Tweet analysis and visualization

Matteo Milone

1 Introduzione

In questo progetto ho utilizzato varie tecnologie per la raccolta dei tweets legati agli attuali 4 migliori giocatori dell'NBA. Ciò è stato realizzato usando Kafka e Spark per lo streaming dei dati ed altre operazioni. Il progetto è stato sviluppato in usando Python come linguaggio ed infine i tweets raccolti vengono anche salvati in MongoDB per avere la possibilità di poter usare i dati raccolti per applicazioni diverse e future. Per questioni di praticità i grafici proposti sono stati realizzati in Python. Ogni tecnologia utilizzata nel progetto è stata installata ed usata all'interno di containers Docker.

2 Aspetti teorici

2.1 Pipeline

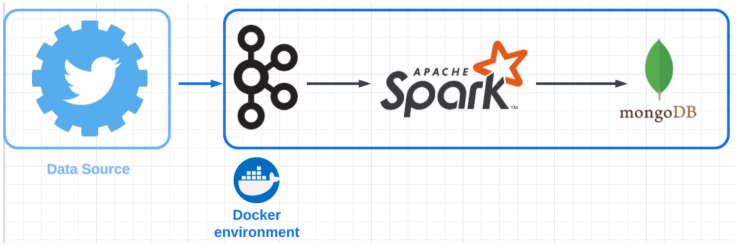

Figure 1: pipeline

2.2 Perchè Docker?

Ho deciso di utilizzare Docker per l'utilizzo delle tecnologie usate per il progetto poiché la possibilità di creare containers in cui installare ogni risorsa (sottoforma di immagini) di cui ho necessitato rende l'ambiente di sviluppo facilmente riproducibile ovunque e vengono ridotte al minimo le problematiche di

compatibilità o conflitti tra le risorse del sistema. Oltre che a evitare problemi di compatibilità tra i programmi usati e il sistema Docker consente anche un'ottimizzazione dell'utilizzo delle CPU, della memoria e dello storage dei dati. In pratica Docker offre ambienti isolati, portatili, scalabili e gestibili, garantendo la riproducibilità e semplificando le operazioni di sviluppo, distribuzione e gestione dell'infrastruttura.

2.3 Lambda Architecture

Questo schema è da anni un capo saldo nel campo dei Big Data, creato da Nathan Marz, uno dei principali contributori del progetto Apache Storm. Questo modello implementa una metodologia generica in grado di fornire un sistema di elaborazione di Big Data modulare, tollerante ai guasti e in tempo reale. Nessuno strumento singolo offre la capacità di elaborare in modo dinamico flussi di dati arbitrari in tempo reale. Per questo, l'architettura Lambda offre un approccio generale necessario per utilizzare un gruppo di tecnologie divise in diversi layers, ognuno dei quali si occupa di un ruolo specifico. I layers sono i seguenti:

- **Batch layer**;
- **Speed layer**;
- **Serving layer;**
- **Batch view (?).**

Nel lavoro proposto, nei rispettivi casi specifici abbiamo un **Batch layer** dove:

- *Producer*: Acquisisce i dati in tempo reale utilizzando l'API di Twitter e li invia a Kafka per l'elaborazione batch successiva;
- *Consume*r: Legge i dati dal topic di Kafka, esegue l'elaborazione batch e li scrive nel database MongoDB;
- *Preprocessing*: Esegue il pre-processing dei dati, inclusa l'eliminazione delle stop words, lo stemming, ecc.;
- *Sentiment analysis*: Utilizza TextBlob per calcolare il sentiment dei tweet;
- *Word Count*: Calcola la frequenza delle parole per generare un word cloud dei termini più frequenti nei tweet;
- *Scrittura in MongoDB*: Scrive i dati dei tweet e le informazioni di sentiment nel database MongoDB.

Invece, riguardo lo **Speed layer**:

- *Consumer*: Legge i dati dal topic di Kafka in tempo reale per l'elaborazione immediata;

- *Preprocessing*: Esegue il pre-processing dei dati in tempo reale;
- Sentiment analysis: Utilizza TextBlob per calcolare il sentiment dei tweet in tempo reale;
- Scrittura in MongoDB: Scrive i dati dei tweet e le informazioni di sentiment nel database MongoDB in tempo reale.

Abbiamo poi il **Serving laye**r, dove abbiamo:

- Data visualization: Utilizza Matplotlib per generare grafici e visualizzare i risultati dell'analisi del sentiment e del word cloud.

Infine menzione speciale per la Batch View. MongoDB è stato usato, come già detto, per la memorizzazione dei dati dei tweets e altre informazioni, che possono essere utilizzate per generare batch views o altre applicazioni come ad esempio time series.

2.4 Data source and streaming

I dati utilizzati in quest'analisi sono stati raccolti attraverso la libreria *Tweepy* che consente di collegarsi all'API di Twitter. I dati raccolti sono tweets riguardanti i principali giocatori dell'NBA in questo momento, oggetto di molta attenzione per gli appassionati visto che al momento dello sviluppo di questa analisi sono in corso i play-off di NBA. L'utilizzo di Kafka consente di poter integrare più fonti di dati diverse in un sistema di streaming. Kafka funziona come intermediario tra data sources e data sinks. In questo caso abbiamo utilizzato solamente Twitter come fonte dati, ma ho deciso di utilizzare comunque Kafka poiché è un'ottima soluzione per la gestione e l'utilizzo dei flussi di dati in real-time.

2.5 Zookeeper

Kafka ha bisogno di Zookeeper per eseguire le sue attività. Zookeeper è un sistema di coordinamento distribuito che fornisce servizi centralizzati per la gestione e la sincronizzazione dei nodi di un cluster. Zookeeper garantisce inoltre l'elevata disponibilità di Kafka fornendo la replica dei dati di configurazione su più nodi.

2.5.1 Creazione Topics

Una volta eseguiti Kafka e Zookeeper all'interno di Docker, c'è bisogno di creare un Topic per connettere il Producer e il Consumer che andremo a utilizzare. Per creare il topic basta digitare dal terminale del container dove è in esecuzione l'immagine di Kafka in Docker il seguente comando:

```
kafka-topics --create --topic nome_topics --bootstrap-server proprio_bs
```

Basta crearlo solo una volta, rimane salvato anche per le applicazioni successive.

2.6 Kafka Producer

Kafka producer è un elemento essenziale (insieme al Consumer) nella creazione di flussi di dati in real-time e nei Big Data in generale. Consente la produzione e l'invio dei dati agli argomenti di Kafka. In questo lavoro il Producer è stato sviluppato in Python e combinato con l'API di Twitter per poter ottenere i tweet pubblicati dai vari utenti e pubblicati all'interno del cluster di Kafka. L'utilizzo di Kafka Producer e Consumer consente di poter combinare dati provenienti da più fonti, gestirli anche se molto voluminosi e inviarli a velocità elevata. L'aspetto più importante è la possibilità di poter lavorare si in modalità sincrona che asincrona, quindi poter eseguire più operazioni in parallelo senza dover attendere il termine di operazioni precedenti.

2.7 Kafka Consumer

Discorso analogo per il Consumer che, supportato da Apache Spark, consente di eseguire operazioni di consumo come analisi, aggregazione, filtraggio o archiviazione dei dati nel cluster. All'interno del Consumer sono state create ed usate alcune funzioni per il preprocessing del testo di ogni tweet e per il conteggio delle parole raccolte. Vedremo, infine, come è stata effettuata una sentiment analysis sfruttando come modello la libreria *textblob*.

2.8 Apache Spark

Per elaborare i dati inizialmente ho creato una sessione Spark. La Spark Session è il punto di partenza e di ingresso per ogni applicazione Spark. Dopo averla inizializzata, usando il motore Spark Streaming ho recuperato i dati dal Producer. In particolare, ho usato la funzione *spark.readStream()* per leggere i dati in real-time dal topic creato e presente in Kafka ed inserito in un dataframe. Con Spark SQL ho creato lo schema per i dati, creato in maniera tale che i dati corrispondano a quelli provenienti dal produttore.

2.8.1 Funzioni e queries

All'interno del Consumer abbiamo usato la funzione *writeStream()* per creare delle queries dalla quali poi abbiamo estratto dei dataframe Pandas attraverso l'utilizzo di Spark SQL. Successivamente ho manipolato i dati e li ho usati per generare due grafici che si aggiornano con un intervallo di 5 secondi circa. I grafici sono stati realizzati usando la libreria *MatPlot* di Python. Inoltre, sempre usando *writeStream()* ho eseguito in parallelo una funzione che salva il datafram spark all'intenro di una collezione di MongoDB.

2.9 Data Visualization

Il primo grafico è un *Word Cloud* contenente le parole (di tutti i tweets raccolti) con il punteggio più alto. Il punteggio associato è relativo al numero di volte che la parola compare. I tweets vengono, quindi, prima processati per eliminare

parti di testo che non sono di interesse all'analisi e poi raggruppati ed inseriti in un nuovo dataframe. Il secondo grafico è un pie-chart che riporta se, rispetto alle keywords proposte, ci sono pareri positivi, negativi o neutrali. Per effettuare la sentiment analysis ho usato textblob. *Textblob* è una libreria NLP di Python capace di effettuare l'analisi dei sentimenti, che consente di determinare se una frase o un documento ha un sentimento positivo, negativo o neutro.

2.10 MongoDB

MongoDB è un database open-source non relazionale orientato ai documenti, noto anche come database NoSQL. Si distingue per la sua flessibilità e scalabilità nel gestire grandi volumi di dati non strutturati o semi-strutturati. MongoDB è utilizzato in una vasta gamma di applicazioni, come applicazioni web, applicazioni mobili, analisi dei dati, Internet of Things (IoT) e molte altre. MongoDB è stato utilizzato per l'archiviazione del dati raccolti. L'archiviazione avviene parallelamente alla generazione dei due grafici prima citati. Le tecnologie utilizzate consentono di poter avere un'alta scalabilità e grado di personalizzazione anche per future e diverse applicazioni. Tutto ciò viene ulteriormente supportato dallo sviluppo del progetto in un ambiente Docker. Una possibile opportunità di miglioramento del progetto potrebbe essere l'integrazione di tools esterni per effettuare la data visualization o comunque poter integrare ulteriori tecnologie per poter interrogare MongoDB, nel quale ho archiviato i dati per poterli avere a disposizione per applicazioni differenti a quelle effettuate e proposte in questo lavoro.

3 Codice utilizzato

In questa sezione andrò a mostrare il codice sviluppato in Python ed usato per realizzare il Producer e il Consumer. Ho omesso le numerose librerie e dipendenze utilizzate affinchè il codice potesse funzionare.

3.1 Producer

```
api_key = ""
api_key_secret = ""
access_token = ""
access_token_secret = ""
bearer_token = ""
```

In queste prime righe vengono inseriti ed utilizzati i valori delle chiavi API e i toke per l'accesso a Twitter.

```
bs = bootstrap-server
producer = KafkaProducer(bootstrap_servers = bs)
topic_name = nome_topic
```

Qui viene prima speicficato l'indirizzo del server bootstrap di Kafka necessari per la connessione e poi viene creato un oggetto KafkaProducer che sarà il responsabile dell'invio dei dati al server Kafka. Infine viene specificato il nome del topic Kafka creato in precedenza in cui verranno inviati i dati.

```
def twitterAuth():
authenticate = tweepy.OAuthHandler(api_key, api_key_secret)
authenticate.set_access_token(access_token, access_token_secret)
authenticate.secure = True
api = tweepy.API(authenticate, wait_on_rate_limit=True)
return api
```

Questa è una funzione che gestisce l'autenticazione a Twitter utilizzando le chiavi API precedentemente estratte. Restituisce un oggetto api che sarà utilizzato per l'accesso alle API di Twitter.

```
class MyStream(tweepy.StreamingClient):
 def on_connect(self):
    print("CONNECTED!")

 def on_data(self, rdata):
    tweet = json.loads(rdata)
    if tweet['data']:
        data = {'message': tweet['data']['text'].replace(',', '')}
        producer.send(topic_name, value=json.dumps(data).encode('utf-8'))
    return True

 def on_error(status_code):
    if status_code == 420:
        return False

def start_streaming(self, keywords):
    for term in keywords:
        self.add_rules(tweepy.StreamRule(term))
    self.filter()
```

Questo blocco definisce una nuova classe chiamata MyStream che eredita dalla classe tweepy.StreamingClient. La classe MyStream sarà utilizzata per gestire lo streaming dei tweet. Il metodo *on_connect* viene chiamato quando la connessione allo streaming di Twitter viene stabilita con successo. In questo caso, viene semplicemente stampato un messaggio "CONNECTED!". Il metodo `on_data` viene chiamato quando vengono ricevuti nuovi dati dallo streaming di Twitter. Il parametro rdata contiene i dati dei tweet in formato JSON. Viene utilizzata la funzione json.loads() per convertire il JSON in un oggetto Python (un dizionario) chiamato tweet. Sempre in questo metodo viene verificato se il dizionario tweet contiene una chiave chiamata 'data'. Se sì, viene creato un nuovo dizionario data con un unico campo 'message', il quale contiene il testo

del tweet. Prima di inviare il dizionario al producer di Kafka, viene rimosso il carattere ',' dal testo del tweet utilizzando il metodo replace(). Infine, il messaggio viene inviato al topic Kafka specificato utilizzando il producer. Il metodo send() del producer di Kafka richiede che il valore da inviare sia di tipo bytes. Pertanto, viene utilizzata la funzione json.dumps(data) per convertire il dizionario data in una stringa JSON, e successivamente la stringa viene codificata in formato di byte utilizzando il metodo .encode('utf-8'). In questo modo, il valore può essere correttamente inviato al topic Kafka utilizzando il producer. Il metodo `on_error` viene chiamato quando si verifica un errore durante lo streaming dei tweet. Se il codice di stato è 420 (errore di limitazione della frequenza), viene restituito False, indicando che lo streaming deve essere interrotto. Il metodo `start_streaming` avvia effettivamente lo streaming dei tweet utilizzando le parole chiave specificate nel parametro keywords. Viene iterato su ciascuna parola chiave e viene aggiunta una regola di streaming utilizzando `self.add_rules`(tweepy.StreamRule(term)). Infine, viene chiamato il metodo self.filter() per avviare l'ascolto dello streaming e filtrare i tweet in base alle regole definite.

```
keywords = ["Tatum", "Jokic", "Doncic", "Embiid"]
```

In questa riga vengono definiti gli argomenti di ricerca (parole chiave) che verranno utilizzati per filtrare i tweet.

```
if __name__ == '__main__':
twitter_stream = MyStream(bearer_token)
twitter_stream.start_streaming(keywords)
```

Python e questa funzione per verificare se lo script viene eseguito direttamente come programma principale anziché essere importato come modulo in un altro script. Questo metodo avvia la ricezione dello streaming dei tweet su Twitter, utilizzando le parole chiave specificate nella lista keywords per filtrare i tweet in base ai contenuti desiderati.

3.2 Consumer

```
bs = bootstrap-server
topic_name = nome_topic
```

Viene specificato il bootstrap server di Kafka (l'indirizzo del server Kafka) e il nome del topic da cui verranno letti i dati.

```
def preprocess(x):
x = x.lower()
tokens = nltk.word_tokenize(x)
tokens_x = [w for w in tokens if w.isalpha()]
stop_x = [word for word in tokens_x if not word in stopwords]
no_rt_x = [word for word in stop_x if not re.search(r'\brt\b', word)\
```

```
 and not re.search(r'http\S+', word) and word != 'message']
stemmer = PorterStemmer()
stemm_x = [stemmer.stem(word) for word in no_rt_x]
return (" ".join(stemm_x))
```

Questa funzione viene utilizzata per eseguire il preprocessing del testo. Prende in input una stringa di testo e applica una serie di operazioni, tra cui la conversione in minuscolo, la tokenizzazione, la rimozione delle stopwords, l'eliminazione dei retweet e dei link, e il stemming delle parole.

```
def analyze_sentiment(text):
blob = TextBlob(text)
sentiment = blob.sentiment.polarity
if sentiment > 0:
    return "POSITIVE"
elif sentiment < 0:
    return "NEGATIVE"
else:
    return "NEUTRAL"
```

Questa funzione utilizza la libreria TextBlob per calcolare il sentiment del testo. Prende in input una stringa di testo, calcola la polarità del sentiment utilizzando TextBlob e restituisce una stringa corrispondente al risultato (POSITIVO, NEGATIVO o NEUTRO).

```
def write_row_in_mongo(batch_df, batch_id):
client = pymongo.MongoClient("uri")
db = client["db_name"]
collection1 = db["collect_name"]
rows = batch_df.toJSON().map(lambda j: json.loads(j)).collect()
for row in rows:
    collection.insert_one(row)
```

Qui viene configurato il client per la connessione al server MongoDB impostando anche il database e la collection in cui memorizzare i dati. Inoltre, Questa funzione viene utilizzata per scrivere i dati in formato batch in MongoDB. Prende in input un DataFrame (`batch_df`) e un ID del batch (`batch_id`). Si connette a MongoDB, seleziona la collezione corretta e inserisce ogni riga del DataFrame come documento individuale. Il metodo toJSON() viene chiamato su `batch_df` per convertire ciascuna riga del DataFrame in una rappresentazione JSON. Questo metodo restituisce un nuovo DataFrame in cui ogni riga contiene una stringa JSON che rappresenta i dati della riga corrispondente. Successivamente, viene chiamato il metodo map() sul DataFrame JSON, che applica una funzione a ciascuna riga del DataFrame. Nell'esempio, la funzione lambda lambda j: json.loads(j) viene utilizzata per convertire ciascuna stringa JSON in un dizionario Python, utilizzando la funzione json.loads(). Infine, il metodo collect() viene chiamato sul DataFrame mappato per ottenere tutti i dati come

una lista di dizionari Python. Questa lista di dizionari rappresenta le righe convertite del DataFrame originale in formato JSON.

```
    findspark.init()
spark = SparkSession.builder \
    .master("local[*]") \
    .appName("TwitterSentimentAnalysis") \
    .config("spark.mongodb.uri", "uri.db.collection") \
    .config("spark.jars.packages", "org.mongodb.spark:mongo-spark-connector_2.12:3.0.1") \
    .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.12:3.1.2") \
    .getOrCreate()
```

Questa parte configura Spark per l'elaborazione dei dati. findspark.init() inizializza l'integrazione di Spark con l'ambiente Python utilizzando la libreria findspark. Questo è necessario per trovare e utilizzare l'installazione di Spark nel sistema. SparkSession.builder creaun'istanza di SparkSession, che rappresenta l'entry point per l'utilizzo di funzionalità di alto livello di Spark, come il supporto per l'esecuzione di query SQL e la gestione dei dati. Vengono impostate alcune opzioni, come il master locale, il nome dell'applicazione, l'URI di connessione per MongoDB e le dipendenze necessarie per il connettore Spark-Kafka e il connettore Spark-MongoDB. .master("local[*]") specifica che l'applicazione Spark verrà eseguita in modalità locale, utilizzando tutti i core disponibili sulla tua macchina. .getOrCreate() restituisce l'istanza di SparkSession, creandola se non esiste già o restituendo quella esistente se presente. In pratica, ottieni l'istanza di SparkSession pronta per essere utilizzata per le operazioni successive.

```
    sc = spark.sparkContext
    sc.setLogLevel('ERROR')
```

Viene creato il contesto Spark utilizzando la sessione Spark precedentemente configurata. Viene anche impostato il livello di log su "ERROR" per ridurre l'output di log indesiderato.

```
    schema = StructType([StructField("message", StringType())])
```

Viene definito lo schema del flusso di dati, che specifica il tipo di dato per ogni colonna.

```
    df = spark.readStream \
       .format("kafka") \
       .option("kafka.bootstrap.servers", bs) \
       .option("subscribe", topic_name) \
       .option("startingOffsets", "latest") \
       .option("header", "true") \
       .load() \
       .selectExpr("CAST(value AS STRING) as message")
```

df = spark.readStream crea un nuovo DataFrame chiamato df utilizzando il metodo readStream() dell'oggetto SparkSession. readStream() viene utilizzato per leggere dati in streaming da una sorgente di dati. .format("kafka") specifica il formato della sorgente di dati, che è "kafka" nel nostro caso. Indica che stiamo leggendo i dati da un topic di Apache Kafka. .option("kafka.bootstrap.servers", bs) imposta l'opzione "kafka.bootstrap.servers" con il valore di bs. bs rappresenta gli indirizzi dei server di bootstrap di Kafka, che sono utilizzati per connettersi ai nodi del cluster Kafka. .option("subscribe", topicname) specifica l'opzione "subscribe" con il valore di topicname. topicname rappresenta il nome del topic Kafka da cui leggere i dati. .option("startingOffsets", "latest") imposta l'opzione "startingOffsets" a "latest". Ciò significa che il processo di lettura in streaming inizierà a leggere i dati dal punto più recente disponibile nel topic Kafka. In altre parole, verranno letti solo i messaggi che arrivano dopo l'inizio del processo di streaming. .option("header", "true") imposta l'opzione "header" a "true". Questo indica che il primo record nel topic Kafka contiene l'intestazione delle colonne del DataFrame. .load() carica effettivamente i dati in streaming dal topic Kafka utilizzando le opzioni specificate in precedenza. .selectExpr("CAST(value AS STRING) as message") applica un'operazione di selezione sul DataFrame caricato. Utilizzando l'espressione CAST(value AS STRING) as message, viene effettuato il casting del valore del messaggio Kafka (che di default è in formato binario) in una stringa e viene assegnato al nuovo alias "message". In pratica, stiamo selezionando solo la colonna "value" del DataFrame e rinominandola come "message".

```
preprocess_udf = udf(preprocess)
df = df.withColumn("cleaned_data",
preprocess_udf(df.message)).dropna()
```

Viene definita una User-Defined Function (UDF) per applicare la funzione di preprocessing al DataFrame. La funzione di preprocessing viene applicata alla colonna "message" e il risultato viene assegnato a una nuova colonna chiamata `cleaned_data`. Vengono inoltre eliminati eventuali record con valori nulli.

```
words_df =
df.select(explode(split("cleaned_data"," ")).alias("word"))
counts_df = words_df.groupBy("word").count()
```

`words_df` è un nuovo dataframe dove vengono applicate una serie di operazioni per ottenere un DataFrame che contiene una colonna chiamata "word", che rappresenta ogni parola individuale estratta dai dati puliti. split(`cleaned_data`, " ") divide la colonna `cleaned_data` in base allo spazio (" ") creando un array di parole. explode() viene utilizzato per "esplodere" l'array di parole in modo che ogni parola venga trattata come una singola riga nel DataFrame risultante. Infine, alias("word") viene utilizzato per rinominare la colonna risultante come "word".

`counts_df` = `words_df`.groupBy("word").count(): In questa riga, viene creato un nuovo DataFrame chiamato `counts_df`. Viene applicata l'operazione

groupBy("word") per raggruppare le righe del DataFrame `words_df` in base al valore della colonna "word". Successivamente, viene chiamato il metodo count() per contare il numero di occorrenze di ogni parola all'interno del DataFrame.

```
sentiment_udf = udf(lambda text: analyze_sentiment(text), StringType())
df = df.withColumn("sentiment", sentiment_udf(df.cleaned_data))
sents_df = df.select("sentiment").groupBy("sentiment").count()
```

Viene definita una UDF per applicare la funzione di analisi del sentiment al DataFrame. La funzione di analisi del sentiment viene applicata alla colonna `cleaned_data` e il risultato viene assegnato a una nuova colonna chiamata "sentiment". Successivamente, viene eseguito il conteggio delle occorrenze dei diversi sentimenti utilizzando il metodo groupBy() e count().

```
df = df.withWatermark("timestamp", "10 minutes")
```

Si applica un watermark al dataframe in streaming. Il watermark è un meccanismo utilizzato in Spark Structured Streaming per gestire il ritardo dei dati e l'eliminazione dei dati scaduti nel flusso in streaming. Indica il limite massimo di ritardo consentito per l'elaborazione dei dati in streaming. In questo caso i dati con un timestamp più vecchio di 10 minuti rispetto al tempo di elaborazione corrente non saranno usati per ulteriori elaborazioni, quindi vengono considerati scaduti e non saranno inclusi nelle aggregazioni o nelle query successive.

```
MNGquery = df.writeStream.queryName("tweets") \
.foreachBatch(write_row_in_mongo2).start()
```

Viene definita una query di scrittura in streaming per il DataFrame "df". La query viene assegnata a una variabile chiamata "MNGquery" e viene specificato il nome della query come "tweets". Viene utilizzata la funzione foreachBatch() per applicare la funzione per salvare i dati in mongo a ogni batch di dati. Infine, la query viene avviata con il metodo start().

```
word_count_query = counts_df.writeStream \
    .outputMode("complete") \
    .format("memory") \
    .queryName("") \
    .option("path", "") \
    .option("checkpointLocation", "") \
    .option("forceDeleteTempCheckpointLocation", "true") \
    .trigger(processingTime="30 seconds") \
    .start()

sentiment_query = sents_df.writeStream \
    .outputMode("complete") \
    .format("memory") \
    .queryName("") \
```

```
    .option("path", "") \
    .option("checkpointLocation", "c") \
    .option("forceDeleteTempCheckpointLocation", "true") \
    .trigger(processingTime="30 seconds") \
    .start()
```

.outputMode("complete") imposta la modalità di output del risultato dell'operazione di scrittura in streaming come "complete". Ciò significa che l'output conterrà tutti i risultati aggregati del DataFrame `counts_df`. In caso si fosse usato "update" o "append", rispettivamente sarebbero stati inclusi solo i nuovi risultati nell'output e quelli precedenti non verranno modificati o rimossi, mentre nel secondo caso l'output non conserverà i risultati precedenti ai nuovi appena genertati dell'elaborazione in streaming. .format("memory") specifica il formato di output per l'operazione di scrittura in streaming come "memory". Ciò indica che l'output verrà salvato in memoria come una tabella temporanea. .queryName(""): Assegna un nome alla query di scrittura in streaming. .option("path", "") imposta l'opzione "path" per specificare la directory di output dei risultati dell'operazione di scrittura in streaming come "results". Questo è il percorso in cui verranno salvati i risultati dell'operazione. .option("checkpointLocation", "") specifica il percorso della directory di checkpoint per l'operazione di scrittura in streaming come "checkpoint". Il checkpoint è una directory utilizzata da Spark per salvare i metadati e lo stato dell'elaborazione in streaming. .option("forceDeleteTempCheckpointLocation", "true") imposta l'opzione "forceDeleteTempCheckpointLocation" su "true". Questo indica a Spark di eliminare automaticamente la directory temporanea del checkpoint quando la query di scrittura in streaming viene fermata. Usata per evitare frequenti errori di collisione tra directories. .trigger(processingTime="30 seconds"): Imposta la frequenza con cui verranno avviate le operazioni di scrittura in streaming. In questo caso, verranno avviate ogni 30 secondi ("30 seconds"). .start(): Avvia effettivamente l'operazione di scrittura in streaming.

```
while True:
  # Word Cloud
  word_count_data = spark.sql("SELECT * FROM WORDQ").toPandas()
  # ...
  # Visualizzazione della Word Cloud

  # Sentiment Analysis
  sentiment_data = spark.sql("SELECT * FROM SENTQ").toPandas()
  # ...
  # Visualizzazione dell'analisi del sentiment

  # Display the plots
  # ...
  # Mostra i grafici

  time.sleep(5)  # Wait for 5 seconds before checking again
```

In questo blocco di codice, viene eseguita un'iterazione continua per visualizzare i risultati in tempo reale. Viene recuperato il DataFrame dei conteggi delle parole e dei sentimenti utilizzando Spark SQL e viene convertito in un oggetto Pandas DataFrame per la visualizzazione. Successivamente, vengono generati i grafici per la Word Cloud e l'analisi del sentiment utilizzando i dati recuperati. Infine, i grafici vengono mostrati e viene aggiunto un ritardo di 5 secondi prima di eseguire una nuova iterazione.

4 Conclusioni

Le tecnologie utilizzate offrono un'elevata scalabilità e personalizzazione, consentendo di adattarsi a future applicazioni diverse. Inoltre, l'implementazione del progetto in un ambiente Docker offre ulteriore supporto. Un'opportunità di miglioramento potenziale sarebbe l'integrazione di strumenti esterni per la visualizzazione dei dati o l'aggiunta di altre tecnologie per consentire l'interrogazione di un database MongoDB. Ciò consentirebbe di accedere ai dati archiviati per scopi diversi da quelli presentati in questo lavoro e di adattarsi a nuove applicazioni.