

CS777 Term Paper - Kaiwen Zhu, Maryam Baizhigitova

Apache Kafka: An In-depth Analysis of its Management, Performance, Design, Experimentation, and an impact on Cloud Computing.

CS777: Term Paper
Maryam Baizhigitova, Kaiwen Zhu
Boston University

Table of Contents

Introduction.....	2
Historical Background.....	2
Use cases of Apache Kafka.....	3
Apache Kafka Architecture.....	4
Approaches.....	5
Demo Objective.....	5
Demo Flow.....	9
Demo Architecture.....	9
Apache Kafka Environment Setup.....	10
Instructions for Executing the Demo Code.....	12
Demo Code Results.....	12
Dataset Explanation.....	12
Detailed Explanation of the Results.....	12

Introduction

What is the term paper all about?

The term paper explores the development and implementation of a real-time data streaming and processing pipeline using Apache Kafka, paired with a machine learning model for predictive analysis. Specifically, the pipeline is tested with the Iris dataset, a classic in the field of machine learning, to classify iris species based on physical measurements. The paper details the architecture of the pipeline, which includes data production, consumption, processing, and the integration of a feedback loop for enhancing model predictions. The term paper discusses the theoretical aspects of data streaming and real-time analytics, the technical details of Kafka and the KNN algorithm, and the practical implications of such a system in real-world scenarios. It would also be important to discuss the benefits of using Kafka for such applications, like its high performance, reliability, scalability, and fault tolerance.

Why is the topic important?

The topic is significant due to the increasing need for real-time data insights in various industries, such as finance, healthcare, and e-commerce. The ability to process and analyze data in real-time allows organizations to make more informed decisions quickly, which can lead to competitive advantages. Additionally, the integration of a feedback mechanism showcases how predictive models can be continuously improved, a concept that is critical in the era of adaptive and intelligent systems. The project also demonstrates the practical application of Kafka as a scalable solution for handling high-throughput data streams, making it relevant for businesses dealing with large volumes of data.

Historical Background

In the *"Kafka: A Distributed Messaging System for Log Processing"* paper, Jay Kreps and his colleagues introduced Apache Kafka as a solution to the challenges faced by LinkedIn in managing real-time data streams. Published in 2011, this paper outlined the fundamental principles and design philosophy behind Kafka, emphasizing its role as a distributed messaging system tailored for log processing. The authors highlighted the need for a scalable, fault-tolerant platform capable of handling massive volumes of data generated by user interactions on social networking platforms like LinkedIn.

This paper marked the inception of Kafka as a pivotal technology in the realm of distributed systems, setting the stage for its evolution into a versatile platform supporting diverse use cases beyond its initial purpose in log processing. Kreps' insights into the challenges faced by LinkedIn underscored the pressing need for a robust solution to handle the escalating demands of real-time data streams, a need that Kafka effectively addressed and continues to address today.

Raptis and Passarella's *"A survey on networked data streaming with Apache Kafka"*, published in IEEE Access in 2023, provides valuable insights into the evolution and adoption of Kafka in the years following its inception. The survey delves into various aspects of Kafka's architecture, capabilities, and applications, offering a comprehensive overview of its role in modern data streaming scenarios. By examining the experiences and challenges faced by organizations deploying Kafka, Raptis and Passarella shed light on the practical implications of this technology and its impact on the field of real-time data processing.

Through an exploration of real-world experiences and challenges encountered by organizations deploying Kafka, Raptis and Passarella provide invaluable insights into the practical implications of this

technology. Their work underscores Kafka's pivotal role in shaping the landscape of real-time data processing, offering a comprehensive understanding of its significance in driving innovation and efficiency across industries.

In a more recent article titled "*What is Apache Kafka?*" published in April 2020, G. Tangelmayer and colleagues offer a contemporary perspective on Kafka's features and functionalities. This article serves as a practical guide for understanding Kafka's core concepts, deployment scenarios, and integration with other technologies. By elucidating Kafka's role in enabling real-time data streaming and event-driven architectures, Tangelmayer et al. underscore its relevance in addressing the growing demands of modern data-driven businesses.

Tangelmayer et al. elucidate Kafka's pivotal role in facilitating real-time data streaming and enabling event-driven architectures, highlighting its importance in meeting the evolving needs of data-driven businesses. Their insights underscore Kafka's enduring relevance as a foundational technology for organizations seeking to harness the power of real-time data for strategic decision-making and operational efficiency.

Use Cases of Apache Kafka

Messaging

Kafka works well as a replacement for a more traditional message broker. Message brokers are used for a variety of reasons (to decouple processing from data producers, to buffer unprocessed messages, etc). In comparison to most messaging systems Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large scale message processing applications. Messaging uses are often comparatively low-throughput, but may require low end-to-end latency and often depend on the strong durability guarantees Kafka provides. In this domain Kafka is comparable to traditional messaging systems such as ActiveMQ or RabbitMQ.

Website Activity Tracking

The original use case for Kafka was to be able to rebuild a user activity tracking pipeline as a set of real-time publish-subscribe feeds. This means site activity (page views, searches, or other actions users may take) is published to central topics with one topic per activity type. These feeds are available for subscription for a range of use cases including real-time processing, real-time monitoring, and loading into Hadoop or offline data warehousing systems for offline processing and reporting.

Metrics

Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.

Stream Processing

Many users of Kafka process data in processing pipelines consisting of multiple stages, where raw input data is consumed from Kafka topics and then aggregated, enriched, or otherwise transformed into new topics for further consumption or follow-up processing. For example, a processing pipeline for recommending news articles might crawl article content from RSS feeds and publish it to an "articles" topic; further processing might normalize or deduplicate this content and publish the cleansed article content to a new topic; a final processing stage might attempt to recommend this content to users. Such processing pipelines create graphs of real-time data flows based on the individual topics. Starting in 0.10.0.0, a light-weight but powerful stream processing library called Kafka Streams is available in

Apache Kafka to perform such data processing as described above. Apart from Kafka Streams, alternative open source stream processing tools include Apache Storm and Apache Samza.

Event Sourcing

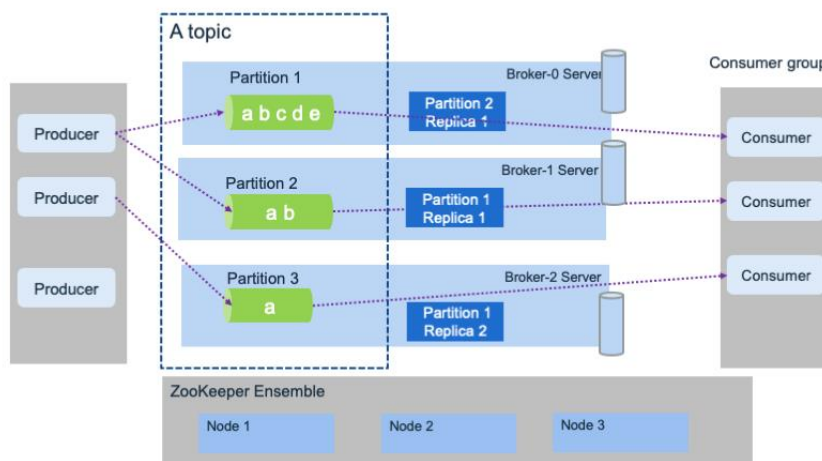
Event sourcing is a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.

Commit Log

Kafka can serve as a kind of external commit-log for a distributed system. The log helps replicate data between nodes and acts as a re-syncing mechanism for failed nodes to restore their data. The log compaction feature in Kafka helps support this usage. In this usage Kafka is similar to Apache BookKeeper project.

Apache Kafka Architecture

Kafka Architecture



Kafka is a distributed streaming platform that is used to build real-time data pipelines and streaming apps. It's scalable, fault-tolerant, and it allows you to publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.

Key components and design elements depicted in the diagram:

- **Producers:** These are the processes that publish data (messages) to Kafka topics. Each message consists of a key, a value, and a timestamp.
- **Topic:** A topic is a category or feed name to which messages are published. In Kafka, topics are multi-subscriber, and they can have zero or more consumers that subscribe to the data written to it.
- **Partitions:** Topics are split into partitions, which allow for the data to be parallelized. This means that each partition can be hosted on a different server, allowing for multiple consumers to read from a topic in parallel.
- **Brokers:** Kafka clusters consist of multiple brokers. Each broker is a Kafka server that stores data and serves clients. In the diagram, each partition has replicas across different brokers, which ensures high availability and fault tolerance.

- **Replicas:** Replicas are copies of partitions. They are used for redundancy. The diagram shows that each partition has multiple replicas, with each replica being on a different broker.
- **Consumer Group:** A consumer group includes the processes that subscribe to a topic and process the data. Within a consumer group, each consumer reads from exclusive partitions of the topic, so that each message is only read by a single consumer in the group.

Apache Kafka Approaches

Apache Kafka is a robust platform for handling real-time data streams, and it comes equipped with various techniques to ensure efficient data processing, fault tolerance, and high availability. At its core, Kafka utilizes message serialization and log compaction to optimize data storage and retrieval, enabling stateful applications to maintain comprehensive data histories. Producers serialize data into formats such as JSON, Avro, or Protobuf before transmission, and consumers deserialize the data upon reception. Log compaction helps in preserving the latest state of data, which is critical for applications requiring a full dataset for processing.

Kafka's architecture is designed for scalability and fault tolerance, achieved through features like partitioning and replication. Partitions allow Kafka to spread data across multiple nodes in the cluster, enabling parallel processing and enhanced performance. Replication ensures that data is copied across different brokers, safeguarding against data loss in the event of a node failure. Kafka also offers exactly once semantics (EOS) to prevent message duplication, a vital feature for many business applications where data accuracy is paramount.

In addition to data integrity, Kafka provides mechanisms for efficient resource management and secure data handling. Through consumer offset management and backpressure handling, Kafka ensures that consumers process messages at their own pace without overwhelming the system. Security measures like SSL/TLS encryption, SASL authentication, and ACLs for authorization help maintain the confidentiality and integrity of the data. Finally, Kafka's monitoring tools and the Kafka Connect framework facilitate the seamless integration of Kafka with a multitude of other systems, making it an indispensable tool for modern data-driven applications.

Main Objective for Both Demos

The main objective of the project is to architect, develop, and iteratively improve a real-time analytical pipeline using Apache Kafka, integrating machine learning to classify data streams effectively.

Building upon this, Demo 1, labeled as the "Enhanced Machine Learning Pipeline," aims to optimize and extend the capabilities of the initial system. It prioritizes advancements in data processing efficiency, predictive accuracy, system scalability, and robustness of the feedback mechanism. This enhanced demo is designed to demonstrate significant improvements over the baseline in handling larger volumes of data with greater complexity and providing a more sophisticated approach to real-time data-driven decision-making.

The Demo 2, focuses on establishing a baseline system that sets up the fundamental Kafka infrastructure, integrates a simple machine learning model, and implements a basic feedback loop for predictions. This serves as a proof of concept and a foundation for refinement.

Collectively, both demos are intended to illustrate the progression from a working prototype to an optimized solution, showcasing the application of Kafka in streaming analytics and the integration of machine learning for actionable insights in a live environment.

Demo 1

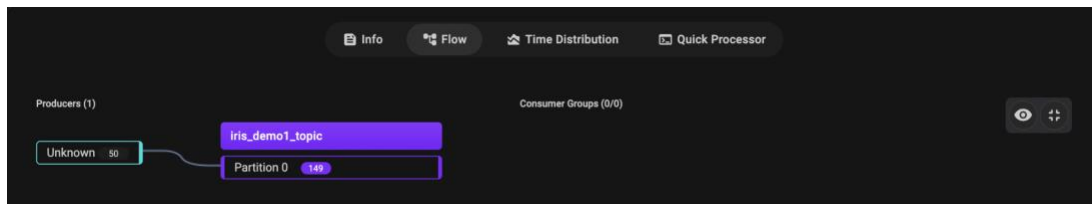
Demo 1 Objective

Demo 1 Enhanced Machine Learning Pipeline with Real-Time Kafka Integration showcases an advanced implementation of a machine learning pipeline that utilizes Apache Kafka for robust real-time data processing. The objective is to develop a real-time machine learning framework that processes streaming data from Kafka, makes predictions with a KNN model on the Iris dataset, dynamically retrains the model with new or corrected data, and incorporates a feedback loop to improve model accuracy over time.

Demo 1 Flow

The screenshots are of the Kafka management interface in Kadeck, showing information about Kafka producers, topics, and the messages within a topic.

There are two topics in demo 1. The first one is `iris_demo1_topic` (shown below), which simulates the process of continuously publishing data from the iris dataset, which is stored in brokers and then consumed for knn model prediction.



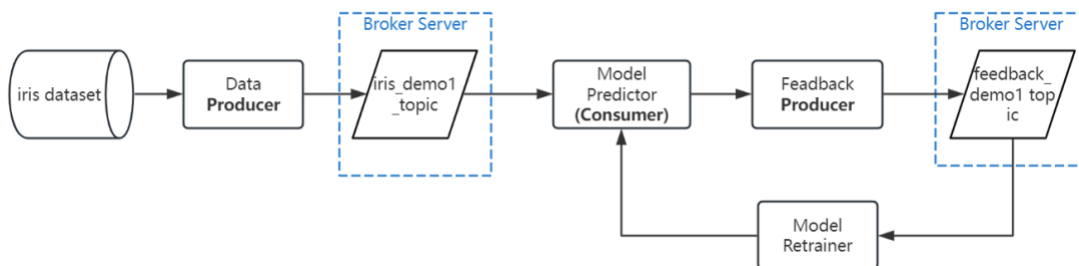
Timestamp	Part	Off	Key	Value	Id	SepalLeng...	SepalWidt...	PetalLeng...	PetalWidt...	Species
2024-03-20 23:33:10:554	0	0	null	{ "id": 1, "S...	1	5.1	3.5	1.4	0.2	Iris-setosa
2024-03-20 23:33:10:664	0	1	null	{ "id": 2, "S...	2	4.9	3	1.4	0.2	Iris-setosa
2024-03-20 23:33:10:773	0	2	null	{ "id": 3, "S...	3	4.7	3.2	1.3	0.2	Iris-setosa
2024-03-20 23:33:10:880	0	3	null	{ "id": 4, "S...	4	4.6	3.1	1.5	0.2	Iris-setosa
2024-03-20 23:33:10:986	0	4	null	{ "id": 5, "S...	5	5	3.6	1.4	0.2	Iris-setosa
2024-03-20 23:33:11:092	0	5	null	{ "id": 6, "S...	6	5.4	3.9	1.7	0.4	Iris-setosa
2024-03-20 23:33:11:198	0	6	null	{ "id": 7, "S...	7	4.6	3.4	1.4	0.3	Iris-setosa
2024-03-20 23:33:11:300	0	7	null	{ "id": 8, "S...	8	5	3.4	1.5	0.2	Iris-setosa
2024-03-20 23:33:11:406	0	8	null	{ "id": 9, "S...	9	4.4	2.9	1.4	0.2	Iris-setosa
2024-03-20 23:33:11:511	0	9	null	{ "id": 10, "S...	10	4.9	3.1	1.5	0.1	Iris-setosa
2024-03-20 23:33:11:620	0	10	null	{ "id": 11, "S...	11	5.4	3.7	1.5	0.2	Iris-setosa
2024-03-20 23:33:11:726	0	11	null	{ "id": 12, "S...	12	4.8	3.4	1.6	0.2	Iris-setosa
2024-03-20 23:33:11:832	0	12	null	{ "id": 13, "S...	13	4.8	3	1.4	0.1	Iris-setosa
2024-03-20 23:33:11:939	0	13	null	{ "id": 14, "S...	14	4.3	3	1.1	0.1	Iris-setosa
2024-03-20 23:33:12:045	0	14	null	{ "id": 15, "S...	15	5.8	4	1.2	0.2	Iris-setosa
2024-03-20 23:33:12:151	0	15	null	{ "id": 16, "S...	16	5.7	4.4	1.5	0.4	Iris-setosa
2024-03-20 23:33:12:260	0	16	null	{ "id": 17, "S...	17	5.4	3.9	1.3	0.4	Iris-setosa
2024-03-20 23:33:12:362	0	17	null	{ "id": 18, "S...	18	5.1	3.5	1.4	0.3	Iris-setosa
2024-03-20 23:33:12:466	0	18	null	{ "id": 19, "S...	19	5.7	3.8	1.7	0.3	Iris-setosa
2024-03-20 23:33:12:577	0	19	null	{ "id": 20, "S...	20	5.1	3.8	1.5	0.3	Iris-setosa

The second topic is `feedback_demo1`, which is the prediction feedback for potential model adjustments. It contains messages from the automatically generated feedback by model predictor and the feedback from the user, as detailed in the table shown in the subsequent screenshot. This setup facilitates its use in both automated and manual processes, aimed at validating predictions and generating feedback, thereby enabling continuous improvement of the model's performance.

Timestamp	Part	Off	Key	Value
2024-03-20 23:35:57:118	0	0	null	{ "id": 71, "SepalLengthCm": 5.9, "SepalWidthCm": 3.2, "Pet...
2024-03-20 23:35:57:123	0	1	null	{ "id": 73, "SepalLengthCm": 6.3, "SepalWidthCm": 2.5, "Pet...
2024-03-20 23:35:57:132	0	2	null	{ "id": 84, "SepalLengthCm": 6, "SepalWidthCm": 2.7, "Petal...
2024-03-20 23:35:57:145	0	3	null	{ "id": 107, "SepalLengthCm": 4.9, "SepalWidthCm": 2.5, "P...
2024-03-20 23:47:31:625	0	4	null	{ "record_id": "123", "correct_label": "Iris-versicolor" }

Showing 5 of 5 records scanned.

Demo 1 Architecture



Data Producer: The data producer reads the Iris dataset and streams it to the `iris_stream` topic. This part serves to simulate real-time data streaming by setting the hardcoded sleep time.

Model Predictor: A Kafka consumer that uses the streamed data for real-time predictions with a pre-trained machine learning model and sends prediction feedback for potential model adjustments. It uses the Iris dataset to train a K-Nearest Neighbors (KNN) classifier, consumes new data from a Kafka topic, makes predictions, and sends feedback on predictions through another Kafka topic.

Feedback Consumer: This part initializes a Kafka consumer for consuming messages from a Kafka topic. It uses the `KafkaConsumer` class from the `kafka-python` library to subscribe to a topic named `'feedback_demo1'` and print out each message it consumes.

Feedback Producer: In a real-world application, this could be part of an application interface where users confirm or correct predictions, thereby generating feedback. Combined with the automatically

generated feedback above, this can be used either as an automated system or as a manual process to validate predictions and produce feedback.

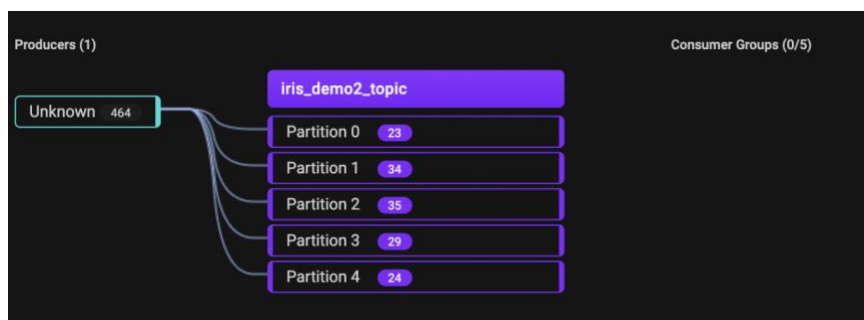
Model Retrainer: Triggered based on a schedule or specific messages, to retrain the model with all available data, including new or corrected labels. The goal is to improve model accuracy by incorporating corrected labels or new data.

Demo 2

Demo 2 Objective

The objective of Demo 2 is to design and implement a foundational real-time data processing and machine learning pipeline using Apache Kafka. This demo aims to establish a baseline architecture capable of streaming data ingestion, basic processing, and the application of a preliminary machine learning model for the classification of the Iris dataset species. The focus is on demonstrating the integration of Kafka with machine learning workflows to achieve real-time analytics. Additionally, Demo 2 seeks to implement an initial feedback mechanism for model predictions, laying the groundwork for iterative improvements. The ultimate goal is to validate the core concept of using Kafka for real-time data streaming and processing in conjunction with machine learning for predictive analytics, providing a solid foundation for subsequent enhancements and optimizations in Demo 1.

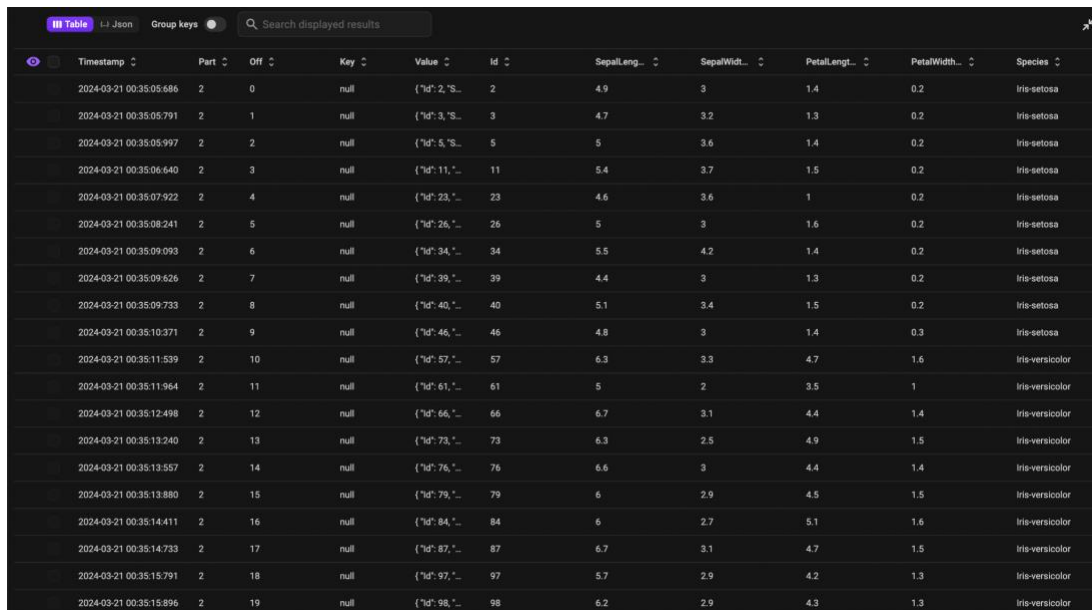
Demo 2 Flow



The screenshot appears to show a Kafka management interface Kadeck, displaying information about Kafka producers, topics, partitions, and the messages within a topic.

At the top, there's a section titled "Producers (1)" with a producer named "Unknown" linked to the topic `iris_demo2_topic`, which is further subdivided into five partitions (Partition 0 to Partition 4). This indicates that data is being produced and sent to the `iris_demo2_topic` topic, and the topic's data is distributed across these five partitions. Below, a table displays a stream of messages that have been produced to the `iris_demo2_topic`. The table includes columns for timestamp, partition, offset, key, value, and ID, along with the data content, which appears to include the features `SepalLengthCm`, `SepalWidthCm`, `PetalLengthCm`, `PetalWidthCm`, and a label `Species`. This resembles

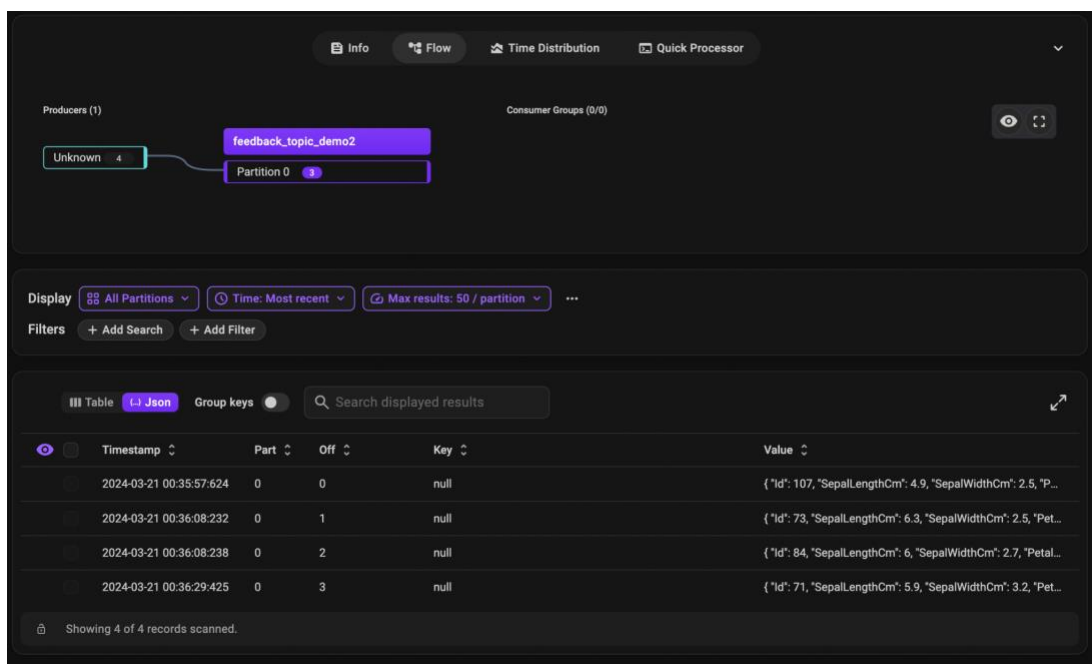
the structure of the Iris dataset, where each message corresponds to a set of measurements for an iris flower and its corresponding species classification.



Timestamp	Part	Off	Key	Value	Id	SepalLength	SepalWidth	PetalLength	PetalWidth	Species
2024-03-21 00:35:05.686	2	0	null	{"Id": 2, "S...	2	4.9	3	1.4	0.2	Iris-setosa
2024-03-21 00:35:05.791	2	1	null	{"Id": 3, "S...	3	4.7	3.2	1.3	0.2	Iris-setosa
2024-03-21 00:35:05.997	2	2	null	{"Id": 5, "S...	5	5	3.6	1.4	0.2	Iris-setosa
2024-03-21 00:35:06.640	2	3	null	{"Id": 11, "S...	11	5.4	3.7	1.5	0.2	Iris-setosa
2024-03-21 00:35:07.922	2	4	null	{"Id": 23, "S...	23	4.6	3.6	1	0.2	Iris-setosa
2024-03-21 00:35:08.241	2	5	null	{"Id": 26, "S...	26	5	3	1.6	0.2	Iris-setosa
2024-03-21 00:35:09.093	2	6	null	{"Id": 34, "S...	34	5.5	4.2	1.4	0.2	Iris-setosa
2024-03-21 00:35:09.626	2	7	null	{"Id": 39, "S...	39	4.4	3	1.3	0.2	Iris-setosa
2024-03-21 00:35:09.733	2	8	null	{"Id": 40, "S...	40	5.1	3.4	1.5	0.2	Iris-setosa
2024-03-21 00:35:10.371	2	9	null	{"Id": 46, "S...	46	4.8	3	1.4	0.3	Iris-setosa
2024-03-21 00:35:11.539	2	10	null	{"Id": 57, "S...	57	6.3	3.3	4.7	1.6	Iris-versicolor
2024-03-21 00:35:11.964	2	11	null	{"Id": 61, "S...	61	5	2	3.5	1	Iris-versicolor
2024-03-21 00:35:12.498	2	12	null	{"Id": 66, "S...	66	6.7	3.1	4.4	1.4	Iris-versicolor
2024-03-21 00:35:13.240	2	13	null	{"Id": 73, "S...	73	6.3	2.5	4.9	1.5	Iris-versicolor
2024-03-21 00:35:13.557	2	14	null	{"Id": 76, "S...	76	6.6	3	4.4	1.4	Iris-versicolor
2024-03-21 00:35:13.880	2	15	null	{"Id": 79, "S...	79	6	2.9	4.5	1.5	Iris-versicolor
2024-03-21 00:35:14.411	2	16	null	{"Id": 84, "S...	84	6	2.7	5.1	1.6	Iris-versicolor
2024-03-21 00:35:14.733	2	17	null	{"Id": 87, "S...	87	6.7	3.1	4.7	1.5	Iris-versicolor
2024-03-21 00:35:15.791	2	18	null	{"Id": 97, "S...	97	5.7	2.9	4.2	1.3	Iris-versicolor
2024-03-21 00:35:15.896	2	19	null	{"Id": 98, "S...	98	6.2	2.9	4.3	1.3	Iris-versicolor

There are no consumer groups currently active, as indicated by "Consumer Groups (0/5)", meaning that while data is being produced and stored in Kafka, there are no consumers actively processing this data at the moment.

This flow demonstrates a Kafka setup that is actively handling the production of messages. The interface provides visibility into the system's real-time operation, and it's a helpful tool for managing and monitoring Kafka's data flow.



The interface shows a Kafka flow diagram with one producer group 'Unknown' (4) connected to a topic 'feedback_topic_demo2'. The topic has one partition 'Partition 0' (1). Below the flow diagram, there are filters and a table of messages.

Display: All Partitions, Time: Most recent, Max results: 50 / partition

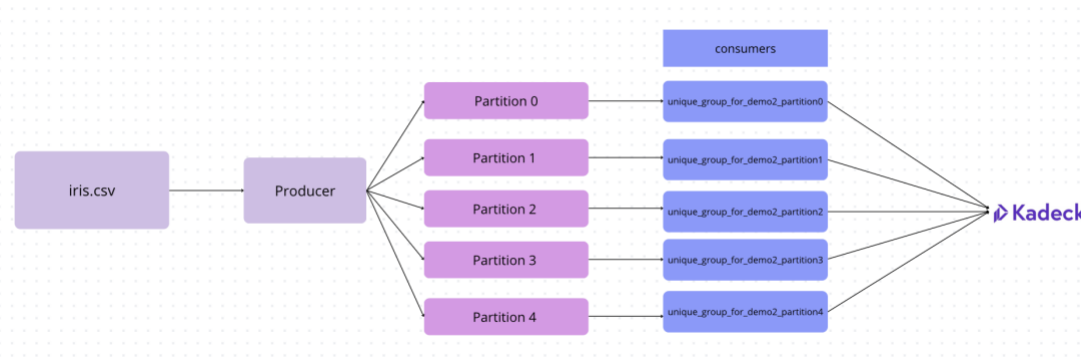
Filters: + Add Search, + Add Filter

Timestamp	Part	Off	Key	Value
2024-03-21 00:35:57.624	0	0	null	{"Id": 107, "SepalLengthCm": 4.9, "SepalWidthCm": 2.5, "P...
2024-03-21 00:36:08.232	0	1	null	{"Id": 73, "SepalLengthCm": 6.3, "SepalWidthCm": 2.5, "Pet...
2024-03-21 00:36:08.238	0	2	null	{"Id": 84, "SepalLengthCm": 6, "SepalWidthCm": 2.7, "Petal...
2024-03-21 00:36:29.425	0	3	null	{"Id": 71, "SepalLengthCm": 5.9, "SepalWidthCm": 3.2, "Pet...

Showing 4 of 4 records scanned.

The image above depicts a Kafka management interface showing the flow of messages to a Kafka topic designated for feedback, that includes wrong predictions based on the knn model.

Demo 2 Architecture



- **Producer:** In this case, the producer is sending data from a file named `iris.csv`, which likely contains the famous Iris flower dataset.
- **Partitions (0-4):** In Kafka, a topic is split into partitions to allow for data to be distributed and scaled out across multiple nodes for fault tolerance and increased throughput.
- **Consumers:** Each partition has a corresponding consumer identified by a unique consumer group ID (e.g., `unique_group_for_demo2_partition0`). This suggests each consumer is intended to exclusively read from its respective partition.
- **Kadeck:** Kadeck is a tool that is managing or monitoring the Kafka data streams and consumer processes.

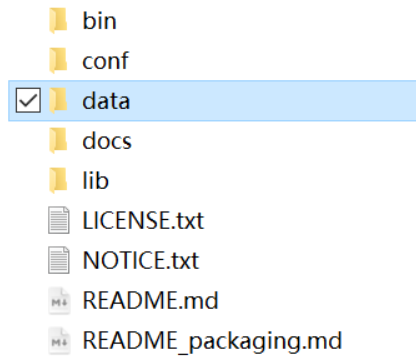
Apache Kafka Environment Setup

Environment Setup for Windows:

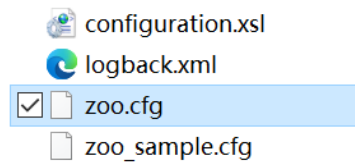
System: windows 10

Installation

1. Install JDK (Java Development Kit)
Java version: 1.8.0
2. Download Apache ZooKeeper
 - version: 3.8.4
 - <https://zookeeper.apache.org/releases.html>
3. Configure the zookeeper
 - a. Decompress the Zookeeper installation package
 - b. Create a data file



- c. Copy the zoo_sample.cfg file and rename it as zoo.cfg in config file

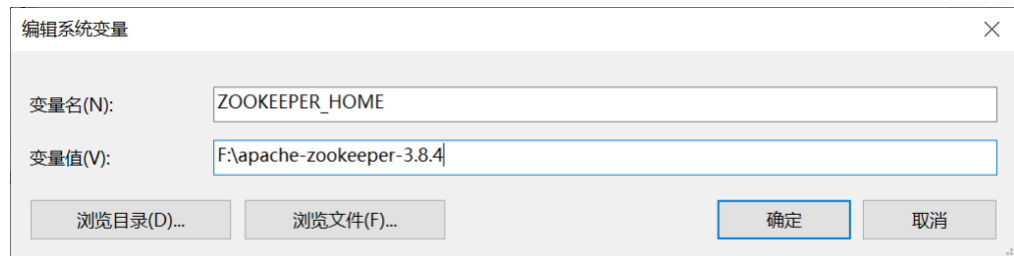


- d. Modify the configuration file zoo.cfg

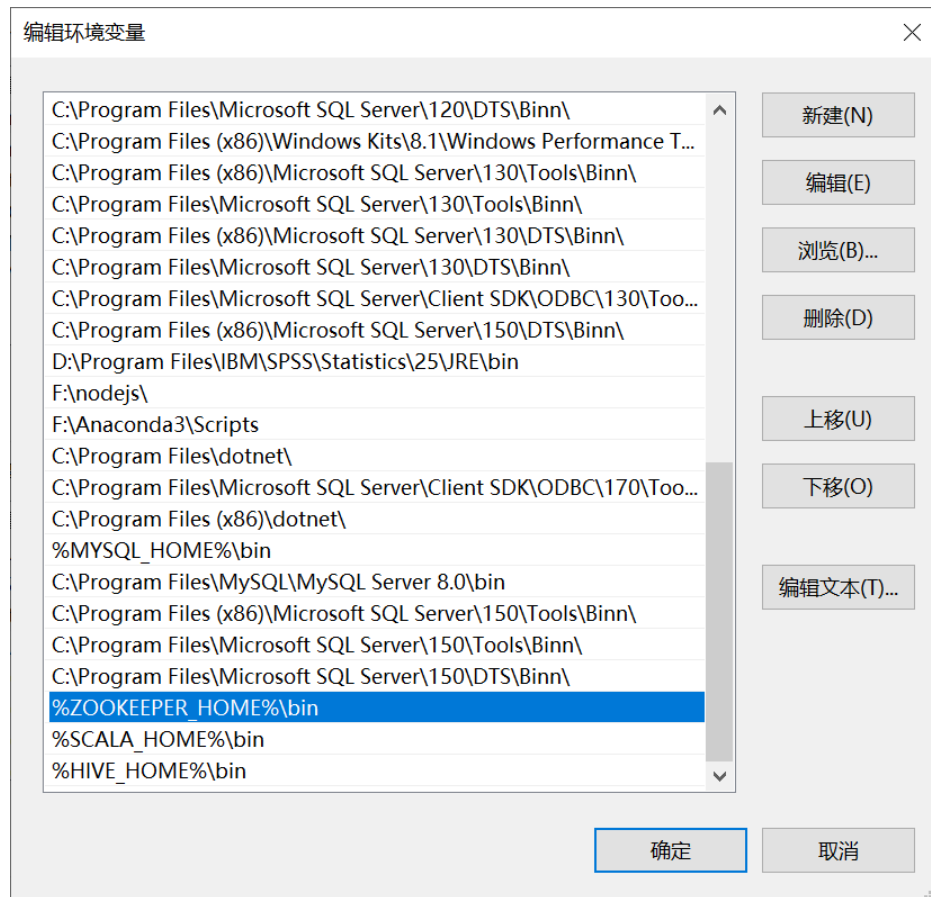
`dataDir=F:\apache-zookeeper-3.8.4\data`

Set dataDir to the location of the data file you just created.

- e. Add system variables for Zookeeper



- f. Add %ZOOKEEPER_HOME%\bin to environment variable Path

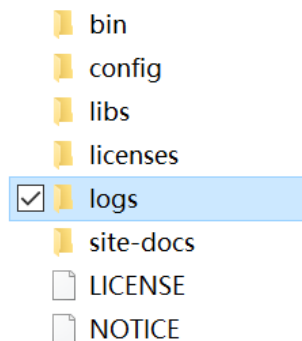


4. Download Apache Kafka

- version: kafka_2.12-3.7.0
- <https://kafka.apache.org/downloads>

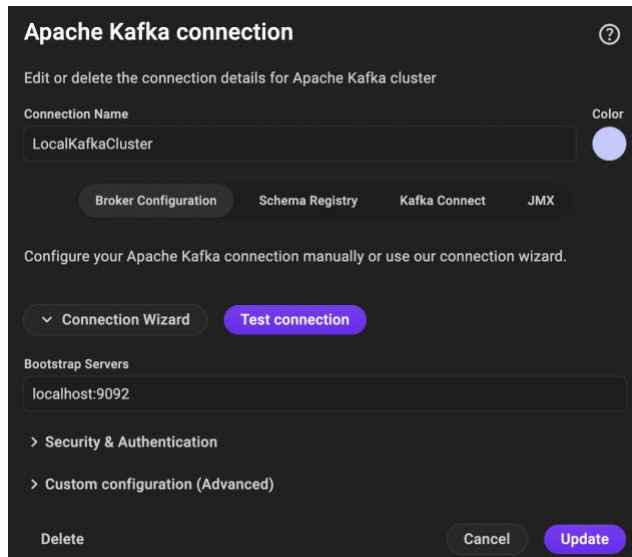
5. Configure the Kafka

- Decompress the Apache Kafka installation package
- Create a new directory logs in the Kafka installation directory



- Modify the configuration file server.properties in config file
 - Set log.dir to the location of the logs file you just created.
log.dirs=F:\kafka_2.12-3.7.0\logs
 - Modify listeners parameters
listeners=PLAINTEXT://localhost:9092

cluster name, such as 'localClusterYourName', and set the bootstrap server to localhost:9092. Use the 'Test Connection' feature to ensure the cluster is working.



Instructions on Executing the Demo 1:

1. Start ZooKeeper

Open the command window and enter the “zkServer” command to start the Zookeeper service.

```
C:\WINDOWS\system32>zkServer
```

The Zookeeper service is running properly if the following information is displayed:

```
2024-03-20 23:38:12,055 [myid:] - INFO [main:o.a.z.s.a.JettyAdminServer@196] - Started AdminServer on address 0.0.0.0, port 8080 and command URL /commands
2024-03-20 23:38:12,064 [myid:] - INFO [main:o.a.z.s.ServerCnxnFactory@169] - Using org.apache.zookeeper.server.NIOServerCnxnFactory as server connection factory
2024-03-20 23:38:12,067 [myid:] - WARN [main:o.a.z.s.ServerCnxnFactory@309] - maxCnxns is not configured, using default value 0.
2024-03-20 23:38:12,071 [myid:] - INFO [main:o.a.z.s.NIOServerCnxnFactory@652] - Configuring NIO connection handler with 10s sessionless connection timeout,
2 selector thread(s), 16 worker threads, and 64 kB direct buffers.
2024-03-20 23:38:12,076 [myid:] - INFO [main:o.a.z.s.NIOServerCnxnFactory@660] - binding to port 0.0.0.0/0.0.0.0:2181
2024-03-20 23:38:12,103 [myid:] - INFO [main:o.a.z.s.w.WatchManagerFactory@42] - Using org.apache.zookeeper.server.watch.WatchManager as watch manager
2024-03-20 23:38:12,104 [myid:] - INFO [main:o.a.z.s.w.WatchManagerFactory@42] - Using org.apache.zookeeper.server.watch.WatchManager as watch manager
2024-03-20 23:38:12,106 [myid:] - INFO [main:o.a.z.s.ZKDatabase@132] - zookeeper.snapshotSizeFactor = 0.33
2024-03-20 23:38:12,111 [myid:] - INFO [main:o.a.z.s.ZKDatabase@152] - zookeeper.snapshotLogCount=500
2024-03-20 23:38:12,118 [myid:] - INFO [main:o.a.z.s.p.SnapStream@61] - zookeeper.snapshot.compression.method = CHECKED
2024-03-20 23:38:12,124 [myid:] - INFO [main:o.a.z.s.p.FileSnap@85] - Reading snapshot F:\apache-zookeeper-3.8.4\data\version-2\snapshot.be
2024-03-20 23:38:12,136 [myid:] - INFO [main:o.a.z.s.DataTree@1717] - The digest in the snapshot has digest version of 2, with zxid as 0xbe, and digest valu
e as 112801181268
2024-03-20 23:38:12,163 [myid:] - INFO [main:o.a.z.a.ZKAuditProvider@42] - ZooKeeper audit is disabled.
2024-03-20 23:38:12,177 [myid:] - INFO [main:o.a.z.s.p.FileTxnSnapLog@372] - 19 txns loaded in 24 ms
2024-03-20 23:38:12,177 [myid:] - INFO [main:o.a.z.s.ZKDatabase@289] - Snapshot loaded in 66 ms, highest zxid is 0xd1, digest is 113886594933
2024-03-20 23:38:12,182 [myid:] - INFO [main:o.a.z.s.p.FileTxnSnapLog@479] - Snapshotting: 0xd1 to F:\apache-zookeeper-3.8.4\data\version-2\snapshot.d1
2024-03-20 23:38:12,186 [myid:] - INFO [main:o.a.z.s.ZooKeeperServer@558] - Snapshot taken in 3 ms
2024-03-20 23:38:12,197 [myid:] - INFO [main:o.a.z.s.RequestThrottler@75] - zookeeper.request_throttler.shutdownTimeout = 10000 ms
2024-03-20 23:38:12,197 [myid:] - INFO [ProcessThread(sid:0 cport:2181)::o.a.z.s.PreRequestProcessor@138] - PreRequestProcessor (sid:0) started, reconfigE
nabled=false
2024-03-20 23:38:12,308 [myid:] - INFO [main:o.a.z.s.ContainerManager@84] - Using checkIntervalMs=60000 maxPerMinute=10000 maxNeverUsedIntervalMs=0
2024-03-20 23:38:30,892 [myid:] - INFO [SessionTracker:o.a.z.s.ZooKeeperServer@643] - Expiring session 0x10002f50c990000, timeout of 180000ms exceeded
```

2. Start Kafka

Open the cmd, go to the Kafka installation directory (F:\kafka_2.12-3.7.0), and enter the following command to start the Kafka service:

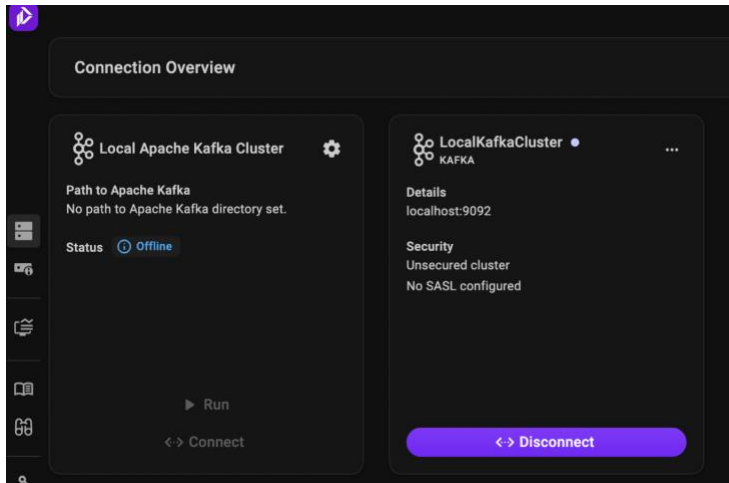
```
F:\kafka_2.12-3.7.0>. \bin\windows\kafka-server-start.bat .\config\server.properties
```


If the following information is displayed, the Kafka service is running properly:

```

2024-03-21 04:03:46.896 INFO [TransactionCoordinator id=0] Starting up. (kafka.coordinator.transaction.TransactionCoordinator)
2024-03-21 04:03:46.902 INFO [TxnMarkerSenderThread-0] Starting (kafka.coordinator.transaction.TransactionMarkerChannelManager)
2024-03-21 04:03:46.902 INFO [TransactionCoordinator id=0] Startup complete. (kafka.coordinator.transaction.TransactionCoordinator)
2024-03-21 04:03:47.008 INFO [ExpirationReaper-0-AlterAcls] Starting (kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
2024-03-21 04:03:47.086 INFO [/config/changes-event-process-thread] Starting (kafka.common.ZkNodeChangeNotificationListener$ChangeEventProcessThread)
2024-03-21 04:03:47.144 INFO [SocketServer listenerType=ZK_BROKER, nodeId=0] Enabling request processing. (kafka.network.SocketServer)
2024-03-21 04:03:47.148 INFO [Awaiting socket connections on localhost:9092. (kafka.network.DataPlaneAcceptor)
2024-03-21 04:03:47.209 INFO [Kafka version: 3.7.0 (org.apache.kafka.common.utils.AppInfoParser)
2024-03-21 04:03:47.210 INFO [Kafka commitId: 2ae524ed625438c5 (org.apache.kafka.common.utils.AppInfoParser)
2024-03-21 04:03:47.212 INFO [Kafka startTimeMs: 1711008227202 (org.apache.kafka.common.utils.AppInfoParser)
2024-03-21 04:03:47.215 INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
2024-03-21 04:03:47.719 INFO [zk-broker-0-to-controller-alter-partition-channel-manager]: Recorded new controller, from now on will use node localhost:9092 (id: 0 rack: null) (kafka.server.NodeToControllerRequestThread)
2024-03-21 04:03:47.746 INFO [zk-broker-0-to-controller-forwarding-channel-manager]: Recorded new controller, from now on will use node localhost:9092 (id: 0 rack: null) (kafka.server.NodeToControllerRequestThread)
    
```

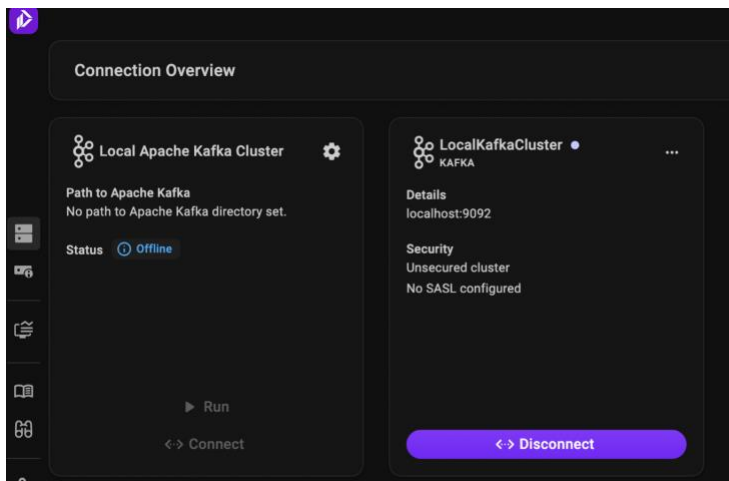
3. Open Kadeck and connect to the cluster



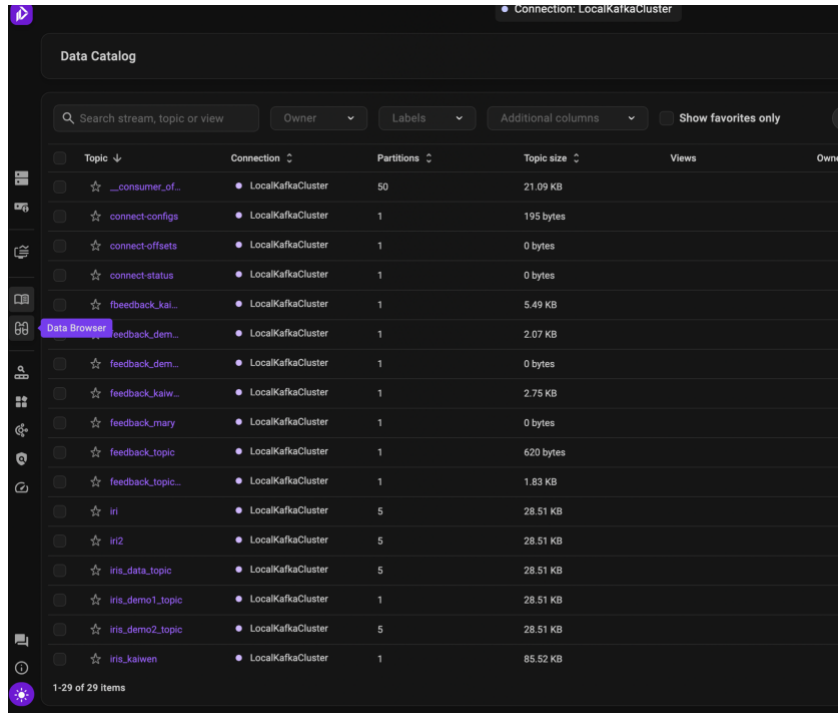
4. Execute your Python code directly in a Jupyter notebook or a .py script
5. Check the Kafka management interface in Kadeck and see the performance

Instructions on Executing the Demo 2:

- To run any producer or consumer from your local machine, with Kadeck open and connected to your cluster, ensure you have the kafka-python library installed by running `pip install kafka-python`. Then, you can execute your Python code directly in a Jupyter notebook or a .py script.



- In Kadeck, navigate to 'Manage Connections' to view all active topics, data flow, and messages. Consumers can also be managed and monitored within Kadeck, which provides a comprehensive view of your Kafka environment. Remember to keep both the Zookeeper and Kafka server Terminal sessions running while working with Kafka. Closing these will stop Zookeeper and Kafka, disrupting the message brokering functions.



The screenshot shows the Kadeck Data Catalog interface. At the top, there's a search bar and filters for 'Owner', 'Labels', and 'Additional columns'. A 'Show favorites only' toggle is also present. Below this is a table of Kafka topics. The table has columns for 'Topic', 'Connection', 'Partitions', 'Topic size', 'Views', and 'Owner'. The 'Data Browser' tab is selected in the left sidebar. The table lists various topics, including consumer-related topics, connect-related topics, and several 'iris' topics. The 'iris_kaiwen' topic is highlighted at the bottom.

Topic	Connection	Partitions	Topic size	Views	Owner
__consumer_of_...	LocalKafkaCluster	50	21.09 KB		
connect-configs	LocalKafkaCluster	1	195 bytes		
connect-offsets	LocalKafkaCluster	1	0 bytes		
connect-status	LocalKafkaCluster	1	0 bytes		
ffeedback_kai...	LocalKafkaCluster	1	5.49 KB		
feedback_dem...	LocalKafkaCluster	1	2.07 KB		
feedback_dem...	LocalKafkaCluster	1	0 bytes		
feedback_kaiw...	LocalKafkaCluster	1	2.75 KB		
feedback_mary	LocalKafkaCluster	1	0 bytes		
feedback_topic	LocalKafkaCluster	1	620 bytes		
feedback_topic...	LocalKafkaCluster	1	1.83 KB		
iri	LocalKafkaCluster	5	28.51 KB		
iri2	LocalKafkaCluster	5	28.51 KB		
iris_data_topic	LocalKafkaCluster	5	28.51 KB		
iris_demo1_topic	LocalKafkaCluster	1	28.51 KB		
iris_demo2_topic	LocalKafkaCluster	5	28.51 KB		
iris_kaiwen	LocalKafkaCluster	1	85.52 KB		

1-29 of 29 items

Demo 1 Code Results

Dataset Explanation: The dataset used is the well-known Iris dataset, which consists of 150 samples of iris flowers. Each sample has four features: sepal length, sepal width, petal length, and petal width, measured in centimeters. The dataset has three species of iris: setosa, versicolor, and virginica.

Enhanced Machine Learning Pipeline with Real-Time Kafka Integration

Data Producer ¶

The data producer reads the Iris dataset and streams it to the iris_stream topic. This part serves to simulate real-time data streaming by setting the hardcoded sleep time.

```
In [2]: # Import required libraries
import json # For converting Python dictionaries to JSON formatted strings
import time # For adding delays in the loop to simulate streaming
from kafka import KafkaProducer # Kafka library to produce messages
import pandas as pd
from kafka.errors import KafkaError # For handling Kafka-specific errors

# Initialise Kafka producer
# Specifies the Kafka server to connect to and the method to serialize the message values to JSON formatted strings.
producer = KafkaProducer(bootstrap_servers='localhost:9092', value_serializer=lambda v: json.dumps(v).encode('utf-8'))

df = pd.read_csv('iris.csv')

for _, row in df.iterrows():
    # Convert the row to a dictionary, suitable for JSON serialisation
    message = row.to_dict()
    try:
        # Attempt to send the message to the 'iris_stream' Kafka topic
        producer.send('iris_demo1_topic', value=message)
        print(f'Sent: {message}') # Print a confirmation that the message was sent
    except KafkaError as e:
        # If sending fails, catch the KafkaError and print the error message
        print(f'Failed to send message: {e}')

    time.sleep(0.1) # Wait for 0.1 seconds before sending the next message to simulate real-time data streaming

# After all messages have been sent, close the producer to free up resources
producer.close()

# Print a confirmation that all messages have been sent
print('All messages sent to Kafka topic \'iris_demo1_topic\'.')
```

```
Sent: {'Id': 1, 'SepalLengthCm': 5.1, 'SepalWidthCm': 3.5, 'PetalLengthCm': 1.4, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 2, 'SepalLengthCm': 4.9, 'SepalWidthCm': 3.0, 'PetalLengthCm': 1.4, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 3, 'SepalLengthCm': 4.7, 'SepalWidthCm': 3.2, 'PetalLengthCm': 1.3, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 4, 'SepalLengthCm': 4.6, 'SepalWidthCm': 3.1, 'PetalLengthCm': 1.5, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 5, 'SepalLengthCm': 5.0, 'SepalWidthCm': 3.6, 'PetalLengthCm': 1.4, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
```

Model Predictor

A Kafka consumer that uses the streamed data for real-time predictions with a pre-trained machine learning model and sends prediction feedback for potential model adjustments. It uses the Iris dataset to train a K-Nearest Neighbors (KNN) classifier, consumes new data from a Kafka topic, makes predictions, and sends feedback on predictions through another Kafka topic.

```
In [3]: # Import necessary libraries
from kafka import KafkaConsumer, KafkaProducer # For interacting with Kafka
import json # For serialisation/deserialisation of data
from sklearn.neighbors import KNeighborsClassifier # KNN classifier from scikit-learn
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import pandas as pd

# Load and prepare the dataset
df = pd.read_csv('iris.csv')

X = df[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']].values
y = df['Species'].values

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialise and train the KNN model
knn = KNeighborsClassifier(n_neighbors=5) # Instantiate the KNN model with 5 neighbors
knn.fit(X_train, y_train)

# Initialise Kafka consumer for consuming incoming data
consumer = KafkaConsumer(
    'iris_demo1_topic',
    bootstrap_servers=['localhost:9092'], # List of Kafka server addresses
    auto_offset_reset='earliest', # Start reading at the earliest message
    consumer_timeout_ms=10000, # Stop consuming if no message for 10 seconds
    value_deserializer=lambda x: json.loads(x.decode('utf-8')) # Deserialize messages from JSON
)

# Initialise Kafka producer for sending feedback
producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'], # List of Kafka server addresses
    value_serializer=lambda x: json.dumps(x).encode('utf-8') # Serialize messages as JSON
)
```

```

    producer.send('feedback_demo1', value=message) # Send feedback message to Kafka topic
    producer.flush() # Ensure message is sent
    print(f'Feedback: {message}')
except KafkaError as e:
    print(f'Failed to send message: {e}') # Handle any errors during sending

# Prepare lists to store true labels and predictions for evaluation
true_labels = []
predictions = []

for message in consumer: # Consume messages from Kafka
    # Extract features from the message
    features = [message.value[f] for f in ['SepallengthCm', 'SepalWidthCm', 'PetallengthCm', 'PetalWidthCm']]
    true_label = message.value['Species'] # Extract the true label

    predicted_label = knn.predict([features])[0] # Predict the label based on the features

    # Store true labels and predictions for later evaluation
    true_labels.append(true_label)
    predictions.append(predicted_label)

    print(f'Predicted Species: {predicted_label}, True Species: {true_label}')

    # If prediction is incorrect, send feedback
    if predicted_label != true_label:
        feedback_message = message.value
        feedback_message['PredictedSpecies'] = predicted_label # Add the predicted label to the feedback
        send_feedback(feedback_message) # Send feedback

# After processing all messages, print a report on model performance
print(classification_report(true_labels, predictions))

# Close the consumer and producer to release resources
consumer.close()
producer.close()

```

```

Predicted Species: Iris-versicolor, True Species: Iris-versicolor
Predicted Species: Iris-versicolor, True Species: Iris-versicolor
Predicted Species: Iris-versicolor, True Species: Iris-versicolor
Predicted Species: Iris-versicolor, True Species: Iris-versicolor
Predicted Species: Iris-versicolor, True Species: Iris-versicolor
Predicted Species: Iris-versicolor, True Species: Iris-versicolor
Predicted Species: Iris-versicolor, True Species: Iris-versicolor
Feedback: {'Id': 71, 'SepallengthCm': 5.9, 'SepalWidthCm': 3.2, 'PetallengthCm': 4.8, 'PetalWidthCm': 1.8, 'Species': 'Iris-versicolo
r', 'PredictedSpecies': 'Iris-virginica'}
Predicted Species: Iris-versicolor, True Species: Iris-versicolor

```

Feedback Consumer

This part initializes a Kafka consumer for consuming messages from a Kafka topic. It uses the `KafkaConsumer` class from the `kafka-python` library to subscribe to a topic named `feedback_demo1` and print out each message it consumes.

```

In [4]: from kafka import KafkaConsumer

# Initialise Kafka consumer for consuming feedback messages
feedback_consumer = KafkaConsumer(
    'feedback_demo1',
    bootstrap_servers=['localhost:9092'],
    group_id='consumer_wrong', # Set the consumer group to 'consumer_wrong'
    auto_offset_reset='earliest',
    consumer_timeout_ms=10000,
    value_deserializer=lambda x: json.loads(x.decode('utf-8'))
)

# Consume messages from the Kafka topic
for message in feedback_consumer:
    print(f'Feedback received: {message.value}')

# Close the consumer to release system resources
feedback_consumer.close()

Feedback received: {'record_id': '123', 'correct_label': 'Iris-versicolor'}
Feedback received: {'Id': 71, 'SepallengthCm': 5.9, 'SepalWidthCm': 3.2, 'PetallengthCm': 4.8, 'PetalWidthCm': 1.8, 'Species': 'Iris-versi
color', 'PredictedSpecies': 'Iris-virginica'}
Feedback received: {'Id': 73, 'SepallengthCm': 6.3, 'SepalWidthCm': 2.5, 'PetallengthCm': 4.9, 'PetalWidthCm': 1.5, 'Species': 'Iris-versi
color', 'PredictedSpecies': 'Iris-virginica'}
Feedback received: {'Id': 84, 'SepallengthCm': 6.0, 'SepalWidthCm': 2.7, 'PetallengthCm': 5.1, 'PetalWidthCm': 1.6, 'Species': 'Iris-versi
color', 'PredictedSpecies': 'Iris-virginica'}
Feedback received: {'Id': 107, 'SepallengthCm': 4.9, 'SepalWidthCm': 2.5, 'PetallengthCm': 4.5, 'PetalWidthCm': 1.7, 'Species': 'Iris-virg
inica', 'PredictedSpecies': 'Iris-versicolor'}
Feedback received: {'Id': 71, 'SepallengthCm': 5.9, 'SepalWidthCm': 3.2, 'PetallengthCm': 4.8, 'PetalWidthCm': 1.8, 'Species': 'Iris-versi
color', 'PredictedSpecies': 'Iris-virginica'}
Feedback received: {'Id': 73, 'SepallengthCm': 6.3, 'SepalWidthCm': 2.5, 'PetallengthCm': 4.9, 'PetalWidthCm': 1.5, 'Species': 'Iris-versi
color', 'PredictedSpecies': 'Iris-virginica'}
Feedback received: {'Id': 84, 'SepallengthCm': 6.0, 'SepalWidthCm': 2.7, 'PetallengthCm': 5.1, 'PetalWidthCm': 1.6, 'Species': 'Iris-versi
color', 'PredictedSpecies': 'Iris-virginica'}
Feedback received: {'Id': 107, 'SepallengthCm': 4.9, 'SepalWidthCm': 2.5, 'PetallengthCm': 4.5, 'PetalWidthCm': 1.7, 'Species': 'Iris-virg
inica', 'PredictedSpecies': 'Iris-versicolor'}

```

Feedback Producer

In a real-world application, this could be part of an application interface where users confirm or correct predictions, thereby generating feedback. Combined with the automatically generated feedback above, this can be used either as an automated system or as a manual process to validate predictions and produce feedback.

```
In [8]: from kafka import KafkaProducer
import json

# Initialize Kafka producer for sending feedback messages
producer = KafkaProducer(bootstrap_servers='localhost:9092',
                        value_serializer=lambda x: json.dumps(x).encode('utf-8'))

# Define a function to send feedback to a Kafka topic
def send_feedback(record_id, correct_label):
    feedback_message = {
        'record_id': record_id, # Unique identifier for the record being corrected.
        'correct_label': correct_label # The correct label for the record
    }
    # Send the feedback message to the 'feedback_demo1' topic
    producer.send('feedback_demo1', value=feedback_message)
    producer.flush()

# Send an example feedback message
# In a real scenario, 'record_id' and 'correct_label' would be dynamically determined
send_feedback(record_id="123", correct_label="Iris-versicolor")

# Print confirmation that feedback was sent
print("Feedback sent to 'feedback_demo1'."

Feedback sent to 'feedback_demo1'.
```

Model Retrainer

Triggered based on a schedule or specific messages, to retrain the model with all available data, including new or corrected labels. The goal is to improve model accuracy by incorporating corrected labels or new data.

```
In [5]: import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Define a function to retrain the model with updated feedback data
def retrain_model_with_feedback():
    # Load the dataset, which is now assumed to include feedback corrections
    df = pd.read_csv('iris.csv') # Assumes 'iris.csv' is the updated dataset file path

    X = df[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']].values
    y = df['Species'].values

    # Splitting dataset into training and testing
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Initialize and train the KNN model
    knn = KNeighborsClassifier(n_neighbors=5)
    knn.fit(X_train, y_train)

    # Evaluate model accuracy
    predictions = knn.predict(X_test)
    print(f"Accuracy after retraining: {accuracy_score(y_test, predictions)}")

# Trigger the retraining process manually
retrain_model_with_feedback()

Accuracy after retraining: 1.0
```

Demo 2 Code Results

Dataset Explanation: The dataset used is the well-known Iris dataset, which consists of 150 samples of iris flowers. Each sample has four features: sepal length, sepal width, petal length, and petal width, measured in centimeters. The dataset has three species of iris: setosa, versicolor, and virginica.

Term Paper: Exploration of Apache Kafka

Setting up Kafka topics

- Setting up Kafka topic to automate infrastructure setup and ensure topic is configured correctly before deploying apps that produce or consume Kafka messages.
- Dividing the topic to the partitions.

```
In [1]: # KafkaAdminClient: Used for administrative operations with Kafka, like creating topics.
# NewTopic: A class used to specify the properties of a topic to be created.
from kafka.admin import KafkaAdminClient, NewTopic

# The AdminClient is used to manage and inspect topics, brokers, and other Kafka objects.
admin_client = KafkaAdminClient(
    bootstrap_servers="localhost:9092" # a list of strings (host:port), establish connection to the Kafka cluster.
)

topic_name = 'iris_demo2_topic'

# Create a list that contains the new topic's configuration.
# num_partitions: The number of partitions for the topic. Partitions are the parallelism factor in Kafka.
# replication_factor: The number of replicates for each partition in the topic. A factor of 1 means no replication.
topic_list = [NewTopic(name=topic_name, num_partitions=5, replication_factor=1)]

# Create the topic(s) in the Kafka cluster.
# new_topics: A list of NewTopic objects that represent the topics to be created.
# validate_only: If False, the topics will be created now; if True, the request will only be validated.
admin_client.create_topics(new_topics=topic_list, validate_only=False)

Out[1]: CreateTopicsResponse_v3(throttle_time_ms=0, topic_errors=[(topic='iris_demo2_topic', error_code=0, error_message=None)])
```

Data Producer

- Implementing a process to serialize and stream data from a CSV file into a Kafka topic in real-time.
- This process efficiently simulates a real-world scenario where data is continuously generated and needs to be processed and streamed to a Kafka topic in real time

```
In [2]: import json # Use for serializing the messages to a JSON formatted string.
import time # Use to introduce a delay between sending messages to simulate real-time data streaming.
from kafka import KafkaProducer # The Kafka client class for producing messages and sending them to Kafka topics.
import pandas as pd
from kafka.errors import KafkaError # Exception class for handling errors in Kafka operations.

In [3]: import time
start_time = time.time()

# Initialize a Kafka producer instance.
# bootstrap_servers specifies the Kafka server to connect to.
# value_serializer is a function to serialize the message value to json and then encode it to UTF-8 bytes.
# This setup allows sending Python dictionaries as messages in a json format.
producer = KafkaProducer(bootstrap_servers='localhost:9092', value_serializer=lambda v: json.dumps(v).encode('utf-8'))

In [4]: # Load data from a CSV file into a pandas dataframe.
df = pd.read_csv('iris.csv')

# Iterate over each row in the dataframe.
for _, row in df.iterrows():
    message = row.to_dict() # Convert the current row to a dictionary.
    try:
        # producer.send sends the message to the specified Kafka topic ('iris_demo2_topic').
        # value argument is the message to be sent, serialized as a JSON string.
        producer.send('iris_demo2_topic', value=message)

        # If sending is working, print confirmation.
        print(f"Sent: {message}")
    except KafkaError as e:
        # If sending fails, print an error message.
        print(f"Failed to send message: {e}")
    time.sleep(0.1) # Simulating real-time streaming, wait for 0.1 seconds before sending the next message.
```

```

# Close the producer instance once all messages have been sent.
producer.close()
end_time = time.time()

print("All messages sent to Kafka topic 'iris_stream'.")
print(f"Producer creation time takes {end_time - start_time} seconds")

Sent: {'Id': 1, 'SepalLengthCm': 5.1, 'SepalWidthCm': 3.5, 'PetalLengthCm': 1.4, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 2, 'SepalLengthCm': 4.9, 'SepalWidthCm': 3.0, 'PetalLengthCm': 1.4, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 3, 'SepalLengthCm': 4.7, 'SepalWidthCm': 3.2, 'PetalLengthCm': 1.3, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 4, 'SepalLengthCm': 4.6, 'SepalWidthCm': 3.1, 'PetalLengthCm': 1.5, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 5, 'SepalLengthCm': 5.0, 'SepalWidthCm': 3.6, 'PetalLengthCm': 1.4, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 6, 'SepalLengthCm': 5.4, 'SepalWidthCm': 3.9, 'PetalLengthCm': 1.7, 'PetalWidthCm': 0.4, 'Species': 'Iris-setosa'}
Sent: {'Id': 7, 'SepalLengthCm': 4.6, 'SepalWidthCm': 3.4, 'PetalLengthCm': 1.4, 'PetalWidthCm': 0.3, 'Species': 'Iris-setosa'}
Sent: {'Id': 8, 'SepalLengthCm': 5.0, 'SepalWidthCm': 3.4, 'PetalLengthCm': 1.5, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 9, 'SepalLengthCm': 4.4, 'SepalWidthCm': 2.9, 'PetalLengthCm': 1.4, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 10, 'SepalLengthCm': 4.9, 'SepalWidthCm': 3.1, 'PetalLengthCm': 1.5, 'PetalWidthCm': 0.1, 'Species': 'Iris-setosa'}
Sent: {'Id': 11, 'SepalLengthCm': 5.4, 'SepalWidthCm': 3.7, 'PetalLengthCm': 1.5, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 12, 'SepalLengthCm': 4.8, 'SepalWidthCm': 3.4, 'PetalLengthCm': 1.6, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 13, 'SepalLengthCm': 4.8, 'SepalWidthCm': 3.0, 'PetalLengthCm': 1.4, 'PetalWidthCm': 0.1, 'Species': 'Iris-setosa'}
Sent: {'Id': 14, 'SepalLengthCm': 4.3, 'SepalWidthCm': 3.0, 'PetalLengthCm': 1.1, 'PetalWidthCm': 0.1, 'Species': 'Iris-setosa'}
Sent: {'Id': 15, 'SepalLengthCm': 5.8, 'SepalWidthCm': 4.0, 'PetalLengthCm': 1.2, 'PetalWidthCm': 0.2, 'Species': 'Iris-setosa'}
Sent: {'Id': 16, 'SepalLengthCm': 5.7, 'SepalWidthCm': 4.4, 'PetalLengthCm': 1.5, 'PetalWidthCm': 0.4, 'Species': 'Iris-setosa'}
Sent: {'Id': 17, 'SepalLengthCm': 5.4, 'SepalWidthCm': 3.9, 'PetalLengthCm': 1.3, 'PetalWidthCm': 0.4, 'Species': 'Iris-setosa'}
Truncated

In [5]: from kafka import KafkaConsumer, KafkaProducer, TopicPartition
import json
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
import pandas as pd

In [6]: # Load and prepare the dataset
df = pd.read_csv('iris.csv')

# The features (X) are the measurements of the iris flowers:
# Sepal Length, Sepal Width, Petal Length, and Petal Width.
X = df[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']].values

# The target (y) is the species of the iris flower.
y = df['Species'].values

# 20% of the data will be used for testing, while the rest will be used for training.
# random_state=42 is used to seed the random number generator for reproducibility.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and train the KNN model
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

Out[6]: KNeighborsClassifier
KNeighborsClassifier()

In [7]: start_time = time.time()

# Initialize Kafka producer for sending feedback
producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    value_serializer=lambda x: json.dumps(x).encode('utf-8')
)

```



```
In [8]: def send_feedback(message):
        try:
            producer.send('feedback_topic_demo2', value=message) # Attempt to send the message to the topic.
            producer.flush() # Flush the producer to force all queued messages to be sent to the Kafka broker.
            print(f"Feedback: {message}") # Print a confirmation that the feedback message has been sent.
        except KafkaError as e:
            print(f"Failed to send message: {e}") # otherwise, print an error
```

Data Consumers

- Multiple consumers are potentially run, one for each partition of the topic, to parallelize processing and increase throughput.

```
In [9]: # Function to create and run a consumer for a specified partition
def run_consumer_for_partition(partition_id):
    # Initialize a KafkaConsumer with specific configurations.
    consumer = KafkaConsumer(
        bootstrap_servers='localhost:9092', # Kafka cluster connection info.
        auto_offset_reset='earliest', # Start reading at the earliest message.
        consumer_timeout_ms=10000, # Stop listening if no message is received within 10 seconds.
        value_deserializer=lambda x: json.loads(x.decode('utf-8')), # Deserialize messages from JSON.
        group_id=f'unique_group_for_demo2_partition_{partition_id}' # Unique group_id for each partition
    )
    # Assign the consumer to a specific partition of the topic.
    partition = TopicPartition('iris_demo2_topic', partition_id)
    consumer.assign([partition])

    # Process messages received from the partition.
    for message in consumer:
        # Extract features for prediction from the message.
        features = [message.value[f] for f in ['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']]
        # Extract the true label from the message.
        true_label = message.value['Species']
        # Use the pre-trained KNN model to predict the label based on the features.
        predicted_label = knn.predict([features])[0]

        # Print the partition ID, predicted label, and true label.
        print(f"Partition: {partition_id}, Predicted Species: {predicted_label}, True Species: {true_label}")

    # If the predicted label does not match the true label, send feedback.
    if predicted_label != true_label:
        feedback_message = message.value
        feedback_message['PredictedSpecies'] = predicted_label
        send_feedback(feedback_message) # Call the send_feedback function to send the corrected message.

    # Close the consumer to release resources.
    consumer.close()
```

Data Feedback

[illegible]

Conclusion

- The producer creation and prediction time are logged, providing insight into the performance and efficiency of the streaming process.

- The consumers results display a list of predicted species along with the true species. The partitions from which the consumers read the messages are also listed, emphasizing the distributed nature of Kafka's topic consumption.
- Instances where the predicted species matches the true species suggest correct model performance.
- The feedback loop is not explicitly shown in the results but is mentioned as part of the process. It's a mechanism to correct the model in case of wrong predictions, enhancing the model's accuracy over time.

Overall, the term paper illustrates a Kafka-based data pipeline integrated with a machine learning model, highlighting the ability of Kafka to handle real-time data operations combined with predictive analytics. A Kafka consumer, set up in multiple instances to parallelize processing, received the messages. Each message contained data from the Iris dataset, and the K-Nearest Neighbors (KNN) algorithm was used to predict the species based on the features in the message. If the predicted species did not match the actual species (true label), feedback was sent to another Kafka topic for correction or reprocessing. The output shows a series of messages with predictions that match the true labels, indicating that the model is generally performing well.