HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Big Data Storage and Processing

Topic: Crypto sentiment analysis

Advisor: Dr. Tran Van Dang

Student: Huu Tuong Tu

Hanoi, Jan 2025

Chapter 1: Introduction

1.1 Problem Statement

- In the cryptocurrency market, the significant volatility of the market and individual assets poses substantial risks for investors. Both positive and negative fluctuations can be influenced by information disseminated on social media platforms. Whenever unfavorable news regarding an asset emerges, the volume of related information on social media tends to surge rapidly, triggering negative reactions from the community toward that asset.
- Therefore, there is a need for a tool to monitor trending information about these digital assets on social media platforms, particularly on Twitter, which serves as an official communication channel for many cryptocurrency platforms. This tool could help users identify the most discussed digital assets on social media within a specific time frame and evaluate the community's sentiment toward these assets.

1.2 Objectives and Scope

- Through thorough research and analysis, the objective of this study is to develop a tool that provides users with an optimal overview of social media information streams. The main tasks of the study include:
- Data Collection: Gather data on the top 100 cryptocurrencies with the best performance using APIs from analytical platforms like Coingecko. Additionally, collect pricing data for these assets from the Binance exchange.
- Social Media Data Analysis: Collect tweets containing hashtags directly related to specific assets (e.g., #BTC, #ETH, #SOL) commonly used in discussions about these assets, as well as hashtags related to cryptocurrency topics. Beyond data collection, the study aims to integrate a semantic analysis feature to evaluate whether tweets are expressing positive or negative sentiments toward the mentioned assets.
- Trend Analysis: Analyze the collected data to identify trends in social media discussions and correlate this with price fluctuations of the assets. This will

provide users with actionable insights into the relationship between social media sentiment and asset performance.

Chapter 2: Overview of the Model

2.1 Lambda Architecture

Lambda Architecture is a data processing framework designed to handle large-scale data by leveraging both batch and streaming-processing methods. In Big Data systems, there are typically two types of data:

- Batch Data: The complete historical data is already available in the system.
- Real-Time Data: Data that is continuously generated in real time.

Lambda Architecture uses batch processing to provide a comprehensive and accurate view of the batch data. Simultaneously, it utilizes real-time streaming processing to generate real-time insights. The outputs from both views can be combined to produce the final output. This architecture aims to balance latency, throughput, and fault tolerance effectively.

Lambda Architecture relies on an immutable data model, where the data source is append-only and serves as a system of record. It is designed to ingest and process events that include timestamps, which are appended to the existing data instead of overwriting it.

2.2 System Description

The system is built using the Lambda Architecture.

- Real-Time Data Integration: Real-time data is connected through Twitter's API for tweet data and Binance’s API for cryptocurrency data.
- Data Processing Pipeline: The data is processed and sent to Kafka under specific topics. From Kafka, the data is ingested into Spark for further processing.
- Batch Data Storage: The data is also stored in HDFS for batch processing and using Spark’s streaming API to handle streams and save the results into MongoDB.
- Visualization and Monitoring: A web application built with Flask is used to monitor and observe data retrieved from MongoDB.

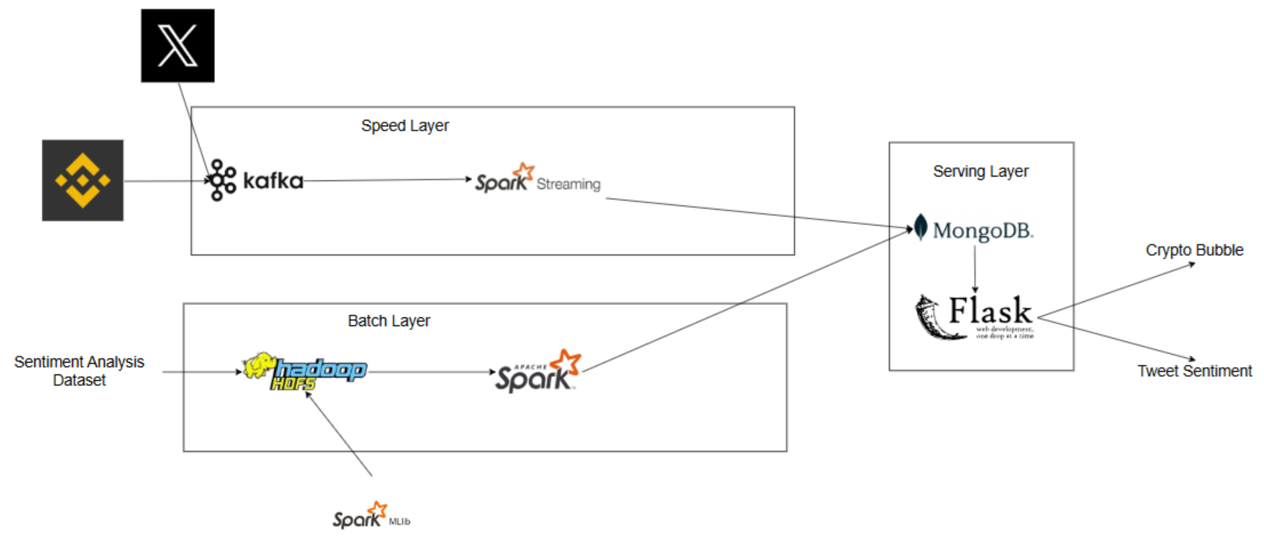


Fig 1: Lambda Architecture

Chapter 3: System Components

3.1 Kafka

Kafka is a distributed data storage platform used for real-time data processing. The main components of Kafka include:

1. Producer: A producer is an application that generates and sends data to the Kafka server. The data consists of formatted messages sent as byte arrays to Kafka.
2. Consumer: Consumers in Kafka subscribe to topics. They are identified by group names, and multiple consumers can read from the same topic. Once data is received, consumers can process it based on specific requirements by adding custom code.
3. Cluster: A Kafka cluster is a set of servers, where each set is referred to as a broker.
4. Broker: A broker is a Kafka server that acts as a bridge between message publishers and message consumers, enabling them to exchange messages.
5. Topic: Data in Kafka is transmitted via topics. Different topics are created for transmitting data to various applications.
6. Partitions: Kafka is a distributed messaging system, and Kafka servers can be set up in a cluster. When a topic receives a large number of messages simultaneously, it can be divided into multiple partitions, which are shared among Kafka servers in the cluster. Each partition is small and independent of the others.
7. Consumer Group: A consumer group is a collection of consumers that consume messages from the Kafka server. Within a consumer group, the handling of messages is shared among the consumers.
8. Zookeeper: Zookeeper is used to manage and coordinate brokers in the Kafka cluster.

3.2 HDFS

HDFS, short for Hadoop Distributed File System, is a distributed data storage system used by Hadoop. Its primary function is to provide high-performance access to data stored across Hadoop clusters. Hadoop consists of four main modules:

1. Hadoop Common: This module contains the essential Java libraries and utilities required by other Hadoop modules. These libraries provide an abstract layer for the file system and operating system and include Java scripts to initiate Hadoop.
2. Hadoop YARN: YARN (Yet Another Resource Negotiator) is a framework for managing processes and resources within clusters.
3. Hadoop Distributed File System (HDFS): HDFS is a distributed file system designed to provide high-throughput access to data, enabling efficient data mining and processing for applications.
4. Hadoop MapReduce: MapReduce is a system built on YARN for processing large datasets in parallel across a distributed environment.

3.3 Spark

Apache Spark is a unified analytics engine designed for large-scale data processing.

Key Components of Apache Spark

1. Spark Streaming: Spark Streaming is an extension of Apache Spark that enables it to handle real-time or near real-time data processing. It divides the data stream into a continuous sequence of micro-batches, which can then be processed using Apache Spark's APIs.
2. Data Integration: Spark provides a standardized interface for reading from and writing to various data stores, including JSON, HDFS, CSV, Apache Parquet, ...

Popular databases like MongoDB are also supported through specialized connectors available in the Spark Packages ecosystem.

3.4 MongoDB

MongoDB is a document-oriented NoSQL database designed for high performance, high availability, and scalability. Key Features of MongoDB:

1. Document-Based Storage: MongoDB stores data in a flexible, JSON-like format called BSON (Binary JSON), allowing for schema-less data

structures. Each document can contain arrays and nested fields, making it suitable for handling complex data structures.

2. High Scalability: MongoDB supports horizontal scaling through sharding, where data is distributed across multiple servers to handle large-scale datasets efficiently.
3. Real-Time Data Processing MongoDB excels in real-time applications due to its ability to quickly process and query unstructured or semi-structured data.
4. Integration with Apache Spark: MongoDB can be integrated with Apache Spark using the MongoDB Connector for Spark, enabling users to leverage Spark's distributed processing capabilities while accessing data stored in MongoDB.
5. Rich Query Language: MongoDB supports a powerful query language that allows for filtering, sorting, and aggregation of data using pipelines, similar to SQL.
6. High Availability: MongoDB ensures data redundancy and fault tolerance through its replica set architecture, where multiple copies of data are maintained across servers.

3.5 Flask

Flask is a lightweight web framework for Python, making it an excellent choice for beginners looking to create simple websites. Despite its minimalistic design, Flask is highly extensible and can be used to develop complex web applications. Key features of flask

Versatility: Flask provides all the tools, libraries, and technologies required to build web applications. These applications can range from blogs and wikis to web-based calendars or web3 platforms.

Microframework Design: Flask is categorized as a microframework because it does not depend on specific tools or libraries. This flexibility offers advantages: Lightweight and easy to use. Minimal dependency on frequent updates. Easier to identify and address security vulnerabilities.

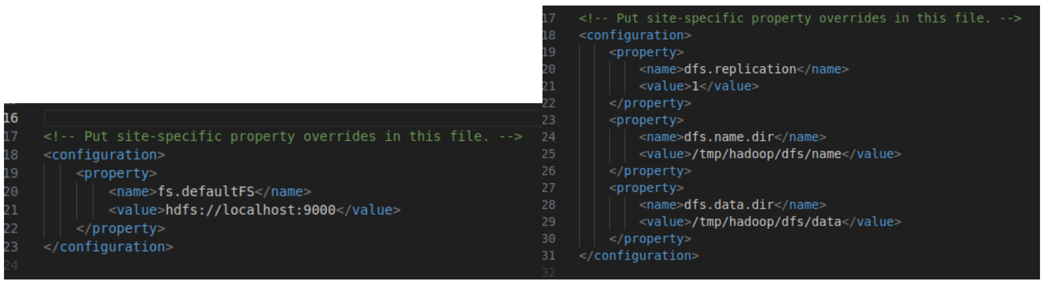
Chapter 4: Experimental Setup

In this study, the following clusters were deployed to facilitate data processing and analysis:

- Kafka Cluster: Used as a distributed messaging system for real-time data ingestion and streaming.
- Hadoop Cluster: Utilized for distributed storage and batch processing of large-scale datasets via HDFS.
- MongoDB Cluster: Implemented as a NoSQL database for storing and querying document-based data.
- Spark Cluster: Employed for distributed data processing, enabling both real-time streaming and batch analytics.

4.1 HDFS

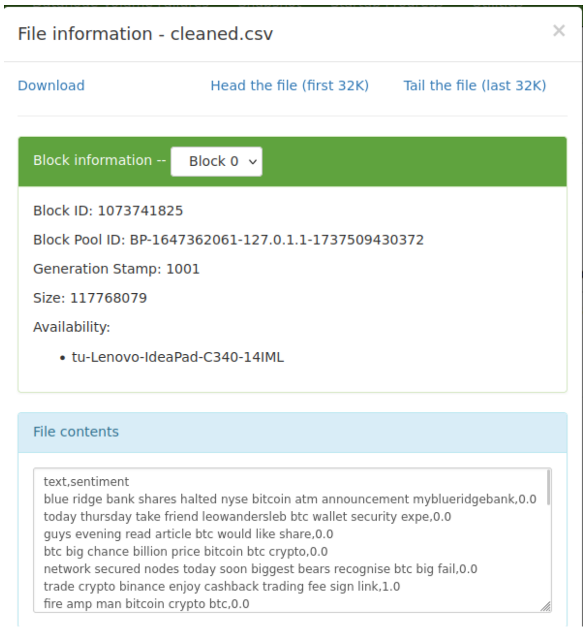
With HDFS, we used batch-processing to handle large datasets. The data is stored in directories for tweets, in CSV format, which includes the tweet content and its corresponding sentiment.



```xml
<!-- Put site-specific property overrides in this file. -->
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

```xml
<!-- Put site-specific property overrides in this file. -->
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.name.dir</name>
        <value>/tmp/hadoop/dfs/name</value>
    </property>
    <property>
        <name>dfs.data.dir</name>
        <value>/tmp/hadoop/dfs/data</value>
    </property>
</configuration>
```

HDFS Config



Block ID: 1073741825
Block Pool ID: BP-1647362061-127.0.1.1-1737509430372
Generation Stamp: 1001
Size: 117768079
Availability:

- tu-Lenovo-IdeaPad-C340-14IML

```
text,sentiment
blue ridge bank shares halted nyse bitcoin atm announcement myblueridgebank,0.0
today thursday take friend leowandersleb btc wallet security expe,0.0
guys evening read article btc would like share,0.0
btc big chance billion price bitcoin btc crypto,0.0
network secured nodes today soon biggest bears recognise btc big fail,0.0
trade crypto binance enjoy cashback trading fee sign link,1.0
fire amp man bitcoin crypto btc,0.0
```

Data store on HDFS

This data is then passed through SparkML, where we apply linear regression and TF-IDF (Term Frequency-Inverse Document Frequency) techniques for training the model. Data can read from hdfs and trained using SparkML

4.2 Kafka

For Kafka, we develope the Kafka Producer using Python libraries to fetch data from Coingecko, Binance, and Twitter.

- Twitter: Tweets are fetched and pushed into the Kafka topic called tweets_coin.
- Binance and Coingecko: Data related to cryptocurrencies from these platforms is pushed into the Kafka topic called coingecko_coin.

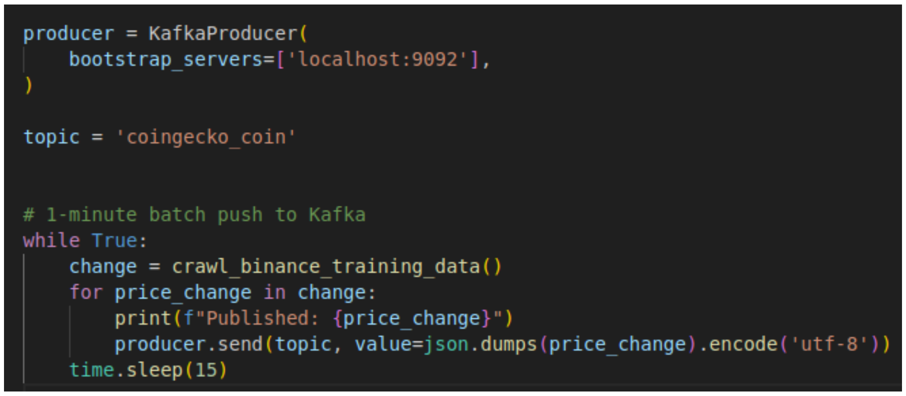
First, we need to run Zookeeper and Kafka using the default configuration:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
bin/kafka-server-start.sh config/server.properties
```

To create a topic, we can run the following command:

```
bin/kafka-topics.sh --create --topic coingecko_coin --bootstrap-server localhost:9092 --partitions 1 --
replication-factor 1
```

Next, to stream and push data to Kafka, we can write a basic script:

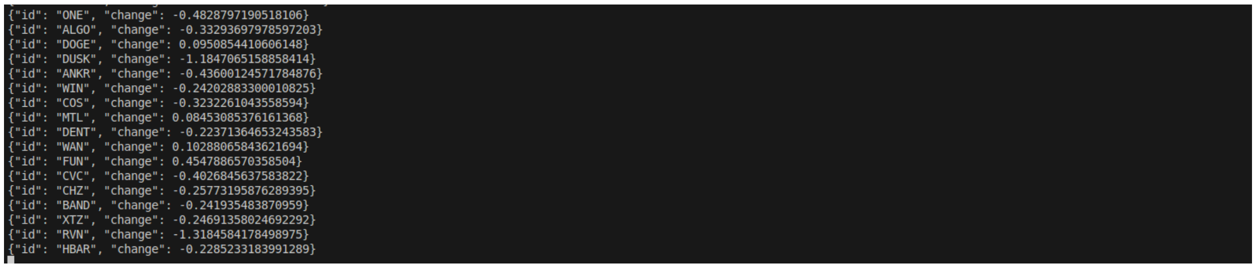


```python
producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
)

topic = 'coingecko_coin'


# 1-minute batch push to Kafka
while True:
    change = crawl_binance_training_data()
    for price_change in change:
        print(f"Published: {price_change}")
        producer.send(topic, value=json.dumps(price_change).encode('utf-8'))
    time.sleep(15)
```

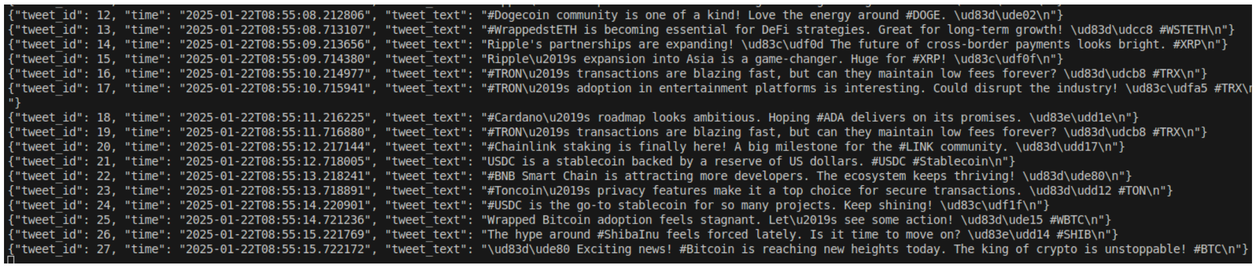
To check whether data is pushed to kafka:

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic coingecko_coin --from-beginning
```

Data on Kafka:



```
{"id": "ONE", "change": -0.4828797190518106}
{"id": "ALGO", "change": -0.33293697978597203}
{"id": "DOGE", "change": 0.0950854410606148}
{"id": "DUSK", "change": -1.1847065158858414}
{"id": "ANKR", "change": -0.43600124571784876}
{"id": "WIN", "change": -0.24202883300010825}
{"id": "COS", "change": -0.3232261043558594}
{"id": "MTL", "change": 0.08453085376161368}
{"id": "DENT", "change": -0.22371364653243583}
{"id": "WAN", "change": 0.10288065843621694}
{"id": "FUN", "change": 0.4547886570358504}
{"id": "CVC", "change": -0.4026845637583822}
{"id": "CHZ", "change": -0.25773195876289395}
{"id": "BAND", "change": -0.241935483870959}
{"id": "XTZ", "change": -0.24691358024692292}
{"id": "RVN", "change": -1.3184584178498975}
{"id": "HBAR", "change": -0.2285233183991289}
```



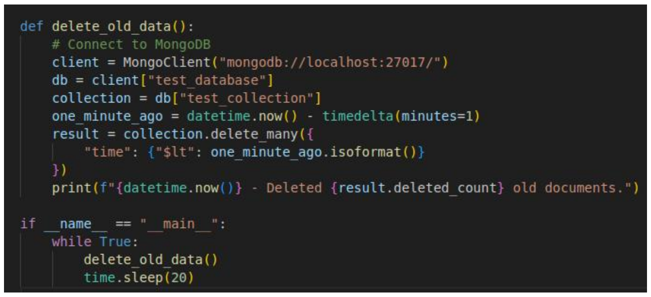
```
{"tweet_id": 12, "time": "2025-01-22T08:55:08.212806", "tweet_text": "#Dogecoin community is one of a kind! Love the energy around #DOGE. \ud83d\ude02\n"}
{"tweet_id": 13, "time": "2025-01-22T08:55:08.713107", "tweet_text": "#WrappedstETH is becoming essential for DeFi strategies. Great for long-term growth! \ud83d\udcc8 #WSTETH\n"}
{"tweet_id": 14, "time": "2025-01-22T08:55:09.213656", "tweet_text": "Ripple's partnerships are expanding! \ud83c\udf0d The future of cross-border payments looks bright. #XRP\n"}
{"tweet_id": 15, "time": "2025-01-22T08:55:09.714380", "tweet_text": "Ripple\u2019s expansion into Asia is a game-changer. Huge for #XRP! \ud83c\udf0f\n"}
{"tweet_id": 16, "time": "2025-01-22T08:55:10.214977", "tweet_text": "#TRON\u2019s transactions are blazing fast, but can they maintain low fees forever? \ud83d\udcb8 #TRX\n"}
{"tweet_id": 17, "time": "2025-01-22T08:55:10.715941", "tweet_text": "#TRON\u2019s adoption in entertainment platforms is interesting. Could disrupt the industry! \ud83c\udfa5 #TRX\
"}
{"tweet_id": 18, "time": "2025-01-22T08:55:11.216225", "tweet_text": "#Cardano\u2019s roadmap looks ambitious. Hoping #ADA delivers on its promises. \ud83e\udd1e\n"}
{"tweet_id": 19, "time": "2025-01-22T08:55:11.716880", "tweet_text": "#TRON\u2019s transactions are blazing fast, but can they maintain low fees forever? \ud83d\udcb8 #TRX\n"}
{"tweet_id": 20, "time": "2025-01-22T08:55:12.217144", "tweet_text": "#Chainlink staking is finally here! A big milestone for the #LINK community. \ud83d\udd17\n"}
{"tweet_id": 21, "time": "2025-01-22T08:55:12.718005", "tweet_text": "USDC is a stablecoin backed by a reserve of US dollars. #USDC #Stablecoin\n"}
{"tweet_id": 22, "time": "2025-01-22T08:55:13.218241", "tweet_text": "#BNB Smart Chain is attracting more developers. The ecosystem keeps thriving! \ud83d\ude80\n"}
{"tweet_id": 23, "time": "2025-01-22T08:55:13.718891", "tweet_text": "#Toncoin\u2019s privacy features make it a top choice for secure transactions. \ud83d\udd12 #TON\n"}
{"tweet_id": 24, "time": "2025-01-22T08:55:14.220901", "tweet_text": "#USDC is the go-to stablecoin for so many projects. Keep shining! \ud83c\udf1f\n"}
{"tweet_id": 25, "time": "2025-01-22T08:55:14.721236", "tweet_text": "Wrapped Bitcoin adoption feels stagnant. Let\u2019s see some action! \ud83d\ude15 #WBTC\n"}
{"tweet_id": 26, "time": "2025-01-22T08:55:15.221769", "tweet_text": "The hype around #ShibaInu feels forced lately. Is it time to move on? \ud83e\udd14 #SHIB\n"}
{"tweet_id": 27, "time": "2025-01-22T08:55:15.722172", "tweet_text": "\ud83d\ude80 Exciting news! #Bitcoin is reaching new heights today. The king of crypto is unstoppable! #BTC\n"}
```

For the topic coingecko_coin, data saved on Kafka is in the format:
{"id": token_name, "change": percentage_change_in_5_minutes}
For the topic tweets_coin, data saved on Kafka is in the format:
{"tweet_id": id_of_tweet, "time": tweet_time, "tweet_text": content}

4.3 MongoDB

MongoDB will be divided into two collections to store the streaming data. After Spark reads the tweet data from Kafka and pushes it to MongoDB, a job will run automatically to delete any data older than 1 minute. This ensures that the application always serves sentiment data with the most up-to-date information, providing users with real-time insights.

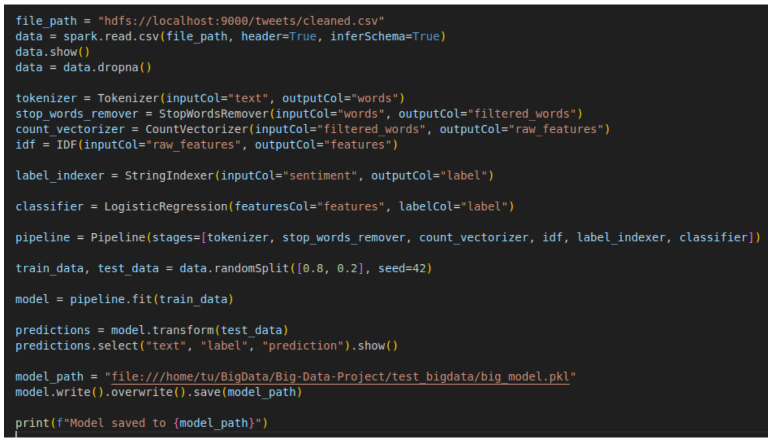
For coin price data, the old data will be overwritten with the latest price updates to ensure that only the most current coin price information is available, reflecting the most recent price changes. This approach keeps the data fresh and accurate for analysis and decision-making.



```python
def delete_old_data():
    # Connect to MongoDB
    client = MongoClient("mongodb://localhost:27017/")
    db = client["test_database"]
    collection = db["test_collection"]
    one_minute_ago = datetime.now() - timedelta(minutes=1)
    result = collection.delete_many({
        "time": {"$lt": one_minute_ago.isoformat()}
    })
    print(f"{datetime.now()} - Deleted {result.deleted_count} old documents.")

if __name__ == "__main__":
    while True:
        delete_old_data()
        time.sleep(20)
```

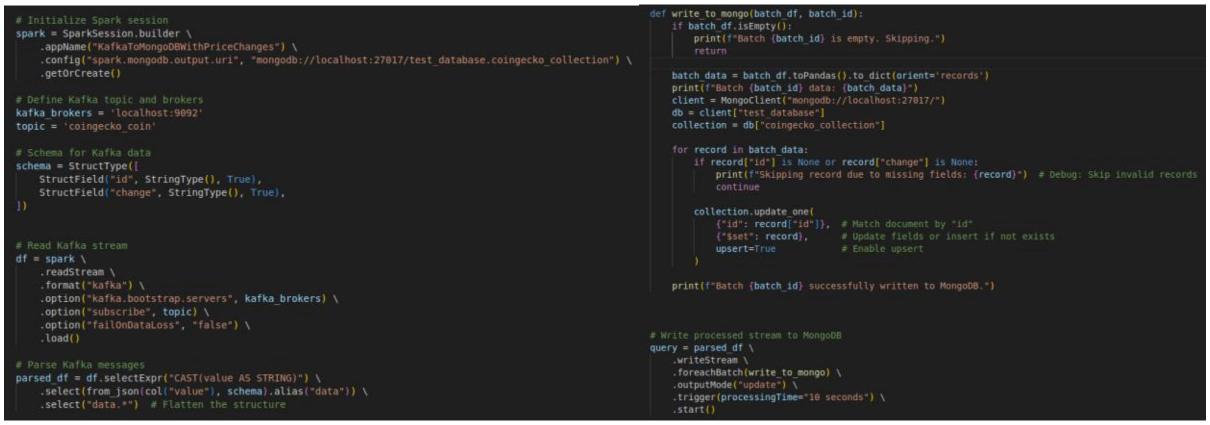
4.4 Spark

First, we need to train a model using data from hdfs using sparkml:



```python
file_path = "hdfs://localhost:9000/tweets/cleaned.csv"
data = spark.read.csv(file_path, header=True, inferSchema=True)
data.show()
data = data.dropna()

tokenizer = Tokenizer(inputCol="text", outputCol="words")
stop_words_remover = StopWordsRemover(inputCol="words", outputCol="filtered_words")
count_vectorizer = CountVectorizer(inputCol="filtered_words", outputCol="raw_features")
idf = IDF(inputCol="raw_features", outputCol="features")

label_indexer = StringIndexer(inputCol="sentiment", outputCol="label")

classifier = LogisticRegression(featuresCol="features", labelCol="label")

pipeline = Pipeline(stages=[tokenizer, stop_words_remover, count_vectorizer, idf, label_indexer, classifier])

train_data, test_data = data.randomSplit([0.8, 0.2], seed=42)

model = pipeline.fit(train_data)

predictions = model.transform(test_data)
predictions.select("text", "label", "prediction").show()

model_path = "file:///home/tu/BigData/Big-Data-Project/test_bigdata/big_model.pkl"
model.write().overwrite().save(model_path)

print(f"Model saved to {model_path}")
```

Now, we have a pretrained sentiment analysis model. Our goal is to stream data from Kafka to MongoDB and process sentiment analysis it using this model. To read Kafka streaming data and write it to MongoDB, we use two connectors provided by the Spark package. For streaming read from Kafka we use *org.apache.spark:spark-sql-kafka-0-10_2.12:3.0.1*, and for streaming push data to MongoDB, we use *mongo-spark-connector_2.12-2.4.1.jar*. We wrote a basic script to stream data from Kafka to MongoDB. In this script, we use **trigger(processingTime="10 seconds")**, which streams data from Kafka to MongoDB every 10 seconds. This configuration is necessary due to limited RAM. If sufficient memory is available and we want **true streaming** (continuous data flow without batching), we can easily achieve this by removing the trigger configuration.



```python
# Initialize Spark session
spark = SparkSession.builder \
    .appName("KafkaToMongoDBWithPriceChanges") \
    .config("spark.mongodb.output.uri", "mongodb://localhost:27017/test_database.coingecko_collection") \
    .getOrCreate()

# Define Kafka topic and brokers
kafka_brokers = 'localhost:9092'
topic = 'coingecko_coin'

# Schema for Kafka data
schema = StructType([
    StructField("id", StringType(), True),
    StructField("change", StringType(), True),
])


# Read Kafka stream
df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", kafka_brokers) \
    .option("subscribe", topic) \
    .option("failOnDataLoss", "false") \
    .load()

# Parse Kafka messages
parsed_df = df.selectExpr("CAST(value AS STRING)") \
    .select(from_json(col("value"), schema).alias("data")) \
    .select("data.*")  # Flatten the structure
```

```python
def write_to_mongo(batch_df, batch_id):
    if batch_df.isEmpty():
        print(f"Batch {batch_id} is empty. Skipping.")
        return

    batch_data = batch_df.toPandas().to_dict(orient='records')
    print(f"Batch {batch_id} data: {batch_data}")
    client = MongoClient("mongodb://localhost:27017/")
    db = client["test_database"]
    collection = db["coingecko_collection"]

    for record in batch_data:
        if record["id"] is None or record["change"] is None:
            print(f"Skipping record due to missing fields: {record}")  # Debug: Skip invalid records
            continue

        collection.update_one(
            {"id": record["id"]},  # Match document by "id"
            {"$set": record},      # Update fields or insert if not exists
            upsert=True            # Enable upsert
        )

    print(f"Batch {batch_id} successfully written to MongoDB.")


# Write processed stream to MongoDB
query = parsed_df \
    .writeStream \
    .foreachBatch(write_to_mongo) \
    .outputMode("update") \
    .trigger(processingTime="10 seconds") \
    .start()
```

To stream read data from Kafka and write to MongoDB, we run following this command:

```
/home/tu/BigData/Big-Data-Project/test_bigdata/spark-3.5.3-bin-hadoop3/bin/spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.0.1 --jars /home/tu/BigData/Big-Data-Project/test_bigdata/mongo-spark-connector_2.12-2.4.1.jar /home/tu/BigData/Big-Data-Project/test_bigdata/spark/binance_kafka_to_mongo_spark.py
```

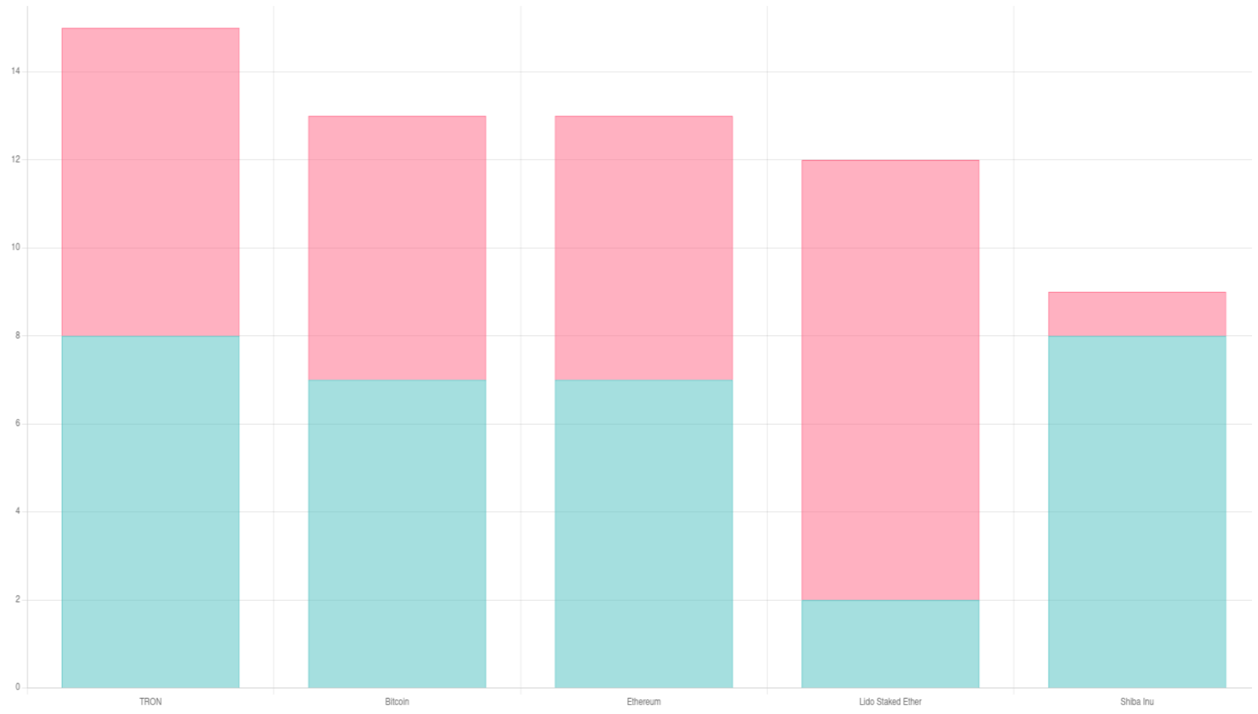
For tweet data, after fetching the tweets from Kafka, these tweets are passed through a sentiment analysis model trained using linear regression. The sentiment is then attached to the tweet, and the updated data is pushed to MongoDB.
For coin price data, since we have already calculated the price difference before pushing the data to Kafka, we only need to stream and fetch the coin price difference from Kafka and then push it to MongoDB.

4.5 Serving Layer

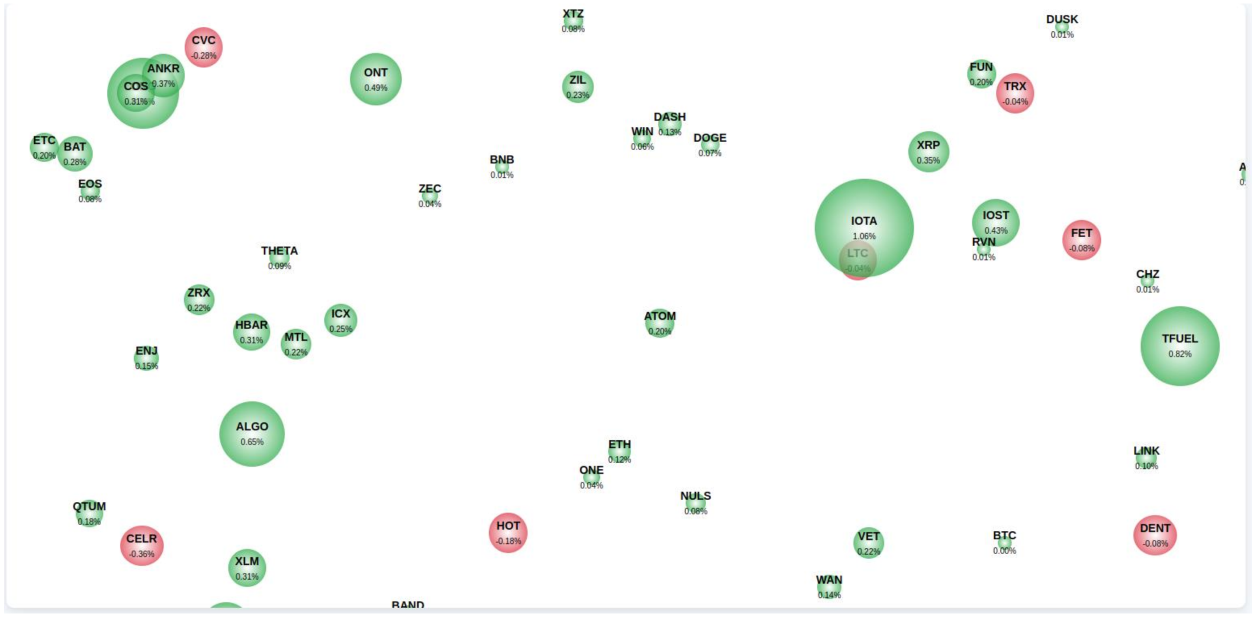
In this project, we chose the Flask framework for building the back end, and HTML with JavaScript for front-end development. Our website consists of two main views:

1. **Coin Sentiment Analysis**: This view displays sentiment scores (negative and positive) for coins that have been most mentioned on Twitter.
2. **Crypto Bubble**: This view tracks and visualizes fluctuations in the values of various coins, highlighting those that have experienced the most significant price changes.

Flask serves as the backend server, enabling it to read data from MongoDB and provide the processed information to the frontend for visualization. These views will provide users with valuable insights into the current trends and sentiment in the cryptocurrency market.



Sentiment Analysis



Bubble Crypto

Chapter 5: Conclusion and Future Directions

Through this project, our team has researched, studied, and built a basic data pipeline using technologies such as Kafka, HDFS, Spark, MongoDB, and Flask.

In the future, we aim to further expand our data pipeline, allowing for more diverse data analysis methods and supporting additional features on our platform. For instance, we envision enabling users to utilize the data in the database for building and testing alpha models and other advanced analytics tools.

We are excited about the potential to continuously develop this pipeline, incorporating new functionalities to better serve user needs and enhance the system's capabilities.