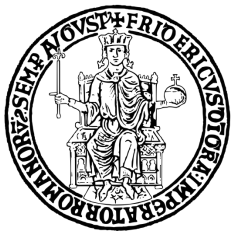


UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

2024

HARDWARE AND SOFTWARE PER BIG DATA

Prepared by:

Hassan Muhammad Siraj
Muhammad Sameer Kazi
Shuja Ur Rehman

Contents

Big Data

Big data refers to massive datasets that are too large or complex to be analyzed using traditional data processing methods. These datasets are typically characterized by the three V's of big data:

- **Volume:** Big data sets can range from terabytes (TB) to exabytes (EB) or more, making them significantly larger than traditional data sets.
- **Velocity:** Big data is generated and produced at an unprecedented rate, often in real-time or near real-time. This requires data processing systems to be able to handle and analyze data streams quickly and efficiently.
- **Variety:** Big data encompasses a wide variety of data formats, including structured data from databases, semi-structured data from logs and documents, and unstructured data from images, videos, and audio. This diversity poses challenges in data integration and analysis.

How is big data used?

The applications of big data are vast and expanding across various industries, including:

- **Finance:** Big data is used to analyze financial markets, assess credit risk, detect fraudulent activities, and develop personalized financial products.
- **Healthcare:** Big data is used to track patient health records, identify potential health risks, develop new treatments, and improve disease prevention strategies.
- **Retail:** Big data is used to track customer purchases, preferences, and shopping behaviors to personalize product recommendations, optimize inventory management, and enhance customer engagement.

What are the challenges of big data?

While big data offers immense potential, it also presents challenges that need to be addressed:

- **Data storage and management:** Storing and managing massive datasets requires specialized infrastructure and expertise.
- **Data integration and heterogeneity:** Integrating data from various sources, each with its own format and structure, can be complex and time-consuming.
- **Data analysis and visualization:** Extracting meaningful insights from big data requires advanced analytics techniques and tools.

- **Data security and privacy:** Protecting sensitive data from unauthorized access and ensuring data privacy compliance are critical considerations.

- **Skills shortage:** The demand for skilled professionals with expertise in big data analytics is high, leading to a talent shortage in some areas.

Apache Kafka

Apache kafka is a distributed streaming platform that handles high-volume real-time data feeds. It is a popular choice for building real-time data pipelines and applications, as it provides a reliable, scalable, and fault-tolerant way to ingest, process, and store data.

Key components:

Apache Kafka consists of the following key components:

1. **Brokers:** Brokers are the central servers in a Kafka cluster that are responsible for storing and managing messages. They form the backbone of the Kafka infrastructure and are responsible for ensuring the reliable delivery, persistence, and replication of messages across the cluster..

2. **Producers:** Producers are applications that generate and send messages to Kafka brokers. They publish messages to topics, which are like queues for storing messages. Producers are responsible for categorizing and organizing messages into topics and ensuring they are delivered to the appropriate brokers.

3. **Consumers:** Consumers are applications that subscribe to Kafka topics and receive messages from them. They process the messages and perform the desired actions, such as storing information in a database, triggering alerts, or generating reports. Consumers are responsible for consuming messages in a timely manner and processing them correctly.

4. **ZooKeeper:** Zookeeper is a distributed coordination service that maintains the overall health and organization of a Kafka cluster. It ensures that all brokers are aware of each other, that messages are routed correctly, and that the cluster remains synchronized in case of failures.

5. **Working principle:** When a producer sends a message to a topic, Kafka brokers replicate the message across multiple partitions. Each partition is assigned a unique ID and is stored on a different broker. This replication ensures data durability and fault tolerance. Consumers subscribe to one or more topics and receive messages from the partitions accordingly. They can work in either at-least-once or exactly once mode, depending on the desired consistency level.

Key features:

Apache Kafka offers several key features that make it a popular choice for real-time data processing:

1. **High throughput:** Kafka can handle high volumes of data with low latency, making it ideal for real-time applications.

2. **Scalability:** Kafka can be scaled horizontally by adding more brokers to the cluster, enabling it to handle increasing data volumes and throughput.

3. **Fault tolerance:** Kafka is a highly fault-tolerant system that can continue to operate even in the event of failures. This is due to its replication and ZooKeeper coordination mechanism.

4. **Durability:** Kafka messages are replicated across multiple brokers, ensuring that data is not lost even if some brokers fail.

5. **Event streaming:** Kafka supports event streaming, which means that messages can be processed as they are produced, without waiting for them to be all stored. This makes it ideal for applications that require real-time response.

Use cases:

Apache Kafka is used in a wide variety of use cases across various industries, including:

1. **Log aggregation:** Kafka can be used to collect and store logs from applications and servers, enabling businesses to monitor the health and performance of their systems.

2. **Real-time analytics:** Kafka can be used to stream data to analytics engines for real-time analysis. This can be used for tasks such as fraud detection, anomaly detection, and user behavior analysis.

3. **Stream processing:** Kafka can be used to process data streams in real time. This can be used for tasks such as data enrichment, transformation, and filtering.

4. **Event-driven architectures:** Kafka can be used to build event-driven architectures, which are systems that react to events as they occur. This can be used to automate processes, improve efficiency, and reduce latency.

Benefits of using Kafka:

The benefits of using Apache Kafka include:

1. **Real-time insights:** Kafka allows businesses to analyze and respond to data in real time, enabling them to make better decisions and improve their operations.

2. **Scalability:** Kafka can be scaled horizontally to handle increasing data volumes and throughput, making it a cost-effective solution for growing businesses.

3. **Fault tolerance:** Kafka is a highly fault-tolerant system that can continue to operate even in the event of failures, minimizing downtime and data loss.

4. **Versatility:** Kafka can be used for a wide variety of use cases, making it a versatile tool for building real-time data pipelines and applications.

MapReduce

MapReduce is a distributed programming model that breaks down a large data processing task into smaller, parallelizable tasks. It comprises three main components:

1. **Job Tracker:** The job tracker is responsible for managing the overall MapReduce job execution, including scheduling mapper and reducer tasks, tracking task progress, and ensuring data distribution.

2. **Task Trackers:** Task trackers are deployed on worker nodes and handle the execution of individual mapper and reducer tasks. They receive tasks from the job tracker, perform the assigned processing, and report back the results.

3. **Distributed File System (DFS):** The DFS, typically Hadoop Distributed File System (HDFS), provides reliable storage for the intermediate and final output of MapReduce jobs. It distributes data across multiple nodes for efficient processing and fault tolerance.

MapReduce Working Principles:

A MapReduce job consists of two phases:

1. **Map Phase:** The map phase processes each input data record and produces a set of key-value pairs. The mapper function extracts relevant information from the input record and transforms it into a key-value pair. The key is typically a unique identifier for the data element, and the value is the processed data or a transformation of it.

2. **Reduce Phase:** The reduce phase receives the intermediate key-value pairs from the map phase and aggregates them based on the key. The reducer function takes a group of key-value pairs with the same key and combines them, producing a smaller set of key-value pairs with aggregated data.

PySpark

PySpark is a high-level Python API for Apache Spark, a distributed data processing framework for large-scale data analysis. It enables users to leverage the power of Spark's distributed computing capabilities from within Python. PySpark's architecture comprises the following components:

1. **SparkContext**: The SparkContext is the entry point to the Spark cluster. It manages the connection to the Spark cluster, allocates resources, and handles tasks distribution.

2. **RDDs (Resilient Distributed Datasets):** RDDs are the fundamental data abstraction in Spark. They represent collections of partitioned data that are distributed across multiple nodes in a cluster.

3. **Spark Core:** The Spark Core library provides the core functionality for distributed data processing, including task scheduling, fault tolerance, and resource management.

4. **Spark SQL:** The Spark SQL library enables structured data processing using SQL-like queries. It provides a DataFrame API for working with structured data and integrates with various data sources, including databases and CSV files.

5. **MLlib (Machine Learning Library):** The MLlib library provides a collection of machine learning algorithms for data analysis and prediction. It includes algorithms for classification, regression, clustering, and dimensionality reduction.

PySpark Working Principles:

PySpark utilizes a parallel processing paradigm to efficiently handle large-scale data analysis tasks. It breaks down data processing into smaller, parallelizable tasks that can be executed on multiple machines in the cluster. This distributed approach enables efficient processing of large datasets, even on commodity hardware.

1. **Data Retrieval:** Data is retrieved from various sources, such as local files, HDFS, or databases, and loaded into Spark as RDDs or DataFrames.

2. **Data Transformation:** Transformations are applied to RDDs or DataFrames to clean, filter, and prepare the data for analysis.

3. **Data Analysis:** Machine learning algorithms in MLlib are used to analyze the data, extracting patterns and insights.

4. **Results Aggregation:** The results of the analysis are aggregated and presented in a meaningful way, such as visualizations or summary statistics.

Benefits of Using PySpark:

PySpark offers several benefits for big data analysis:

1. **Ease of Use:** PySpark provides a Python API, making it accessible to users familiar with Python.

2. **Scalability:** PySpark can scale horizontally to handle increasing data volumes and processing demands.

3. **Fault Tolerance:** Spark's distributed architecture and fault tolerance mechanism ensure that the system can continue to operate even if nodes fail.

4. **Versatility:** PySpark can handle both structured and unstructured data, making it suitable for a wide range of data analysis tasks.

5. **Integration with Diverse Tools:** PySpark can integrate with various data sources, tools, and libraries, providing flexibility in data processing workflows.

Our Project and Code With Explanation

Environment: Colab

Libraries Installation

```
!pip install kafka-python
!pip install confluent_kafka
!pip install pyspark
!pip install sweetviz
```

!pip install yfinance

is a Python library that allows you to download historical market data from Yahoo Finance. It provides an easy way to access stock market data, including historical prices, dividends, and splits.

Example Use Case: Fetching historical stock prices for analysis, building financial models, or creating stock market visualizations.

!pip install pyspark

PySpark is the Python API for Apache Spark, a fast and general-purpose cluster computing system. It provides high-level APIs in Python, Java, Scala, and R, making it easier to build parallel processing applications

Example Use Case: Analyzing large datasets, distributed data processing, machine learning, and building scalable data pipelines

!pip install sweetviz

sweetviz is a Python library that generates beautiful, high-density visualizations for data exploration. It helps in comparing datasets, identifying patterns, and understanding the structure of the data

Example Use Case: Exploratory Data Analysis (EDA) during the initial stages of a data science project to gain insights into the dataset's characteristics.

!pip install kafka-python

kafka-python is a Python client for Apache Kafka, a distributed streaming platform. It allows you to produce and consume messages to/from Kafka topics, facilitating real-time data processing.

Example Use Case: Building applications that need to ingest, process, or distribute streaming data using Kafka as the messaging backbone.

!pip install confluent_kafka

confluent_kafka is a Python client for Apache Kafka, similar to kafka-python. It's developed by Confluent, the company founded by the creators of Kafka. It provides additional features and enhancements.

Example Use Case: Similar to kafka-python, used for producing and consuming messages in Kafka-based applications

Importing Libraries

```
import os
from datetime import datetime
import time
import threading
import json
```

```
from kafka import KafkaProducer
from kafka.errors import KafkaError
import pandas as pd
import yfinance as yf
import sweetviz as sv
import pandas as pd
from pyspark.sql import SparkSession
from pyspark.ml.clustering import KMeans
from pyspark.sql.functions import col
from datetime import date
from pyspark.sql import functions as F
from pyspark.sql.window import Window
from pyspark.sql import Row
from pyspark.ml.feature import VectorAssembler, StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
```

Crawling and appending data to Dataframe from Yahoo finance

In below attached code we fetched the 1 year historical data of 93 different companies from yahoo finance using yfinance library and created dataFrame as stock_data.

```
def get_stock_data(ticker, start_date, end_date):
    # Download historical data for the specified ticker symbol
    data = yf.download(ticker, start=start_date, end=end_date);
    data['Symbol'] = ticker
    return data

def get_multiple_stocks_data(tickers, start_date, end_date):
    # Create an empty DataFrame to store the combined data
    stock_data = pd.DataFrame()
    #c = 0
    # Loop through each ticker and fetch historical data
    for ticker in tickers:
      print(ticker)
      combined_data = get_stock_data(ticker, start_date, end_date)
      stock_data = stock_data.append(combined_data)
    return stock_data
# Specify the ticker symbol and date range
start_date = "2023-01-01"
end_date = "2024-01-01"

# Specify additional ticker symbols
all_tickers =
['AAPL','T','GOOG','MO','AA','AXP','BABA','ABT','AMAT','AMGN','AIG','ALL','ADBE','GOOGL','ACN','ABBV',
'MT','LLY','APA','ADP','AKAM','NLY','ADSK','ADM','WBA','PANW','AMD','AVGO','EA','AEM','APD','AMBA','NV
S','LULU','ARRY','A','ORLY','AZO','AN','AZN','BUD','BDX','AB','AFL','ADI','ACIW','AMP','AMTD','AEO','NVO','A
LTR','PAA','AAP','FNMA','UBS','ARLP','ATI','ADT','AVB','LH','AVY','AON','ADC','AYI','ASML','AMT','ACM','ARI'
,'AR','AAN','BAH','ALB','AIZ','SAIC','CAR','AU','APH','AMX','JKHY','AMKR','AEIS','VRSK','APO','RBA','MAA','A
SX','ARCO','ANET','AIR','WAB','RS','PKG','AMG']
```

```
# Get historical data for multiple stocks
stock_data = get_multiple_stocks_data(all_tickers, start_date, end_date)
#Resetting Index
stock_data = stock_data.reset_index()

stock_data['Date']
```

For this we created two functions:

get_stock_data

It Downloads historical stock market data for a specified ticker symbol within a given date range using the yfinance library. DataFrame containing historical stock data with an additional 'Symbol' column indicating the ticker symbol.

get_multiple_stocks_data

It Iterates through a list of ticker symbols, fetches historical data for each stock using the **get_stock_data** function, and combines the data into a single DataFrame. Combined DataFrame containing historical data for all specified stocks

When we download historical data using yfinance, the **date** becomes the index of the DataFrame. By Calling **reset_index()** it will reset the index of the DataFrame to the default integer-based index and move the existing index (in this case, the 'Date' column) back to a regular column.

```
stock_data
```

	Date	Open	High	Low	Close	Adj Close	Volume	Symbol
0	2023-01-03	130.279999	130.899994	124.169998	125.070000	124.374802	112117500	AAPL
1	2023-01-04	126.889999	128.660004	125.080002	126.360001	125.657639	89113600	AAPL
2	2023-01-05	127.129997	127.769997	124.760002	125.019997	124.325081	80962700	AAPL
3	2023-01-06	126.010002	130.289993	124.889999	129.619995	128.899506	87754700	AAPL
4	2023-01-09	130.470001	133.410004	129.889999	130.149994	129.426544	70790800	AAPL
...	...	...	...	...	...	...	...	...
3745	2023-12-22	75.550003	75.643997	72.790001	73.430000	73.430000	7138800	TTD
3746	2023-12-26	73.680000	73.820000	72.269997	73.589996	73.589996	3697200	TTD
3747	2023-12-27	73.849998	74.489998	73.349998	74.080002	74.080002	2541100	TTD
3748	2023-12-28	73.790001	74.339996	73.220001	73.400002	73.400002	2571600	TTD
3749	2023-12-29	73.300003	73.481003	71.639999	71.959999	71.959999	3764400	TTD

3750 rows × 8 columns

Sweetviz report for data analysis

```
# Convert 'Date' column to datetime type
```

```
stock_data['Date'] = pd.to_datetime(stock_data['Date'])

# Group by 'Date' and 'Stock' and pivot the DataFrame
grouped_data = stock_data.groupby(['Date', 'Symbol']).first().unstack()

# Flatten the multi-level column index
grouped_data.columns = ['_'.join(map(str, col)).strip() for col in
grouped_data.columns.values]

# Reset index to make 'Date' a column again
grouped_data.reset_index(inplace=True)

# Display the resulting DataFrame
print(grouped_data)
advert_report = sv.analyze(grouped_data.filter(like = 'Close'))
advert_report.show_html('Stocks_raw_report.html')
```

stock_data['Date'] = pd.to_datetime(stock_data['Date'])

This line converts the 'Date' column in the stock_data DataFrame to the datetime type using the pd.to_datetime function from the pandas library.

grouped_data = stock_data.groupby(['Date', 'Symbol']).first().unstack()

This line groups the stock_data DataFrame by the 'Date' and 'Symbol' columns and then applies the first() function to each group. The unstack() method is then used to pivot the DataFrame, making 'Symbol' values become columns.

grouped_data.columns = ['_'.join(map(str, col)).strip() for col in grouped_data.columns.values]

This line flattens the multi-level column index created by the pivot operation. It uses a list comprehension to concatenate the levels of the multi-index into single strings, separated by underscores.

grouped_data.reset_index(inplace=True)

This line resets the index of the grouped_data DataFrame, making 'Date' a column again. The inplace=True argument modifies the DataFrame in place

print(grouped_data)

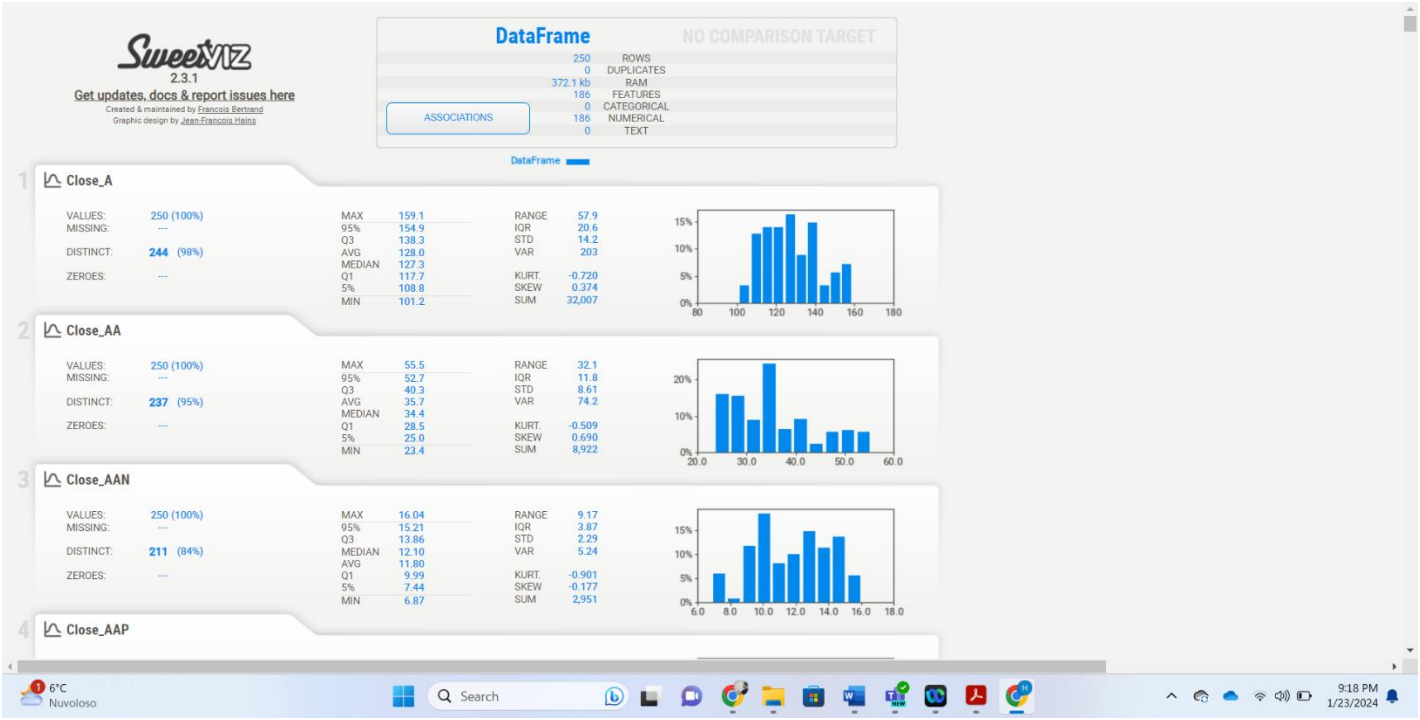
This line prints the resulting DataFrame (grouped_data) to the console

advert_report = sv.analyze(grouped_data.filter(like='Close'))

advert_report.show_html('Stocks_raw_report.html')

These lines use the sweetviz library to generate an analysis report (advert_report) for columns in grouped_data that contain the string 'Close'. The filter(like='Close') method is used to select only the columns with names containing 'Close'. Finally, the show_html method generates an HTML report and saves it to a file named 'Stocks_raw_report.html'.

Overall, this code takes historical stock data (stock_data), performs data manipulation and transformation operations, and then generates an analysis report using sweetviz for columns related to closing stock prices.



Transfering Pandas Data to pyspark DF and then to RDD

```
spark = SparkSession.builder.appName("StockDataRDD").getOrCreate()
stock_data = stock_data.rename(columns={"Adj Close": "Adj_Close"})
stock_data_df = spark.createDataFrame(stock_data)

# Convert Python DataFrame to PySpark RDD
stock_data_rdd = stock_data_df.rdd
stock_data_df.show()
```

SparkSession.builder.appName("StockDataRDD").getOrCreate():

We Initialized a Spark session with the **StockDataRDD** name. The appName method sets the application name, and getOrCreate either gets an existing Spark session or creates a new one.

stock_data = stock_data.rename(columns={"Adj Close": "Adj_Close"}):

We Renamed the 'Adj Close' column to 'Adj_Close' in the original Pandas DataFrame. Column names with spaces can sometimes cause issues in PySpark, so this renaming is done to adhere to PySpark's requirements.

spark.createDataFrame(stock_data):

We Converted the Pandas DataFrame (stock_data) to a PySpark DataFrame (stock_data_df).

stock_data_df.show():

Displays the content of the PySpark DataFrame.

```
+-------------------+------------------+------------------+------------------+------------------+------------------+---------+------+
|               Date|              Open|              High|               Low|             Close|         Adj_Close|   Volume|Symbol|
+-------------------+------------------+------------------+------------------+------------------+------------------+---------+------+
|2023-01-03 00:00:00|130.27999877929688|130.89999389648438|124.16999816894531|125.06999969482422|124.37480163574219|112117500|  AAPL|
|2023-01-04 00:00:00|126.88999938964844|128.66000366210938|125.08000183105469|126.36000061035156|125.65763854980469| 89113600|  AAPL|
|2023-01-05 00:00:00|127.12999725341797| 127.7699966430664|124.76000213623047| 125.0199966430664|124.32508087158203| 80962700|  AAPL|
|2023-01-06 00:00:00|126.01000213623047| 130.2899932861328|124.88999938964844| 129.6199951171875|128.89950561523438| 87754700|  AAPL|
|2023-01-09 00:00:00|130.47000122070312|133.41000366210938|129.88999938964844|130.14999389648438|129.42654418945312| 70790800|  AAPL|
|2023-01-10 00:00:00|130.25999450683594|131.25999450683594| 128.1199951171875|130.72999572753906| 130.0033416748047| 63896200|  AAPL|
|2023-01-11 00:00:00|            131.25|133.50999450683594| 130.4600067138672|133.49000549316406| 132.7480010986328| 69458900|  AAPL|
|2023-01-12 00:00:00| 133.8800048828125|134.25999450683594|131.44000244140625|133.41000366210938|   132.66845703125| 71379600|  AAPL|
|2023-01-13 00:00:00|132.02999877929688| 134.9199981689453|131.66000366210938|134.75999450683594|134.01095581054688| 57809700|  AAPL|
|2023-01-17 00:00:00| 134.8300018310547| 137.2899932861328| 134.1300048828125|135.94000244140625|135.18438720703125| 63646600|  AAPL|
|2023-01-18 00:00:00|136.82000732421875|138.61000061035156|135.02999877929688| 135.2100067138672| 134.4584503173828| 69672800|  AAPL|
|2023-01-19 00:00:00| 134.0800018310547|            136.25|133.77000427246094|135.27000427246094| 134.5181121826172| 58280400|  AAPL|
|2023-01-20 00:00:00|135.27999877929688|138.02000427246094|134.22000122070312| 137.8699951171875| 137.1036376953125| 80223600|  AAPL|
|2023-01-23 00:00:00| 138.1199951171875|143.32000732421875|137.89999389648438|141.11000061035156|140.32565307617188| 81760300|  AAPL|
|2023-01-24 00:00:00|140.30999755859375|143.16000366210938| 140.3000030517578|142.52999877929688| 141.7377471923828| 66435100|  AAPL|
|2023-01-25 00:00:00|140.88999938964844|142.42999267578125|138.80999755859375|141.86000061035156| 141.0714874267578| 65799300|  AAPL|
|2023-01-26 00:00:00| 143.1699981689453|            144.25|141.89999389648438| 143.9600067138672|143.15982055664062| 54105100|  AAPL|
|2023-01-27 00:00:00|143.16000366210938|147.22999572753906| 143.0800018310547|145.92999267578125| 145.1188507080078| 70555800|  AAPL|
|2023-01-30 00:00:00| 144.9600067138672| 145.5500030517578|142.85000610351562|             143.0| 142.2051544189453| 64015300|  AAPL|
|2023-01-31 00:00:00| 142.6999969482422|144.33999633789062|142.27999877929688| 144.2899932861328| 143.4879608154297| 65874500|  AAPL|
+-------------------+------------------+------------------+------------------+------------------+------------------+---------+------+
only showing top 20 rows
```

Performing feature engineering using RDD map to form Daily_return and Cumulative_Return feature

```
# Create a PySpark session
spark = SparkSession.builder.appName("StockDataRDD").getOrCreate()

#stock_data_rdd = spark.sparkContext.parallelize(data)
# Function to calculate returns for each stock
def calculate_returns(partition_index, iterator):

    prev_close_price = {}
    cumulative_return = {}
    for row in iterator:
        date, open, high, low, closing_price, adj, vol, symbol = row

        # Initialize cumulative_return for the stock_symbol if not exists
        cumulative_return.setdefault(symbol, {'value': 0.0})

        # Initialize prev_close_price for the stock_symbol if not exists
        prev_close_price.setdefault(symbol, {'value': None})

        # Check if this is a new stock_symbol and reset prev_close_price
        if prev_close_price[symbol]['value'] is None:
            prev_close_price[symbol]['value'] = closing_price

        daily_return = (closing_price / prev_close_price[symbol]['value'] - 1) if (prev_close_price[symbol]['value']
is not None and closing_price is not None) else None
        cumulative_return[symbol]['value'] += daily_return if daily_return is not None else 0.0

        yield Row(
            Date=date,
            Open=open,
            High=high,
            Low=low,
            Close=closing_price,
```

```
            Adj_Close=adj,
            Volume=vol,
            Symbol=symbol,
            DailyReturn=daily_return,
            CumulativeReturn=cumulative_return[symbol]['value']
        )

        prev_close_price[symbol]['value'] = closing_price


# Sort the RDD by Date and StockSymbol, then calculate returns using mapPartitionsWithIndex
sorted_rdd = stock_data_rdd.sortBy(lambda x: (x[7], x[0]))
indexed_rdd = sorted_rdd.mapPartitionsWithIndex(calculate_returns)

# Show the resulting RDD
indexed_rdd.collect()
```

calculate_returns function:

This function is defined to calculate daily and cumulative returns for each stock symbol within a partition of the RDD.

It uses a dictionary (prev_close_price) to keep track of the previous closing price for each stock symbol and another dictionary (cumulative_return) to store the cumulative return for each stock symbol.

For each row in the iterator, it calculates the daily return, updates the cumulative return, and yields a new Row with the calculated values.

The function is applied to each partition using mapPartitionsWithIndex.

sorted_rdd:

The RDD (stock_data_rdd) is sorted by Date and StockSymbol using the sortBy transformation.

indexed_rdd:

The mapPartitionsWithIndex transformation is used to apply the calculate_returns function to each partition of the sorted RDD. It calculates returns for each stock symbol within a partition.

indexed_rdd.collect():

Finally, collect() is called to retrieve the results as a list. This operation brings the data from the distributed RDD to the driver program, which is feasible if the size of the resulting data is not too large.

```
[Row(Date=datetime.datetime(2023, 1, 3, 0, 0), Open=130.27999877929688, High=130.89999389648438, Low=124.16999816894531, Close=125.06999969482422, Adj_Close=124.37480163574219, Volume=112117500,
Symbol='AAPL', DailyReturn=0.0, CumulativeReturn=0.0),
 Row(Date=datetime.datetime(2023, 1, 4, 0, 0), Open=126.88999938964844, High=128.66000366210938, Low=125.08000183105469, Close=126.36000061035156, Adj_Close=125.65763854980469, Volume=89113600,
Symbol='AAPL', DailyReturn=0.010314231379827232, CumulativeReturn=0.010314231379827232),
 Row(Date=datetime.datetime(2023, 1, 5, 0, 0), Open=127.12999725341797, High=127.7699966430664, Low=124.76000213623047, Close=125.0199966430664, Adj_Close=124.3250732421875, Volume=80962700,
Symbol='AAPL', DailyReturn=-0.010604653061194957, CumulativeReturn=-0.00029042168136772517),
 Row(Date=datetime.datetime(2023, 1, 6, 0, 0), Open=126.01000213623047, High=130.2899932861328, Low=124.88999938964844, Close=129.6199951171875, Adj_Close=128.89950561523438, Volume=87754700,
Symbol='AAPL', DailyReturn=0.036794101724815675, CumulativeReturn=0.03650368004344795),
 Row(Date=datetime.datetime(2023, 1, 9, 0, 0), Open=130.47000122070312, High=133.41000366210938, Low=129.88999938964844, Close=130.14999389648438, Adj_Close=129.42657470703125, Volume=70790800,
Symbol='AAPL', DailyReturn=0.004088865910060413, CumulativeReturn=0.04059254595350836),
 Row(Date=datetime.datetime(2023, 1, 10, 0, 0), Open=130.25999450683594, High=131.25999450683594, Low=128.1199951171875, Close=130.72999572753906, Adj_Close=130.0033416748047, Volume=63896200,
Symbol='AAPL', DailyReturn=0.004456410743407302, CumulativeReturn=0.045048956696915665),
 Row(Date=datetime.datetime(2023, 1, 11, 0, 0), Open=131.25, High=133.50999450683594, Low=130.4600067138672, Close=133.49000549316406, Adj_Close=132.74801635742188, Volume=69458900, Symbol='AAPL',
```

Transfering RDD data to Pyspark dataframe

```
stock_data_with_returns_df = indexed_rdd.toDF()
```

```
stock_data_with_returns_df.show()
```

indexed_rdd.toDF():

In this case, indexed_rdd is the RDD containing the calculated returns, and toDF() is applied to convert an RDD into a DataFrame in PySpark and stored in stock_data_with_returns_df.

```
+-------------------+------------------+------------------+------------------+------------------+------------------+---------+------+--------------------+--------------------+
|               Date|              Open|              High|               Low|             Close|         Adj_Close|   Volume|Symbol|         DailyReturn|    CumulativeReturn|
+-------------------+------------------+------------------+------------------+------------------+------------------+---------+------+--------------------+--------------------+
|2023-01-03 00:00:00|130.27999877929688|130.89999389648438|124.16999816894531|125.06999969482422|124.37480163574219|112117500|  AAPL|                 0.0|                 0.0|
|2023-01-04 00:00:00|126.88999938964844|128.66000366210938|125.08000183105469|126.36000061035156|125.65763854980469| 89113600|  AAPL|0.010314231379827232|0.010314231379827232|
|2023-01-05 00:00:00|127.12999725341797| 127.7699966430664|124.76000213623047| 125.0199966430664| 124.3250732421875| 80962700|  AAPL|-0.01060465306119...|-2.90421681367725...|
|2023-01-06 00:00:00|126.01000213623047| 130.2899932861328|124.88999938964844| 129.6199951171875|128.89950561523438| 87754700|  AAPL|0.036794101724815675| 0.03650368004344795|
|2023-01-09 00:00:00|130.47000122070312|133.41000366210938|129.88999938964844|130.14999389648438|129.42657470703125| 70790800|  AAPL|0.004088865910060413| 0.04059254595350836|
|2023-01-10 00:00:00|130.25999450683594|131.25999450683594| 128.1199951171875|130.72999572753906| 130.0033416748047| 63896200|  AAPL|0.004456410743407302|0.045048956696915665|
|2023-01-11 00:00:00|            131.25|133.50999450683594| 130.4600067138672|133.49000549316406|132.74801635742188| 69458900|  AAPL| 0.02111229140844828| 0.06616124810536395|
|2023-01-12 00:00:00| 133.8800048828125|134.25999450683594|131.44000244140625|133.41000366210938|   132.66845703125| 71379600|  AAPL|-5.99309519533908...| 0.06556193858583004|
|2023-01-13 00:00:00|132.02999877929688| 134.9199981689453|131.66000366210938|134.75999450683594|134.01092529296875| 57809700|  AAPL|0.010119112567793076| 0.07568105115362311|
|2023-01-17 00:00:00| 134.8300018310547| 137.2899932861328| 134.1300048828125|135.94000244140625|135.18438720703125| 63646600|  AAPL|0.008756366745847899| 0.08443741789947101|
|2023-01-18 00:00:00|136.82000732421875|138.61000061035156|135.02999877929688| 135.2100067138672|134.45846557617188| 69672800|  AAPL|-0.00536998465814...| 0.07906743324132226|
|2023-01-19 00:00:00| 134.0800018310547|            136.25|133.77000427246094|135.27000427246094| 134.5181121826172| 58280400|  AAPL|4.437360817584057...| 0.07951116932308067|
|2023-01-20 00:00:00|135.27999877929688|138.02000427246094|134.22000122070312| 137.8699951171875|137.10365295410156| 80223600|  AAPL| 0.01922074933545259| 0.09873191865853326|
|2023-01-23 00:00:00| 138.1199951171875|143.32000732421875|137.89999389648438|141.11000061035156|140.32566833496094| 81760300|  AAPL| 0.02350043960188808| 0.12223235826042134|
|2023-01-24 00:00:00|140.30999755859375|143.16000366210938| 140.3000030517578|142.52999877929688|141.73776245117188| 66435100|  AAPL|0.010063058343160014| 0.13229541660358135|
|2023-01-25 00:00:00|140.88999938964844|142.42999267578125|138.80999755859375|141.86000061035156| 141.0714874267578| 65799300|  AAPL|-0.00470075194473818| 0.12759466465884317|
|2023-01-26 00:00:00| 143.1699981689453|            144.25|141.89999389648438| 143.9600067138672|143.15980529785156| 54105100|  AAPL|0.014803370185255682| 0.14239803484409885|
|2023-01-27 00:00:00|143.16000366210938|147.22999572753906| 143.0800018310547|145.92999267578125| 145.1188507080078| 70555800|  AAPL|0.013684258613779976| 0.15602293457878883|
|2023-01-30 00:00:00| 144.9600067138672| 145.5500030517578|142.85000610351562|             143.0|142.20513916015625| 64015300|  AAPL|-0.02007870464417...|  0.1360042229937013|
|2023-01-31 00:00:00| 142.6999969482422|144.33999633789062|142.27999877929688| 144.2899932861328| 143.4879608154297| 65874500|  AAPL|0.009020932070858745| 0.14502515506456004|
+-------------------+------------------+------------------+------------------+------------------+------------------+---------+------+--------------------+--------------------+
only showing top 20 rows
```

Pivot table on Date with group by on Symbols

```
pivot_df = (
    stock_data_with_returns_df.groupBy("Symbol")
    .pivot("Date")
    .agg(F.first("DailyReturn").alias("DailyReturn"), F.first("CumulativeReturn").alias("CumulativeReturn"))
    .orderBy("Symbol")
)

pivot_df.show()
```

stock_data_with_returns_df.groupBy("Symbol"):

Groups the DataFrame stock_data_with_returns_df by the 'Symbol' column. This is the key for the pivot operation.

.pivot("Date"):

Performs a pivot operation on the grouped DataFrame, where the unique values in the 'Date' column become columns in the resulting DataFrame.

.agg(...):

Aggregates the values in each pivoted column. Here, it uses the F.first function to get the first value (since we expect one value per group) and aliases the resulting columns as "DailyReturn" and "CumulativeReturn".

.orderBy("Symbol"):

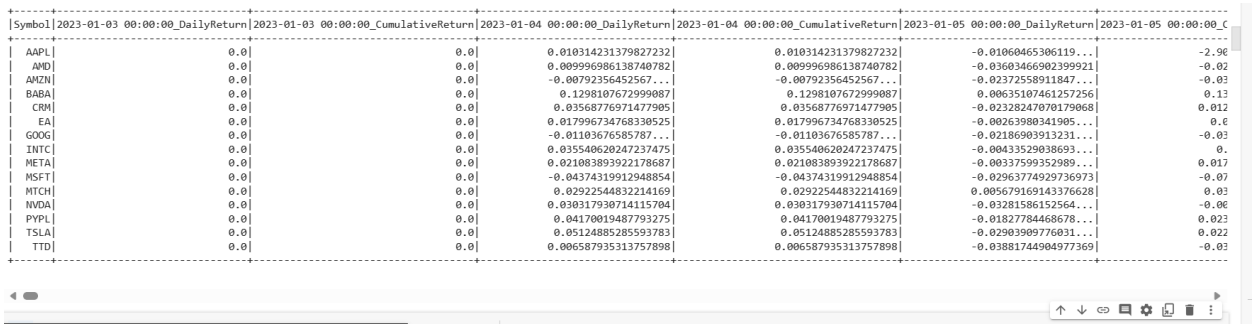
Orders the resulting DataFrame by the 'Symbol' column.

pivot_df.show():

Displays the resulting pivoted DataFrame.

The resulting pivot_df DataFrame has rows representing unique stock symbols and columns representing dates. It contains values for "DailyReturn" and "CumulativeReturn" for each date and stock symbol.

This pivot operation is used to reshape our data to have a clearer view of how values change over time for different stocks.



Symbol	2023-01-03 00:00:00_DailyReturn	2023-01-03 00:00:00_CumulativeReturn	2023-01-04 00:00:00_DailyReturn	2023-01-04 00:00:00_CumulativeReturn	2023-01-05 00:00:00_DailyReturn	2023-01-05 00:00:00_C
AAPL	0.0	0.0	0.010314231379827232	0.010314231379827232	-0.01060465306119...	-2.9
AMD	0.0	0.0	0.009996986138740782	0.009996986138740782	-0.03603466902399921	-0.02
AMZN	0.0	0.0	-0.00792356452567...	-0.00792356452567...	-0.02372558911847...	-0.03
BABA	0.0	0.0	0.1298107672999087	0.1298107672999087	0.00635107461257256	0.13
CRM	0.0	0.0	0.035687769714779 05	0.035687769714779 05	-0.02328247070179068	0.012
EA	0.0	0.0	0.017996734768330525	0.017996734768330525	-0.00263980341905...	0.0
GOOG	0.0	0.0	-0.01103676585787...	-0.01103676585787...	-0.02186903913231...	-0.03
INTC	0.0	0.0	0.035540620247237475	0.035540620247237475	-0.00433529038693...	0.
META	0.0	0.0	0.021083893922178687	0.021083893922178687	-0.00337599352989...	0.017
MSFT	0.0	0.0	-0.04374319912948854	-0.04374319912948854	-0.02963774929736973	-0.07
MTCH	0.0	0.0	0.029225448322141 69	0.029225448322141 69	0.005679169143376628	0.03
NVDA	0.0	0.0	0.030317930714115704	0.030317930714115704	-0.03281586152564...	-0.00
PYPL	0.0	0.0	0.04170019487793275	0.04170019487793275	-0.01827784468678...	0.023
TSLA	0.0	0.0	0.05124885285593783	0.05124885285593783	-0.02903909776031...	0.022
TTD	0.0	0.0	0.0065879353133757898	0.0065879353133757898	-0.03881744904977369	-0.03

Analyzing appropriate number of clusters for applying K means

```
feature_columns=pivot_df.columns
feature_columns=feature_columns[1:]
# Assemble the feature columns into a vector column
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
assembled_data = assembler.transform(pivot_df)

# Scale the features
scaler = StandardScaler(inputCol="features", outputCol="scaled_features",
withStd=True, withMean=True)
scaled_df = scaler.fit(assembled_data).transform(assembled_data)

# Run k-means clustering for a range of values of k
k_values = range(2, 11)  # Adjust the range as needed
sse = []  # Sum of squared distances

for k in k_values:
    kmeans = KMeans(k=k, seed=1)
    model = kmeans.fit(assembled_data)
    sse.append(model.summary.trainingCost)

# Plot the elbow curve
plt.plot(k_values, sse, marker='o')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Sum of Squared Distances (SSE)')
plt.title('Elbow Method for Optimal k')
plt.show()
```

VectorAssembler:

Assembles the daily return columns into a single vector column named "features" using VectorAssembler. This step is necessary for the subsequent scaling and clustering.

StandardScaler:

Scales the features using StandardScaler. This step is important in K-Means clustering, especially when features have different scales.

Specify Range of k Values:

The range of k values is defined from 2 to 10 (inclusive) using k_values.

Iterate Over k Values:

A loop is used to iterate over each k value in the specified range.

For each k value, a new instance of the KMeans algorithm is created (KMeans(k=k, seed=1), and the model is fitted to the assembled data (assembled_data).

Calculate Sum of Squared Distances (SSE):

The sum of squared distances (SSE) is obtained from the model.summary.trainingCost attribute, and the value is appended to the sse list.

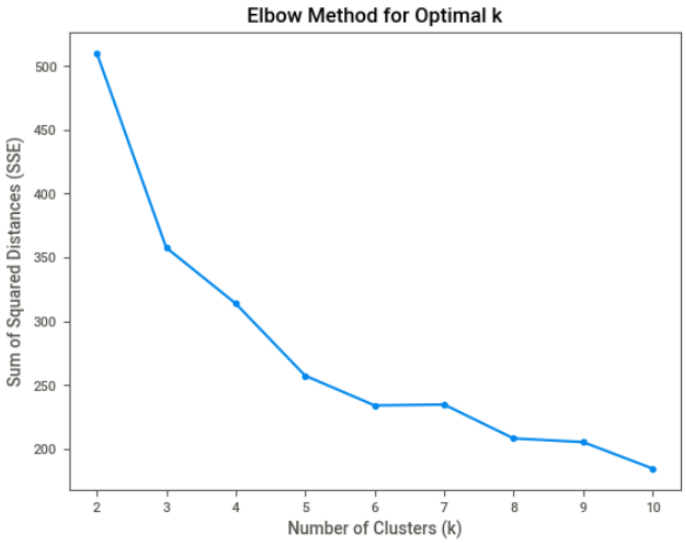
Plot the Elbow Curve:

The elbow curve is plotted using matplotlib (plt.plot()).

The x-axis represents the number of clusters (k), and the y-axis represents the corresponding SSE.

The plot helps visualize the trade-off between the number of clusters and the quality of clustering, aiming to find the "elbow" point where adding more clusters does not significantly reduce SSE.

The purpose of the elbow method is to identify the optimal number of clusters, and the user can inspect the plot to determine the point where the SSE starts to stabilize (the "elbow" point).



Applying K-mean cluster

```
# Apply K-Means clustering
num_clusters = 6  # Adjust as needed
kmeans = KMeans(k=num_clusters, seed=42, featuresCol="scaled_features",
predictionCol="cluster")
model = kmeans.fit(scaled_df)
result_df = model.transform(scaled_df)

# Display the results
result_df.select("Symbol", "cluster").show
```

KMeans:

Initializes a K-Means clustering model with the specified number of clusters (num_clusters), seed, and features and prediction columns.

model = kmeans.fit(scaled_df):

Fits the K-Means model to the scaled data.

result_df = model.transform(scaled_df):

Applies the trained K-Means model to the scaled data, adding a new column "cluster" to the DataFrame, indicating the cluster assignment for each stock.

result_df.select("Symbol", "cluster").show():

Selects and displays the "Symbol" and "cluster" columns from the resulting DataFrame. This will show the stock symbols along with their assigned clusters.

Converting to Pandas from pyspark

```
pandas_df = result_df.toPandas()
def calculate_average(lst):
    return sum(lst) / len(lst)

pandas_df['average_column'] = pandas_df['features'].apply(lambda x: calculate_average(x))
```

pandas_df = result_df.toPandas():

Converts the PySpark DataFrame result_df to a Pandas DataFrame and stored in pandas_df. This operation brings the data from the distributed Spark cluster to a single machine.

def calculate_average(lst):

Defines a custom Python function calculate_average that takes a list (lst) as input and calculates the average of its elements.

pandas_df['average_column'] = pandas_df['features'].apply(lambda x: calculate_average(x)):

Creates a new column named 'average_column' in the Pandas DataFrame pandas_df.

The column is populated by applying the custom calculate_average function to each row in the 'features' column. The apply function is used to apply a custom function to each element in the specified column.

After executing this code, the resulting Pandas DataFrame pandas_df will have a new column 'average_column' containing the average value of the elements in the 'features' column for each row.

Visual representation

```
fig = plt.figure()
ax = fig.add_subplot(111)
scatter = ax.scatter(pandas_df['average_column'],pandas_df['Symbol'],c=pandas_df['cluster'],s=50)
ax.set_title('K-Means Clustering')
ax.set_xlabel('Avg_of_daily_and_cumulative_return')
ax.set_ylabel('Symbol')
plt.colorbar(scatter)
```

import matplotlib.pyplot as plt:

Imports the Matplotlib library for creating plots.

import seaborn as sns:

Imports the Seaborn library for statistical data visualization.

fig = plt.figure():

Creates a new Matplotlib figure.

ax = fig.add_subplot(111):

Adds a subplot to the figure. The subplot is a single plot in this case.

scatter = ax.scatter(...):

Creates a scatter plot using the data from the Pandas DataFrame pandas_df.

The x-axis is 'average_column', the y-axis is 'Symbol', and the color of each point is determined by the 'cluster' column.

s=50 specifies the size of the markers.

ax.set_title('K-Means Clustering'):

Sets the title of the plot.

ax.set_xlabel('Avg_of_daily_and_cumulative_return'):

Sets the label for the x-axis.

ax.set_ylabel('Symbol'):

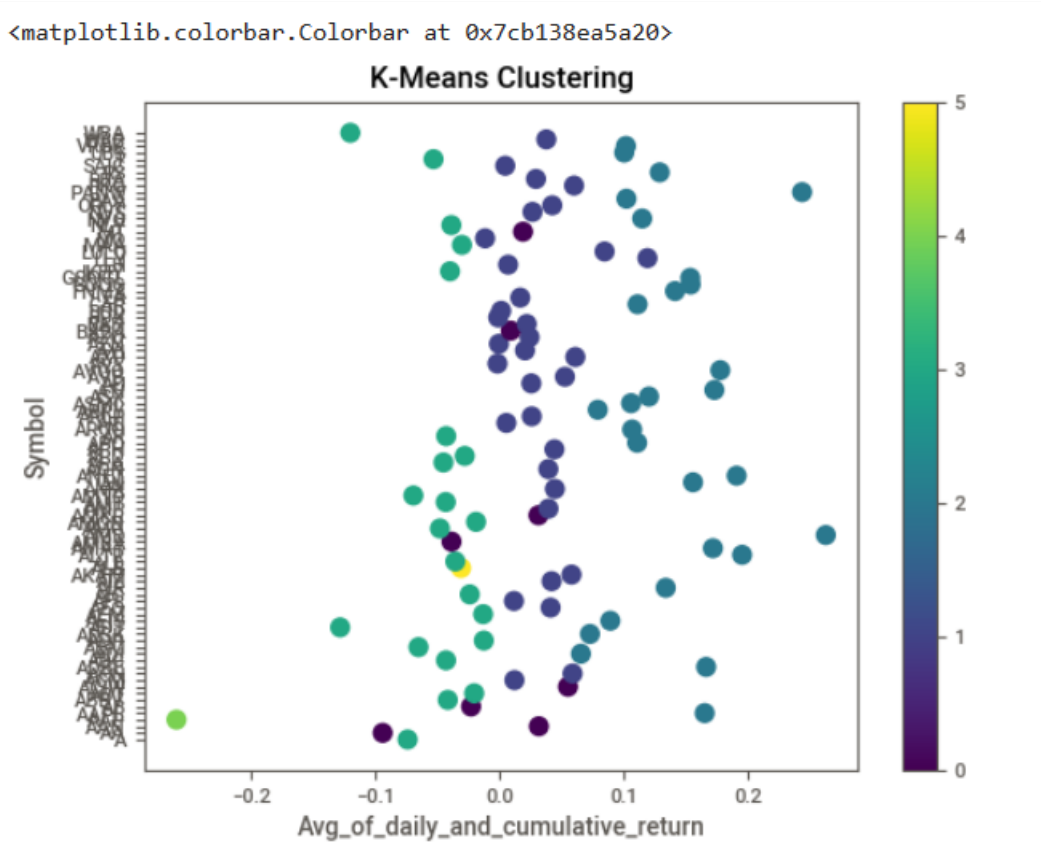
Sets the label for the y-axis.

plt.colorbar(scatter):

Adds a colorbar to the plot, representing the different clusters.

This code produces a scatter plot where each point corresponds to a stock symbol. The x-axis represents the average of daily and cumulative returns ('average_column'), the y-axis represents the stock symbols ('Symbol'), and the color of each point indicates the cluster assignment obtained from K-Means clustering.

Seaborn and Matplotlib are commonly used libraries for data visualization in Python. The resulting plot provides insights into how stocks are grouped based on their average returns. Adjustments can be made to enhance the plot further, depending on specific visualization requirements.




```
sns.catplot(data=pandas_df, x="cluster", y="Symbol")
```

import matplotlib.pyplot as plt:

Imports the Matplotlib library for creating plots.

import seaborn as sns:

Imports the Seaborn library for statistical data visualization.

sns.catplot(...):

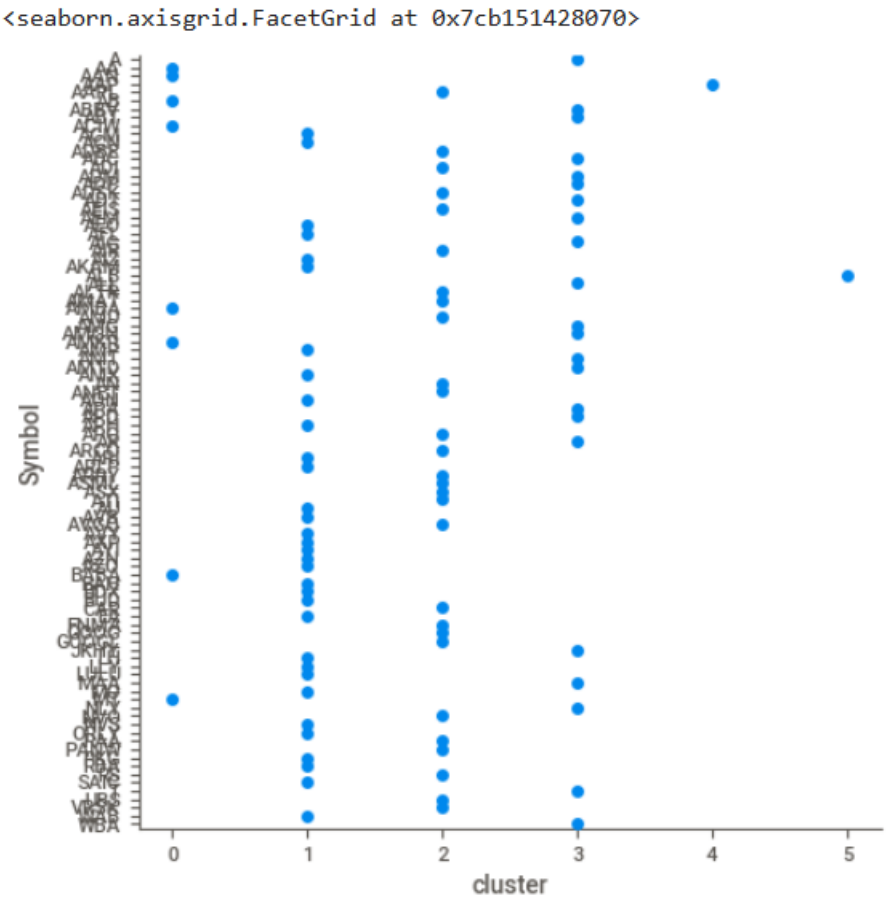
Creates a categorical plot using Seaborn's catplot function.

The data parameter is set to the Pandas DataFrame pandas_df.

x="cluster" specifies the column representing the cluster assignments on the x-axis.

y="Symbol" specifies the column representing the stock symbols on the y-axis.

This code generates a categorical plot where each point represents a stock symbol, and the points are distributed across different clusters on the x-axis. The resulting plot helps visualize how stock symbols are grouped within each cluster.



Setting up Kafka

```
!curl -sSOL https://archive.apache.org/dist/kafka/3.3.1/kafka_2.13-3.3.1.tgz
!tar -xzf kafka_2.13-3.3.1.tgz
```

The provided commands are using curl and tar to download and extract the Kafka distribution version 3.3.1.

!curl -sSOL https://archive.apache.org/dist/kafka/3.3.1/kafka_2.13-3.3.1.tgz:

curl is a command-line tool for making HTTP requests. In this case, it is used to download a file.

The flags:

-s (or --silent) makes the download silent, suppressing the progress meter.

-S (or --show-error) shows an error message if an error occurs.

-O (or --remote-name) saves the file with the same name as the remote file.

-L (or --location) follows redirects, if any.

The URL specifies the location of the Kafka distribution version 3.3.1 on the Apache archive.

!tar -xzf kafka_2.13-3.3.1.tgz:

tar is a command-line tool for archiving files.

The flags:

-x extracts files from an archive.

-z decompresses the archive using gzip.

-f specifies the file name of the archive.

The archive file being extracted is kafka_2.13-3.3.1.tgz.

After running these commands, the Kafka distribution version 3.3.1 will be downloaded and extracted in the current working directory. This process is commonly used to set up a local Kafka environment for development or testing purposes.

Creating Kafka Topics

```
!./kafka_2.13-3.3.1/bin/zookeeper-server-start.sh -daemon ./kafka_2.13-3.3.1/config/zookeeper.properties
!./kafka_2.13-3.3.1/bin/kafka-server-start.sh -daemon ./kafka_2.13-3.3.1/config/server.properties
!echo "Waiting for 10 secs until kafka and zookeeper services are up and running"
!sleep 10
```

The provided commands are starting Kafka and ZooKeeper services in the background, then waiting for a brief period to allow the services to start up. Let's break down each command:

!./kafka_2.13-3.3.1/bin/zookeeper-server-start.sh -daemon ./kafka_2.13-3.3.1/config/zookeeper.properties:

./kafka_2.13-3.3.1/bin/zookeeper-server-start.sh starts the ZooKeeper server.

-daemon runs the process in the background (as a daemon).

./kafka_2.13-3.3.1/config/zookeeper.properties specifies the configuration file for ZooKeeper.

!./kafka_2.13-3.3.1/bin/kafka-server-start.sh -daemon ./kafka_2.13-3.3.1/config/server.properties:

./kafka_2.13-3.3.1/bin/kafka-server-start.sh starts the Kafka broker/server.

-daemon runs the process in the background.

./kafka_2.13-3.3.1/config/server.properties specifies the configuration file for the Kafka broker.

!echo "Waiting for 10 secs until kafka and zookeeper services are up and running":

This command simply prints a message to the console indicating that there is a brief wait.

!sleep 10:

sleep pauses the execution for the specified duration (in seconds). In this case, it waits for 10 seconds.

The purpose of waiting for a few seconds after starting the services is to allow Kafka and ZooKeeper to fully initialize and become operational. This is a common practice to ensure that the services are up and running before proceeding with any further operations or configurations.

After running these commands, Kafka and ZooKeeper should be running in the background, and the script waits for 10 seconds to ensure that they have sufficient time to initialize before any subsequent actions.

Creating Kafka Topics

```
import os
for i in all_tickers:
  command = str('./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --
replication-factor 1 --partitions 1 --topic ') +  i
  print(command)
```

```
os.system(command)
os.system('sleep 2')
```

The provided Python script is using a loop to create Kafka topics for each stock symbol in the all_tickers list. It is executing the Kafka command-line tool kafka-topics.sh with specific parameters to create topics. Here's the breakdown of the script:

Loop through each stock symbol (i) in the all_tickers list:

The loop iterates over each stock symbol in the list.

Construct the Kafka command to create a topic:

The command variable is constructed using an f-string. It includes the necessary parameters for creating a Kafka topic.

The --topic parameter is set to the current stock symbol (i).

Print the command:

The constructed Kafka command is printed to the console for informational purposes.

Execute the command using os.system:

The os.system function is used to execute the Kafka command, creating a topic for the current stock symbol.

Pause for 2 seconds before creating the next topic:

os.system('sleep 2') introduces a 2-second pause before creating the next topic. This helps avoid potential issues with rapid topic creation.

This script is a straightforward way to automate the creation of Kafka topics for each stock symbol in the all_tickers list. The --bootstrap-server, --replication-factor, and --partitions parameters are set to specific values in the command. Adjustments can be made based on specific requirements and Kafka configuration.

```
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic AAPL
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic AMD
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic AMZN
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic BABA
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic CRM
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic EA
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic GOOG
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic INTC
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic META
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic MSFT
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic MTCH
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic NVDA
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic TSLA
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic PYPL
./kafka_2.13-3.3.1/bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --replication-factor 1 --partitions 1 --topic TTD
```

Inserting Stock name and cluster number to Kafka

```
new_data = {}
for ticker in all_tickers:
  new_data[ticker] = (pandas_df.loc[(pandas_df['Symbol'] ==
ticker)][['Symbol','cluster']]).to_dict()


def error_callback(exc):
    raise Exception('Error while sendig data to kafka: {0}'.format(str(exc)))
```

```
def write_to_kafka(topic_name, items):
  count=0
  producer = KafkaProducer(bootstrap_servers=['127.0.0.1:9092'])
  for key in items:
    producer.send(topic_name, key=key.encode('utf-8')).add_errback(error_callback)
    count+=1
  producer.flush()
  print("Wrote {0} messages into topic: {1}".format(count, topic_name))

for i in all_tickers:
  write_to_kafka(i, new_data[i])
```

Conclusion:

In the final analysis, the formation of clusters hinges on the meticulous consideration of daily returns, a metric derived from comparing the closing prices of consecutive days. This approach allows for a granular assessment of the market dynamics on a day-to-day basis, offering insights into the short-term fluctuations in stock prices.

Furthermore, the incorporation of cumulative returns into the clustering process proves to be instrumental. Cumulative returns serve as a comprehensive measure, encapsulating the total change in the investment price over a specified duration. This inclusive perspective enhances the efficacy of the K-means clustering algorithm by providing a more holistic view of the historical performance of stocks.

By combining daily and cumulative returns within the clustering framework, we achieve a great understanding of stock behavior. The resultant graph, color-coded to represent distinct clusters, serves as a visual aid for discerning patterns in stock performance. This visual representation becomes a valuable tool for investors seeking to identify clusters that not only demonstrate favorable daily returns but also exhibit sustained growth over time, as depicted by cumulative returns.

In conclusion, the integration of both daily and cumulative returns in the clustering analysis enhances the robustness of the evaluation process. The visual interpretation of the clustered data provides stakeholders with a clear and actionable perspective on the relative strengths of different stock groupings. This comprehensive approach aligns with the goal of making informed investment decisions based on a multifaceted assessment of stock performance.