Sentiment Analysis of Twitter data through Kafka and Spark

Università degli Studi di Napoli Federico II
Authors: Colantuono Antonio, Cozzolino Vittorio, Mazzocchi Mattia

January 9, 2026

Contents

Chapter 1

Problem Description

1.1 Context of the Problem

Sentiment Analysis, often called opinion mining, is a branch of Natural Language Processing (NLP) that uses algorithms to identify and categorize the emotional tone of a piece of text.

At its core, the goal is to determine the polarity of a statement. Advanced models can go further, detecting specific emotions like anger, joy, or urgency. This means being able to convert a stream of global conversations into a more structured, real-time map of public opinion. This is expecially true on Twitter where it is possible to gather a lot of data on how people feel about specific events, product launches, etc.

The analysis of Twitter data deals with several problems due to the nature of data itself such as the presence of sarcasm that makes harder understanding the sentiment of a tweet as well as the presence of words that NLP models are not aware of and therefore can be misunderstood

1.2 Technical Requirements

The sentiment analysis problem must be addressed respecting specific constraints:

- Kafka being the stream processor. Tweets can be treated as real-time events and therefore there is the need for a system that is able to ingest great amount of events in a small window of time without crashing.
- Spark implementation with Python in order to implement NLP models that allow to perform a sentiment analysis.

1.3 Objectives

The goal of this project is to provide an end-to-end data pipeline that is able to **ingests** data simulating tweets being fetched from Twitter and being managed using Kafka, **process** the data provided by Kafka topic through Spark in order to perform a sentiment analysis and finally **compare** different possible configuration to provide a better solution.

Chapter 2

System Architecture

2.1 Overview

According to the technical requirments there are going to be the main tools used to develop the pipeline: **Kafka** and **Spark**.

Kafka
Kafka is a distributed event store and stream-processing platform whose aim is to provide high-performance pipelines with a high-throughput and low latency to deal with real-time data.

In a more practical way, Kafka allows the creation of **Topics** towards which producers can send their data. Topics can later be divided into **partitions** that enables several consumers to read data at once instead of only one at a time. In order to have more reliability, the partitions can be **replicated**. Both partitions and replicas implementation requires the Kafka cluster to be composed by more than a single broker.

Spark
Spark is an analytics engine for large-scale data processing. Spark was developed in response to the limitations of MapReduce allowing to perform computations **in-memory** instead of disk-based operations and introducing a more efficient data processing model.

2.2 High-level Pipeline

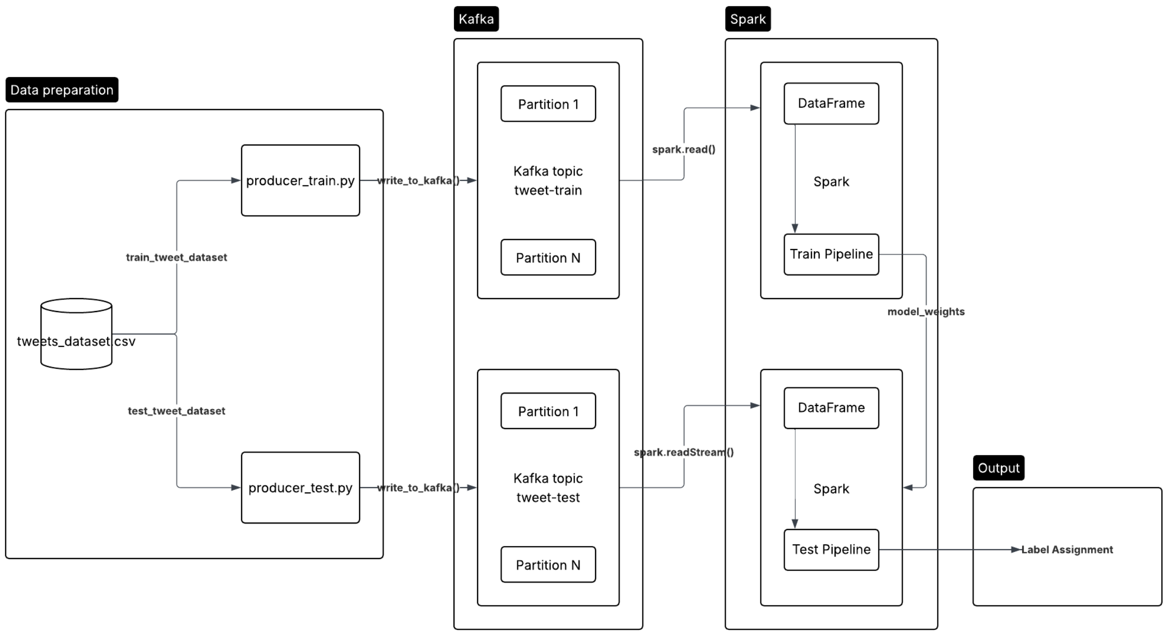


Figure 2.1: Architecture Pipeline

The pipeline is composed by three main stages:

- **Data preparation**: data is being wrangled and then split into training and test parts in order to be sent to Kafka topics. Tweet-train receives tweets in batch while Tweet-test receives tweets at a certain rate in order to simulate tweets being provided by a hypothetical API.
- **Kafka**: data is ingested and managed thanks to the created Kafka cluster
- **Spark**: training data is used to train a Spark NLP model. The weights generated by this model are then provided to another instance of the same model whose goal is to perform classification on the unseen data of the test data.

2.3 Implementation

The proposed pipeline has been implemented on a Linux based OS where several dependencies were mandatory for the pipeline to work:

- Java 17.0.17
- Python 3.10

After the environment dependencies have been provided then it is required to setup Kafka.

The Kafka version used is the **2.13-4.1.1**. This version of Kafka also allows for the cluster of server to be set up through the use of KRaft, an alternative metadata management solution to the Zookeeper.

2.3.1 Kafka Cluster Setup

In order to have a Kafka cluster running, three steps need to be performed:

1. Generate the configuration file for the server(s)
2. Format the server(s) configuration file
3. Start the server(s)

1. Server(s) configuration files generation
The following code shows a 2-servers scenario.

Kafka package provides a server default layout that can be customized based on the required setup. It is required to make a copy of such file for as many servers as needed.

```
cp ${KAFKA_DIR}/config/server.properties ${KAFKA_DIR}/config/server
    -1.properties
```

Set the server ID.

```
sed -i "s/^node.id=.*/node.id=1" ${KAFKA_DIR}/config/server-1.
    properties
```

Set the listeners.

```
sed -i "s|^listeners=.*|listeners=PLAINTEXT://localhost:9092,
    CONTROLLER://localhost:9093|" ${KAFKA_DIR}/config/server-1.
    properties
sed -i "s|^advertised.listeners=.*|advertised.listeners=PLAINTEXT
    ://localhost:9092|" ${KAFKA_DIR}/config/server-1.properties
```

Set the logs folder.

```
sed -i "s|^log.dirs=.*|log.dirs=${KAFKA_DIR}/logs/server-1-logs|" $
    {KAFKA_DIR}/config/server-1.properties
```

Set the controller quorum voters.

```
echo controller.quorum.voter="1@localhost:9093,2@localhost:9095" >>
     ${KAFKA_DIR}/config/server-1.properties
```

2. Server(s) configuration files formatting
Setting up a cluster of server requires the generation of a UUID that is shared among all the servers in the cluster

```
KAFKA_CLUSTER_ID = ${KAFKA_DIR}/bin/kafka-storage.sh random-uuid
```

Such UUID is then used to format the server(s) configuration files

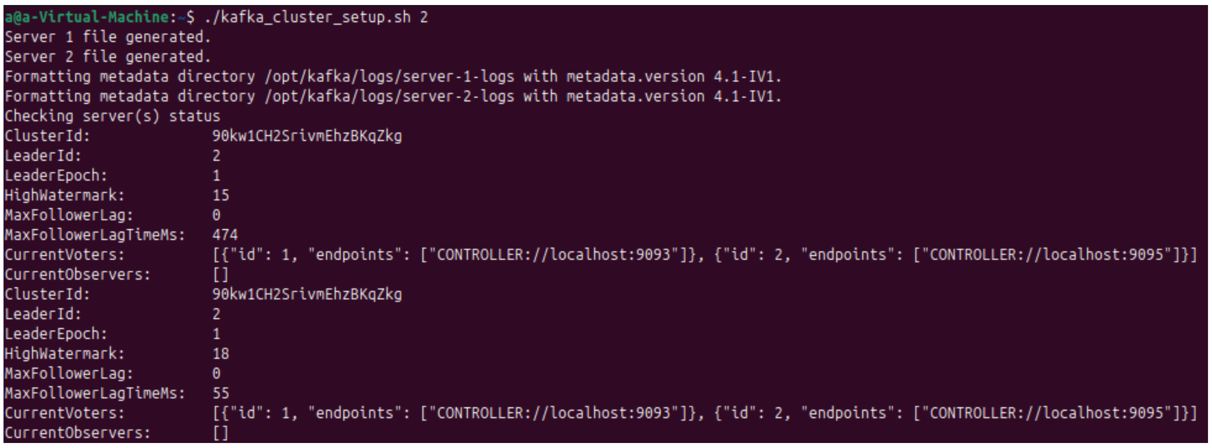
```
${KAFKA_DIR}/bin/kafka-storage.sh format -t ${KAFKA_CLUSTER_ID} -c
    ${KAFKA_DIR}/config/server-1.properties
```

3. Start the server(s)

```
${KAFKA_DIR}/bin/kafka-server-start.sh -daemon ${KAFKA_DIR}/config/
    server-1.properties
```

Verify that the server(s) are working properly

```
${KAFKA_DIR}/bin/kafka-metadata-quorum.sh --bootstrap-controller
    localhost:9093 describe --status
```



```
a@a-Virtual-Machine:~$ ./kafka_cluster_setup.sh 2
Server 1 file generated.
Server 2 file generated.
Formatting metadata directory /opt/kafka/logs/server-1-logs with metadata.version 4.1-IV1.
Formatting metadata directory /opt/kafka/logs/server-2-logs with metadata.version 4.1-IV1.
Checking server(s) status
ClusterId:              90kw1CH2SrivmEhzBKqZkg
LeaderId:               2
LeaderEpoch:            1
HighWatermark:          15
MaxFollowerLag:         0
MaxFollowerLagTimeMs:   474
CurrentVoters:          [{"id": 1, "endpoints": ["CONTROLLER://localhost:9093"]}, {"id": 2, "endpoints": ["CONTROLLER://localhost:9095"]}]
CurrentObservers:       []
ClusterId:              90kw1CH2SrivmEhzBKqZkg
LeaderId:               2
LeaderEpoch:            1
HighWatermark:          18
MaxFollowerLag:         0
MaxFollowerLagTimeMs:   55
CurrentVoters:          [{"id": 1, "endpoints": ["CONTROLLER://localhost:9093"]}, {"id": 2, "endpoints": ["CONTROLLER://localhost:9095"]}]
CurrentObservers:       []
```

Figure 2.2: Kafka cluster setup showcase

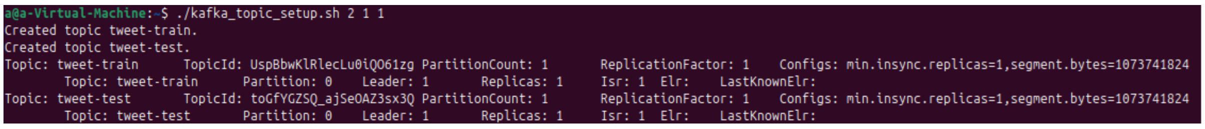
2.3.2 Kafka Topic setup

It is possible to create topics after the cluster of servers has been generated. For the purpose of this project two topics are going to be created: **tweet-train** and **tweet-test**.

```
${KAFKA_DIR}/bin/kafka-topics.sh --create --topic tweet-train --
    bootstrap-server localhost:9092,localhost:9094
${KAFKA_DIR}/bin/kafka-topics.sh --create --topic tweet-test --
    bootstrap-server localhost:9092,localhost:9094
```

Verify that the topic are working properly

```
${KAFKA_DIR}/bin/kafka-topics.sh --describe --topic tweet-train --
    bootstrap-server localhost:9092,localhost:9094
${KAFKA_DIR}/bin/kafka-topics.sh --describe --topic tweet-test --
    bootstrap-server localhost:9092,localhost:9094
```



```
a@a-Virtual-Machine:~$ ./kafka_topic_setup.sh 2 1 1
Created topic tweet-train.
Created topic tweet-test.
Topic: tweet-train      TopicId: UspBbwKlRlecLu0iQO61zg PartitionCount: 1       ReplicationFactor: 1    Configs: min.insync.replicas=1,segment.bytes=1073741824
        Topic: tweet-train      Partition: 0    Leader: 1       Replicas: 1     Isr: 1  Elr:    LastKnownElr:
Topic: tweet-test       TopicId: toGfYGZSQ_ajSeOAZ3sx3Q PartitionCount: 1       ReplicationFactor: 1    Configs: min.insync.replicas=1,segment.bytes=1073741824
        Topic: tweet-test       Partition: 0    Leader: 1       Replicas: 1     Isr: 1  Elr:    LastKnownElr:
```

Figure 2.3: Kafka topic setup showcase

2.3.3 Providing training data

This part of the pipeline has been implemented through the use of Python. The following packages have been used in order to connect to the Kafka topic:

- kafka-python 2.3.0

and to wrangle data:

- pandas 2.3.3
- pyarrow 22.0.0
- scikit-learn 1.8.0

Now that the topics are available to be used by the producer it is possible to provide the training part of the starting dataset of tweets to the *tweet-train* topic.

```
def write_to_kafka(topic_name, messages, labels):
  producer = KafkaProducer(bootstrap_servers=BOOTSTRAP_SERVERS)
    for data in range(10000):
          producer.send(topic_name,
                key=labels.iat[data].encode('utf-8'),
                value=messages.iat[data,0].encode('utf-8'))

write_to_kafka("tweet-train", x_train_df, y_train_df)
```

2.3.4 Spark processing

The Spark processing has been implemented through the use of Python. In order to allow Spark to work properly, the following parameters have been used when launching the *spark-submit* command line:

```
spark-submit \
--driver-memory 24g \
--executor-memory 16g \
--packaged "org.apache.spark:spark-sql-kafka-0-10_2.12:3.4.4,com.
   johnsnowlabs.nlp:spark-nlp_2.12:6.3.0" \
--files "log4j.propeties" \
--conf "spark.driver.extraJavaOptions=-Dlog4j.configuration=file:
   log4j.properties" \
--conf "spark.executor.extraJavaOptions=-Dlog4j.configuration=file:
   log4j.properties" \
--conf "spark.serializer=org.apache.spark.serializer.KryoSerializer
   " \
--conf "spark.kryoserializer.buffer.max=2000M"
```

and the following environment and Python packages have been used:

- pyspark 3.4.4
- java 17.0.17
- scala 2.12.17
- spark-nlp 6.3.0

Data reading

Data is ready to be read from the *tweet-train* topic from Spark. In order to execute any action through Spark, a SparkSession is required.

```
spark = SparkSession \
        .builder \
        .appName("Twitter") \
        .getOrCreate()
```

It is then possible to read data from the Kafka *tweet-train* topic

```
df = spark \
        .read \
        .format("kafka") \
        .option("kafka.bootstrap.servers", BOOTSTRAP_SERVERS) \
        .option("subscribe", "tweet-train") \
        .load()
```

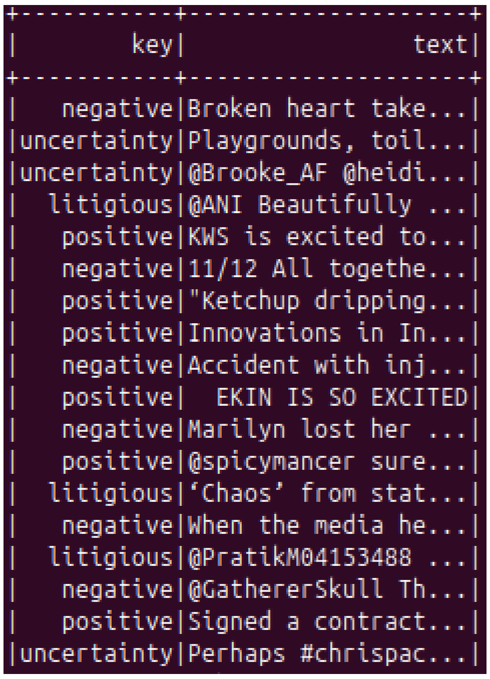
and requires to be casted from binary to string

```
df = df.selectExpr("CAST(key AS STRING)", "CAST(value as STRING) as
    text")
```

Model training

Spark allows the use of **transformers** to apply transformation to data and not executing them before an **action** on said data has been called. The transfomers are implemented through the **Annotators** that end up creating the data processing pipeline.

```
document_assembler = DocumentAssembler() \
...

embeddings = UniversalSentenceEncoder.pretrained(name="tfhub_use",
   lang="en") \
...

classifier = ClassifierDLApproach() \
...

finisher = Finisher() \
...

indexer = StringIndexer() \
...

pipeline = Pipeline() \
.setStages([
document_assembler,
embeddings,
dl,
finisher,
indexer])
```

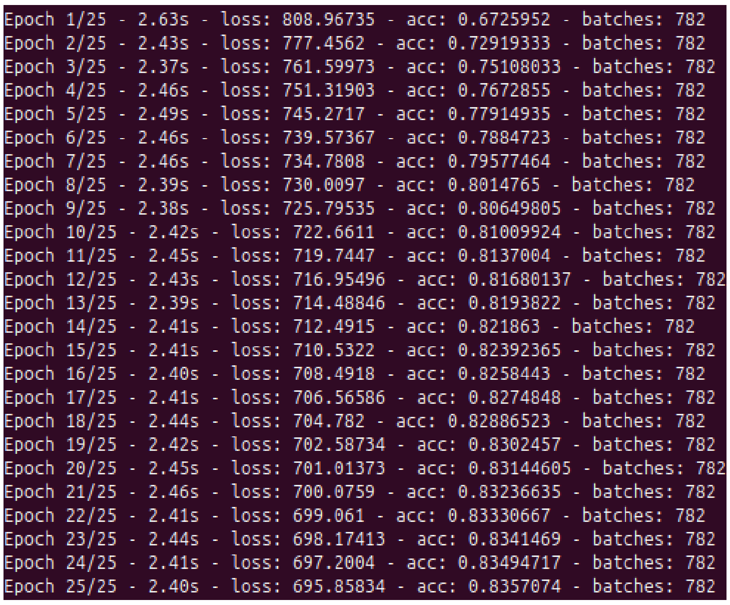


```
+-----------+--------------------+
|        key|                text|
+-----------+--------------------+
|   negative|Broken heart take...|
|uncertainty|Playgrounds, toil...|
|uncertainty|@Brooke_AF @heidi...|
|  litigious|@ANI Beautifully ...|
|   positive|KWS is excited to...|
|   negative|11/12 All togethe...|
|   positive|"Ketchup dripping...|
|   positive|Innovations in In...|
|   negative|Accident with inj...|
|   positive|  EKIN IS SO EXCITED|
|   negative|Marilyn lost her ...|
|   positive|@spicymancer sure...|
|  litigious|‘Chaos’ from stat...|
|   negative|When the media he...|
|  litigious|@PratikM04153488 ...|
|   negative|@GathererSkull Th...|
|   positive|Signed a contract...|
|uncertainty|Perhaps #chrispac...|
```

Figure 2.4: Data read from Kafka *tweet-train* topic

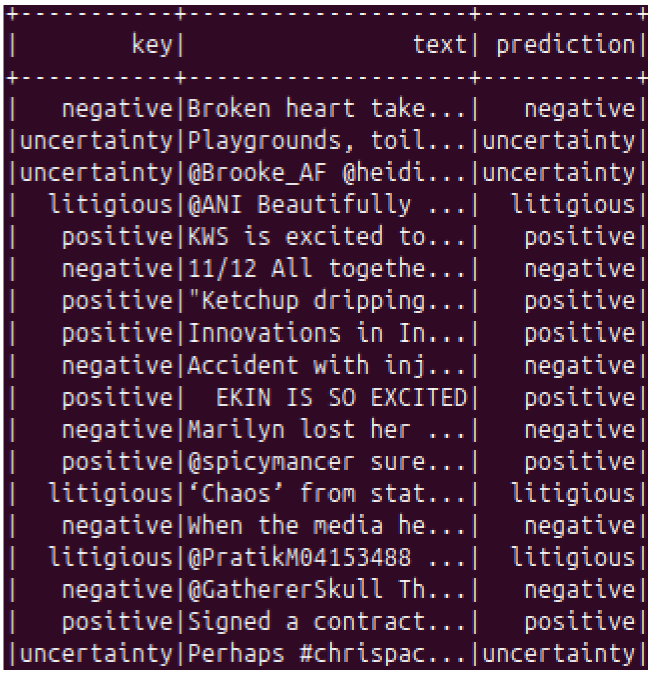
The whole pipeline result and the classifier weights are then saved

```
result = pipeline.fit(df)
result.write().overwrite().save("model_weights")
```



```
Epoch 1/25 - 2.63s - loss: 808.96735 - acc: 0.6725952 - batches: 782
Epoch 2/25 - 2.43s - loss: 777.4562 - acc: 0.72919333 - batches: 782
Epoch 3/25 - 2.37s - loss: 761.59973 - acc: 0.75108033 - batches: 782
Epoch 4/25 - 2.46s - loss: 751.31903 - acc: 0.7672855 - batches: 782
Epoch 5/25 - 2.49s - loss: 745.2717 - acc: 0.77914935 - batches: 782
Epoch 6/25 - 2.46s - loss: 739.57367 - acc: 0.7884723 - batches: 782
Epoch 7/25 - 2.46s - loss: 734.7808 - acc: 0.79577464 - batches: 782
Epoch 8/25 - 2.39s - loss: 730.0097 - acc: 0.8014765 - batches: 782
Epoch 9/25 - 2.38s - loss: 725.79535 - acc: 0.80649805 - batches: 782
Epoch 10/25 - 2.42s - loss: 722.6611 - acc: 0.81009924 - batches: 782
Epoch 11/25 - 2.45s - loss: 719.7447 - acc: 0.8137004 - batches: 782
Epoch 12/25 - 2.43s - loss: 716.95496 - acc: 0.81680137 - batches: 782
Epoch 13/25 - 2.39s - loss: 714.48846 - acc: 0.8193822 - batches: 782
Epoch 14/25 - 2.41s - loss: 712.4915 - acc: 0.821863 - batches: 782
Epoch 15/25 - 2.41s - loss: 710.5322 - acc: 0.82392365 - batches: 782
Epoch 16/25 - 2.40s - loss: 708.4918 - acc: 0.8258443 - batches: 782
Epoch 17/25 - 2.41s - loss: 706.56586 - acc: 0.8274848 - batches: 782
Epoch 18/25 - 2.44s - loss: 704.782 - acc: 0.82886523 - batches: 782
Epoch 19/25 - 2.42s - loss: 702.58734 - acc: 0.8302457 - batches: 782
Epoch 20/25 - 2.45s - loss: 701.01373 - acc: 0.83144605 - batches: 782
Epoch 21/25 - 2.46s - loss: 700.0759 - acc: 0.83236635 - batches: 782
Epoch 22/25 - 2.41s - loss: 699.061 - acc: 0.83330667 - batches: 782
Epoch 23/25 - 2.44s - loss: 698.17413 - acc: 0.8341469 - batches: 782
Epoch 24/25 - 2.41s - loss: 697.2004 - acc: 0.83494717 - batches: 782
Epoch 25/25 - 2.40s - loss: 695.85834 - acc: 0.8357074 - batches: 782
```

Figure 2.5: ClassifierDLApproach training



```
+-----------+--------------------+-----------+
|        key|                text| prediction|
+-----------+--------------------+-----------+
|   negative|Broken heart take...|   negative|
|uncertainty|Playgrounds, toil...|uncertainty|
|uncertainty|@Brooke_AF @heidi...|uncertainty|
|  litigious|@ANI Beautifully ...|  litigious|
|   positive|KWS is excited to...|   positive|
|   negative|11/12 All togethe...|   negative|
|   positive|"Ketchup dripping...|   positive|
|   positive|Innovations in In...|   positive|
|   negative|Accident with inj...|   negative|
|   positive|  EKIN IS SO EXCITED|   positive|
|   negative|Marilyn lost her ...|   negative|
|   positive|@spicymancer sure...|   positive|
|  litigious|'Chaos' from stat...|  litigious|
|   negative|When the media he...|   negative|
|  litigious|@PratikM04153488 ...|  litigious|
|   negative|@GathererSkull Th...|   negative|
|   positive|Signed a contract...|   positive|
|uncertainty|Perhaps #chrispac...|uncertainty|
```

Figure 2.6: Data being classified during training

Model testing

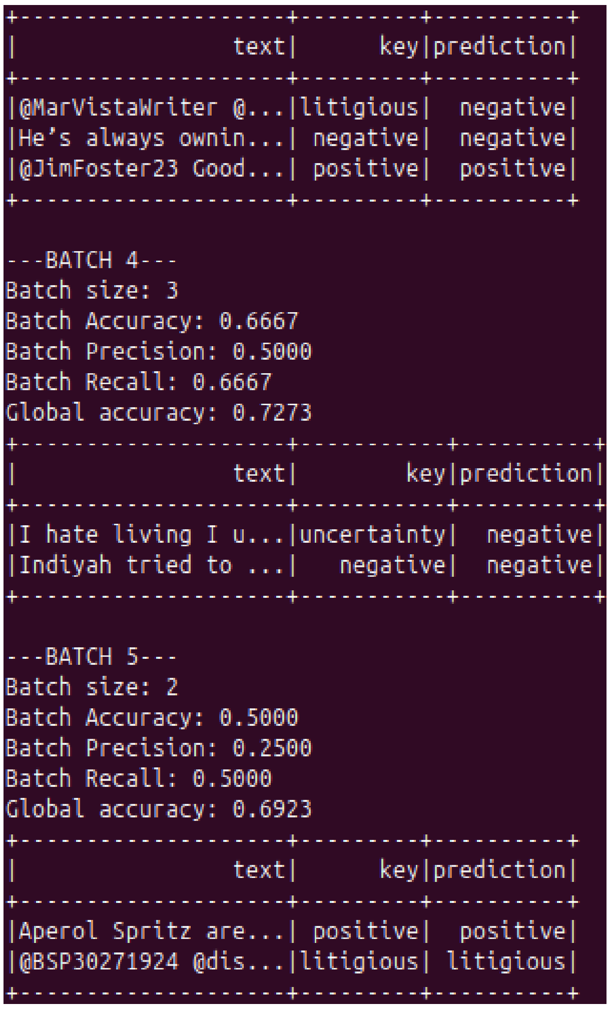
Now that the model has been trained it is possible to test its performance on unseed data.

It is first necessary to setup a connection with the Kafka topic *tweet-test* in *readStream* mode that allows for continuous process of data reading from a Kafka topic.

```
df = spark \
.readStream \
.format("kafka") \
.option("kafka.bootstrap.servers", BOOTSTRAP_SERVERS) \
.option("subscribe", "tweet-test") \
.load()
```

import the pipeline results from the training and use the model weights to classify test data.

```
model = PipelineModel.load("model_weights")
...
out = df.writeStream \
        .outputMode("append") \
        .format("console") \
        .foreachBatch(evaluate_batch) \
        .start()

out.awaitTermination()
```



```
+--------------------+---------+----------+
|                text|      key|prediction|
+--------------------+---------+----------+
|@MarVistaWriter @...|litigious|  negative|
|He’s always ownin...| negative|  negative|
|@JimFoster23 Good...| positive|  positive|
+--------------------+---------+----------+

---BATCH 4---
Batch size: 3
Batch Accuracy: 0.6667
Batch Precision: 0.5000
Batch Recall: 0.6667
Global accuracy: 0.7273
+--------------------+-----------+----------+
|                text|        key|prediction|
+--------------------+-----------+----------+
|I hate living I u...|uncertainty|  negative|
|Indiyah tried to ...|   negative|  negative|
+--------------------+-----------+----------+

---BATCH 5---
Batch size: 2
Batch Accuracy: 0.5000
Batch Precision: 0.2500
Batch Recall: 0.5000
Global accuracy: 0.6923
+--------------------+---------+----------+
|                text|      key|prediction|
+--------------------+---------+----------+
|Aperol Spritz are...| positive|  positive|
|@BSP30271924 @dis...|litigious| litigious|
+--------------------+---------+----------+
```

Figure 2.7: Data being classified during testing

Chapter 3

Experimental results

3.1 Model performance metrics

Given the previously illustrated architecture, the following performance have been obtained

Model training

Through the use of pyspark.ml.MulticlassClassificationEvaluator and after training the model for 25 epochs, the following metrics have been registered

- Accuracy: $83\% \pm 1\%$
- Recall: $83\% \pm 1\%$
- Precision $83\% \pm 1\%$

Model testing

Through the use of pyspark.ml.MulticlassClassificationEvaluator and after classifying the test data, the following metrics have been registered

- Accuracy $74.5\% \pm 1.5\%$

3.2 Producer and Consumer latency

It is possible to create a Kafka cluster composed of one or more server. This allows to provide parallelization to both producer writings and consumer readings which should decrease the latency of the operations.

Althought, in a single machine scenario, this parallelization could have more drawbacks than advantages due to the overhead created by the system itself.
In order to test this, two cluster configurations have been compared:

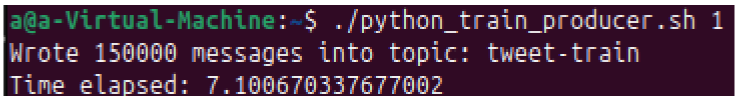
- 1-server cluster
- 10-servers cluster with 10 partitions

In both scenarios, 150000 tweets have been written and read.

1-server cluster

The following time elapsed have been registered for both producer writings and consumer readings:

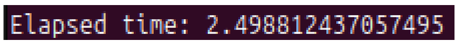
- Writing: 7 ± 0.5 seconds



```
a@a-Virtual-Machine:~$ ./python_train_producer.sh 1
Wrote 150000 messages into topic: tweet-train
Time elapsed: 7.100670337677002
```

Figure 3.1: 1-server cluster data writing time

- Reading: 2.5 ± 0.5 seconds

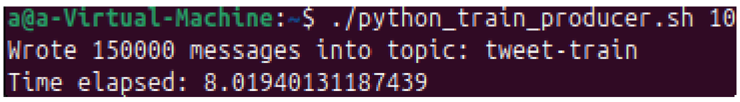


```
Elapsed time: 2.498812437057495
```

Figure 3.2: Spark reading from the 1-server cluster

10-servers cluster

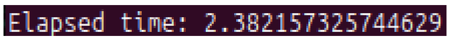
- Writing: 8 ± 0.5 seconds



```
a@a-Virtual-Machine:~$ ./python_train_producer.sh 10
Wrote 150000 messages into topic: tweet-train
Time elapsed: 8.01940131187439
```

Figure 3.3: 1-server cluster data writing time

- Reading: 2.3 ± 0.5 seconds



```
Elapsed time: 2.382157325744629
```

Figure 3.4: Spark reading from the 10-servers cluster

Results

Spark reading seems to take a very similar time, both in the 1-server and 10-servers scenario while the time it requires to write to both cluster configurations appears to have a wider difference, with the 1-server cluster taking about $\sim$ 1 second less. This could be due the overhead created by the configuration itself not being compensated by the parallelization effect.