University of Pisa
Department of Computer Science

Tweet Sentiment Analysis using Kafka and Spark : Elon Musk use case

Yohannis Kifle Telila

Student ID : **621821**

Distributed Enabling Platforms - 534AA
Academic year 2021/2022

Contents

Chapter 1

Introduction

1.1 Problem description

In recent years, social media websites have turned into a source of a wide range of information. This is due to the nature of social media, which allows users to generate real-time data, such as commenting about their perspectives on a range of topics, debate current events, and express emotions for topics or trending debates. Twitter, one of a social media platform which is estimated to have about 396.5 million monthly active users and these users create approximately 867 million tweets a day[1]. These figures reflect Twitter's worldwide reach and potential effect globally.

Sentiment analysis (also known as opinion mining) is a natural language processing (NLP) approach for determining if statement is positive, negative, or neutral. Sentiment analysis can be performed for different purposes. For instance, Sentiment analysis is essential tool to detect and understand customer feelings[2]. Companies that use these tool can understand how customers feels about their product and improve their product or service. However, the sheer amount and speed of Twitter's streaming data poses a significant technical hurdle. To process these amount of data we need a system that is capable of horizontally scalable. Furthermore, the volume and the velocity of these tweets could vary in time. For such unpredictable and flexible environment we need a system that is fast, fault-tolerant and scalable as the need for computation goes up.

This motivated me to use Apache spark to do the real-time sentiment analysis on tweet analysis on **Elon Musk** being a big of him.

1.2 Background and State of the Art

Every day, Twitter processes over 400 billion events in real time and generates data on a petabyte (PB) scale. These data are ingested from a variety of event sources, which are created on a variety of platforms and storage systems, including Hadoop,

Vertica, Manhattan distributed databases, Kafka, Twitter Eventbus, GCS, BigQuery, and PubSub [3].

The old architecture of Twitter is a lambda architecture[1] composed of batch and real-time processing pipe-line. The batch components uses HDFS to store the data and scalding[2] which is MapReduce framework developed by their team to process data. For real-time data source they have used kafka topics. They have been using storm[3] to process these data in real-time until they switched to Heron framework developed by their own team to match their growth [4].

The latest architecture of Twitter is built on top of their own data centers and Google cloud platform. The architecture is depicted in Fig 1.1.

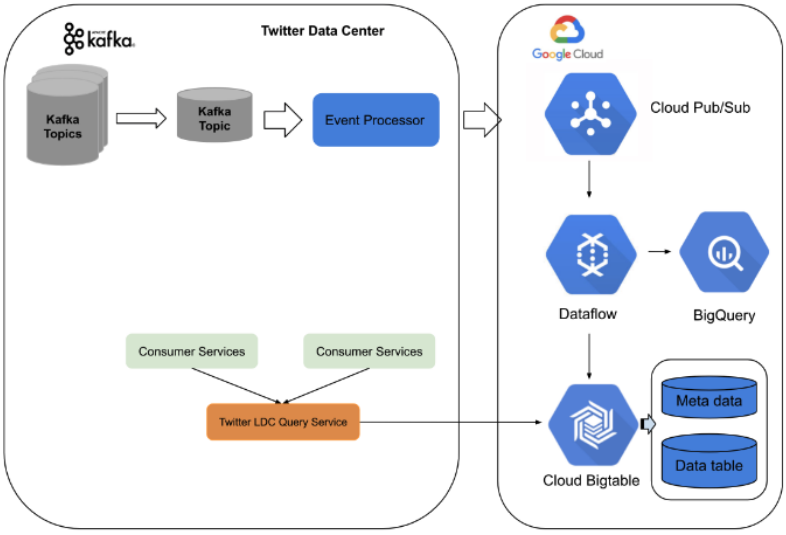


Figure 1.1: Twitter architecture - Kafka and Data Flow

On the twitter data-center the events are preprocessed(transformations and remapping of fields) and will be sent to kafka-topics. On the Google cloud these events are processed and aggrigated in real-time using Twitter internal framework built on top of DataFlow[4]. Lastly, the aggregated counts with query keys are written to Bigtable.

[1]Lambda architecture is a way of processing massive data that provides access to batch and stream-processing methods with a hybrid approach.

[2]https://twitter.github.io/scalding/

[3]https://storm.apache.org/

[4]https://cloud.google.com/dataflow

Chapter 2

Conceptual Design

I propose a simple architecture composed of Apache kafka, Apache spark cluster that can process sentiment of tweets in real-time and dashboard server to visualize results in real-time. The architecture depicted in Fig 2.1 shows the conceptual design of my architecture and help us to see the individual components of this project and how they are connected together. The main components are:

- Kafka producer
- Apache Kafka cluster
- Spark cluster(kafka consumer) and
- Dashboard web application.

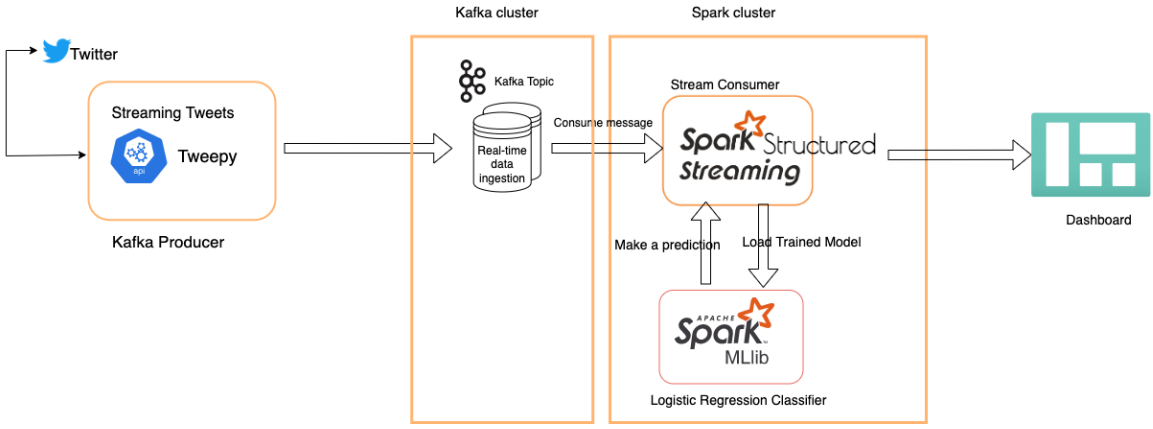


Figure 2.1: Overview of Application Architecture

The kafka producer is program that fetches tweets in real-time and publishes to a kafka topic in a kafka cluster. Apache Spark structured streaming will be used to create the processing layer since it offers excellent computational performance in a distributed architecture. This layer will handle all of the basic data processing, such as transformation, aggregation, and sentiment analysis in real-time. We can simply scale horizontally as additional processing power is required because Spark is a distributed computing environment.

2.1 Apache Kafka Cluster

The kafka cluster decouples the communication between stream of tweets(kafka producer) and Spark structured streaming(kafka consumer) allowing us to create multiple producers from different data sources and increase our consumers to process the streams faster.

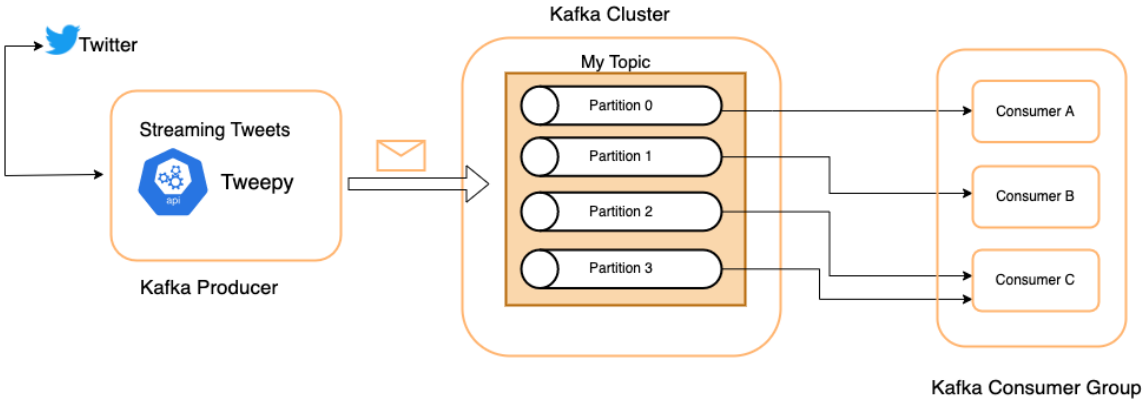


Figure 2.2: Overview of Kafka workflow diagram

The kafka produce request message will be sent from the kafka producer and these messages are stored in the kafka brokers from which the consumer(Spark structured streaming application) will be able to consume the messages and produce the final results.

2.2 Apache Spark Cluster

The spark cluster, depicted on Fig:2.1 acts as a Kafka consumer which run a spark streaming application. The cluster contains driver program and multiple worker nodes. The application is connected to a kafka server and subscribed to on of the "Topics" and continuously receive stream of messages over batches. These batches are processed through different transformations including prediction of sentiments and aggregation to get the total sentiment counts(sentiment prediction count). I have also created another stream that filters out tweets with negative sentiments(filter Negative tweets).

The result from the two streams are sent to the web server via HTTP POST request to the dashboard server. The dashboard displays total number of tweets processed, total number of tweets with a negative, positive and neutral sentiment and sample of tweets with negative sentiments.

Chapter 3

Tools exploited

When designing this system, the main consideration that I took was to design a system that is scalable, fast and that can handle large amounts of stream of data reliably in real-time and efficiently in a fault tolerable fashion. To achieve this design goal I exploited different distributed enabling tools. Lets see them in detail.

3.1 Apache Kafka

Apache Kafka is a distributed publish-subscribe messaging system. Because stream sizes and velocity changes over time, using Kafka provides the extra benefit of horizontal scaling. Additionally, data segmentation, low latency, and the capacity to manage a large number of different streams are all aspects that makes Kafka a strong fit for the system that I wanted to design.

Kafka acts as a data pipeline in my application, providing streaming data. The Kafka producer is built using in python, connects to the Twitter Data API and begins publishing data to Kafka cluster.

3.2 Apache Spark: Structured Streaming

Apache Spark structured streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. Stream processing applications nowadays requires much faster processing of the data and Spark's resilient distributed dataset (RDD) allows it to speed up computation and transparently keep data in memory and only persist data to a disk what's necessary or to optimize computation.

I have used structured streaming to continuously consume messages from the kafka broker and analyse the sentiment of the tweets(messages). Whenever my streaming application starts it loads the model which is persisted on a disk and make the pre-

diction on that.

Spark structured streaming does come by default with the package that allows us to read streams from kafka source. For that I have added *org.apache.spark:spark-sql-kafka-0-10_2.12:3.2.1* package to read the streams.

3.3 Apache Spark: Machine Learning Library(MLib)

MLlib is Apache Spark's scalable machine learning library that is easy to use, scalable and contains many algorithms and utilities[1] that makes it easier to combine multiple algorithms into a single pipeline, or workflow. Machine learning classification, regression, clustering and many other algorithms are included in this library. For my purpose since I will be doing sentiment classification I used Logistic regression Estimator to train my sentiment classifier model. MLlib also comes with Estimators which are, as the name suggests are used to evaluate estimator performance metrics over test data.

This library also comes with different utilities that can be used as feature transformation such as standardization, normalization, StopWordRemovers and Tokenizers. I have utilized some of the feature transformers to pre-process the tweet before making the prediction which will be discussed in the implementation section.

Model selection also known as hyperparameter tuning which is a techniques used to find the best model parameters for the given model was performed using tools provided in MLlib[2]. Implementation detail will be provided in the next chapter.

After training, the model has been persisted to a disk to avoid retraining the model all over again. The structured streaming application will be able to load the model and start making the prediction without doing the training again.

3.4 John Snow Labs' Spark NLP

John Snow Labs' Spark NLP is an open source text processing library for Python, Java, and Scala built on top of Apache Spark. It provides production-grade, scalable, and trainable versions of the latest research in natural language processing[3].

I wanted to experiment with this tool because it provides state-of the art feature transformers at scale by extending Apache Spark natively.

[1]https://spark.apache.org/mllib/
[2]https://spark.apache.org/docs/latest/ml-tuning.html
[3]https://www.johnsnowlabs.com/spark-nlp

Chapter 4

Implementation

In this chapter, I will discuss implementation details and results I achieved from training a classifier model.

4.1 Data collection and pre-processing

For training my model, I collected twitter sentiment dataset from 3 different data sources[1]. I collected 182,521 records from these three datasets. The dataset contains two fields, first field is the *text*, representing twets and the second column is *polarity* representing sentiment of the tweet. The *polarity* column has three different values representing the three sentiments. **0** - Neutral sentiment, **1** - Positive Sentiment and **2** - Negative Sentiment. After removing rows with missing values, I ended up with 182329 records. Sentiment distribution is depicted at Table: 4.1

Polarity	Amount
0	59111
1	75482
2	47736

Table 4.1: Polarity distribution

I removed usernames and links from the text since these information cannot express any sentiment.

4.2 Spark MLlib - Training Model

To train my model chose Logistic regression classification which is a simple classification algorithm used to predict the probability of a target variable. I experimented with two models and the model has been persisted to disk after training.

[1]url of the datasets: https://www.kaggle.com/code/kritanjalijain/twitter-sentiment-analysis-lstm

4.2.1 Logistic regression

I applied three transformations to convert the text(tweets) to a vector so that it can be used to train my model. The stages are described below.

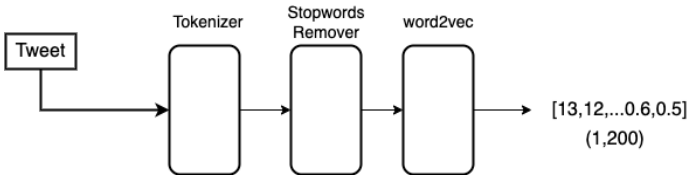


Figure 4.1: Tweet vectorization stages

I used word2vec pre-trained model to convert the tweets to a vector of dimension 200. The dataset has been devided to 85% for training and 15% for testing. Additional 20% of th training set has been reserved for validation set. I performed model selection over the following hyperparameters.

Parameters	Search space
Tolerance(tol)	$[0.01, 0.001, 0.0004]$
Threshold(threshold)	$[0.4, 0.5, 0.6]$
Regularization parameter(regParam)	$[0.008, 0.0008, 0.00008]$
Maximum iteration(maxIter)	$[1000, 5000, 10000]$

Table 4.2: Model selection parameters

I get accuracy of 63% with best parameters (regParam=0.00008,tol=0.0004,threshold=0.4, maxIter=10000).

	precision	recall	f1-score	support
0	0.64	0.61	0.62	9032
1	0.63	0.73	0.68	11262
2	0.59	0.48	0.53	7001
accuracy			0.63	27295
macro avg	0.62	0.61	0.61	27295
weighted avg	0.62	0.63	0.62	27295

Figure 4.2: Classification result

4.2.2 Logistic regression with JohnSnow NLP

JohnSnow NLP provides many state of the art feature transformers. I added some of the transformers to build new feature vectorizer.

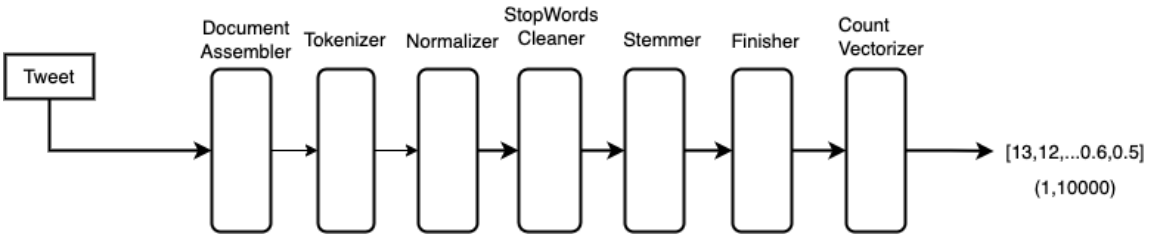


Figure 4.3: Tweet vectorization stages

The documentAssemble prepares the data into a format that is processable by Spark NLP. Then, the tokenizer tokenizes the data and Normalizer is annotator that cleans out tokens. StopWordsCleaner removes the stop words from the tokens and Stemmer converts the token to their root words. Finisher converts annotation results into a format that easier to use. CountVectorizer converts the tokens to a vector. I selected dictionary size of 10000, so the final output will be a vector of size 10000. I performed hyper-parameter search over the parameter specified at Table: 4.2. These all transformations are based on spark and can be scaled as our data set gets bigger.

I get accuracy of 82% with best parameter (regParam=0.008,tol=0.001,threshold=0.4, maxIter=1000) which significantly better that the previous model. Since the main goal of this project is not getting higher classification accuracy I will use this model to analysis how my spark application perform.

	precision	recall	f1-score	support
0	0.80	0.87	0.83	9032
1	0.85	0.82	0.83	11262
2	0.79	0.74	0.76	7001
accuracy			0.82	27295
macro avg	0.81	0.81	0.81	27295
weighted avg	0.82	0.82	0.81	27295

Figure 4.4: Classification result

4.3 Kafka consumer : Structured streaming

My kafka consumer is spark structured streaming application. First, it loads the model and start reading the stream from kafka topic. I created two streams, one outputs the total number of counts grouped by sentiments to the sink(sentiment prediction count) and the second outputs negative tweets to the sink(Filter Negative tweets) with *update* output mode. The sink processes each batch and post the results to the dashboard server. The total number of tweets and filtered negative tweets are sent to the dashboard server for visualization.

4.4 Kafka producer : Tweepy

I utilized Tweepy[2], an easy-to-use Python library for accessing the Twitter API that lets us to integrate the Twitter API into our Kafka Producer, to stream data from the Twitter API to Kafka topic. The Twitter requests fetched from from Twitter API are packaged into a kafka producer record with specified topic(in my case, **Elon Musk**). Then the producer serializes and partitions the message and will be will be sent in a batch to a partition leader. Ultimately the partition leader will send the message the "kafka consumer". I used *json_serializer* as a value serializer and I didn't use key for the topic.

Twitter allows to fetch maximum of 100 results per single request with **Essential** access level. I collected *date created*, *text*(tweet text),*tweet id*, *username* and *profile url* informations. It took on average 0.65 seconds to process one request and publish to kafka.

4.5 Dashboard server

The dashboard server is a web app built using flask. On the dash we can monitor total number of tweets processed, positive tweets, negative tweets and neutral tweets. Additionally, recent negative tweets are continuously displayed on the dashboard.

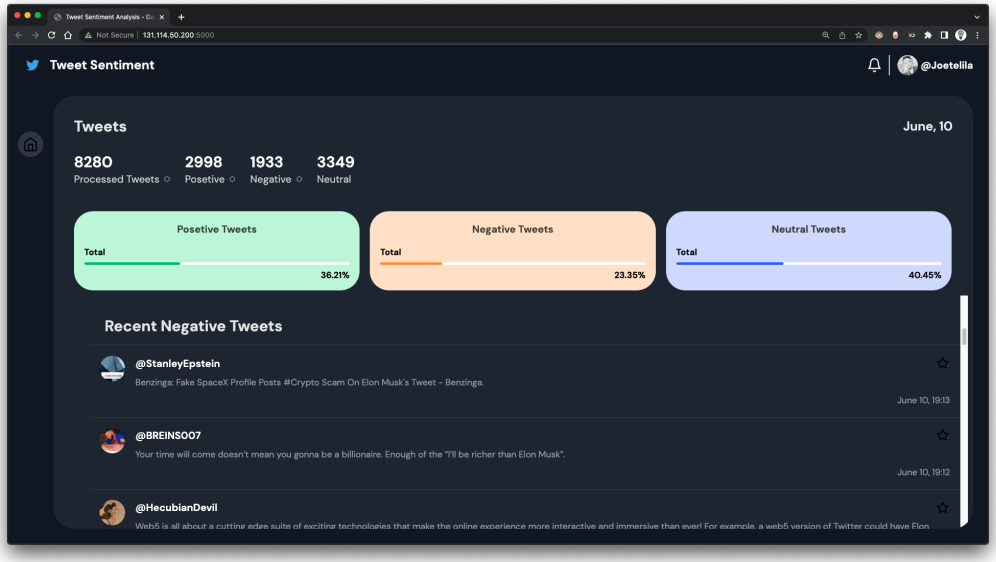


Figure 4.5: Classification result

[2]https://github.com/tweepy/tweepy

Chapter 5

Experiments

5.1 Environment setup and measurements

All the experiments are performed on a remote cluster machines provided to us. The driver node is Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz with 4 cores and has 32GB of RAM. The worker nodes are also Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz but with 2 cores and 8GB of memory. In total we have 8 worker nodes and 1 driver node.

I performed the experiment by changing the number of worker nodes and analysing 3 key measurements. These measurements are collected from Streaming Query Statistics page of spark ui [5].

- `Input rate(rec/sec):` Represents rate at which data is arriving.
- `Process rate(rec/sec):` Rate at which spark is processing data.
- `Batch duration(msec):` The duration at which each batch is being processed.

Prior to conducting the experiment, Memory and CPU consumption data for the driver and all of the worker nodes were gathered and are shown in the following tables 5.15.2. I conducted the experiments on the weekends with the expectation that the cluster would be less overloaded by then. As seen on the table all of the resources were idle, as I had expected. The "load average" over 1, 5 and 15 minutes provides a comparative indication of how long it will take for things to be completed. Higher values indicate an issue or an overworked machine.

	Memory			%CPU	%CPU Load average		
	Total(KB)	Free(KB)	Used(KB)	idle	1min	5min	15min
	32946488	31845960	1100528	99.9	0	0.01	0.05
%Mem	100	96.6597	3.3403				

Table 5.1: Driver node Memory and CPU status

Node 01

	Memory			%CPU	%CPU Load average		
	Total(KB)	Free(KB)	Used(KB)	idle	1min	5min	15min
	8174160	8007664	166496	99.8	0.07	0.03	0.04
%Mem	100	97.9631	2.0792				

Node 02

	Memory			%CPU	%CPU Load average		
	Total(KB)	Free(KB)	Used(KB)	idle	1min	5min	15min
	8174160	8009396	164764	99.8	0.00	0.01	0.04
%Mem	100	97.9843	2.0157				

Node 03

	Memory			%CPU	%CPU Load average		
	Total(KB)	Free(KB)	Used(KB)	idle	1min	5min	15min
	8174160	8009396	164764	99.8	0.00	0.01	0.04
%Mem	100	97.9572	2.0157				

Node 04

	Memory			%CPU	%CPU Load average		
	Total(KB)	Free(KB)	Used(KB)	idle	1min	5min	15min
	8174152	8000536	173616	99.8	0.01	0.01	0.04
%Mem	100	97.8760	2.1701				

Node 05

	Memory			%CPU	%CPU Load average		
	Total(KB)	Free(KB)	Used(KB)	idle	1min	5min	15min
	8174152	8008536	165616	100	0.00	0.01	0.03
%Mem	100	97.9739	2.0680				

Node 06

	Memory			%CPU	%CPU Load average		
	Total(KB)	Free(KB)	Used(KB)	idle	1min	5min	15min
	8174160	8008688	165472	99.8	0.03	0.03	0.04
%Mem	100	97.9757	2.0662				

Node 05

	Memory			%CPU	%CPU Load average		
	Total(KB)	Free(KB)	Used(KB)	idle	1min	5min	15min
	8174160	8006648	167512	100	0.03	0.03	0.05
%Mem	100	97.9507	2.0922				

Node 06

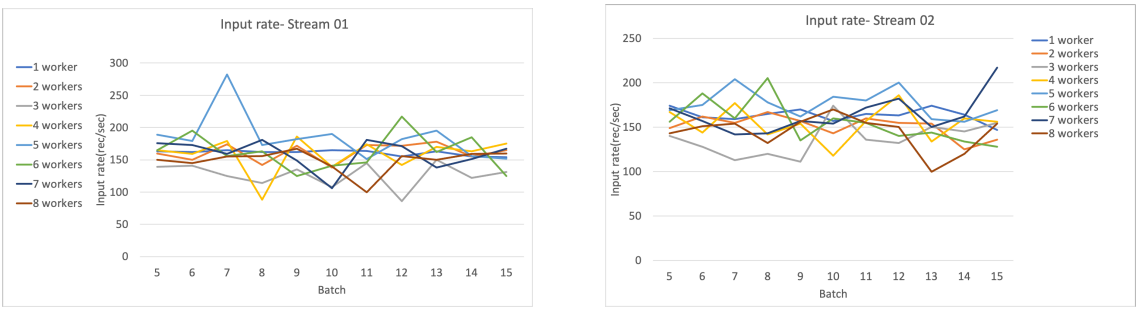
	Memory			%CPU	%CPU Load average		
	Total(KB)	Free(KB)	Used(KB)	idle	1min	5min	15min
	8174160	8008539	165621	100	0.01	0.02	0.03
%Mem	100	97.9738	2.0681				

Table 5.2: Worker nodes Memory and CPU status

5.2 Results

In my experiment, I discovered that the data gathered in the first five batches are mostly outliers and are not included in the experiment. Starting with the fifth batch, all important performance indicators were gathered.

I performed the structured streaming query as fast as feasible to handle Apache Kafka data. The streaming task will process all accessible data in Kafka for each batch. The **Input rate** 5.1 is in the range of **150**±50 records/sec in both **stream 01** and **stream 02** varying number of worker nodes; which is quite reasonable given the fact that the kafka is producing at the rate **150 records/sec**.

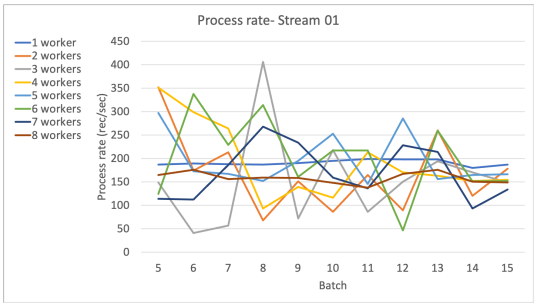


(a) Stream 01 (b) Stream 02

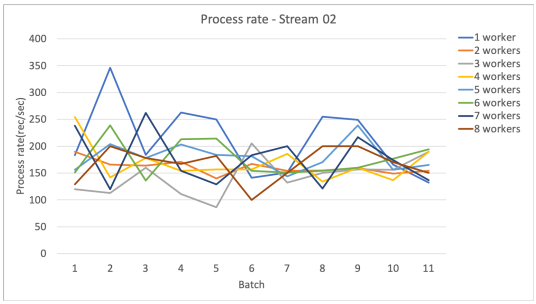
Figure 5.1: Input Rate

From the **process rate** 5.2 we can understand that spark is able to process at the rate at which data is arriving in both Streams and varying number of worker nodes.

This could be a good indication that the **input rate** is lower than what the spark stream can handle.



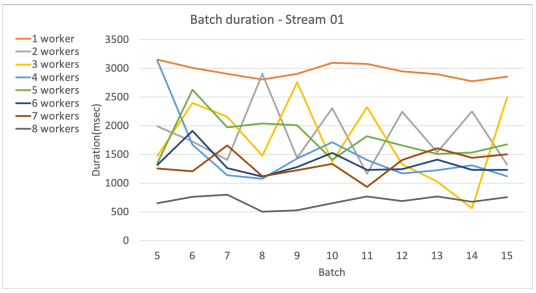
(a) Stream 01



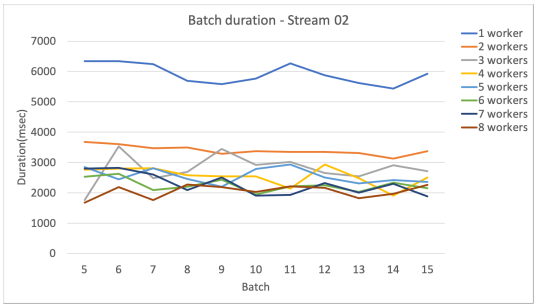
(b) Stream 02

Figure 5.2: Process Rate

The main difference we can notice in both scenario is in **batch duration** 5.3. Intuitively, if the stream processing capacity of cluster is not enough, the **batch duration** or latency will rise. This can be observed on both **Stream 01** and **Stream 02** using **1 worker** node where on average it took 3 sec and 6 sec to process a single batch. We can improve this time by adding more resources/hardware to the cluster. We can observe this on the same figure the decrease in **batch duration** as we increase the number of worker nodes.



(a) Stream 01



(b) Stream 02

Figure 5.3: Batch Duration

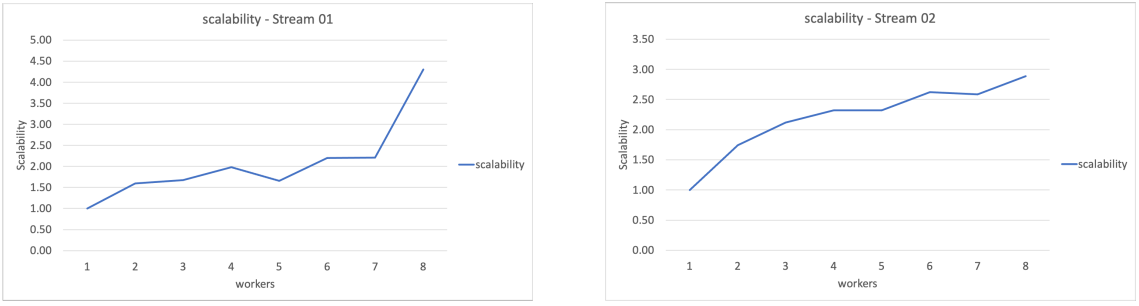
5.2.1 Scalability

Scalability is the ration between the parallel execution time with parallelism degree equal to 1 and the parallel execution time with degree equal to nw (number of worker node used). Mathematically scalability can be expressed as:

$$S_c(nw) = \frac{T_{par}(1)}{T_{par}(nw)} \tag{5.1}$$

The value we would get from calculating scalability will tell us how much we can improve when we add more computing resources. Ideally we would expect a linear

increase in scalability as we add more workers to our cluster but its impossible to achieve this result because of the overhead that will be introduced.



(a) Stream 01 (b) Stream 02

Figure 5.4: Scalability

Result shown on Figure 5.4 give us the scalability we can achieve as we add more resources to our cluster. On **Stream 01** we achieved scalability of **4.5x** with 8 worker nodes whereas scalability of **3x** on **Stream 01**. The theoretical outcome could not be realized, since these ideal estimates indicate a maximum scalability that we can reach. However, the cost of communication and synchronization further degrades performance.

5.3 Conclusion

This report has described an approach to perform sentiment analysis based on Apache Spark, Kafka, and MLlib, scalable frameworkds. Using spark cluster I was able process realtime tweets efficiently. The most important fact is that this system can adapt to the changing amount of streams. If the input rate gets bigger, we can add more consumer to the topic without making significant change to the code or if the rate gets smaller, we can remove some workers or decrease the resource of our cluster so that we don't waste processing power.

Finally, I was able to understand the principles of distributed systems as a result of this project. Implementing this application to analyze real-time sentiment taught me how to utilize Spark to address large data and stream problems.

5.4 Folder structure and how to run the code

The project folder **Sentiment-analysis-using-kafka-spark-final** contains three subfolders. **dashboardServer** folder contains the implementation for web dashboard server. **data** folder contains the data used for training. **pipeline_lr_js_model**(logistic regression with transformation from johnsnowlab) and **pipeline_lr_model** are the

two models persisted on a disk. **Report** folder contains this report files.

To run the project first, start kafka zookeeper and kafka server.

```
#!/bin/bash
$ zookeeper-server-start.sh config/zookeeper.properties
```

```
#!/bin/bash
$ kafka-server-start.sh config/server.properties
```

Then, create a kafka topic with 5 partitions.

```
#!/bin/bash
$ kafka-topics.sh --create --bootstrap-server 131.114.50.200:9092 \
--partitions 5 --topic ElonMusk
```

Next, launch kafka producer.

```
#!/bin/bash
$ python3 /kafka-script/tweet_producer.py
```

Now we are ready to submit our spark job.

```
#!/bin/bash
$ spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.2.1,\
                 com.johnsnowlabs.nlp:spark-nlp_2.12:3.4.4  \
                --master spark://131.114.50.200:7079 \
                 kafka-script/tweet_consumer.py
```

Finally, launch web dashboard server using below command inside *dashboardServer* folder.

```
#!/bin/bash
$ flask run -h 131.114.50.200
```

Bibliography

[1] *Twitter statistics 2022: How many people use twitter?* https://thesmallbusinessblog.net/twitter-statistics/, Accessed: 2022-06-06.

[2] G. Hu, P. Bhargava, S. Fuhrmann, S. Ellinger, and N. Spasojevic, "Analyzing users' sentiment towards popular consumer industries and brands on twitter," *CoRR*, vol. abs/1709.07434, 2017. arXiv: 1709.07434. [Online]. Available: http://arxiv.org/abs/1709.07434.

[3] *Processing billions of events in real time at twitter*, https://blog.twitter.com/engineering/en_us/topics/infrastructure/2021/processing-billions-of-events-in-real-time-at-twitter-, Accessed: 2022-06-06.

[4] S. Kulkarni *et al.*, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 239–250, ISBN: 9781450327589. DOI: 10.1145/2723372.2742788. [Online]. Available: https://doi.org/10.1145/2723372.2742788.

[5] *A look at the new structured streaming ui in apache spark 3.0*, https://databricks.com/blog/2020/07/29/a-look-at-the-new-structured-streaming-ui-in-apache-spark-3-0.html, Accessed: 2022-06-06.