Twitter Profanity Filter

Big Data Models and Algorithms, University of L'Aquila.

Stefano Valentini - m. 254825

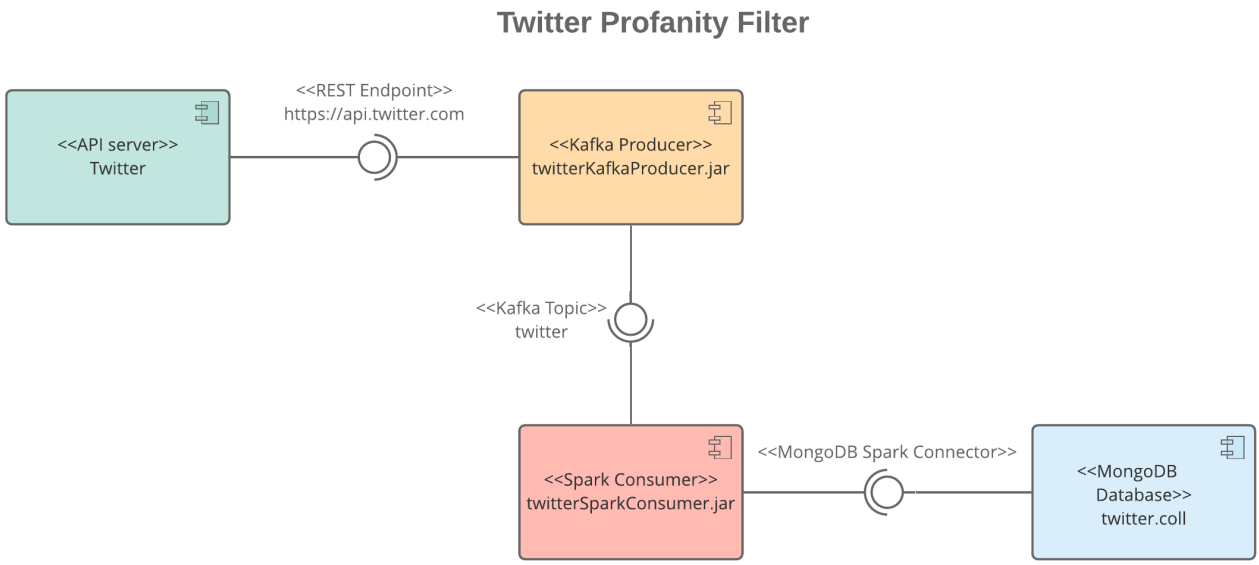
demo: https://youtu.be/TvMuIvu6a5E

Table of Contents

Introduction

The goal of this project is to build an application that is capable of retrieving the twitter's tweet streaming, apply a filter, and store on a MongoDb database only the tweets that are written in english and do not contain a series of banned words (in this case the Google's profanity words list).

Architecture



In a nutshell: the tweet's stream is retrieved by the Kafka Producer using the `twitter4j` library, that offers ready-to-use functions to interact with the Twitter's API Server.
For each tweet, the Kafka Producer asynchronously publishes a record to the `twitter` topic; since the Spark Consumer is subscribed to this topic it will receive each tweet; it will then apply the specified filter and it will save the received records to the MongoDB database in batches, each X milliseconds (where X is configurable inside the properies file).

Kafka Producer

The Kafka Producer is responsible for retrieving the tweet stream and to publish the received tweets to the `twitter` topic.
The tweet json structure can be found in the `twitter-kafka-producer/src/main/resources/tweet.json` file; for this project we are only interested in the `lang` and `text` fields but, since we want the producer to be independent w.r.t. the application's goal, we will publish the whole json anyway.
The `twitter4j` library works by connecting to the Twitter API endpoint and by instantiating a `Listener` object, whose methods will be invoked according to the responses that are received from Twitter; in particular we are interested in the `onStatus` method, that will be called when a new status is received.
The `onStatus` invokes the publish procedure, that is responsible of building the Kafka record and to publish it to the `twitter` topic.

Spark Consumer

The Spark Consumer is responsible for fetching the tweets from the `twitter` topic, apply a filter to the stream, and save the results to a MongoDB database.
It is initialized by instantiating a `SparkSession` object, to be used to create the input `Dataset` object.
The input dataset is constructed by specifiyng that the rows are in the `kafka` format, and that they have to be retrieved from the address specified in the properies file and from the `twitter` topic.

```
Dataset<Row> inputDf = sparkSession
    .readStream()
    .format("kafka")
    .option("kafka.bootstrap.servers", KafkaConfiguration.get("address"))
    .option("subscribe", KafkaConfiguration.get("topic"))
    .load();
```

Since that `value` field of the kafka record is not readable by Spark we need to cast it to `string`; now that we have a readable value we need to structurize it so to be able to query and filter it by accessing its fields.
By starting from a `twitter.json` model file we can extract its schema by importing this single file into a dummy Spark dataset; this operation is performed only once, at the start of the execution.
Now that we have a string value and a schema we can adapt that string value to the schema we built, so to construct a structurized dataset.
Having a dataset whose rows are structured allows us to access its fields by using the field names themselves so we can finally apply the filter:

```
Dataset<Row> filteredDf = inputDfAsJsonFlattened
    .filter(inputDfAsJsonFlattened.col("lang").equalTo("en")
        .and(not(inputDfAsJsonFlattened.col("text")
                .rlike("(^|\\s)("+String.join("|", bannedWords)+")(\\s|$)"))));
```

The code above builds a new dataset by filtering the input one: first we check that the `lang` field is equal to `en`, then we check that the `text` field (that contains the text body of the tweet) does not contain any of the words loaded from the `banned-words.txt` configuration file (through the `FilterConfiguration` class).

The filtered dataset is then prepared to be inserted into the database; with the code below we are commanding that every `processingTime` milliseconds each row of the dataset that has still not be processed (`append`) must be sent to the database sink (`MongoDbSink`).

```
StreamingQuery databaseOutput = filteredDf.writeStream()
    .outputMode("append")
    .trigger(Trigger.ProcessingTime(processingTime, TimeUnit.MILLISECONDS))
    .foreach(new MongoDbSink())
    .start();

databaseOutput.awaitTermination();
```

A sink in Spark is a class that performs some kind of processing on the input dataset (sink, meaning that everything is thrown there); in our case our sink extends the spark's `ForeachWriter` class by implementing its methods: `open`, `process`, `close`.

- In the `open` method, that is invoked each time a batch is processed (i.e., every `processingTime` milliseconds), we initialize the connection to the database and we instantiate a list that will contain all the rows that will be then saved.
- In the `process` method, that is executed after the `open`, we simply add the current row to the previously declared list.
- In the `close` method, that is executed after the last row has been processed, we create a MongoDB document for each value in the list that we built, using as id the `id_str` field of the tweet and the `text` field as value.

Conclusions

By looking at the database we can see that we have about **50 insertion per second** and this seems right because:

- we know that there are obout 5800 tweets/s **but** the free access Twitter API "returns a small random sample of all public statuses": since the enterprise paid API "delivers a 10% random sample of the realtime Twitter Firehose", we can assume that our free access API samples an amount of tweets that is around the **3-5%** of the Twitter Firehose (about **300** tweets/s);
- we are only looking at tweets in english by using the `lang` property that is inferred by the Twitter's algorithms (so, alot of tweets may have this property as not defined or are wrongly categorized);
- we are filtering a list of 1703 common "bad" words (like "ugly", for example).