can be performed in batch mode. This may minimize latency and make incremental processing possible (Gupta & Saxena, 2016). Structured Streaming intends to provide end-to-end real-time applications that combine streaming with batch and interactive analysis, it includes a wealth of operational options, such as rollbacks, code modifications, and hybrid streaming/batch execution. Structured Streaming achieves high performance via Spark SQL's code generation engine and can outperform other streaming application like Apache Flink due to the low-level nature of many streaming APIs. It incorporates numerous stream processing system concepts, such as separating processing time from event time, windowing, and triggers. (Armbrust et al., 2018). As demonstrated in Figure 5, the core concept of Structured Streaming is to consider a live data stream as a table to which rows are continually appended (Ivanov & Taaffe, 2018). Window one of the features presented is an aggregation operation based on time. As streams are practically considered as continually appended tables and each row in such a table contains a timestamp a registered time at the point the data is generated, actions on windows may be specified inside the query and each query can specify distinct window according to the timestamp generated as the event time. The three types of window operations for event time in Structured Streaming namely, sliding, tumbling and session(Spark Guide, 2023). Structured streaming also requires specifying the schema for the incoming data and arrives in micro batch for continuous process.

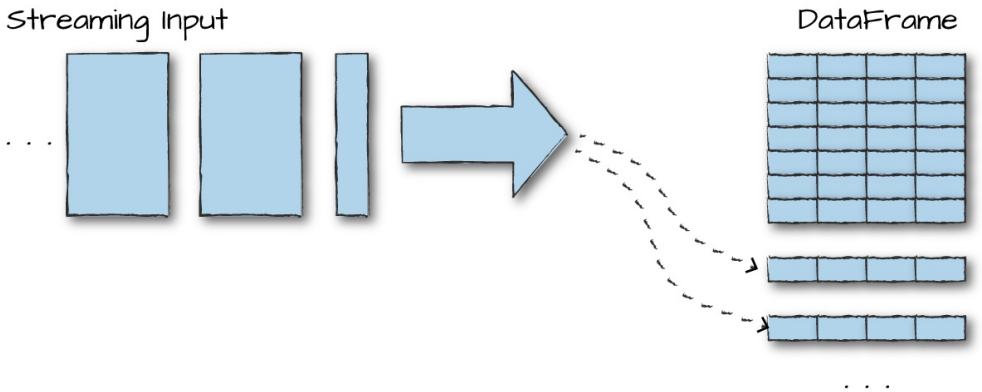


Figure 4 Appending Structured streaming input (Chambers & Zaharia, 2018)

4.2.3 Data Lake

Streaming sink is one way to write data from structured streaming into a more flexible data storage framework to enable adapting data privacy and compliance, for example removing duplicate or private records hence the introduction of Data Lake, (Liu et al., 2020) used this mechanism for an efficient querying operation. The data lake technology adopted for this framework is Delta Lake created by Databricks, is an open-source ACID (atomicity, consistency, isolation, durability) table storage layer and the API is compatible with Apache core. Delta Lake employs a transaction log compressed into the Apache Parquet format to offer ACID features, time travel, and substantially quicker metadata operations. It utilizes this architecture to deliver advanced capabilities like autonomous data layout optimization, upserts, caching, and audit logs. The objects themselves are encoded as Parquet, making it simple to develop connectors for engines that already support Parquet processing.

The characteristics of Delta Lake UPSERT, DELETE, and MERGE actions efficiently rewrite the relevant objects to execute revisions to archived data and compliance procedures, streaming I/O efficiency, and caching. Delta Lake is implemented primarily as a storage format and a set of client access protocols, making it easy to manage and highly available while providing clients with direct, high-bandwidth access to the object store (Armbrust et al., 2020). Parquet enables efficient compression and encoding algorithms that allows minimizing data storage costs and enhances the efficacy of searching data. The column-striping and assembly-language methods used to construct Parquet which are designed for storing huge data-blocks and organized into row groups and contains metadata information which is necessary to locate and identify data values (Sharma et al., 2018).

4.3 Apache Spark: Machine Learning Modelling (MLlib)

Artificial intelligence, and specifically machine learning, has been employed in various ways by the research community to transform several different and even heterogeneous data sources into high quality facts and knowledge, delivering outstanding capabilities to accurate pattern finding (Assefi et al., 2017). Machine Learning is a field of study that involves the development of algorithms and models that enable computers to learn from data. The discipline of Machine Learning involves endowing computers with the capacity to learn without the need for explicit programming (Géron, 2019).

This phase implements the application of machine learning models based on the type of dataset to be analyzed. The major problem with iterative calculations is the reliance on the intermediate data (Chambers & Zaharia, 2018). However, implementing machine learning algorithms on massive and complex datasets is computationally costly, and it requires a very significant number of logical and physical resources, such as data file space, CPU, and memory (Bahga Vijay Madisetti, 2019). Apache Spark MLlib is one of the most famous platforms for large-scale data analysis for data collection and cleaning, feature engineering and selection, training and adjusting large-scale supervised and unsupervised machine learning models and deploying these models in production (Assefi et al., 2017), Regression and Clustering algorithm were explore to test the framework. Figure 6 shows the inner workflow components of Spark

MLlib, overall the selection of a machine learning algorithm depends on the factors of the problem objective.

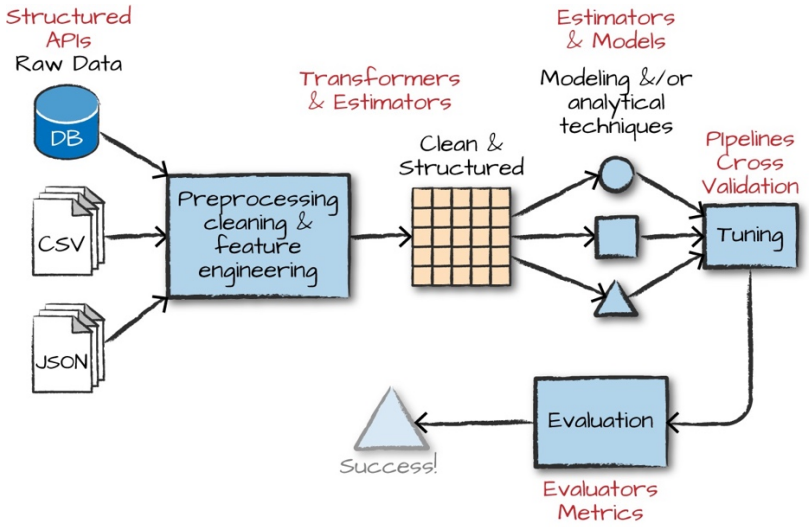


Figure 5 Spark Machine Learning Workflow (Chambers & Zaharia, 2018)

4.3.1 Supervised Learning and Unsupervised Learning

Learning can be defined as the cognitive process of enhancing one's ability to perform a specific task, such as the detection of fraudulent credit card activity. Various training methodologies, including decision trees, decision forests, logistic regression, support vector machines, neural networks, kernel machines, and Bayesian classifiers, have been employed to enhance the learning experience. The techniques employed in acquiring knowledge of from data frequently incorporate concepts from optimization theory or numerical analysis. The particular structure of machine-learning predicaments, such as the objective function or the integrated function typically being a summation of numerous terms, motivates advancements in this field. The techniques employed in acquiring knowledge from data frequently incorporate concepts from optimization theory or numerical analysis. The particular structure of machine-learning predicaments, such as the objective function or the integrated function typically being a summation of numerous terms. (Jordan & Mitchell, 2015)

Supervised learning is a method where a model learns from experience of labelled data called training for regression, classification or prediction. Regression is a supervised model for finding pattern in the linear relationship between a data features and the outcome commonly on numerical or quantitative data. For example "set *X* is some subset of *R3* (three ultrasound measurements), and set labels, *Y,* is the set of real numbers (the weight in grams)" (Shalev-Shwartz & Ben-David, 2014) and also classification

Unsupervised learning is when a model make prediction on unlabeled data. Clustering groups similar data according to the distribution and partitioned into sub sets called clusters (Shalev-Shwartz & Ben-David, 2014) to explore a dataset. Kmeans algorithm implemented is an iterative clustering process for determining clusters by reducing the distance of the mean value of a cluster center vectors and cluster members on a dataset.

5. FRAMEWORK TESTING

5.1 Modules Installation and Spark Session

The first step is to install the core modules which is Apache Spark and other packages like Apache Kafka and Delta Lake that are part of the framework.

Spark Installation

To install Spark, the package can be downloaded on the official website https://spark.apache.org/downloads.html based on the hardware machine it will run, for testing this framework Spark was run on MacOS and it runs on Java Virtual Machine (JVM) so installation of Java is a requirement.

Packages Installation

First step is to use 'pip install' to install the Kafka-python, findspark and PySpark to be able to use Python API on spark and then import into a Jupyter notebook IDE environment. The *findspark.init()* then installs the packages with the *- -conf* configuration command. The Kafka package is then integrated into Spark structured streaming, and Delta lake spark package. The spark version 3.3.0 was installed for the framework test. The following is the package installation code;

```
// import os, findspark and pyspark

import os
import findspark
import pyspark

// define packages and spark version

SCALA_VERSION = '2.12'
SPARK_VERSION = '3.3.0'

// integrate other packages like kafka and delta lake on spark core modules

os.environ['PYSPARK_SUBMIT_ARGS'] = f'--packages org.apache.spark:spark-sql-kafka-0-10_{SCALA_VERSION}:{SPARK_VERSION},org.apache.spark:spark-avro_2.12:3.3.2,io.delta:delta-core_2.12:2.2.0 --conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" –conf "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog" --conf spark.driver.extraJavaOptions=-Djdk.logger.allowReflection=true --conf spark.executor.extraJavaOptions=-Djdk.logger.allowReflection=true pyspark-shell'

// initialize findspark to add packages to spark core

findspark.init()
```

Initialize Spark Session

The SparkSession can be regarded as the mixture of various contexts, which also encompasses the StreamingContext. The establishment of this foundation serves as the basis for constructing Structured Streaming. The SparkSession serves as the primary interface for connecting PySpark code to the Spark cluster. The process of creating a Directed Acyclic Graph (DAG) for a given job and determining the allocation of specific tasks to executor or worker nodes is carried out by the driver node. The collection of variables and methods is intrinsically

static in the context of the executors, signifying that every executor obtains a replica of the variables and methods from the driver. In the event that the executor modifies said variables or overwrites the methods during task execution, such alterations are made in isolation from the copies of other executors and the variables and methods of the driver remain unaffected (Drabas et al., 2016) .

The nodes that have been established for executable code are set to operate in cluster mode according to design. In other word, a spark session is to initialize the spark high level core which is the entry for a spark job to begin and the important module for the framework is the SQL, on which is the underlying for Structure streaming. The Catalyst Optimizer is implemented in Spark SQL. The optimiser was developed utilising functional programming structures and was devised with dual objectives which is to facilitate the integration of novel optimisation methodologies and functionalities into Spark SQL, while also enabling third-party developers to enhance the optimizer's capabilities. This could involve incorporating rules tailored to specific data sources, accommodating additional data types, and other similar enhancements (Kakarla et al., 2021).

The code below instantiate a local spark session called Kafka Stream as demonstrated below where the * alongside the '*import*' command means all the methods of the module should be imported to be used in the code function. *getOrCreate()* is to create the spark UI interface where execution mode can be visualized;

```
// import spark functions, data types and spark sql to initiate SparkSession

from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *

// create the session with getOrCreate()

spark = SparkSession.builder \
    .appName("KafkaStream") \
    .getOrCreate()


// set Spark logging level to ERROR to avoid various other logs on console.

spark.sparkContext.setLogLevel("ERROR")

// set Spark schema inference to true

spark.sql("set spark.sql.streaming.schemaInference=true");

// this indicates the spark session has been created with the spark UI assessed
from port 40404 where the execution tasks are analyzed

SparkContext

Spark UI
Version
       v3.3.1
Master
       local[*]
AppName
       KafkaStream
```

5.2 CASE STUDY ONE: Weather Sensor Analysis with RandomForest Regression

This case study is to test a structured data stream on the framework and applied a RandomForest regression to predict Air Quality Index in the city of Nottingham which is defined by the Geographical coordinates (latitude= 52.950001 and longitude= -1.150000) of the location. The data stream source is from open weather *https://openweathermap.org/api/air-pollution* for Air pollution API hourly forecast of polluting gases, descriptions about the dataset parameters and API call can be found on the website. The reminder of the section demonstrated the implementation and outcome of the case study.

5.2.1 Streaming Weather Data Collection, Data Ingestion and Schema

Data Collection

The collected data API response from open weather is in Json format from the API request; the best way to extract json object is to use the requests and json python libraries

```
// import python json and request function

import requests
import json
```

Open weather required to create an API key on the website which is included in the API url with the coordinate of the location where the air pollution data will be extracted

```
// create a URL with the open weather API request to fetch data from it

url   =   "http://api.openweathermap.org/data/2.5/air_pollution/forecast?lat=52.950001&lon=-1.150000&appid=2d557f868d673c2659a653d495a004dd"

// use the requests.get() function to parse the data stream

response = requests.get(url)

// create an if statement to get data when site is available with code 200 for normal site operation and print an error message if site is unavailable

if response.status_code == 200:
    data = response.json()
else:
    print("Error fetching data from API")
```

Kafka Data Ingestion

In order to install Kafka, first download the latest stable version on the website into a folder on the system where kafka will be hosted alongside Apache Spark extract and to be able to consume the streaming data from the source first change directory to where Kafka is installed then run the following command to start Kafka script with ***kafka_2.13-3.0.0 % bin/kafka-server-start.sh config/server.properties***

and create a Kafka topic called weathertest where the data will be stored with this script command ***bin./kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic weathertest***

```
//import KafkaProducer the producer is used to fetch data from the API response

from kafka import KafkaProducer
```

Setting up ***KafkaProducer()*** function with the broker servers called bootstrap_servers indicates the host machine address 127.0.0.1 on port address 9092 and serialize data to the appropriate json-encoded format. The ***value_serializer*** variable refers to the process of converting data structures into a particular format in this case JSON with the ***json.dumps()*** function and encoded to utf-8 in order to efficiently manage format of the data the origin and destination.

The ***producer.send()*** function publish the formatted json weather data or event to the kafka topic 'weatherdata'

```
// creating kafka producer to extract data from open weather api

producer = KafkaProducer(bootstrap_servers=['127.0.0.1:9092'],
                         value_serializer=lambda x: json.dumps(x).encode('utf-8'))

// send data to Kafka topic with the producer.send()

producer.send('weatherdata', data)
producer.flush()
```

Structure Stream Consume

The structured streaming consumes data as a data frame ***df*** from the Kafka streaming producer; the ***spark.readStream()*** method read the data from the specified source with ***format()*** indicates that the data source is been derived from the kafka producer and the ***option()*** provides the configurations of the producer source to subscribe to the created kafka topic and ***load()*** checks and returns the streaming data values as a dataFrame which is asset of row of the dataset with the defined schema.

Dataframe are distributed collections of data that are immutable, in contrast to RDDs, dataframes are structured in a manner where data is arranged into designated columns with specific names and the primary purpose is to facilitate the processing of extensive datasets with greater ease it bears resemblance to tables in the realm of relational databases (Drabas et al., 2016).

- *bootstrap.servers: kafka server ports*
- *subscribe: subscribe specify the kafka topic that holds the data*
- *startingOffsets*: is a policy applied to begin data extraction from the earliest generated.

```
// structured streaming acquiring streaming data from kafka producer

df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "127.0.0.1:9092") \
    .option("subscribe", "weatherdata") \
    .option("startingOffsets", "earliest") \
    .load()
```

The schema output of **df** as seen below is a set of fields typically from the Kafka source and this case study is interested in the value field where the actual content resides.

```
// print the schema of df

root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

Convert the value of the kafka data stream to string format select the value that contains the weather data and timestamp when the data event occurs with the sql expression ***selectExpr()***

```
// create a dataframe that cast the kafka value to string and selects the timestamp
of the event
df = df.selectExpr("CAST(value AS STRING)","timestamp")
```

Schema

The stream data schema is define using the ***struct()*** function schema method provides the schema of the data json and it includes the data type for instance aqi which represents the air quality index (aqi) is in integer (int) type while the polluting gases has double data type, therefore, the schema defines the components of the polluting gases, aqi and the data format as an array.

```
// define a schema with the parameter of the air pollution data response

schema = "array<struct<main:struct<aqi:int>, components:struct<co:double, no:double,
no2:double, o3:double, so2:double, pm2_5:double, pm10:double, nh3:double>, dt:bigint>>"
```

The ***get_json_object()*** function is used to extract schema values the and separate with ***from_json()*** into different columns and explode the nested json values to separate then with the ***explode()*** function

```
// df_new is the new dataframe with the values of each components objects called
coord and list

df_new = df.select(get_json_object(col("value"),
"$.coord").alias("coord"),explode(from_json(get_json_object(col("value"),"$.list"),
schema)).alias("exploded_col"),"timestamp")
```

Fetching the nested polluting gases and air quality index data into columns with a new dataframe ***df_n*** that will be used for further analysis on the dataset using the ***col()*** *to create*

columns, rename with alias() function and use ***select()*** to place the columns into the dataframe as demonstrated below;

```
// df_n is the new data frame with the desired data values

df_n = df_new.select(col("exploded_col.main.aqi").alias("aqi"),
               col("exploded_col.components.co").alias("co"),
               col("exploded_col.components.no").alias("no"),
               col("exploded_col.components.no2").alias("no2"),
               col("exploded_col.components.o3").alias("o3"),
               col("exploded_col.components.so2").alias("so2"),
               col("exploded_col.components.pm2_5").alias("pm2_5"),
               col("exploded_col.components.pm10").alias("pm10"),
               col("exploded_col.components.nh3").alias("nh3"),
               col("exploded_col.dt").alias("dt"),col("timestamp").alias("timestamp"))
```

The ***printSchema()*** function prints the metadata of the dataframe 'df_n' that shows the weather data parameters, co, no, no2, 03, so2, pm2_5, pm10, nh3 which are the air polluting gases symbols, 'aqi' is the Air quality index measure and timestamp is the event time

```
// showing variables with their datatypes
df_n.printSchema()
```

```
root
 |-- aqi: integer (nullable = true)
 |-- co: double (nullable = true)
 |-- no: double (nullable = true)
 |-- no2: double (nullable = true)
 |-- o3: double (nullable = true)
 |-- so2: double (nullable = true)
 |-- pm2_5: double (nullable = true)
 |-- pm10: double (nullable = true)
 |-- nh3: double (nullable = true)
 |-- dt: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- minutes: integer (nullable = true)
```

5.2.3 Statistical Operation: Aggregation and Queries

Though a transformation operation has happened with the spark readStream it is just an instance of the data so to derive the actually data an action called query has to be called to create the data itself. Therefore a memory sink output the data by creating a table named ***query*** name that receives data continuously for further actions or query or for output to the delta lake. The ***outputMode*** named append to add the late record of the stream which is output once. Aggregate functions pertain to the process of consolidating multiple rows into a singular value summary through grouping (Kakarla et al., 2021). Aggregation operations ***(.agg***) like sum, count were applied on the data stream query, the process involves retrieving the most recent data from the streaming data source, conducting incremental processing to revise the outcome, and subsequently disposing of the original source data and retains solely the essential intermediate state data necessary for aggregation result updates and grouped by the ***window()*** operation that is to apply aggregate per window time, this window function enable the execution of mathematical operations on a group of rows with respect to a given row and

valuable when computing analytical functions related to time (Kakarla et al., 2021) and ***withWatermark()*** to check for late data arriving. The process of watermarking enables the engine to continuously track the present event time within the data and attempt to discard the outdated state correspondingly. The watermark of the request can be set through the specification of the time at which the event occurred column (timestamp) and the corresponding limit denoting the proper delay of data with respect to event time.

For an aggregation combined with widow and watermarking example, the code below shows the calculation of the sum and count of 'co' and 'no2' gases and finding the max value of aqi by defining a window of 1 hour of the query on the timestamp column and a watermarking threshold of 2 hours for late data.

```
// create an aggregate window_df_count_sum to find the operations sum, count and max

window_df_count_sum = df_n.withWatermark("timestamp", "2 hour").groupBy(window(df_n.timestamp, "1 hour")).agg({"co": "count", "no2": "count","co": "sum","aqi": "max"})

// use aggregate function to find the max value of aqi within a timestamp window of 2 hours

window_df_max = df_n.groupBy(window(df_n.timestamp, "2 hour")).agg({"aqi": "max"})
```

In order to display a table-like result of the streaming data aggregate, the spark dataframe is converted to pandas data frame with the following defined function, a streaming query is defined and then apply pandas view;

```
// import pandas module
import pandas as pd

// define a function to write each micro-batch to a Pandas DataFrame
def write_to_pandas(df, epoch_id):

// convert the spark dataframe to a pandas dataframe
    pandas_df = df.toPandas()

// print the pandas dataframe
    display(pandas_df)
```

The following are the command needed to create the query; ***writeStream()*** to write an instance to configure the stream output for example the sum and count result shown in table 3 and ***foreachBatch()*** to sink the streaming data into a pandas data frame with the pandas user defined function. The utilisation of foreachBatch operations enables the application of random actions and writing logic on the result of streaming queries and this functionality permits the execution of specific logic and arbitrary operations on the output of each micro-batch. ***outputmode()*** set to complete mode, outputs the complete stream representation of the aggregation and the ***start()*** to begin the computation and return the stream object.

```
// define the streaming query
streaming_query = window_df_count_sum.writeStream \
    .foreachBatch(write_to_pandas) \
    .outputMode("complete") \
    .start()
```

	window			count(no2)	max(aqi)	sum(co)
0	(2023-04-09 22:00:00)	21:00:00,	2023-04-09	96	3	21285.52

Table 3 Output of window aggregate (count, sum and max)

5.2.4 Delta Lake Storage

Structured Streaming can be utilised to write data into a Delta table as well. Delta Lake leverages the transaction log to ensure the delivery of exactly-once processing, even in the presence of concurrent streams or batch queries being executed on the table. The integration of Delta Lake with Spark structured streams is extensive, facilitated by the employment of ***readStream*** and ***writeStream***. Delta Lake effectively addresses several constraints commonly linked with streaming applications and files, such as: ensuring "exactly-once" handling in scenarios involving multiple streams or simultaneously batch processes and effectively identifying newly added data while utilising data as the origin for a stream (Table Streaming Reads and Writes — Delta Lake Documentation, n.d.).

The next stage to extract the streaming data into a delta lake to enable proper data quality and compliance and been able to pull data in the raw stream format for further analysis like visualization and machine learning application. The code following steps demonstrates first creating a query table called **'my_query'** just as shown in the code above but in this case it's sink into memory and extracted from there to pass the data frame into the variable called stream with sql command 'select'.

```
// create a streaming query data table

stream_query = df_n.writeStream \
    .outputMode("append") \
    .format("memory") \
    .queryName("my_query") \
    .start()

stream = spark.sql("SELECT * FROM my_query")
```

Then infer from the query table to delta lake table with the ***stream.write.format()*** in append mode and ***save()*** as a new delta table. The ***format()*** method specify the sink mode which is **'delta'** and table 4 display the structured dataset in delta lake.

```
// import the delta table function
from delta.tables import *

// write the stream to Delta Lake
stream2 =stream.write.format("delta").mode("append").save("/tmp/df_n_table")

// read the streaming from Delta Lake and display as pandas table
stream3 = spark.read.format("delta").load("/tmp/df_n_table")

// load Delta table as DataFrame
display(DeltaTable.forPath(spark, "/tmp/df_n_table").toDF().toPandas())

// needed if it is necessary to restart the query table
for q in spark.streams.active:
    if q.name == "word_my_query":
        q.stop()
```

aqi	co	no	no2	o3	so2	pm2_5	pm10	nh3	dt	timestamp	minutes
3	226.97	0.00	7.28	104.43	3.19	9.65	10.43	3.55	1681070400	2023-04-09 21:16:09.481	16
3	230.31	0.00	7.28	101.57	3.13	11.05	11.87	3.52	1681074000	2023-04-09 21:16:09.481	16
3	233.65	0.00	6.86	100.14	2.68	12.30	13.17	3.45	1681077600	2023-04-09 21:16:09.481	16
2	230.31	0.00	6.00	98.71	2.18	13.00	13.88	3.55	1681081200	2023-04-09 21:16:09.481	16
2	230.31	0.00	6.08	92.98	2.41	15.75	16.68	3.80	1681084800	2023-04-09 21:16:09.481	16

Table 4 Open weather structured data

5.2.5 Random Forest Regression

Random forest (RF) regression is a supervised algorithm with regression ensembled learning that combines predictions from the assemble of the random forest decision tree and output the average prediction, it gives a better prediction compare to a single regression model prediction. The following are the procedure steps for random forest regression model prediction (Kakarla et al., 2021);

- randomly select a subset of features using the featureSubsetStrategy.
- selecting random samples
- Generate a subset of data by selecting specific features and rows from the original dataset.
- Construct a decision tree utilising the subset data.
- Iterate the aforementioned steps to construct the quantity of trees as designated by the user.

By default, PySpark employs 20 trees for the number of trees parameter (numTrees) to obtain the final prediction which has been used in this test model prediction, and it is necessary to pass the data through all the individual trees. The class that receives the highest number of votes is selected as the ultimate output. In the context of regression, it is common practice to obtain the final output by averaging the output of individual trees.

5.2.6 Streaming Data Transformation

The Transformer class is designed to modify data by typically adding a fresh column to the data frame. The primary parameter of the method is typically the sole mandatory requirement, necessitating the provision of a data frame for conversion. The model predicted the air quality index (outcome) based on the 8 polluting gases (co,no,no2,03,so2,pm2_5,pm10,nh3) features which were transformed to become acceptable for the model prediction. Importing the libraries needed to perform machine learning with the random forest regression model, feature transformation and model evaluation

```
// import the machine learning libraries features, regression, and evaluation

from pyspark.ml.feature import *
from pyspark.ml.regression import *
```

```
from pyspark.ml.evaluation import *
```

```
// define the input features variable column names
feature_cols = ["co","no","no2","o3","so2","pm2_5","pm10","nh3"]
```

The Vector Assembler is a transformer that effectively consolidates various numeric columns, including vectors, into a singular column that is represented as a vector. The data transformation step is done using the vector assembler transformer to concatenate the polluting gases columns into a single column to train the RFregressor model which are of numeric types. The data is then split into 70% training data and 30% test data.

```
// define the VectorAssembler to transform input features into a vector column
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
```

```
// split the stream data into training (70%) and test sets (30%)
(trainingData, testData) = stream3.randomSplit([0.7, 0.3])
```

5.2.7 Model Training and Prediction

What is the rationale for using a random forest algorithm? Random forests exhibit greater robustness compared to an individual decision tree and effectively mitigate the issue of overfitting. The elimination of feature selection bias is achieved through the random selection of characteristics through each tree training step. The random forest algorithm utilises a proximity matrix, which has the capability to impute values that are missing (Li et al., 2017). This section shows the training of the random forest regressor model. The vectorized features are feed as the features columns and the target column ***'aqi'*** is feed as the label column. The model and the vector assembler transformer were treated as in a single pipeline called ***rf_pipeline*** and directly applied on the training streaming data and then prediction is done on the test data, which is sink back into a delta table as displayed in table 5 while table 6 shows the prediction aqi column and the actual aqi column.

```
// define the Random Forest Regression model with the features column and target
column
rf = RandomForestRegressor(featuresCol="features", labelCol='aqi')
```

The PySpark ML Pipeline is a comprehensive framework that encompasses a sequence of different phases for transforming and estimating data. This process involves taking in initial data in the form of a data frame, performing requisite data transformations, and ultimately estimating a statistical model using an estimator (Li et al., 2017).

```
// chain the VectorAssembler, and model into a Pipeline
rf_pipeline = Pipeline(stages=[assembler,rf])
// summary output of the random forest regressor model parameter
```

```
rfModel = rf_pipeline_model.stages[1], print(rfModel)
RandomForestRegressionModel:uid=RandomForestRegressor_591083c8f70b,
numTrees=20, numFeatures=8
```

Model Training

Training the streaming Data with RandomForest Regressor

```
// fit/train the pipeline to the streaming data
rf_pipeline_model = rf_pipeline.fit(trainingData)
```

Model Prediction

```
// make predictions on the streaming data using the fitted Pipeline
rf_predictions = rf_pipeline_model.transform(testData)
```

Write Prediction to Delta Lake

```
// write the predictions to Delta Lake
rf_predictions.write.format("delta").mode("append").save("/tmp/rf_tableprediction")
// read the predictions from Delta Lake and display as pandas table
rf_pred_df = spark.read.format("delta").load("/tmp/rf_tableprediction")
display(rf_pred_df.toPandas())
```

aqi	co	no	no2	o3	so2	pm2_5	pm10	nh3	timestamp	features	prediction
2	213.62	0.00	3.21	72.24	1.5	1.15	1.86	2.25	2023-04-09 21:16:09.481	[213.62, 0.0, 3.21, 72.24, 1.25, 1.15, 1.86, 2...	2.000
2	213.62	0.00	3.38	72.24	1.40	1.16	1.91	2.09	2023-04-09 21:16:09.481	[213.62, 0.0, 3.38, 72.24, 1.4, 1.16, 1.91, 2.09]	2.000
2	216.96	0.00	3.47	70.81	1.30	1.31	2.10	2.31	2023-04-09 21:16:09.481	[216.96, 0.0, 3.47, 70.81, 1.3, 1.31, 2.1, 2.31]	2.000

Table 5 Prediction output with vectorized features

```
// select aqi actual feature values and predicted aqi values
```

```
display(rf_pred_df.select("aqi","minutes","prediction").toPandas().head(10))
```

aqi	minutes	prediction
2	16	2.000
2	16	2.000
2	16	2.000
2	16	2.000
2	16	2.000

Table 6 Prediction

5.2.8 Regression Model Evaluation

The Root Mean Square Error (RMSE) is a statistical measure used to evaluate the accuracy of a model's predictions by calculating the square root of the average of the squared differences between the predicted and actual values. The metric mathematical computation in *equation 1* where *m* is the number of features, *x(i)* is the vector of all features, *y(i)* is the label, *X* is the features matrix, *h* is the prediction function and $RMSE(\boldsymbol{X}, h)$ is the cost function in the dataset, this provides an insight into the magnitude of the system's predictive inaccuracies, assigning greater significance to substantial deviations. A model's performance is considered superior when its RMSE values are lower (Géron, 2019). The Mean Squared Error (MSE) is a statistical metric that quantifies the average of the squared differences between the predicted and observed values of the response variable. Reduced Mean Squared Error (MSE) values are indicative of superior model performance. The Root Mean Squared Error (RMSE) bears resemblance to the Mean Squared Error (MSE) metric, yet it possesses the benefit of being expressed in the same units as the dependent variable.

$$\mathrm{RMSE}(\boldsymbol{X}, \mathrm{h}) = \sqrt{\frac{1}{\mathrm{m}} \sum_{\mathrm{i}=1}^{\mathrm{m}} \Big(\mathrm{h}\big(\mathrm{x(i)} - \mathrm{y(i)}\big)\Big) 2}$$

Equation 1 RMSE equation (Géron, 2019)

```
// select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(labelCol='aqi', metricName="rmse")
rmse = evaluator.evaluate(rf_predictions)
print(f"Root Mean Squared Error (MSE) on test data: {rmse}")

Root Mean Squared Error (MSE) on test data: 0.1747352494701839
```

Additional metrics that were investigated for the prediction computation include R-squared and MAE. The coefficient of determination, commonly known as R-squared (R2), quantifies the extent to which the variability in the response variable (y) can be attributed to the explanatory variables (X) included in the model. The R-squared metric ranges from 0 to 1, whereby larger values signify superior model fitting. The Mean Absolute Error (MAE) is a metric that quantifies the mean absolute deviation between the predicted and observed values of the dependent

variable. Overall the RMSE produces a better evaluation compared to R2 and MAE with just a small variation in the error value.

```
// define the evaluation metric
evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="aqi",
metricName="rmse")

// calculate the evaluation metrics for the test data
r2 = evaluator.evaluate(rf_predictions, {evaluator.metricName: "r2"})
mae = evaluator.evaluate(rf_predictions, {evaluator.metricName: "mae"})

// print the evaluation metrics
print(f"R-squared (R2) on test data: {r2}")
R-squared (R2) on test data: 0.1847352494701839

print(f"Mean Absolute Error (MSE) on test data: {mae}")
Mean Absolute Error (MSE) on test data: 0.1947352494701839
```

5.2.9 Streaming Data Visualization

The Python library known as Plotly is an open-source and interactive plotting tool that offers a diverse range of over 40 chart types. These chart types cater to various applications such as statistical in nature, financial, geographical, scientific, and three-dimensional applications. Plotly is a Python-based tool that leverages the plotly.js library to generate visually appealing and interactive web-based visualisations. These visualisations can be integrated into various applications, including Jupyter notebooks, standalone HTML files, and web applications developed using Dash. The Python library for Plotly is occasionally denoted as "plotly.py" in order to distinguish it from the JavaScript library. This visualization framework has been used for plotting the predicted aqi values against the actual air quality index as show on the scattered plot in figure 6.

```
// import the plotly libraries
import plotly.io as pio
pio.renderers.default = "iframe"
import plotly.graph_objects as go
import chart_studio.plotly as py
import plotly.express as px


// plotly user profile sign in details which includes username and token
py.sign_in("adeoye.a.kafayat", "51BL1baAC505SzD7FM8A")
```

```
px.colors.qualitative.Plotly;

// extract 1000 dataset from the prediction data to plot
pred_df = rf_pred_df.toPandas().head(1000)

// using this plotly code snippet to create a scatter plot of AQI vs prediction values
fig = go.Figure()
fig.add_trace(go.Scatter(x=pred_df.index, y=pred_df['aqi'], mode='lines', name='AQI'))
fig.add_trace(go.Scatter(x=pred_df.index, y=pred_df['prediction'], mode='lines',
name='Prediction'))

// add titles and axis labels
fig.update_layout(title='AQI vs Prediction', xaxis_title='Time', yaxis_title='Value')

// show the plot
fig.show()
```

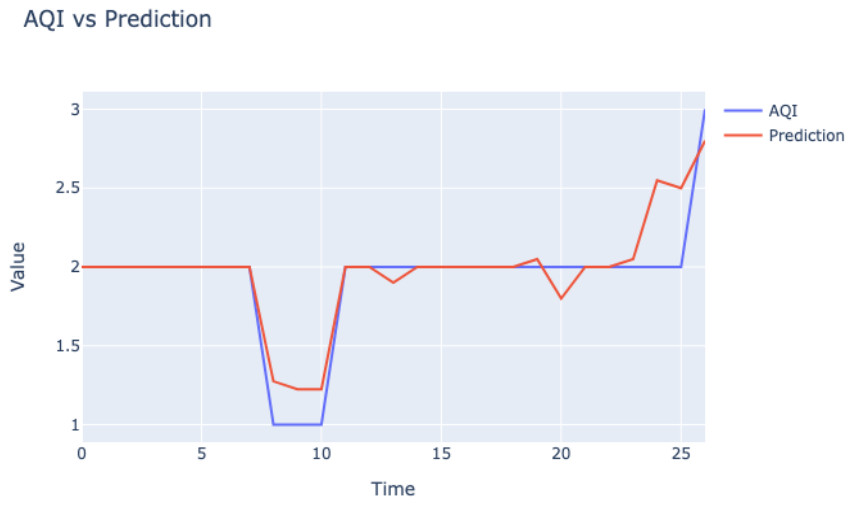


Figure 6 Real Time Air Quality Index versus Prediction

This figure 7 plotly visualization displays the 'Co' level versus the air quality index (aqi)

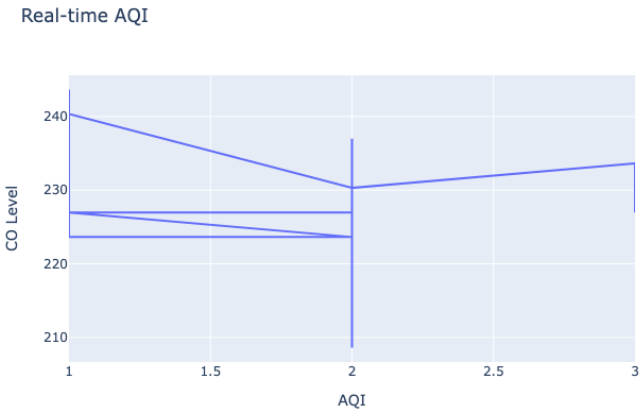


Figure 7 Real Time Air Quality Index for CO Level

5.3 CASE STUDY TWO: News Feed Clustering and Sentimental Analysis

This section is dedicated to a case study on testing the framework on unstructured dataset extracting news topic on climate change from various online media outlet via News API website https://newsapi.org/docs/endpoints/everything with millions of news articles and blogs. An API key is generated to be able to have access to the data endpoint with data format in JSON. Natural Language Processing and clustering techniques is applied to the news topics to determine the sentimental insight on the topic and how common the topic subject occurs in the News.

5.3.1 News Feed Streaming Data Extraction

The extraction of the newsfeed data is done with import of python request and json libraries and an url variable is created to store the streaming data api the api already as a token key appended so there was no need to generate on the NewsAPI website.

```
// import the request and json libraries
import requests
import json
// create the streaming data api url
url = https://newsapi.org/v2/everything?q=climate&apiKey=408f9fb0be5d4aa4b480c794a85db203

// request to extract the data and parse
response_1 = requests.get(url)

// create a function to fetch the news data using the requests.get() function
if response_1.status_code == 200:
```

```
    data_1 = response_1.json()
else:
    print("Error fetching data from API")
```

5.3.2 Data Ingestion and Schema

This section shows the ingestion of the streaming news data into Kafka topic and defining a schema for the data.

Kafka Ingestion

```
// import kafka library
from kafka import KafkaProducer


// set up Kafka producer
producer = KafkaProducer(bootstrap_servers=['127.0.0.1:9092'],
                         value_serializer=lambda x: json.dumps(x).encode('utf-8'))


// send data to Kafka topic called newsfeeddata1
producer.send('newsfeeddata1', data_1)
producer.flush()
```

Schema

This level shows the defined schema for the extracted news json format and data type; ***structType()*** denote the data structure which contains the ***structField()*** that defines the data object name and the data type with ***stringType()*** to cast the data as a string

```
// define the schema for the entire news dataset json objects
json_schema = StructType([
    StructField("status", StringType()),
    StructField("totalResults", StringType()),
    StructField("articles", ArrayType(article_schema))
])


// define the schema called article_schema for the news articles field
article_schema = StructType([
    StructField("title", StringType()),
    StructField("author", StringType()),
    StructField("source", StructType([
        StructField("id", StringType()),
        StructField("name", StringType())
```

```
    ])),
    StructField("publishedAt", TimestampType()),
    StructField("url", StringType())
])

// read streaming data from Kafka topic
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "127.0.0.1:9092") \
    .option("subscribe", "newsfeeddata1") \
    .option("startingOffsets", "earliest") \
    .option("failOnDataLoss", "false") \
    .load()
```

Convert the binary Kafka value to string and parse the JSON objects to extract the following fields from the articles value title, author, id, name, publication date and url to the news site

```
// extract the article fields for further analysis
parsed_df = df.selectExpr("CAST(value AS STRING)") \
              .select(from_json("value", json_schema,
options={"allowUnquotedControlChars":True}).alias("parsed_value")) \
              .selectExpr("parsed_value.status",
                          "parsed_value.totalResults",
                          "inline_outer(parsed_value.articles)").select("title", "author",
"source.*", "publishedAt", "url")

// print the data schema
parsed_df.printSchema()
```

```
root
|-- title: string (nullable = true)
|-- author: string (nullable = true)
|-- id: string (nullable = true)
|-- name: string (nullable = true)
|-- publishedAt: timestamp (nullable = true)
|-- url: string (nullable = true)
```

5.3.3 Delta Lake Storage: Write & Read & Load Stream Query

This section shows the creation of the data stream to write and read data to delta lake storage and the structured output data sample is shown as table 7;

```
// create a query with the writeStream function
stream_query = parsed_df.writeStream \
    .outputMode("append") \
    .format("memory") \
    .queryName("word_my_query") \
    .start()

// select the query into a variable called stream
stream = spark.sql("SELECT * FROM word_my_query")

// write the stream to delta lake

from delta.tables import *
stream1 =stream.write.format("delta").mode("append").save("/tmp/df_title_table")

// read the streaming stream2 from delta lake and display as pandas table
stream2 = spark.read.format("delta").load("/tmp/df_title_table")

// load delta table as dataframe
display(DeltaTable.forPath(spark, "/tmp/df_title_table").toDF().toPandas())
```

title	author	id	name	publishedAt	url
Washington, DC's Cherry Blooms Draw Crowds—and...	Emma Ricketts	None	None	2023-04-01 13:00:00	https://www.wired.com/story/washington-cherry-...
Can Burning Man Pull Out of Its Climate Death ...	Alden Wicker	None	None	2023-04-04 13:00:00	https://www.wired.com/story/burning-man-climat...
Mitsubishi wants to be the world's carbon broker	Justine Calma	None	None	2023-04-28 19:45:59	https://www.theverge.com/2023/4/28/23702343/mi...

Table 7 Structured news dataset

5.4 Natural Language Processing (NLP) Transformation

The field of natural language processing (NLP) pertains to the intersection of computer science and artificial intelligence, with a focus on the interplay between computers and human languages. Specifically, NLP involves the development of computer program that can effectively process and analyze vast quantities of natural language data. The domain of natural language processing is often characterized by various obstacles, which typically encompass the areas of speech recognition, natural language comprehension, and natural language production. Approximately 90% of global data is unstructured and can manifest in various formats such as images, text, audio, and video. Text can manifest in diverse formats, ranging from a catalogue of discrete terms to sentences and even multiple paragraphs that incorporate unique symbols, such as tweets and other forms of punctuation. It may also manifest in various formats such as web pages, HTML documents, and other similar mediums. The data at hand is often subject to noise and lacks cleanliness. The data necessitates treatment and subsequent execution of several preprocessing procedures to ensure the appropriate input data for feature engineering and model construction. If data is not preprocessed, any algorithms constructed using it will not provide any business benefits (Kulkarni & Shivananda, 2019).

```
// pyspark machine learning libraries for clustering analysis, transformation and evalaution
from pyspark.ml.feature import *
from pyspark.ml.clustering import *
from pyspark.sql.functions import *
from pyspark.ml import *
from pyspark.ml.evaluation import *

//  matplotlib visualization libraries for plotting word cloud and scattered plot
from textblob import TextBlob
import matplotlib.pyplot as plt
import streamlit as st
import seaborn as sns
```

The process of preprocessing entails converting unprocessed textual data into a comprehensible structure. Empirical data obtained from real-world scenarios is frequently characterized by incompleteness, inconsistency, and substantial levels of noise, which renders it susceptible to numerous errors. Preprocessing has been demonstrated as an effective approach for addressing such concerns. The process of data preprocessing involves the preparation of unprocessed textual data for subsequent processing. The process of tokenization involves dividing textual data into the smallest semantically meaningful units. Two types of tokenizers are available: sentence tokenizer and word tokenizer. This recipe will feature a word tokenizer, an essential component of text preprocessing for all types of analyses. The following NLP functions tokenize, stop remover, bag of word, vectorizer and count vectorizer were applied for the textual streaming data preprocessing;

5.4.1 Tokenize

Tokenization refers to the procedure of decomposing a given text or document into smaller entities known as tokens. Typically, a token is utilised to denote a solitary word, numerical value, or punctuation symbol. The Tokenizer is a component of Spark MLlib that executes the aforementioned procedure of segmenting an input text into discrete words, eliminating any superfluous spaces, and transforming them to lowercase. The output that ensues is a compilation of tokens that can be subjected to additional processing and utilised for the purpose of textual analysis.

```
// tokenize the title column
tokenizer = Tokenizer(inputCol="title", outputCol="words")
words_df = tokenizer.transform(stream2)
```

5.4.2 Stop Remover

The StopWordsRemover is a transformer that is utilised in Spark NLP to eliminate frequently occurring words, commonly referred to as "stop words", from a particular piece of text. Stop words refer to words that are deemed insignificant within a particular context and are frequently eliminated from text to enhance analysis or processing velocity. Stop words are a class of words that are commonly used in natural language, but are typically excluded from text analysis because they do not carry significant meaning. Typical stop phrases contain "a", "an", "the" at, "and", "or", "in", "on", "at", and "to", among others. The elimination of stop words is a prevalent preparation measure in text analysis processes such as topic modelling, text classification, and processing of natural languages.

```
// remove stop words from the tokenized words
remover = StopWordsRemover(inputCol="words", outputCol="filtered_words")
filtered_df = remover.transform(words_df)
```

5.4.3 Bag of Word: Vectorize

The bag-of-words approach effectively achieves the objective of transforming phrases or sentences into a numerical representation by tallying the frequency of identical words. Within the realm of computer science, the term "bag" is utilised to denote a data structure that functions similarly to an array or list in terms of object tracking. However, in contrast to these aforementioned structures, the order of objects within an a bag is not of significance. Furthermore, in instances where an object is present multiple times, the bag simply records the count of occurrences rather than duplicating the object itself. The Bag of Words approach is commonly referred to as CountVectorizer, wherein the frequency of occurrence of each word is tallied and subsequently represented as a vector. In order to generate a vector, it is necessary to instantiate an object for CountVectorizer and subsequently execute the fit and transform operations concurrently.

```
// vectorize the filtered words
vectorizer = CountVectorizer(inputCol="filtered_words", outputCol="features")
vectorized_df = vectorizer.fit(filtered_df).transform(filtered_df)
```

5.4.4 Word Count

Word count as in table 8 shows the frequency of the word occurrence in the extracted News topics with climate having 308 total words from the real time extraction.

```
// create the CountVectorizer model on the streaming data
cv = CountVectorizer(inputCol="filtered_words", outputCol="features", vocabSize=1000,
minDF=2.0)

// apply the model to the preprocessed stream data
cv_model = cv.fit(filtered_df)

// define a UDF (user defined function) to convert the features column to a string
column
features_to_str = udf(lambda x: ' '.join([str(i) for i in x.indices]), StringType())

// apply the UDF to the features column
result = cv_model.transform(filtered_df)
result = result.withColumn("feature_str", features_to_str(result["features"]))

// split the feature_str column and count the words
word_counts = result.select(explode(split("feature_str", " ")).alias("word"))

// get the vocabulary from the CountVectorizerModel
vocab = cv_model.vocabulary

// create a dataframe with word and count columns
word_count = spark.createDataFrame([(word, count) for word, count in zip(vocab,
word_counts.select("count").rdd.flatMap(lambda x: x).collect())], ["word", "count"])

// order the dataframe by count in descending order
word_count = word_count.orderBy("count", ascending=False)

// replace none values with 0
word_count = word_count.na.fill(0)
```

```
// replace '-' and '...' with an empty string in the 'word' column
word_freq = word_count.withColumn('word', regexp_replace('word', '[-|...]',''))

// display the dataframe
word_freq_dis = display(word_freq.limit(20).na.drop().toPandas())
```

word	count
climate	308
	186
reuters	152
	48
change	46
reuterscom	28
vote	24
activists	24
world	22
resolution	22
crisis	20
targets	20
oil	16

Table 8 Climate Change word count derived with countvectoriser

5.5 Kmeans Clustering: Model Training and Prediction

The objective is to cluster comparable instances into groups. Clustering is a process that involves the identification of instances that share similarities and the subsequent assignment of these instances to clusters or groups that exhibit similar characteristics. The K-Means algorithm is a straightforward algorithm that exhibits the ability to cluster this type of dataset with speed and efficiency, typically requiring only a few iterations. Stuart Lloyd introduced a method for pulse-code modulation at Bell Labs in 1957. However, the publication of this technique was limited to the company until 1982 (Géron, 2019).

Model Training

```
// initially train the pyspark k-means model and setting k = 5
kmeans = KMeans(k=5, seed=1)
model = kmeans.fit(vectorized_df.select("features"))
```

Model Prediction

```
// use transform() to predict the vectorized data frame
kmeans_predictions = model.transform(vectorized_df)

// convert the predicted data frame to pandas for visualization
kmeans_pandas_df = kmeans_predictions.select("title", "prediction").toPandas()
```

K-means Clusters Visualization

The streaming dataset depicted comprises of unlabeled instances that are visibly segregated into distinct clusters *k* as displayed on plot figure 8. The cluster with large numbers of data points shares common words, resulting in more scattered bubbles in those clusters.

```
// visualize the clusters with a scatter plot
plt.scatter(kmeans_pandas_df['prediction'], kmeans_pandas_df.index)
plt.xlabel('Cluster')
plt.ylabel('word_clust_freq')
plt.title('Cluster Assignments')
plt.show()
```

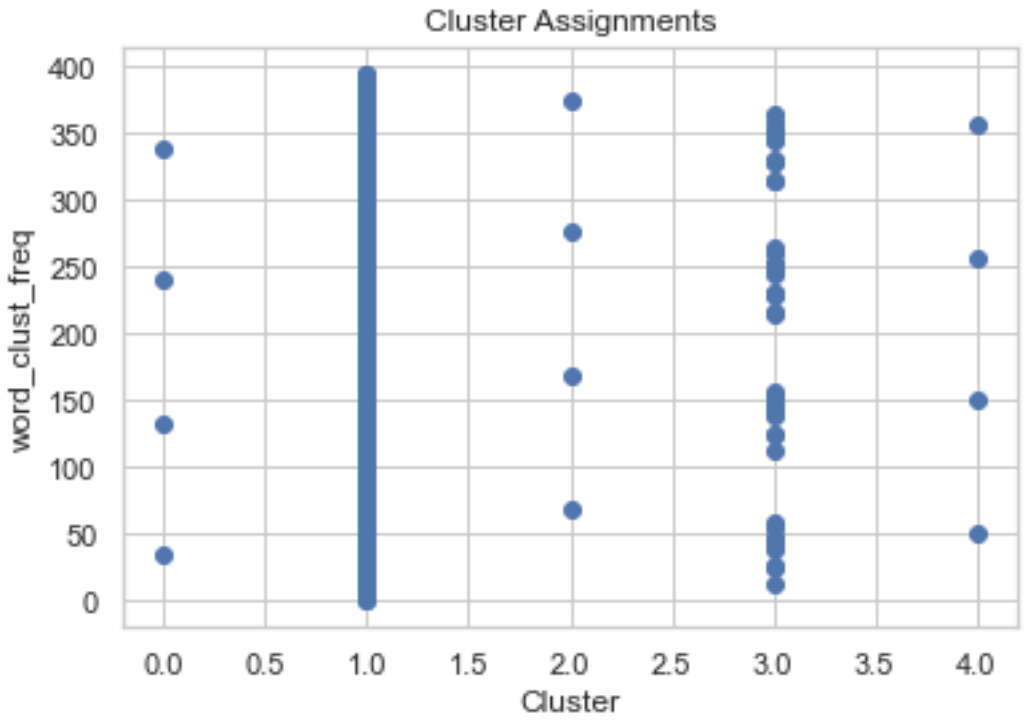


Figure 8 Clustering scattered plot with k = 5 output

```
// filter the kmeans_pandas_df to get the rows in the most clustered cluster
most_clustered_cluster = kmeans_pandas_df[kmeans_pandas_df['prediction'] == 1]

// Combine the titles into a single string
text = ' '.join(most_clustered_cluster['title'])
```

Word Cloud

Figure 9 shows the word cloud of kmeans cluster with parameter *k* set to 5 on the most clustered cluster 1 in figure 8 for the extracted words from the online news climate topics with the word climate been the most common word followed the news outlet name Reuters and the word climate change, and net zero

```
// Create the word cloud
wordcloud = WordCloud(width=800, height=400).generate(text)

// display the word cloud
fig, ax = plt.subplots(figsize=(10, 5))
ax.imshow(wordcloud, interpolation='bilinear')
ax.axis("off")
plt.show()
```

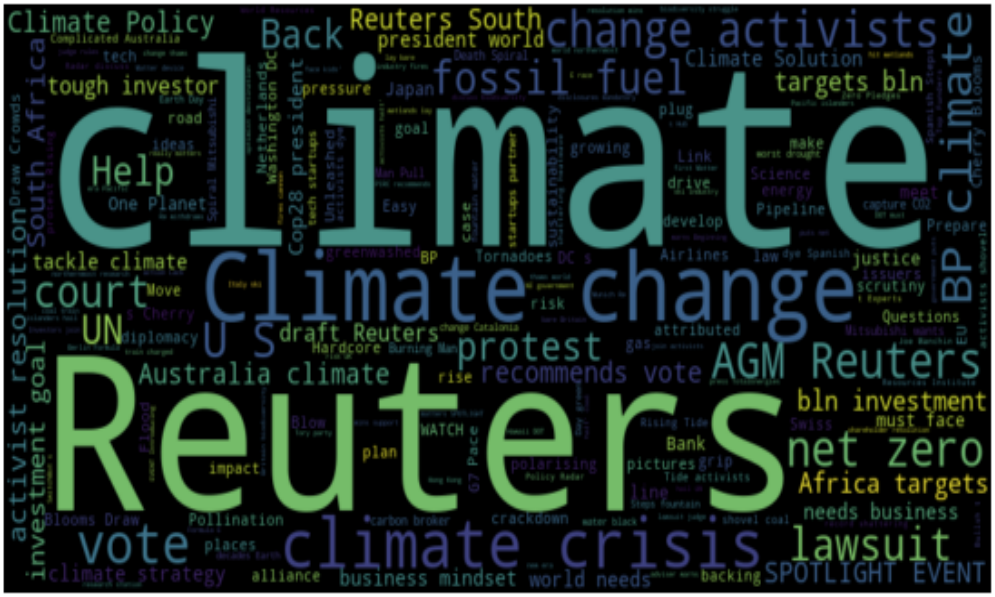


Figure 9 Word cloud for k =5

5.6 K-Means Model Evaluation

The K-Means algorithm has certain limitations that should be taken into consideration. Although K-Means algorithm has several advantages, such as its ability to process data quickly and handle large datasets, it is not without limitations. As observed, it is imperative to execute

the algorithm multiple times to prevent suboptimal solutions. Additionally, determining the number of clusters can be a cumbersome task, hence the use of the evalaution techniques such as silhouette and elbow method to determine the performance of the k-means and select the best parameter *k* cluster number on the streaming data.

5.6.1 Silhouette

The Silhouette score is a metric that quantifies the degree of similarity between an object and its respective cluster, taking into account both the cohesion within the cluster and the separation from other clusters. The scoring system is bounded between negative one and positive one, whereby greater numerical values correspond to superior clustering outcomes. A score that is negative in value indicates the possibility that the sample could have been erroneously assigned to a cluster that is not appropriate.

```
// evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()
silhouette = evaluator.evaluate(kmeans_predictions)
print("Silhouette Score = " + str(silhouette))
```

```
Silhouette Score = -0.013331837391900355
```

5.6.2 Elbow Method: Within-Cluster Sum of Squares (WCSS)

The elbow method is a commonly used technique in determining the most suitable number of clusters for a k-means clustering algorithm. The process entails graphing the Within-Cluster Sum of Squares (WCSS) against the quantity of clusters and designating the point of inflection as the optimal number of clusters. Elbow Method confirms from the plotted graph in figure 10 that optimized clusters k= 4 is a better choice than when cluster k= 5 in the evaluation of the k-means clustering analysis.

```
// calculate Within-Cluster Sum of Squares (WCSS)
evaluator = ClusteringEvaluator()
wcss = evaluator.evaluate(kmeans_predictions_4)
print("Within-Cluster Sum of Squares = " + str(wcss))
```

```
Within-Cluster Sum of Squares = 0.07830905182847443
```

```
// elbow method to find optimal number of clusters
cost = np.zeros(20)
for k in range(2,20):
    kmeans = KMeans().setK(k).setSeed(1).setFeaturesCol("features")
    model = kmeans.fit(vectorized_df)
    predictions = model.transform(vectorized_df)
```

```
    evaluator = ClusteringEvaluator()
    cost[k] = evaluator.evaluate(predictions)

// visualize the elbow method cluster to select the best optimal number
fig, ax = plt.subplots(1,1, figsize =(8,6))
ax.plot(range(2,20),cost[2:20])
ax.set_xlabel('Number of Clusters')
ax.set_ylabel('WSSSE')
plt.show()
```

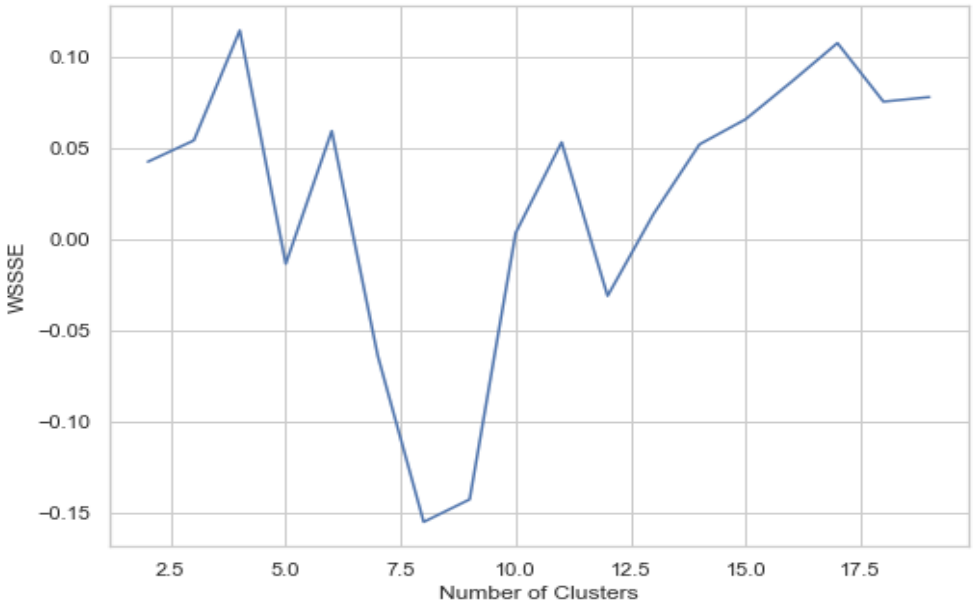


Figure 10 Elbow evaluation method with optimized k =4 cluster

5.6.3 Kmeans Cluster Word Cloud Visualization

```
// train the k-means model setting k = 4
kmeans_4 = KMeans(k=4, seed=1)
model_4 = kmeans.fit(vectorized_df.select("features"))

// use transform() to predict the vectorized data frame
kmeans_predictions_4 = model_4.transform(vectorized_df)

// convert the data frame to pandas for visualization
kmeans_pandas_df_4 = kmeans_predictions_4.select("title", "prediction").toPandas()

// visualize the clusters k = 4 with a scatter plot
```

```
plt.scatter(kmeans_pandas_df_4['prediction'], kmeans_pandas_df_4.index)
plt.xlabel('Cluster')
plt.ylabel('word_clust_freq')
plt.title('Cluster Assignments')
plt.show()
```

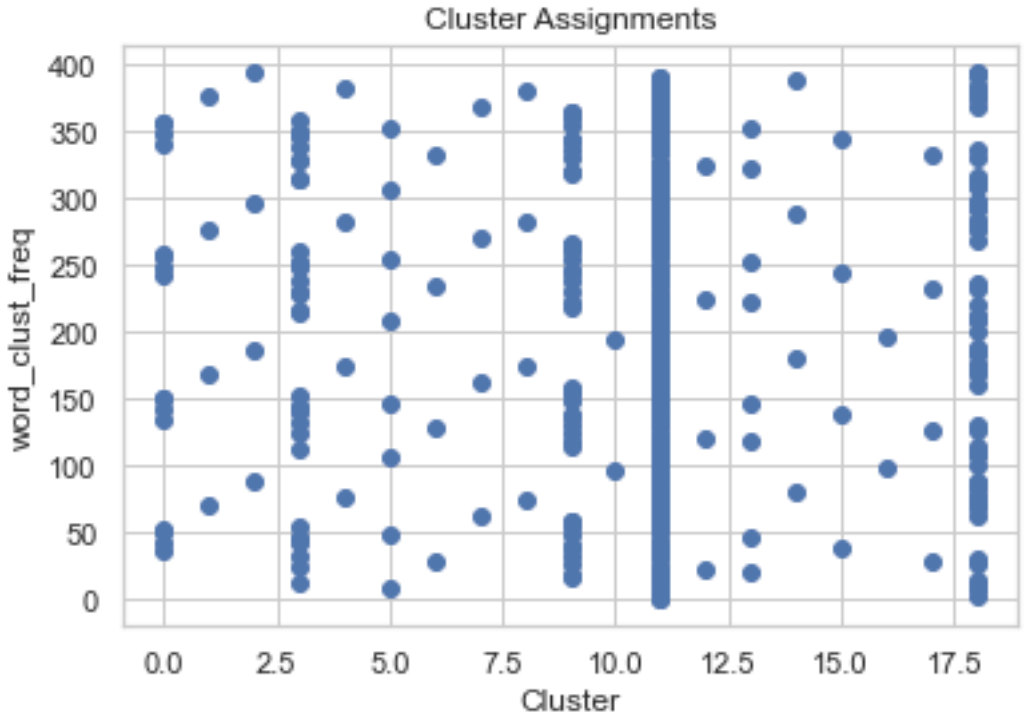


Figure 11 Best Optimized clusters for k = 4

Most of the instances were assigned to the appropriate cluster, but some of the instances were mislabeled for those with the scantiest clusters. Plotting the word cloud on most clustered cluster 11 in figure 12, with the optimised kmeans cluster with parameter *k* set to 4 in figure 11

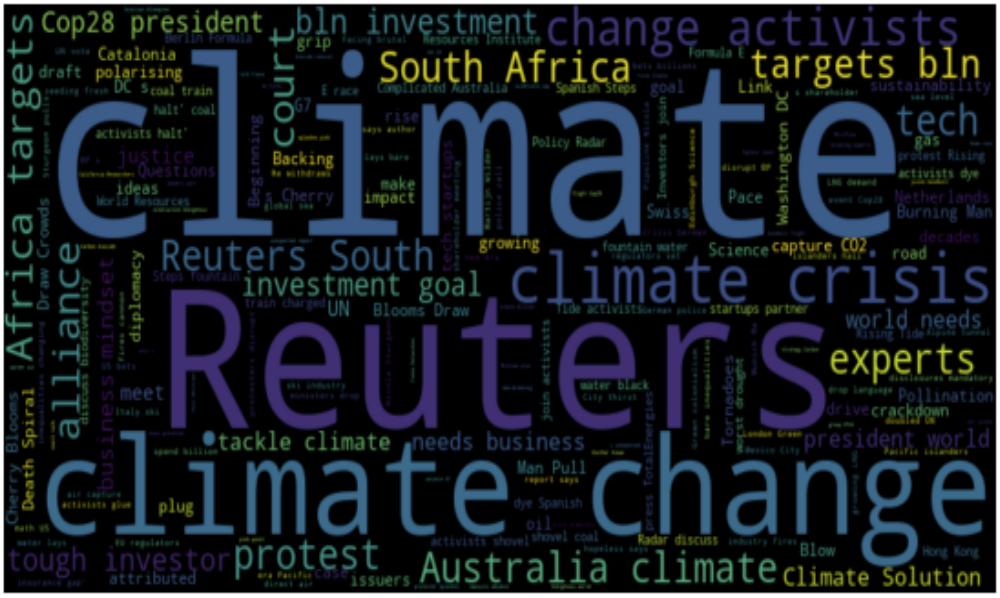


Figure 12 Optimized k= 4 word cloud for most clustered cluster

5.7 Sentimental Analysis

The Sentiment Polarity Distribution depicts the dispersion of sentiment polarity values throughout the entirety of the analysed textual data. The sentiment polarity of a text is a numerical representation of its sentiment, ranging from -1 to 1. A score of -1 denotes an extremely negative sentiment, while a score of 1 indicates an exceedingly positive sentiment. A sentiment polarity score of zero denotes a state of neutral sentiment.

```
// define a function for calculate the sentimental polarity
def get_sentiment(text):
    blob = TextBlob(str(text))
    return blob.sentiment.polarity

// define a variable for the get_sentiment() function with udf
get_sentiment_udf = udf(get_sentiment, FloatType())


// add a new column with the sentiment polarity
vectorized_df_sen = vectorized_df.withColumn('sentiment_polar', get_sentiment_udf('title'))

// Add a new column with the sentiment label
vectorized_df_sen = vectorized_df_sen.withColumn('sentiment_label',
get_sentiment_label_udf('sentiment_polar'))

// define a function to get the sentiment label of positive, neutral and negative
def get_sentiment_label(polarity):
```

```
    if polarity > 0:
        return 'Positive'
    elif polarity < 0:
        return 'Negative'
    else:
        return 'Neutral'

// define a UDF for the get_sentiment_label() function
get_sentiment_label_udf = udf(get_sentiment_label, StringType())

// find the use kmeans for sentimental prediction
kmeans = KMeans(k=4, seed=1)
model_k = kmeans.fit(vectorized_df_sen.select("features"))

// add the predicted cluster column to the DataFrame which contains the sentiment
polarity and label (positive, neutral and negative)
kmeans_predictions_sen = model_k.transform(vectorized_df_sen)

kmeans_predictions_sen = model_k.transform(vectorized_df_sen.select("title","features",
"sentiment_polar", "sentiment_label"))

// convert the DataFrame to pandas for visualization
kmeans_pandas_df_sen = kmeans_predictions_sen.select("title",
"prediction","sentiment_polar","sentiment_label").toPandas()

// visualize the sentimental polarity
sns.boxplot(x='prediction', y='sentiment_polar', data=kmeans_pandas_df_sen)
plt.title('Sentiment Polarity by Cluster')
plt.show()
```

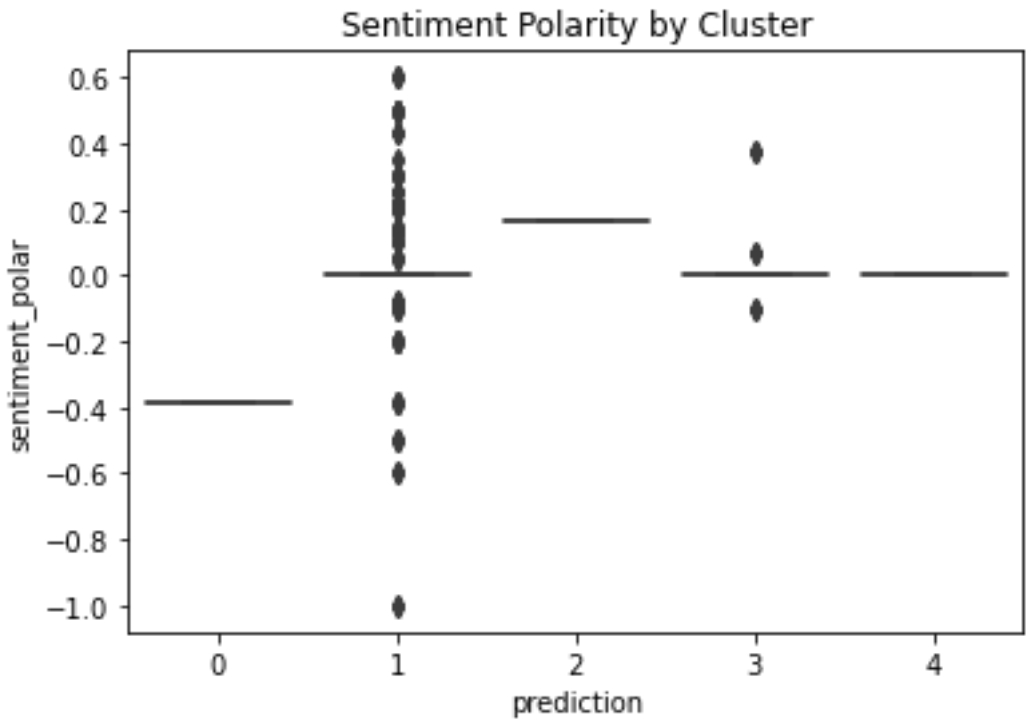


Figure 13 sentimental analysis clustering

The distribution plot shows how frequently each sentiment polarity value occurs in the dataset. For example, if the sentiment polarity distribution is skewed to the left, it means that the dataset contains more negative sentiment text. If the distribution is skewed to the right, it means that the dataset contains more positive sentiment text. A symmetrical distribution indicates a balanced mix of positive, negative, and neutral sentiment meanwhile from the plot it shows the topic climate from the extracted streaming blog has a *neural sentiment*.

```
// visualize the sentimental polarity with labels

g = sns.FacetGrid(kmeans_pandas_df_sen, col='prediction')
g.map(sns.histplot, 'sentiment_label')
plt.suptitle('Sentiment Polarity Distribution by Cluster', y=1.05)
plt.show()
```

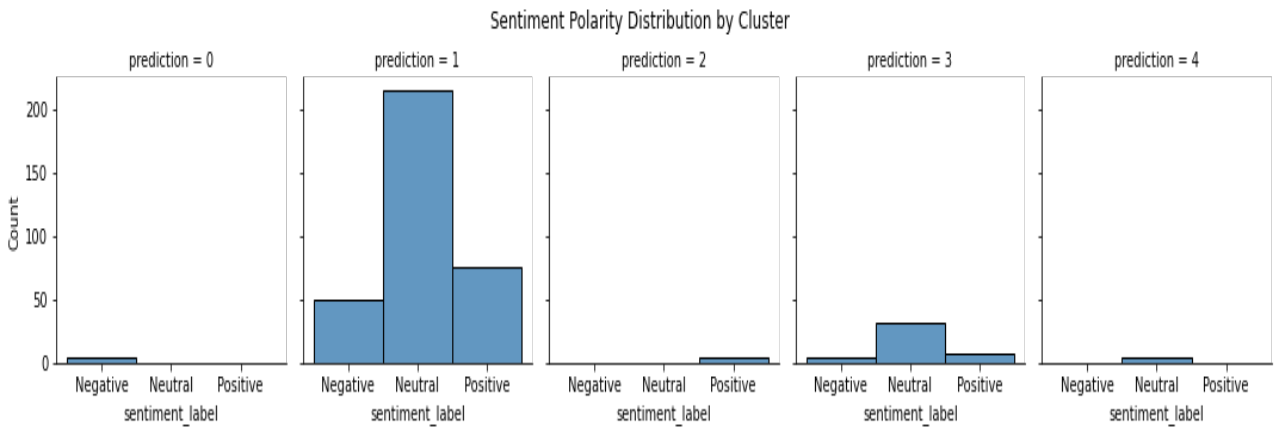


Figure 14 sentimental polarity distribution plot

5.8 Discussion

This section is to discuss the overall aim of this project the working principle of having a resilient system that have the capacity to ingest streaming data, process it to a structure the data, store for further process for different modelling technique depending on the objective, interactive query and visualization of data at the time the event occurs. Apache Spark a general parallel distributed engine's core abstraction, the streaming modules, Kafka streaming messaging system and data lake storage have been adopted in developing the proposed framework which is becoming a standard big streaming data analytics tool. This research project provides further insight into having a robust architecture that enable the extraction of data that are unbounded to time, structure it to a proper data frame, store and apply machine learning algorithm to both structured and unstructured stream data Spark Python object oriented API Pyspark for programming that works well on streaming dataframe. The framework was built on a local computer of 4GB RAM processing unit on Mac IOS operating system and Java as Spark uses Java serialization a way to translate data to suitable structure format from the data source to the consumer sink. It is obvious that the in memory processing ensures faster execution to query streaming data and structured streaming with an underlying sparksql structure allows dataset to be in data frame API which proves to have low latency, reliable and high throughput and uses catalyst optimizer which generates an optimized query The structured streaming executes jobs either as micro batch as a discretized stream and works well in load balancing or continuous processing mode, for this framework it executes with the latter which supports job rescaling at runtime achieving 1 millisecond end-to-end latencies with at-least-once guarantees. The data workload is stored into a non-sql database (data lake) as an output sink to avoid overload of the memory and read directly to apply iterative operations like machine learning which if accessed from the memory will require expensive computational resources while also ensuring data quality and privacy preservation. Spark offers a fast and scalable machine learning library MLLib, the data lake supports storage of large scale streaming data for further data preprocessing and exploration, and it is cost effective and not complicated to implement with Apache spark. It is easier to perform aggregate functions like sum, count, max, compare to (Carcillo et al., 2017) that implemented Cassandra storage for the streaming data and encountered issues.

Two datasets were used to test the reliability of the framework in dealing with dynamic data that streaming fast in near time, weather data was extracted and a schema defined, the data features were transformed using vector assembler before applying a supervised random forest regressor algorithm within a specific time window predict the air quality index based on the polluting gases all features and target feature are numeric values hence the use of a regression. Root mean squared error (rmse), r-squared and mean squared error evaluation metrics were used to evaluate the performance of the predictive model which determined the error rate between the predicted value and actual value of the air quality index on the test stream data and returns an average error accuracy, the lowest and best value of *0.174 (17.4%)* was estimated by the rmse evaluation method. The news feed extracted is an unstructured data and the interested topic was on climate change, natural language processing transformation techniques were performed and sentimental analysis on the occurrence of the word climate change in news is found to be 'Neural' the evaluation of unsupervised K-means clustering algorithm done with elbow method shows that setting k value to 4 is a better performance in predicting the interested key word from the extracted topic.

6. CONCLUSION

The overall aim of this research is to meet the computing demands of huge real time data analytics. It is necessary to design, execute, and manage the necessary pipelines and algorithms within an efficient framework. Scalable and cost effective data processing and management methodology will be crucial for the next generation of analytics applications to act and respond swiftly in circumstances when decisions must be made in real time. The designed framework leveraging Apache Spark Structured Streaming, Kafka, and Data Lake has been tested on two case study which shows evidence that the proposed framework successfully carry out an efficient large data applications to ease the process of real time data analytics from the source to the result expected output regardless of the velocity or pace at which the data is generated or variety of the data format. It also has a balanced latency with high throughput and fault tolerance an ability to restart with backup cluster nodes when a failure occurs, as experimented by (Armbrust et al., 2018) to validate these properties of the system. The case study data were chosen based on their dynamic and unpredictable nature to demonstrate the use of both supervised Regressor and unsupervised machine learning Clustering with textual sentimental analysis and natural language processing transformation which were also evaluated and optimized.

Despite the scalability and ability to append in coming streaming the structured streaming module of the Apache spark has a schema inference limitation. It is discovered that it does not instantly read on schema for streaming data compared to statics or batched data from files or parquet table which means if the json of a stream API is unknow or has large numbered of features it will be difficult to manually define the schema to be able to write as a structured streaming data frame however with the introduction of Kafka it allows to read to console the schema of the data stream to adopt for structured stream. A schema mechanism for data properties or key-value pairs selection needs to be explored to address this issue. Another limitation tackled is having access or storing previous machine learning predicted or analyzed dataset this is where the data lake serves as an advantage to store historical data as it is difficult to access this if the Kafka server or Apache cluster restarts to prevent data loss. It is also noticed that data were collected in micro batch which is static stream data in a near real time in structured streaming supposedly in every 16 minutes although still a decent low latency with thousands of data within the timeframe which leads to having visualization done on only the batched events it does not allow for an appropriate dynamic variation to visualize new events at the same time due to the large scale, unbounded, and unlimited nature of the data unless the visualization method is reconfigured to extract next real time appended data to enable a visualization of the incremental update of stream data to capture new patterns.

The future work could include experimenting with different data streaming sources and format other than in JSON such as Avro, incorporate automated schema inference and visualization methods for the streaming events on time.

7. Appendices

Appendix A Case Study One Code

CASE STUDY ONE: Weather Sensor Analysis with RandomForest Regression

Import modules integrated into spark Kafka and Delta Lake

In [3]:
```
import os
import findspark
import pyspark

SCALA_VERSION = '2.12'
SPARK_VERSION = '3.3.0'

os.environ['PYSPARK_SUBMIT_ARGS'] = f'--packages org.apache.spark:spark-sql-ka
org.apache.spark:spark-avro_2.12:3.3.2,io.delta:delta-core_2.12:2.2.0
--conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension"
--conf "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.Del
--conf spark.driver.extraJavaOptions=-Djdk.logger.allowReflection=true
--conf spark.executor.extraJavaOptions=-Djdk.logger.allowReflection=true pyspa

#findspark.init()
```

Import pyspark libraries

To start up spark session, define schema and data type

In [1]:
```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *

spark = SparkSession.builder \
    .appName("KafkaStream") \
    .getOrCreate()

#Set Spark logging level to ERROR to avoid various other logs on console.
spark.sparkContext.setLogLevel("ERROR")
spark.sql("set spark.sql.streaming.schemaInference=true")
```

Spark session created on local port 4040

In [5]:
```
spark
```

Out[5]:

SparkSession - in-memory

SparkContext

Spark UI

Version	`v3.3.1`
Master	`local[*]`
AppName	`KafkaStream`

Extract weather data with the stream data API

In [2]:
```
import requests
import json

url = "http://api.openweathermap.org/data/2.5/air_pollution/forecast?lat=52.95
response = requests.get(url)

if response.status_code == 200:
    data = response.json()
    # Do something with the data
else:
    print("Error fetching data from API")
```

Kafka Data Ingestion

In [3]:
```
from kafka import KafkaProducer

# Set up Kafka producer
producer = KafkaProducer(bootstrap_servers=['127.0.0.1:9092'],
                         value_serializer=lambda x: json.dumps(x).encode('

# Send data to Kafka topic
producer.send('weatherdata', data)
producer.flush()
```

Structured Streaming DataFrame

In [6]:
```
df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "127.0.0.1:9092") \
    .option("subscribe", "weatherdata") \
    .option("startingOffsets", "earliest") \
    .load()
```

In [2]:
```
df.printSchema()
```

In [7]:
```
df = df.selectExpr("CAST(value AS STRING)","timestamp")
```

In [8]:
```
schema = "array<struct<main:struct<aqi:int>, components:struct<co:double, no:d
```

In [9]:
```
df_new = df.select(get_json_object(col("value"), "$.coord").alias("coord"),exp
```

In [10]:
```
df_n = df_new.select(col("exploded_col.main.aqi").alias("aqi"),
                col("exploded_col.components.co").alias("co"),
                col("exploded_col.components.no").alias("no"),
                col("exploded_col.components.no2").alias("no2"),
                col("exploded_col.components.o3").alias("o3"),
                col("exploded_col.components.so2").alias("so2"),
                col("exploded_col.components.pm2_5").alias("pm2_5"),
                col("exploded_col.components.pm10").alias("pm10"),
                col("exploded_col.components.nh3").alias("nh3"),
                col("exploded_col.dt").alias("dt"),col("timestamp").alias("time
```

In [11]:
```
# extract hour from timestamp column
df_n = df_n.withColumn("minutes", minute("timestamp"))
df_n = df_n.drop(col("hour"))
```

In [12]:
```
window_df_count_sum = df_n.withWatermark("timestamp", "2 hour").groupBy(window
```

In [13]:
```
window_df_max = df_n.groupBy(window(df_n.timestamp, "1 hour")).agg({"aqi": "ma
```

In [14]:
```
import pandas as pd

# Define a function to write each micro-batch to a Pandas DataFrame
def write_to_pandas(df, epoch_id):
    # Convert the Spark DataFrame to a Pandas DataFrame
    pandas_df = df.toPandas()
    # Print the Pandas DataFrame
    display(pandas_df)
```

In [15]:
```
# Define the streaming query
streaming_query = window_df_count_sum.writeStream \
    .foreachBatch(write_to_pandas) \
    .outputMode("complete") \
    .start()
```

	window	count(no2)	max(aqi)	sum(co)
0	(2023-04-09 21:00:00, 2023-04-09 22:00:00)	96	3	21285.52

In [16]:
```
# Define the streaming query
streaming_query = window_df_max.writeStream \
    .foreachBatch(write_to_pandas) \
    .outputMode("complete") \
    .start()
```

	window	max(aqi)
0	(2023-04-09 21:00:00, 2023-04-09 22:00:00)	3

In [17]:
```
df_n.printSchema()
```

```
root
 |-- aqi: integer (nullable = true)
 |-- co: double (nullable = true)
 |-- no: double (nullable = true)
 |-- no2: double (nullable = true)
 |-- o3: double (nullable = true)
 |-- so2: double (nullable = true)
 |-- pm2_5: double (nullable = true)
 |-- pm10: double (nullable = true)
 |-- nh3: double (nullable = true)
 |-- dt: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- minutes: integer (nullable = true)
```

In [18]:
```
stream_query = df_n.writeStream \
    .outputMode("append") \
    .format("memory") \
    .queryName("my_query") \
    .start()

stream = spark.sql("SELECT * FROM my_query")
```

In [19]:
```
from delta.tables import *
# Write the stream to Delta Lake
stream2 =stream.write.format("delta").mode("append").save("/tmp/df_n_table")

# Read the streaming from Delta Lake and display as pandas table
stream3 = spark.read.format("delta").load("/tmp/df_n_table")

# Load Delta table as DataFrame
display(DeltaTable.forPath(spark, "/tmp/df_n_table").toDF().toPandas())
```

	aqi	co	no	no2	o3	so2	pm2_5	pm10	nh3	dt	timestamp	minu
0	3	226.97	0.00	7.28	104.43	3.19	9.65	10.43	3.55	1681070400	2023-04-09 21:16:09.481	
1	3	230.31	0.00	7.28	101.57	3.13	11.05	11.87	3.52	1681074000	2023-04-09 21:16:09.481	
2	3	233.65	0.00	6.86	100.14	2.68	12.30	13.17	3.45	1681077600	2023-04-09 21:16:09.481	
3	2	230.31	0.00	6.00	98.71	2.18	13.00	13.88	3.55	1681081200	2023-04-09 21:16:09.481	
4	2	230.31	0.00	6.08	92.98	2.41	15.75	16.68	3.80	1681084800	2023-04-09 21:16:09.481	
...	...	...	...	...	...	...	...	...	...	...	...	
91	2	226.97	1.08	8.23	60.80	2.65	1.13	1.30	0.66	1681225200	2023-04-09 21:16:09.481	
92	1	226.97	1.23	7.88	58.65	2.62	1.03	1.20	0.74	1681228800	2023-04-09 21:16:09.481	
93	2	223.64	0.85	6.17	62.23	1.86	0.76	0.90	1.00	1681232400	2023-04-09 21:16:09.481	
94	2	220.30	0.30	4.50	71.53	1.45	0.83	1.05	1.24	1681236000	2023-04-09 21:16:09.481	
95	2	216.96	0.03	3.81	80.11	1.42	1.22	1.89	1.50	1681239600	2023-04-09 21:16:09.481	

96 rows × 12 columns

In [21]:
```
stream3.printSchema()
```

```
root
 |-- aqi: integer (nullable = true)
 |-- co: double (nullable = true)
 |-- no: double (nullable = true)
 |-- no2: double (nullable = true)
 |-- o3: double (nullable = true)
 |-- so2: double (nullable = true)
 |-- pm2_5: double (nullable = true)
 |-- pm10: double (nullable = true)
 |-- nh3: double (nullable = true)
 |-- dt: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- minutes: integer (nullable = true)
```

In [ ]:
```
import plotly.io as pio
pio.renderers.default = "iframe"
import plotly.graph_objects as go
import chart_studio.plotly as py
py.sign_in("adeoye.a.kafayat", "51BL1baAC505SzD7FM8A")
```

In [ ]:
```
import plotly.express as px
px.colors.qualitative.Plotly;
```

In [22]:
```
# Machine learning libaraaries
from pyspark.ml.feature import *
from pyspark.ml.classification import *
from pyspark.ml.regression import *
from pyspark.ml.evaluation import *
from pyspark.ml.tree import *
```

In [ ]:
```
# transformation
# Define the input features and target variable column names
feature_cols = ["co","no","no2","o3","so2","pm2_5","pm10","nh3"]
# Define the VectorAssembler to transform input features into a vector column
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
```

In [23]:
```
# Split the data into training and test sets (30% held out for testing)
(trainingData, testData) = stream3.randomSplit([0.7, 0.3])
```

Random Forest Regression

In [27]:
```
# Define the multinomial Regression model with the features column and target
# Train a RandomForest model.
rf = RandomForestRegressor(featuresCol="features", labelCol='aqi')
```

In [28]:
```
from pyspark.ml import *
# Chain the VectorAssembler, VectorIndexer, and Logistic Regression model intc
rf_pipeline = Pipeline(stages=[assembler,rf])
```

In [29]:
```
# Fit the Pipeline to the streaming data
rf_pipeline_model = rf_pipeline.fit(trainingData)
```

In [33]:
```
# Make predictions on the streaming data using the fitted Pipeline
rf_predictions = rf_pipeline_model.transform(testData)
```

In [32]:
```
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(labelCol='aqi', metricName="rmse")
rmse = evaluator.evaluate(rf_predictions)
print(f"Root Mean Squared Error (MSE) on test data: {rmse}")
```

```
Root Mean Squared Error (MSE) on test data: 0.1747352494701839
```

In [34]:
```
rfModel = rf_pipeline_model.stages[1]
print(rfModel)  # summary only
```

```
RandomForestRegressionModel: uid=RandomForestRegressor_591083c8f70b, numTrees=
20, numFeatures=8
```

In [35]:
```
# Write the predictions to Delta Lake
rf_predictions.write.format("delta").mode("append").save("/tmp/rf_tablepredict

# Read the predictions from Delta Lake and display as pandas table
rf_pred_df = spark.read.format("delta").load("/tmp/rf_tableprediction")
display(rf_pred_df.toPandas())
```

	aqi	co	no	no2	o3	so2	pm2_5	pm10	nh3	dt	timestamp	minu
0	2	213.62	0.00	3.21	72.24	1.25	1.15	1.86	2.25	1681174800	2023-04-09 21:16:09.481	
1	2	213.62	0.00	3.38	72.24	1.40	1.16	1.91	2.09	1681182000	2023-04-09 21:16:09.481	
2	2	216.96	0.00	3.47	70.81	1.30	1.31	2.10	2.31	1681171200	2023-04-09 21:16:09.481	
3	2	216.96	0.00	3.60	72.24	1.54	1.28	2.02	2.00	1681185600	2023-04-09 21:16:09.481	
4	2	220.30	0.30	4.50	71.53	1.45	0.83	1.05	1.24	1681236000	2023-04-09 21:16:09.481	
5	2	223.64	0.35	8.40	69.38	2.65	1.27	2.00	1.87	1681196400	2023-04-09 21:16:09.481	
6	2	223.64	0.61	8.65	73.67	3.16	1.36	2.11	1.65	1681207200	2023-04-09 21:16:09.481	
7	2	223.64	0.85	6.17	62.23	1.86	0.76	0.90	1.00	1681232400	2023-04-09 21:16:09.481	
8	1	233.65	0.00	14.74	46.49	3.07	4.60	5.04	1.66	1681344000	2023-04-09 21:16:09.481	

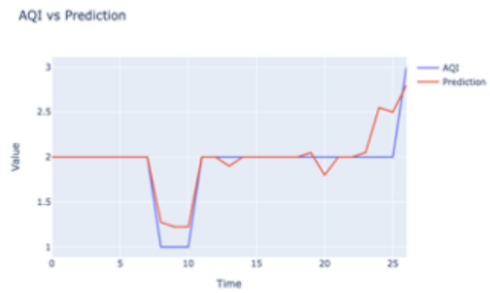
	aqi	co	no	no2	o3	so2	pm2_5	pm10	nh3	dt	timestamp	minu
17	2	216.96	0.33	4.97	79.39	1.65	0.50	0.60	1.09	1681311600	2023-04-09 21:16:09.481	
18	2	216.96	0.57	6.00	81.54	1.94	0.75	1.71	2.57	1681286400	2023-04-09 21:16:09.481	
19	2	220.30	0.00	3.04	87.26	1.55	0.71	1.83	2.85	1681261200	2023-04-09 21:16:09.481	
20	2	230.31	0.18	15.08	60.08	4.47	1.13	1.35	1.60	1681322400	2023-04-09 21:16:09.481	
21	2	208.62	0.08	7.97	69.38	2.35	3.35	3.67	0.94	1681110000	2023-04-09 21:16:09.481	
22	2	216.96	0.86	3.98	86.55	2.27	1.40	1.83	1.63	1681128000	2023-04-09 21:16:09.481	
23	2	226.97	0.00	5.44	77.96	2.06	14.46	14.75	1.22	1681099200	2023-04-09 21:16:09.481	
24	2	230.31	0.00	6.00	98.71	2.18	13.00	13.88	3.55	1681081200	2023-04-09 21:16:09.481	
25	2	230.31	0.00	6.08	92.98	2.41	15.75	16.68	3.80	1681084800	2023-04-09 21:16:09.481	

	aqi	co	no	no2	o3	so2	pm2_5	pm10	nh3	dt	timestamp	minu
26	3	230.31	0.00	7.28	101.57	3.13	11.05	11.87	3.52	1681074000	2023-04-09 21:16:09.481	

In [38]:
```
display(rf_pred_df.select("aqi","minutes","prediction").toPandas().head(10))
```

	aqi	minutes	prediction
0	2	16	2.000
1	2	16	2.000
2	2	16	2.000
3	2	16	2.000
4	2	16	2.000
5	2	16	2.000
6	2	16	2.000
7	2	16	2.000
8	1	16	1.275
9	1	16	1.225

In [5]:
```
# plot the prediction vs actual value
pred_df = rf_pred_df.toPandas().head(1000)

# Create a scatter plot of AQI vs prediction
fig = go.Figure()
fig.add_trace(go.Scatter(x=pred_df.index, y=pred_df['aqi'], mode='lines', name
fig.add_trace(go.Scatter(x=pred_df.index, y=pred_df['prediction'], mode='lines

# Add titles and axis labels
fig.update_layout(title='AQI vs Prediction', xaxis_title='Time', yaxis_title='

# Show the plot
fig.show()
```



Random Forest Evaluation with RMSE, Rsquared and MAE

In [ ]:

```
# Define the evaluation metric
evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="aqi", me

# Calculate the evaluation metrics for the test data
rmse = evaluator.evaluate(rf_predictions, {evaluator.metricName: "rmse"})
r2 = evaluator.evaluate(rf_predictions, {evaluator.metricName: "r2"})
mae = evaluator.evaluate(rf_predictions, {evaluator.metricName: "mae"})

# Print the evaluation metrics
print("RMSE = %g" % rmse)
print("R-squared = %g" % r2)
print("MAE = %g" % mae)
```

Appendix B Case Study Two Code

CASE STUDY TWO: News Feed Clustering and Sentimental Analysis

Extract weather data with the stream data API

In [37]:
```
import requests
import json

url = "https://newsapi.org/v2/everything?q=climate&apiKey=408f9fb0be5d4aa4b480
response_1 = requests.get(url)

if response_1.status_code == 200:
    data_1 = response_1.json()
    #print(data_1)
    # Do something with the data
else:
    print("Error fetching data from API")
```

Kafka Data Ingestion

In [2]:
```
from kafka import KafkaProducer

# Set up Kafka producer
producer = KafkaProducer(bootstrap_servers=['127.0.0.1:9092'],
                         value_serializer=lambda x: json.dumps(x).encode('

# Send data to Kafka topic
producer.send('newsfeeddata1', data_1)
producer.flush()
```

Spark Session Set up

In [4]:
```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *

spark = SparkSession.builder \
    .appName("KafkaNewsStream") \
    .getOrCreate()

#Set Spark logging level to ERROR to avoid various other logs on console.
spark.sparkContext.setLogLevel("ERROR")
spark.sql("set spark.sql.streaming.schemaInference=true");
```

In [41]:
```
spark
```

Out[41]: **SparkSession - in-memory**

SparkContext

Spark UI

Version	`v3.3.1`
Master	`local[*]`
AppName	`KafkaNewsStream`

Define Stream Data Schema and Data Type

In [43]:
```
# define the schema for the articles field
article_schema = StructType([
    StructField("title", StringType()),
    StructField("author", StringType()),
    StructField("source", StructType([
        StructField("id", StringType()),
        StructField("name", StringType())
    ])),
    StructField("publishedAt", TimestampType()),
    StructField("url", StringType())
])

# define the main schema for the entire JSON message
json_schema = StructType([
    StructField("status", StringType()),
    StructField("totalResults", StringType()),
    StructField("articles", ArrayType(article_schema))
])
```

Structured Streaming Module

Setting the "failOnDataLoss" option to "false" means that Spark will not fail the query if it detects data loss, but instead, it will log a warning message. This is useful when you want to tolerate some data loss in your streaming application, and you don't want it to fail every time it detects data loss

In [44]:
```
# read streaming data from Kafka
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "127.0.0.1:9092") \
    .option("subscribe", "newsfeeddata1") \
    .option("startingOffsets", "earliest") \
    .option("failOnDataLoss", "false") \
    .load()
```

Parse to derive data values from Article nested json object

In [45]:
```
# convert the binary Kafka value to string and parse the JSON
parsed_df = df.selectExpr("CAST(value AS STRING)") \
    .select(from_json("value", json_schema, options={"allowUnquotedControlChar
    .selectExpr("parsed_value.status",
                "parsed_value.totalResults",
                "inline_outer(parsed_value.articles)").select("title", "author
```

In [46]:
```
# print schema for article object
parsed_df.printSchema()
```

```
root
 |-- title: string (nullable = true)
 |-- author: string (nullable = true)
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- publishedAt: timestamp (nullable = true)
 |-- url: string (nullable = true)
```

In [47]:
```
import pandas as pd

# Define a function to write each micro-batch to a Pandas DataFrame
def write_to_pandas(df, epoch_id):
    # Convert the Spark DataFrame to a Pandas DataFrame
    pandas_df = df.toPandas()
    # Print the Pandas DataFrame
    display(pandas_df)
```

In [48]:
```
# Define the streaming query
streaming_data = parsed_df.writeStream \
    .foreachBatch(write_to_pandas) \
    .outputMode("append") \
    .start()
```

```
2023-04-30 19:59:38.640 INFO    py4j.clientserver: Python Server ready to rece
ive messages
2023-04-30 19:59:38.643 INFO    py4j.clientserver: Received command c on objec
t id p2
```

	title	author	id	name	publishedAt	
0	Washington, DC's Cherry Blooms Draw Crowds—and...	Emma Ricketts	None	None	2023-04-01 13:00:00	https://www.wired.com/story/washington-
1	Can Burning Man Pull Out of Its Climate Death ...	Alden Wicker	None	None	2023-04-04 13:00:00	https://www.wired.com/story/burning-ma
2	Mitsubishi wants to be the world's carbon broker	Justine Calma	None	None	2023-04-28 19:45:59	https://www.theverge.com/2023/4/28/2370
3	To capture CO2 in the US, climate tech startup...	Justine Calma	None	None	2023-04-21 19:54:20	https://www.theverge.com/2023/4/21/2369
4	The Link Between Tornadoes and Climate Change ...	Angely Mercado	None	None	2023-03-31 19:34:00	https://gizmodo.com/tornadoes-and-clima
...	...	...	...	...	...	
391	The EPA's watchdog is warning about oversight ...	Eric McDaniel	None	None	2023-04-01 10:01:09	https://www.npr.org/2023/04/01/11673346
392	Meet the One Planet Advisory Council, a group ...	Business Insider	None	None	2023-04-03 19:02:05	https://www.businessinsider.com/meet-on
393	The steps that could help address biodiversity...	Catherine Boudreau	None	None	2023-04-24 20:05:52	https://www.businessinsider.com/biodi
394	3 experts discuss the impact of climate change...	Elizabeth Wood	None	None	2023-04-25 18:01:46	https://www.businessinsider.com/experts
395	They found a hole in the ocean near Mexico, an...	Nicholas Carlson	None	None	2023-04-24 16:17:59	https://www.businessinsider.com/taam-j

396 rows × 6 columns

Delta Table

In [49]:
```
stream_query = parsed_df.writeStream \
    .outputMode("append") \
    .format("memory") \
```

```
    .queryName("word_my_query") \
    .start()

stream = spark.sql("SELECT * FROM word_my_query")
```

In [50]:
```
from delta.tables import *
# Write the stream to Delta Lake
stream1 =stream.write.format("delta").mode("append").save("/tmp/df_title_table

# Read the streaming from Delta Lake and display as pandas table
stream2 = spark.read.format("delta").load("/tmp/df_title_table")

# Load Delta table as DataFrame
display(DeltaTable.forPath(spark, "/tmp/df_title_table").toDF().toPandas())
```

	title	author	id	name	publishedAt	
0	Washington, DC's Cherry Blooms Draw Crowds—and...	Emma Ricketts	None	None	2023-04-01 13:00:00	https://www.wired.com/story/washington-c
1	Can Burning Man Pull Out of Its Climate Death ...	Alden Wicker	None	None	2023-04-04 13:00:00	https://www.wired.com/story/burning-man-
2	Mitsubishi wants to be the world's carbon broker	Justine Calma	None	None	2023-04-28 19:45:59	https://www.theverge.com/2023/4/28/237023
3	To capture CO2 in the US, climate tech startup...	Justine Calma	None	None	2023-04-21 19:54:20	https://www.theverge.com/2023/4/21/23690
4	The Link Between Tornadoes and Climate Change ...	Angely Mercado	None	None	2023-03-31 19:34:00	https://gizmodo.com/tornadoes-and-climate
...	...	...	...	...	...	
391	Cop28 President: World Needs Business Mindset ...	msmash	None	None	2023-04-07 21:40:00	https://news.slashdot.org/story/23/04/07/
392	It's Way Too Easy to Get Google's Bard Chatbot...	Vittoria Elliott	None	None	2023-04-05 05:47:22	https://www.wired.com/story/its-way-too-
393	Noah Raford Can Help You Prepare for a Not-So-...	WIRED Staff	None	None	2023-04-26 12:00:00	https://www.wired.com/story/have-a-nice-
394	The UK will spend £100 million to develop its ...	Jon Fingas	None	None	2023-04-24 15:35:07	https://www.engadget.com/the-uk-is-creat
395	The week around the world in 20 pictures	Jim Powell	None	None	2023-04-28 20:35:53	https://www.theguardian.com/artanddesign

396 rows × 6 columns

In [ ]:

Machine Learning

In [149…
```
# Machine learning libararies
from pyspark.ml.feature import *
from pyspark.ml.clustering import *
from pyspark.sql.functions import *
from pyspark.ml import *
from pyspark.ml.evaluation import *
from textblob import TextBlob
import matplotlib.pyplot as plt
import streamlit as st
import seaborn as sns
```

NLP Transformation: tokenizer, stop words, vectorizer

In [52]:
```
# apply transformations to streaming DataFrame

# Tokenize the title column
tokenizer = Tokenizer(inputCol="title", outputCol="words")
words_df = tokenizer.transform(stream2)

# Remove stop words from the tokenized words
remover = StopWordsRemover(inputCol="words", outputCol="filtered_words")
filtered_df = remover.transform(words_df)

# Vectorize the filtered words
vectorizer = CountVectorizer(inputCol="filtered_words", outputCol="features")
vectorized_df = vectorizer.fit(filtered_df).transform(filtered_df)

#pipeline = Pipeline(stages=[tokenizer, stopwords_remover, vectorizer])
```

Frequency

In [58]:
```
# create the CountVectorizer model on the streaming data
cv = CountVectorizer(inputCol="filtered_words", outputCol="features", vocabSiz
cv_model = cv.fit(filtered_df)

# define a UDF to convert the features column to a string column
features_to_str = udf(lambda x: ' '.join([str(i) for i in x.indices]), StringT

# apply the UDF to the features column
result = cv_model.transform(filtered_df)
result = result.withColumn("feature_str", features_to_str(result["features"]))

# split the feature_str column and count the words
word_counts = result \
    .select(explode(split("feature_str", " ")).alias("word")) \
    .groupBy("word") \
    .count() \
    .orderBy("count", ascending=False)
```

In [59]:
```
display(word_counts.toPandas())
```

	word	count
0	0	308
1	1	186
2	2	152
3	3	48
4	4	46
...	...	...
613	608	2
614	569	2
615	611	2
616	542	2
617	572	2

618 rows × 2 columns

In [60]:
```
# get the vocabulary from the CountVectorizerModel
vocab = cv_model.vocabulary

# create a dataframe with word and count columns
word_count = spark.createDataFrame([(word, count) for word, count in zip(vocab

# order the dataframe by count in descending order
word_count = word_count.orderBy("count", ascending=False)

# replace None values with 0
word_count = word_count.na.fill(0)
```

In [61]:
```
# Replace '-' and '...' with an empty string in the 'word' column
word_freq = word_count.withColumn('word', regexp_replace('word', '[-|...]','')
```

In [62]:
```
# display the dataframe
word_freq_dis = display(word_freq.limit(20).na.drop().toPandas())
```

	word	count
0	climate	308
1		186
2	reuters	152
3		48
4	change	46
5	reuterscom	28
6	vote	24
7	activists	24
8	world	22
9	resolution	22
10	crisis	20
11	targets	20
12	oil	16
13	un	16
14	investors	12
15	africa	12
16	bp	12
17	big	12
18	eu	12
19	tech	12

K-Means Clustering

In [114…

```
# Train the k-means model with k = 5
kmeans = KMeans(k=5, seed=1)
model = kmeans.fit(vectorized_df.select("features"))

# Add the predicted cluster column to the DataFrame
kmeans_predictions = model.transform(vectorized_df)

# Convert the DataFrame to pandas for visualization
kmeans_pandas_df = kmeans_predictions.select("title", "prediction").toPandas()
```

In [134…

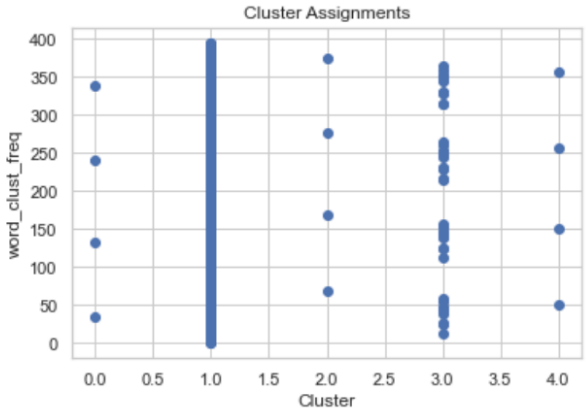
```
display(kmeans_pandas_df)
```

	title	prediction
0	Washington, DC's Cherry Blooms Draw Crowds—and...	1
1	Can Burning Man Pull Out of Its Climate Death ...	1
2	Mitsubishi wants to be the world's carbon broker	1
3	To capture CO2 in the US, climate tech startup...	1
4	The Link Between Tornadoes and Climate Change ...	1
...	...	...
391	Cop28 President: World Needs Business Mindset ...	1
392	It's Way Too Easy to Get Google's Bard Chatbot...	1
393	Noah Raford Can Help You Prepare for a Not-So-...	1
394	The UK will spend £100 million to develop its ...	1
395	The week around the world in 20 pictures	1

396 rows × 2 columns

have a larger number of data points that share common words, resulting in more scattered bubbles in those clusters.

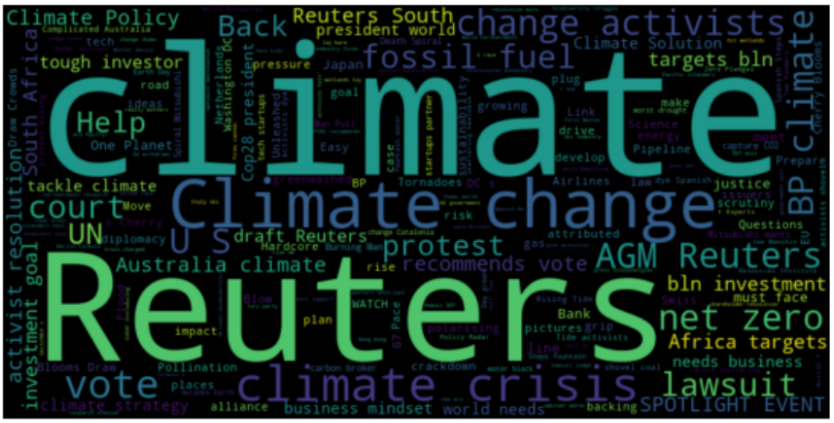
In [115…
```
# Visualize the clusters with a scatter plot
plt.scatter(kmeans_pandas_df['prediction'], kmeans_pandas_df.index)
plt.xlabel('Cluster')
plt.ylabel('word_clust_freq')
plt.title('Cluster Assignments')
plt.show()
```



Word Cloud

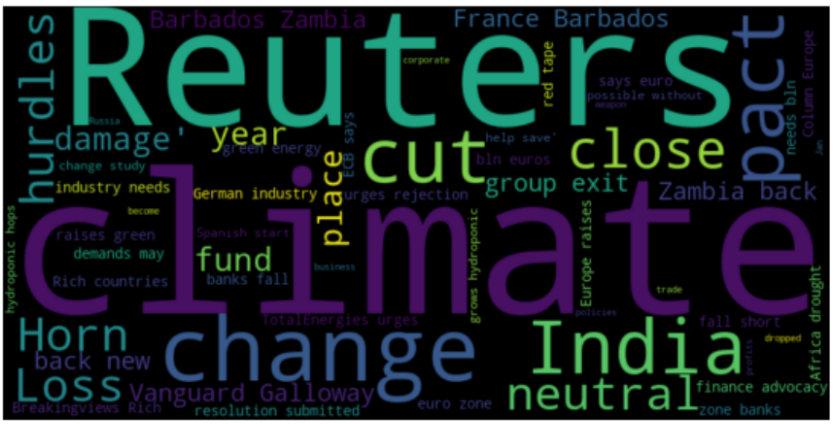
In [121…
```
# Filter the kmeans_pandas_df to get the rows in the most clustered cluster
most_clustered_cluster = kmeans_pandas_df[kmeans_pandas_df['prediction'] == 1]

# Combine the titles into a single string
text = ' '.join(most_clustered_cluster['title'])

# Create the word cloud
wordcloud = WordCloud(width=800, height=400).generate(text)
```

```
# Display the word cloud
fig, ax = plt.subplots(figsize=(10, 5))
ax.imshow(wordcloud, interpolation='bilinear')
ax.axis("off")
plt.show()
```



In [128…

```
# Filter the kmeans_pandas_df to get the rows in the most clustered cluster
most_clustered_cluster = kmeans_pandas_df[kmeans_pandas_df['prediction'] == 3]

# Combine the titles into a single string
text = ' '.join(most_clustered_cluster['title'])

# Create the word cloud
wordcloud = WordCloud(width=800, height=400).generate(text)

# Display the word cloud
fig, ax = plt.subplots(figsize=(10, 5))
ax.imshow(wordcloud, interpolation='bilinear')
ax.axis("off")
plt.show()
```



Increase value of k to 7

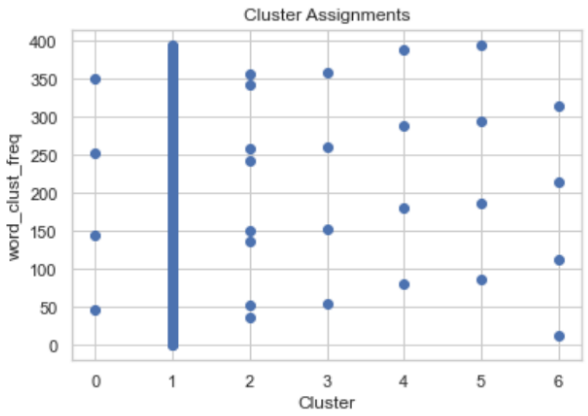
In [135…

```
# Train the k-means model with k = 5
kmeans_7 = KMeans(k=7, seed=1)
model_7 = kmeans_7.fit(vectorized_df.select("features"))
```

```
# Add the predicted cluster column to the DataFrame
kmeans_predictions_7 = model_7.transform(vectorized_df)

# Convert the DataFrame to pandas for visualization
kmeans_pandas_df_7 = kmeans_predictions_7.select("title", "prediction").toPand
```

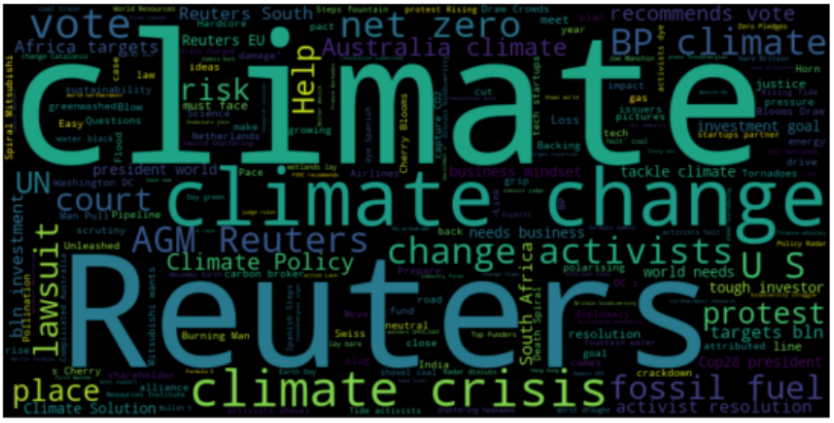
In [136…

```
# Visualize the clusters with a scatter plot
plt.scatter(kmeans_pandas_df_7['prediction'], kmeans_pandas_df_7.index)
plt.xlabel('Cluster')
plt.ylabel('word_clust_freq')
plt.title('Cluster Assignments')
plt.show()
```



In [140…

```
# Filter the kmeans_pandas_df to get the rows in the most clustered cluster
most_clustered_cluster = kmeans_pandas_df_7[kmeans_pandas_df_7['prediction'] =

# Combine the titles into a single string
text = ' '.join(most_clustered_cluster['title'])

# Create the word cloud
wordcloud = WordCloud(width=800, height=400).generate(text)

# Display the word cloud
fig, ax = plt.subplots(figsize=(10, 5))
ax.imshow(wordcloud, interpolation='bilinear')
ax.axis("off")
plt.show()
```



Sentimental analysis

In [92]:
```
def get_sentiment(text):
    blob = TextBlob(str(text))
    return blob.sentiment.polarity

# Define a UDF for the get_sentiment() function
get_sentiment_udf = udf(get_sentiment, FloatType())


# Add a new column with the sentiment polarity
vectorized_df_sen = vectorized_df.withColumn('sentiment_polar', get_sentiment_

# Add a new column with the sentiment label
vectorized_df_sen = vectorized_df_sen.withColumn('sentiment_label', get_sentim
```

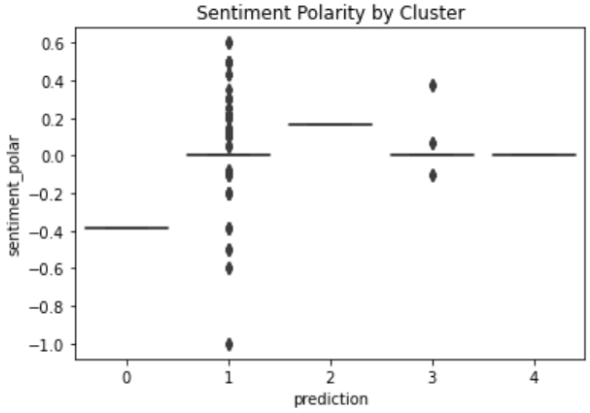
In [71]:
```
# Define a function to get the sentiment polarity of a word
def get_word_sentiment(text):
    blob = TextBlob(text)
    return blob.sentiment.polarity

# Define a UDF for the get_word_sentiment() function
get_word_sentiment_udf = udf(get_word_sentiment, FloatType())
```

In [72]:
```
# Create a new dataframe with the sentiment polarity of each word
sentiment_df = word_count.selectExpr("explode(split(word, ' ')) as word_p") \
                    .withColumn('sentiment_polarity', get_word_sentiment_udf('
                    .filter('sentiment_polarity != 0')
```

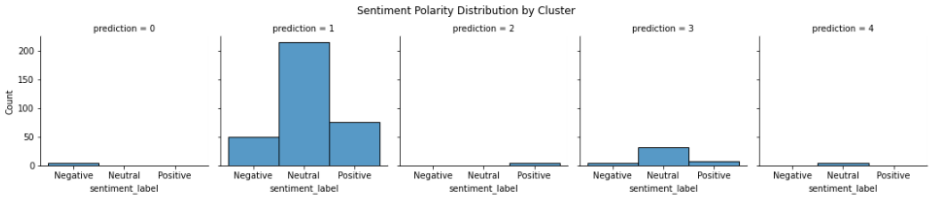
In [73]:
```
# Define a function to get the sentiment label
def get_sentiment_label(polarity):
    if polarity > 0:
        return 'Positive'
    elif polarity < 0:
        return 'Negative'
    else:
        return 'Neutral'

# Define a UDF for the get_sentiment_label() function
get_sentiment_label_udf = udf(get_sentiment_label, StringType())

# Add a new column with the sentiment label
sentiment_df = sentiment_df.withColumn('sentiment_label', get_sentiment_label_
```

In [98]:
```
kmeans = KMeans(k=4, seed=1)
model_k = kmeans.fit(vectorized_df_sen.select("features"))

# Add the predicted cluster column to the DataFrame
#kmeans_predictions_sen = model_k.transform(vectorized_df_sen)

kmeans_predictions_sen = model_k.transform(vectorized_df_sen.select("title","f

# Convert the DataFrame to pandas for visualization
kmeans_pandas_df_sen = kmeans_predictions_sen.select("title", "prediction","se
```

In [95]:
```
sns.boxplot(x='prediction', y='sentiment_polar', data=kmeans_pandas_df_sen)
plt.title('Sentiment Polarity by Cluster')
plt.show()
```



In [99]:
```
g = sns.FacetGrid(kmeans_pandas_df_sen, col='prediction')
g.map(sns.histplot, 'sentiment_label')
plt.suptitle('Sentiment Polarity Distribution by Cluster', y=1.05)
plt.show()
```



Clustering Evaluation

Silhouette Score

In [153…
```
# Evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()

silhouette = evaluator.evaluate(kmeans_predictions)
print("Silhouette Score = " + str(silhouette))
```

```
Silhouette Score = -0.013331837391900355
```

In [154…
```
# Evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()

silhouette = evaluator.evaluate(kmeans_predictions_7)
print("Silhouette Score = " + str(silhouette))
```

```
Silhouette Score = -0.06381339712006173
```

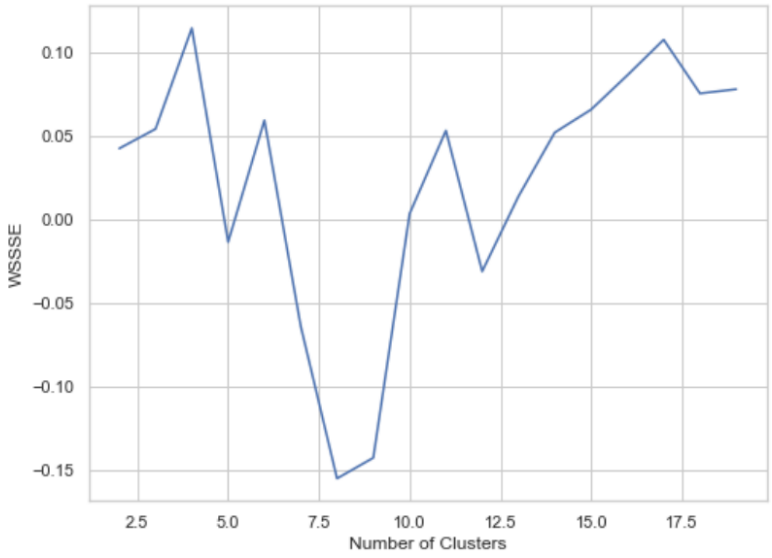
Within-Cluster Sum of Squares (WCSS)

In [156…
```
# Calculate Within-Cluster Sum of Squares (WCSS)
evaluator = ClusteringEvaluator()
wcss = evaluator.evaluate(kmeans_predictions)
print("Within-Cluster Sum of Squares = " + str(wcss))
```

```
Within-Cluster Sum of Squares = -0.013331837391900355
```

Elbow method

In [159…
```
# elbow method to find optimal number of clusters
cost = np.zeros(20)
for k in range(2,20):
    kmeans = KMeans().setK(k).setSeed(1).setFeaturesCol("features")
    model = kmeans.fit(vectorized_df)
    predictions = model.transform(vectorized_df)
    evaluator = ClusteringEvaluator()
    cost[k] = evaluator.evaluate(predictions)

fig, ax = plt.subplots(1,1, figsize =(8,6))
ax.plot(range(2,20),cost[2:20])
ax.set_xlabel('Number of Clusters')
ax.set_ylabel('WSSSE')
plt.show()
```



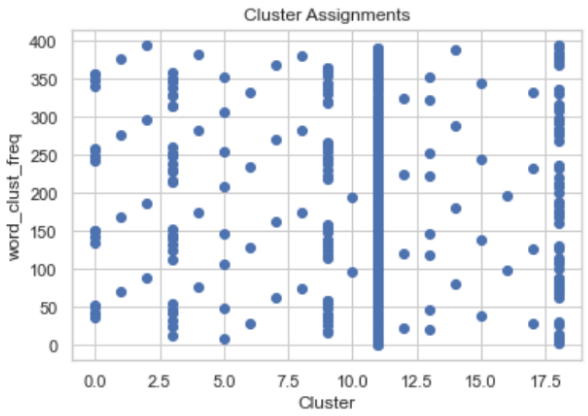
Elbow Method Optimised Clusters k= 4

In [160…
```
# Train the k-means model with k = 5
kmeans_4 = KMeans(k=4, seed=1)
model_4 = kmeans.fit(vectorized_df.select("features"))
```

```
# Add the predicted cluster column to the DataFrame
kmeans_predictions_4 = model_4.transform(vectorized_df)

# Convert the DataFrame to pandas for visualization
kmeans_pandas_df_4 = kmeans_predictions_4.select("title", "prediction").toPand
```

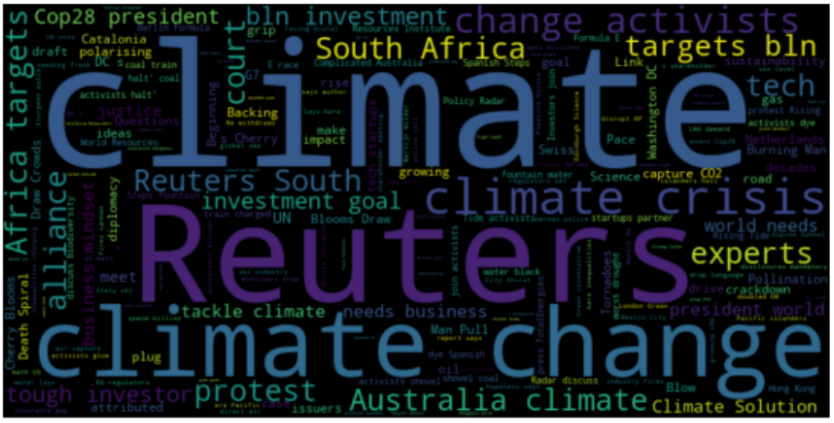
In [161…

```
# Visualize the clusters with a scatter plot
plt.scatter(kmeans_pandas_df_4['prediction'], kmeans_pandas_df_4.index)
plt.xlabel('Cluster')
plt.ylabel('word_clust_freq')
plt.title('Cluster Assignments')
plt.show()
```



In [167…

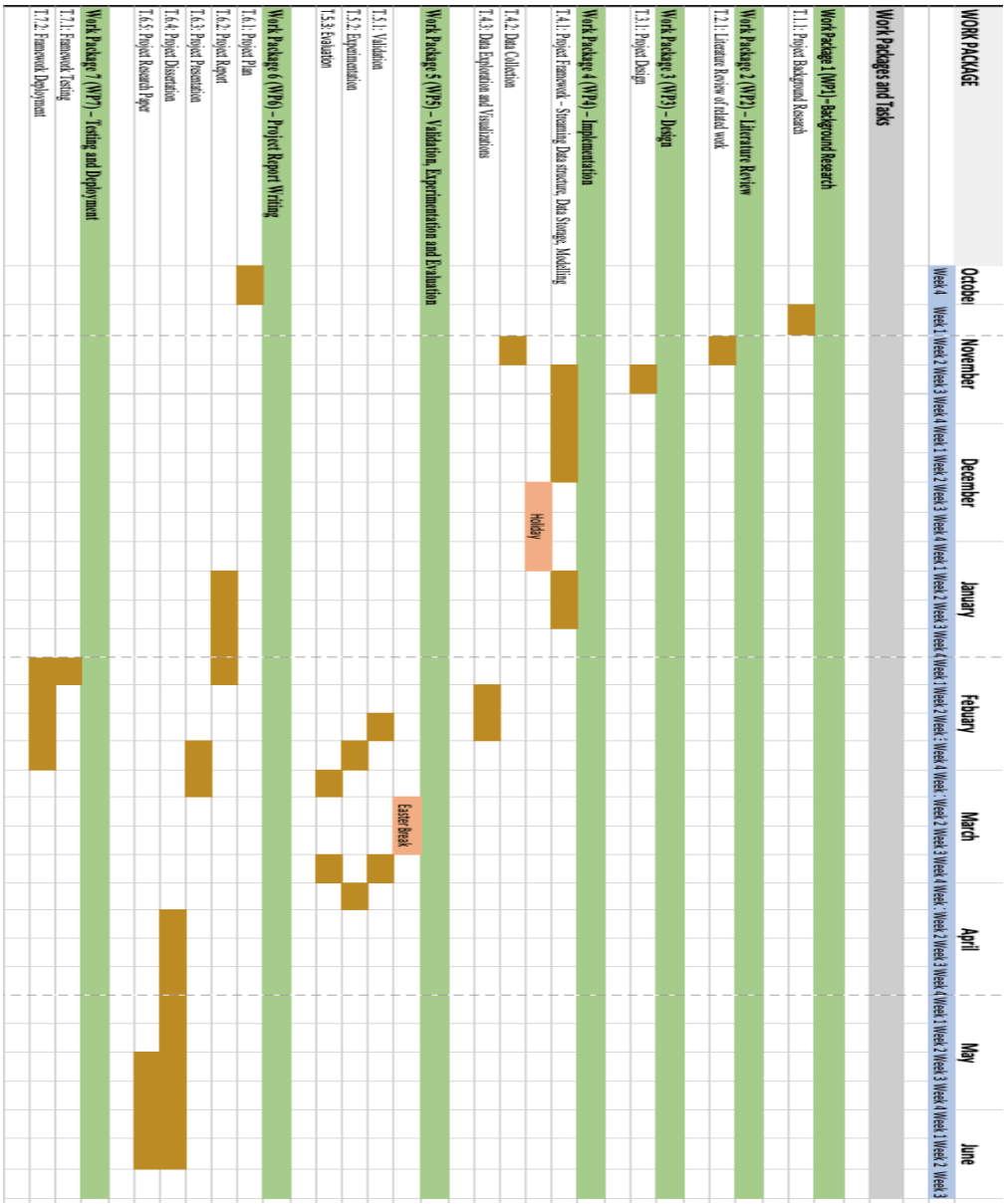
```
# Filter the kmeans_pandas_df to get the rows in the most clustered cluster
most_clustered_cluster = kmeans_pandas_df[kmeans_pandas_df_4['prediction'] ==

# Combine the titles into a single string
text = ' '.join(most_clustered_cluster['title'])

# Create the word cloud
wordcloud = WordCloud(width=800, height=400).generate(text)

# Display the word cloud
fig, ax = plt.subplots(figsize=(10, 5))
ax.imshow(wordcloud, interpolation='bilinear')
ax.axis("off")
plt.show()
```



In [8]:
```
# Calculate Within-Cluster Sum of Squares (WCSS)
evaluator = ClusteringEvaluator()
wcss = evaluator.evaluate(kmeans_predictions_4)
print("Within-Cluster Sum of Squares = " + str(wcss))
```

In [35]:
```
# restart query table if necessary
for q in spark.streams.active:
    if q.name == "word_my_query":
        q.stop()
```

Appendix C (i) Project Management



Appendix C (ii) Meeting Minute Note

The University of Nottingham — School of Computer Science — ima Intelligent Modelling & Analysis

Meeting Record

Supervisor Name: Peer-Olaf Seibers
Your Name: Kafayat Adeoye
Date of last meeting: 3/11/2022

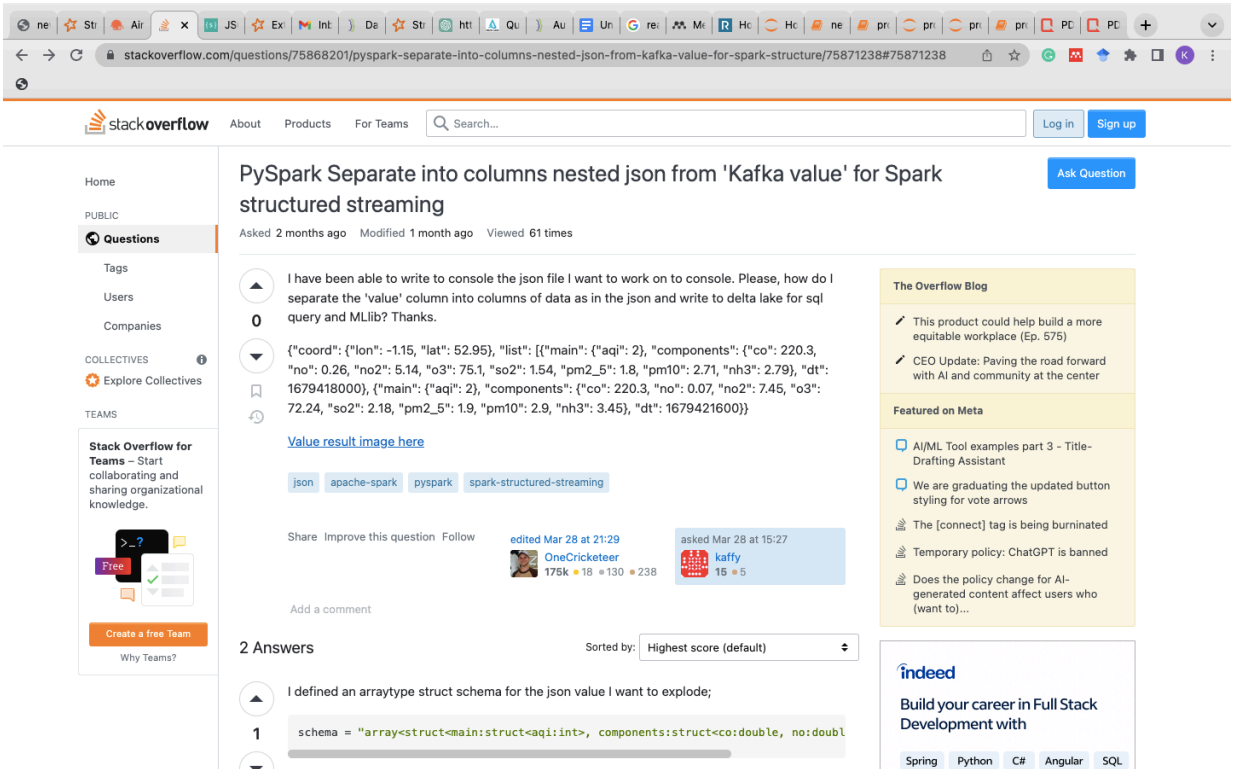
1. What was discussed in the last meeting?
   - We discussed the overall outcome for the project and suggested to experiment with the prototype on a single type of data and algorithm before further evaluation on other types
   - Suggestion on asking questions from researchers if needed

2. What was agreed in the last meeting (your tasks until the next meeting)?
   - I should continue my search for research papers for related work and background of the project topic
   - Create a mind map with the research papers for background, and related work

3. Agenda for current meeting
   - Discussion on the mind map if I come up with any or the papers I have found so far

Page 1 of 1 128 words English (United Kingdom) Focus 196%

Appendix D Stack Overflow



stackoverflow.com/questions/75868201/pyspark-separate-into-columns-nested-json-from-kafka-value-for-spark-structure/75871238#75871238

stackoverflow About Products For Teams Search... Log in Sign up

Home
PUBLIC
Questions
Tags
Users
Companies
COLLECTIVES
Explore Collectives
TEAMS
Stack Overflow for Teams – Start collaborating and sharing organizational knowledge.
Free
Create a free Team
Why Teams?

PySpark Separate into columns nested json from 'Kafka value' for Spark structured streaming

Ask Question

Asked 2 months ago Modified 1 month ago Viewed 61 times

I have been able to write to console the json file I want to work on to console. Please, how do I separate the 'value' column into columns of data as in the json and write to delta lake for sql query and MLlib? Thanks.

0

{"coord": {"lon": -1.15, "lat": 52.95}, "list": [{"main": {"aqi": 2}, "components": {"co": 220.3, "no": 0.26, "no2": 5.14, "o3": 75.1, "so2": 1.54, "pm2_5": 1.8, "pm10": 2.71, "nh3": 2.79}, "dt": 1679418000}, {"main": {"aqi": 2}, "components": {"co": 220.3, "no": 0.07, "no2": 7.45, "o3": 72.24, "so2": 2.18, "pm2_5": 1.9, "pm10": 2.9, "nh3": 3.45}, "dt": 1679421600}]}

Value result image here

json apache-spark pyspark spark-structured-streaming

Share Improve this question Follow

edited Mar 28 at 21:29 OneCricketeer 175k ● 18 ● 130 ● 238

asked Mar 28 at 15:27 kaffy 15 ● 5

Add a comment

2 Answers Sorted by: Highest score (default)

I defined an arraytype struct schema for the json value I want to explode;

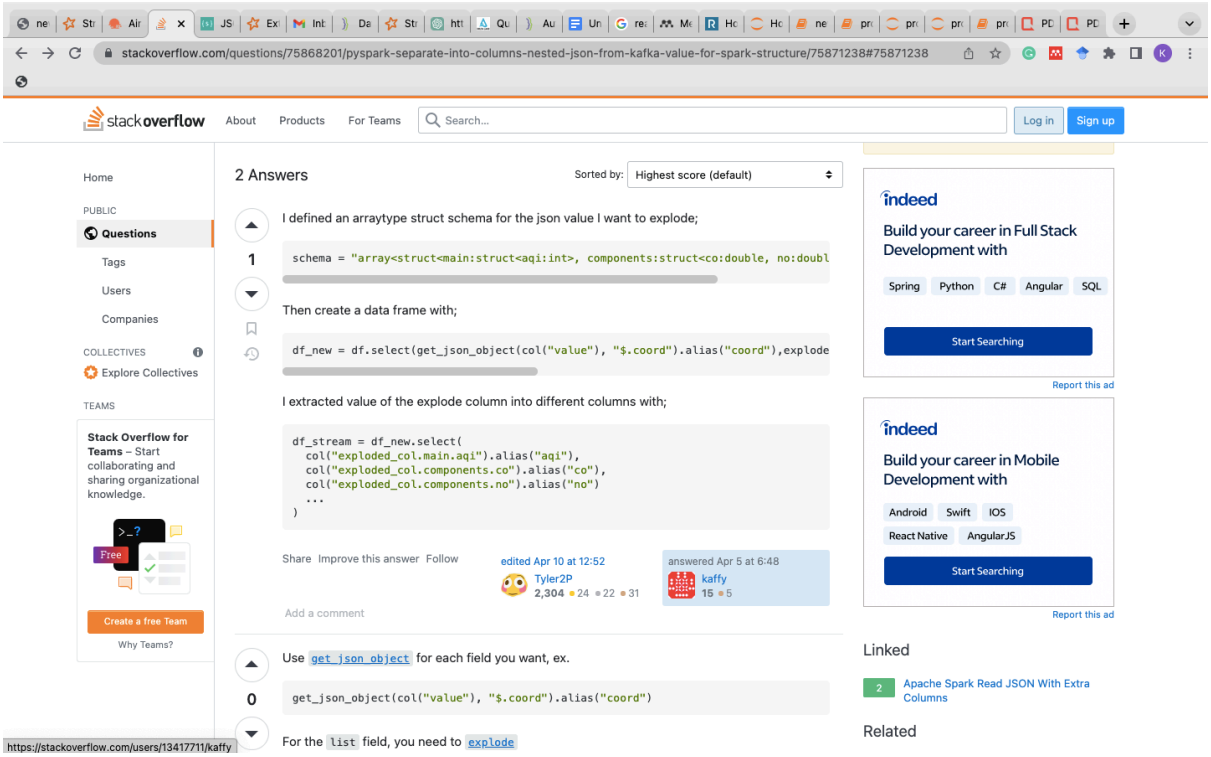
1

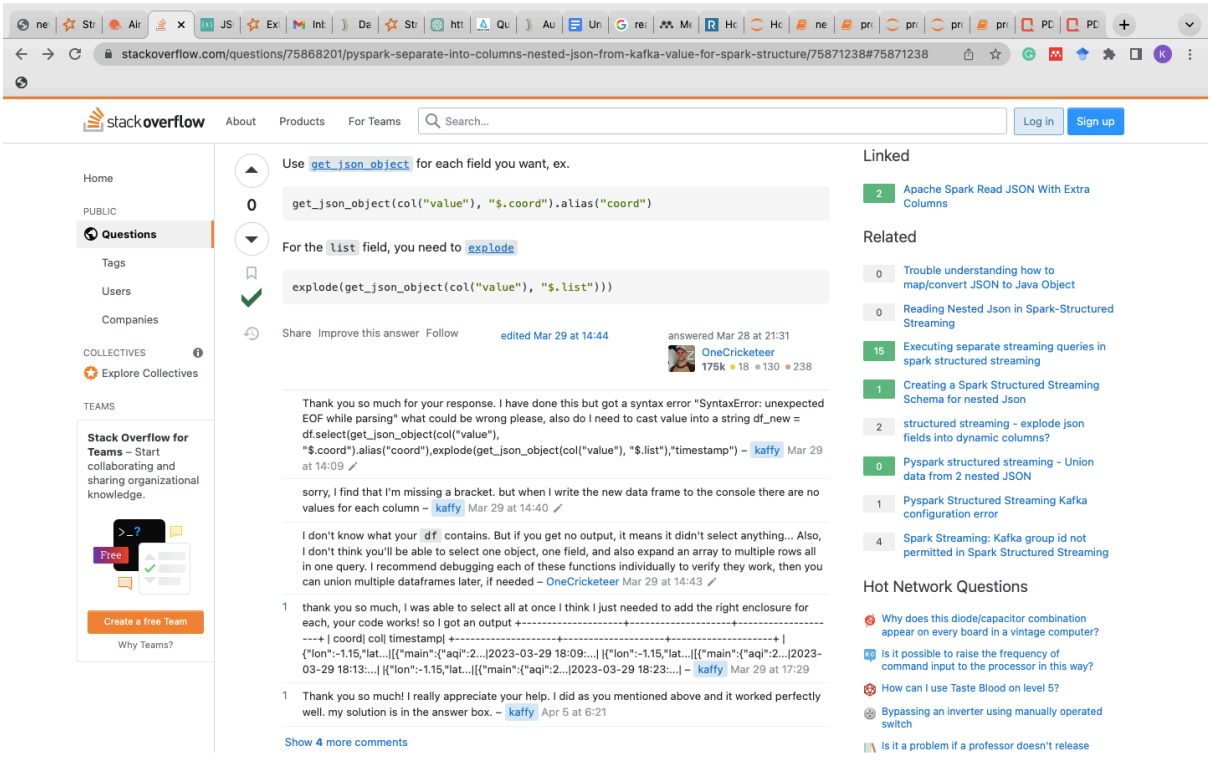
```
schema = "array<struct<main:struct<aqi:int>, components:struct<co:double, no:doubl
```

The Overflow Blog
- This product could help build a more equitable workplace (Ep. 575)
- CEO Update: Paving the road forward with AI and community at the center

Featured on Meta
- AI/ML Tool examples part 3 - Title-Drafting Assistant
- We are graduating the updated button styling for vote arrows
- The [connect] tag is being burninated
- Temporary policy: ChatGPT is banned
- Does the policy change for AI-generated content affect users who (want to)...

indeed
Build your career in Full Stack Development with
Spring Python C# Angular SQL


stackoverflow.com/questions/75868201/pyspark-separate-into-columns-nested-json-from-kafka-value-for-spark-structure/75871238#75871238
stackoverflow
About
Products
For Teams
Search...
Log in
Sign up
Home
PUBLIC
Questions
Tags
Users
Companies
COLLECTIVES
Explore Collectives
TEAMS
Stack Overflow for Teams – Start collaborating and sharing organizational knowledge.
Create a free Team
Why Teams?
2 Answers
Sorted by: Highest score (default)
1
I defined an arraytype struct schema for the json value I want to explode;
schema = "array<struct<main:struct<aqi:int>, components:struct<co:double, no:doubl
Then create a data frame with;
df_new = df.select(get_json_object(col("value"), "$.coord").alias("coord"),explode
I extracted value of the explode column into different columns with;
df_stream = df_new.select(
col("exploded_col.main.aqi").alias("aqi"),
col("exploded_col.components.co").alias("co"),
col("exploded_col.components.no").alias("no")
...
)
Share Improve this answer Follow
edited Apr 10 at 12:52
Tyler2P
2,304 • 24 • 22 • 31
answered Apr 5 at 6:48
kaffy
15 • 5
Add a comment
0
Use get_json_object for each field you want, ex.
get_json_object(col("value"), "$.coord").alias("coord")
For the list field, you need to explode
https://stackoverflow.com/users/13417711/kaffy
indeed
Build your career in Full Stack Development with
Spring
Python
C#
Angular
SQL
Start Searching
Report this ad
indeed
Build your career in Mobile Development with
Android
Swift
IOS
React Native
AngularJS
Start Searching
Report this ad
Linked
2 Apache Spark Read JSON With Extra Columns
Related


stackoverflow.com/questions/75868201/pyspark-separate-into-columns-nested-json-from-kafka-value-for-spark-structure/75871238#75871238
stackoverflow
About
Products
For Teams
Search...
Log in
Sign up
Home
PUBLIC
Questions
Tags
Users
Companies
COLLECTIVES
Explore Collectives
TEAMS
Stack Overflow for Teams – Start collaborating and sharing organizational knowledge.
Create a free Team
Why Teams?
Use get_json_object for each field you want, ex.
0
get_json_object(col("value"), "$.coord").alias("coord")
For the list field, you need to explode
explode(get_json_object(col("value"), "$.list")))
Share Improve this answer Follow
edited Mar 29 at 14:44
answered Mar 28 at 21:31
OneCricketeer
175k • 18 • 130 • 238
Thank you so much for your response. I have done this but got a syntax error "SyntaxError: unexpected EOF while parsing" what could be wrong please, also do I need to cast value into a string df_new = df.select(get_json_object(col("value"), "$.coord").alias("coord"),explode(get_json_object(col("value"), "$.list"),"timestamp") – kaffy Mar 29 at 14:09
sorry, I find that I'm missing a bracket. but when I write the new data frame to the console there are no values for each column – kaffy Mar 29 at 14:40
I don't know what your df contains. But if you get no output, it means it didn't select anything... Also, I don't think you'll be able to select one object, one field, and also expand an array to multiple rows all in one query. I recommend debugging each of these functions individually to verify they work, then you can union multiple dataframes later, if needed – OneCricketeer Mar 29 at 14:43
1 thank you so much, I was able to select all at once I think I just needed to add the right enclosure for each, your code works! so I got an output +--------------------+--------------------+--------------------+ | coord| col| timestamp| +--------------------+--------------------+--------------------+ |{"lon":-1.15,"lat...|[{"main":{"aqi":2...|2023-03-29 18:09:...| |{"lon":-1.15,"lat...|[{"main":{"aqi":2...|2023-03-29 18:13:...| |{"lon":-1.15,"lat...|[{"main":{"aqi":2...|2023-03-29 18:23:...| – kaffy Mar 29 at 17:29
1 Thank you so much! I really appreciate your help. I did as you mentioned above and it worked perfectly well. my solution is in the answer box. – kaffy Apr 5 at 6:21
Show 4 more comments
Linked
2 Apache Spark Read JSON With Extra Columns
Related
0 Trouble understanding how to map/convert JSON to Java Object
0 Reading Nested Json in Spark-Structured Streaming
15 Executing separate streaming queries in spark structured streaming
1 Creating a Spark Structured Streaming Schema for nested Json
2 structured streaming - explode json fields into dynamic columns?
0 Pyspark structured streaming - Union data from 2 nested JSON
1 Pyspark Structured Streaming Kafka configuration error
4 Spark Streaming: Kafka group id not permitted in Spark Structured Streaming
Hot Network Questions
Why does this diode/capacitor combination appear on every board in a vintage computer?
Is it possible to raise the frequency of command input to the processor in this way?
How can I use Taste Blood on level 5?
Bypassing an inverter using manually operated switch
Is it a problem if a professor doesn't release

8. Bibliography

Armbrust, M., Das, T., Torres, J., Yavuz, B., Zhu, S., Xin, R., Ghodsi, A., Stoica, I., & Zaharia, M. (2018). Structured streaming: A declarative api for real-time applications in apache spark. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 601–613. https://doi.org/10.1145/3183713.3190664

Assefi, M., Behravesh, E., Liu, G., & Tafti, A. P. (2017). Big data machine learning using apache spark MLlib. *Proceedings - 2017 IEEE International Conference on Big Data, Big Data 2017*, *2018-January*, 3492–3498. https://doi.org/10.1109/BIGDATA.2017.8258338

Bahga Vijay Madisetti, A. (2019). *Big Data Analytics: A Hands-On Approach*. www.hands-on-books-series.com

Chambers, B. (William A., & Zaharia, M. (2018). *Spark : the definitive guide : big data processing made simple*.

Kakarla, R., Krishnan, S., & Alla, S. (2021). Applied Data Science Using PySpark. In *Applied Data Science Using PySpark*. Apress. https://doi.org/10.1007/978-1-4842-6500-0

Kienzler, R., Rezaul, M., Sridhar, K., Siamak, A., Meenakshi, A., Broderick, R., & Mei, H. S. (2018). *Apache Spark 2: Data Processing and Real-Time Analytics Master complex big data processing, stream analytics, and machine learning with Apache Spark*.

Maas, G., & Garillot, F. (2019). *Stream Processing with Apache Spark Best Practices for Scaling and Optimizing Apache Spark*.

Nishant Garg. (2013). *Apache Kafka*.

Nti, I. K., Quarcoo, J. A., Aning, J., & Fosu, G. K. (2022). A mini-review of machine learning in big data analytics: Applications, challenges, and prospects. *Big Data Mining and Analytics*, *5*(2), 81–97. https://doi.org/10.26599/BDMA.2021.9020028

Salloum, S., Dautov, R., Chen, X., Peng, P. X., & Huang, J. Z. (2016). Big data analytics on Apache Spark. *International Journal of Data Science and Analytics*, *1*(3–4), 145–164. https://doi.org/10.1007/S41060-016-0027-9

Scott, D., Gamov, V., Klein Foreword, D., & Rao, J. (2022). *Kafka in Action*.

Shalev-Shwartz, S., & Ben-David, S. (2014). *Understanding Machine Learning*.

Structured Streaming guide. (n.d.). *Structured Streaming Programming Guide - Spark 3.3.1 Documentation*. Retrieved February 2, 2023, from https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#window-operations-on-event-time

Ait Hammou, B., Ait Lahcen, A., & Mouline, S. (2020). Towards a real-time processing framework based on improved distributed recurrent neural network variants with fastText for social big data analytics. *Information Processing & Management*, *57*(1), 102122. https://doi.org/10.1016/J.IPM.2019.102122

Ali, A. R. (2018). Real-time big data warehousing and analysis framework. *2018 IEEE 3rd International Conference on Big Data Analysis, ICBDA 2018*, 43–49. https://doi.org/10.1109/ICBDA.2018.8367649

Cho, J., Chang, H., Mukherjee, S., Lakshman, T. V, & Van Der Merwe, J. (2017). *Typhoon: An SDN Enhanced Real-Time Big Data Streaming Framework*. *17*. https://doi.org/10.1145/3143361.3143398

Dean, J., & Ghemawat, S. (2008). MapReduce. *Communications of the ACM*, *51*(1), 107–113. https://doi.org/10.1145/1327452.1327492

Demirbaga, U., Wen, Z., Noor, A., Mitra, K., Alwasel, K., Garg, S., Zomaya, A. Y., & Ranjan, R. (2022). AutoDiagn: An Automated Real-Time Diagnosis Framework for Big Data Systems.

IEEE Transactions on Computers, *71*(5), 1035–1048. https://doi.org/10.1109/TC.2021.3070639

Dinesh Jackson, S. R., Fenil, E., Gunasekaran, M., Vivekananda, G. N., Thanjaivadivel, T., Jeeva, S., & Ahilan, A. (2019). Real time violence detection framework for football stadium comprising of big data analysis and deep learning through bidirectional LSTM. *Computer Networks*, *151*, 191–200. https://doi.org/10.1016/J.COMNET.2019.01.028

Drabas, T., Lee, D., & Karau, H. (2016). *Learning PySpark : build data-intensive applications locally and deploy at scale using the combined powers of Python and Spark 2.0*.

Du, G., Long, C., Yu, J., Wan, W., Zhao, J., & Wei, J. (2019). *A Real-time Big Data Framework for Network Security Situation Monitoring*. https://doi.org/10.5220/0007708301670175

Eugster, P. T., Felber, P. A., & Kermarrec, A.-M. (2003). *The Many Faces of Publish/Subscribe*.

Gao, J., & Chiao, S. (2020). *Air Quality Prediction: Big Data and Machine Learning Approaches*. https://doi.org/10.18178/ijesd.2018.9.1.1066

Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow SECOND EDITION Concepts, Tools, and Techniques to Build Intelligent Systems*. http://oreilly.com

Hapke, H., & Nelson, C. (2020). *Building Machine Learning Pipelines Automating Model Life Cycles with TensorFlow*.

Hassaan, M., & Elghandour, I. (2016). A real-time big data analysis framework on a CPU/GPU heterogeneous cluster: A meteorological application case study. *Proceedings - 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, BDCAT 2016*, 168–177. https://doi.org/10.1145/3006299.3006304

Kakarla, R., Krishnan, S., & Alla, S. (2021). Applied Data Science Using PySpark. In *Applied Data Science Using PySpark*. Apress. https://doi.org/10.1007/978-1-4842-6500-0

Kulkarni, A., & Shivananda, A. (2019). Natural Language Processing Recipes. In *Natural Language Processing Recipes*. Apress. https://doi.org/10.1007/978-1-4842-4267-4

Li, J., Cheng, K., Wang, S., Morstatter, F., Trevino, R. P., Tang, J., & Liu, H. (2017). Feature Selection: A Data Perspective. *ACM Comput. Surv*, *50*. https://doi.org/10.1145/3136625

Rybarczyk, Y., & Zalakeviciute, R. (2018). Machine Learning Approaches for Outdoor Air Quality Modelling: A Systematic Review. *Applied Sciences 2018, Vol. 8, Page 2570*, *8*(12), 2570. https://doi.org/10.3390/APP8122570

Sahar Hussain Jambi, by. (2016). *Engineering Scalable Distributed Services for Real-Time Big Data Analytics*.

Shalev-Shwartz, S., & Ben-David, S. (2014). *Understanding Machine Learning*.

Table streaming reads and writes — Delta Lake Documentation. (n.d.). Retrieved November 22, 2022, from https://docs.delta.io/latest/delta-streaming.html

Zheng, Z., Wang, P., Liu, J., & Sun, S. (2015). Real-Time Big Data Processing Framework: Challenges and Solutions. *Appl. Math. Inf. Sci*, *9*(6), 3169–3190. https://doi.org/10.12785/amis/090646

Demirbaga, U., Wen, Z., Noor, A., Mitra, K., Alwasel, K., Garg, S., Zomaya, A. Y., & Ranjan, R. (2022). AutoDiagn: An Automated Real-Time Diagnosis Framework for Big Data Systems. IEEE Transactions on Computers, 71(5), 1035–1048. https://doi.org/10.1109/TC.2021.3070639

Hirschberg, J., & Manning, C. D. (2015). Advances in natural language processing. https://www.science.org

Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. Science,349(6245),255–260. https://doi.org/10.1126/SCIENCE.AAA8415/ASSET/AB2EF18A-576D-464D-B1B6-1301159EE29A/ASSETS/GRAPHIC/349_255_F5.JPEG