SparkR Machine Learning

Jim Harner

1/6/2020

```
Sys.setenv(SPARK_HOME = "/opt/spark")
library(SparkR, warn.conflicts = FALSE)
sparkR.session(appName = "StructuredNetworkWordCount",
               sparkConfig = list(spark.driver.memory = '4g'))
```

```
## Spark package found in SPARK_HOME: /opt/spark
```

```
## Launching java with spark-submit command /opt/spark/bin/spark-submit   --driver-memory "4g" sparkr-sh
```

```
## Java ref type org.apache.spark.sql.SparkSession id 1
```

8.1 Spark Streaming

The analysis of streaming data is becoming essential with the increasing prevalence of real-time data. It is not just Twitter! The Internet of Things (IoT) generates generates huge amounts of data, as do wearables. Sensors are becoming ubiquitous and they collect climate data, health data, ecological data and increasingly, almost anything that supports sensors being embedded. The web generates a continues stream of log data, but far more in terms of browsing habits, financial transactions, etc.

A powerful, robust engine is needed to analyze this data as it arrives. It would be nice if we do not need to learn a completely new programming model.

The material in this section is based on Spark's 2.2.0 Structured Streaming Programming Guide.

8.1.1 Basics

Spark supports structured streaming using the a fault-tolerant streaming engine built on the Spark SQL engine. It is highly scalable and has very low latency. The computations for streaming data are done as in batch computations for static data. The Spark SQL engine runs analyses incrementally and continuously and updates the final result as streaming data continues to arrive.

The Spark DataFrame API supports R for getting streaming aggregations, event-time windows, stream-to-batch joins, among others. The streaming engine ensures end-to-end, exactly-once fault-tolerance guarantees through checkpointing and Write Ahead Logs.

8.1.2 Streaming Example

The Structured Streaming Programming Guide has a simple example to illustrate the concept.

Create a streaming DataFrame that represents text data received from a server listening on localhost:9999, and transform the DataFrame to calculate word counts.

```
# Create DataFrame representing the stream of input lines from connection to localhost:9999
lines_SDF <- read.stream("socket", host = "localhost", port = 9999)

# Split the lines into words
words <- selectExpr(lines_SDF, "explode(split(value, ' ')) as word")
```

```
# Generate running word count
wordCounts_SDF <- count(group_by(words, "word"))
```

The `lines_SDF` represents an unbounded table containing the streaming text data. This table contains one column of strings named `value`, and each line in the streaming text data becomes a row in the table. Next, we have a SQL expression with two SQL functions - split and explode, to split each line into multiple rows with a word each. In addition, we name the new column `word`. Finally, we have defined the `wordCounts_SDF` SparkDataFrame by grouping by the unique values in the SparkDataFrame and counting them.

We now start receiving data and computing the counts. To do this, we set it up to print the complete set of counts (specified by `outputMode = "complete"` to the console every time they are updated. And then start the streaming computation using `start()`. At this point the code does not work since we are not actually streaming data. But the code is given for reference.

```
# Start running the query that prints the running counts to the console
query <- write.stream(wordCounts, "console", outputMode = "complete")
awaitTermination(query)
```

After this code is executed, the streaming computation will have started in the background. The `query` object is a handle to that active streaming query, and we have decided to wait for the termination of the query using `awaitTermination` to prevent the process from exiting while the query is active.

We need a source for streaming data. The UNIX utility Netcat (`nc`) can be used as a data server, but is not yet built into `rspark`. The Linux command should be run in a separate container.

```
nc -lk 9999
```

8.1.3 Programming Model

Spark's Structured Steaming model treats real-time data as a continuously-appended table. Thus, the stream-processing model is very similar to Spark's batch processing model. The standard batch-processing query runs automatically as an incremental query on an unbounded input table.

Every arriving row appended to the Input Table. A query on the input will generate a Result Table everytime a trigger event occurs, e.g., every second.

The output is written to external storage in one of three modes: complete, append, and update. If aggregations are not written the append and update are equivalent.

Event-time data

Event-time is embedded in the data. Often, it is important to operate on this event-time, i.e., you probably want to use the time when the data was generated (that is, event-time in the data), rather than the time Spark receives them.

Event-time is very naturally expressed in this model, i.e, each event from the devices is a row in the table, and event-time is a column value in the row. This allows window-based aggregations (e.g. number of events every minute) to be just a special type of grouping and aggregation on the event-time variable and each time window is a group. Each row can belong to multiple windows/groups.

Fault tolerance

Spark streaming is designed such that the Structured Streaming sources, the sinks and the execution engine reliably track the exact progress of the processing so that it can handle any kind of failure by restarting and/or reprocessing. Every streaming source is assumed to have offsets (similar to Kafka offsets) to track the read position in the stream. The engine uses checkpointing and write ahead logs to record the offset range of the data being processed in each trigger.

8.1.4 Input Sources

Spark has the following built-in data sources:

- File source: reads a file written in a local directory as a stream
- Kafka source: poll data from Kafka
- Socket source: reads UTF-8 text data from a socker connection

The socket source is not fault-tolerant. The example in Section 5.4.3 used a socket source. Kafka is commonly used in production as shown in Streaming Architecture figure in Section 4.1.

```
sparkR.session.stop()
```