Sentiment Analysis of Tweets through Lambda Architecture

Davide Pucci
Matricola: 7064811
davide.pucci@stud.unifi.it

Abstract

This work is a full term project for the course of Parallel Computing, held by professor Marco Bertini at University of Florence. In this project is described a Lambda Architecture to perform real-time sentiment analysis on Tweets in a big data scenario, where thousands of tweets are generated each moment. This implementation makes use of Apache software foundation technologies such as Hadoop, HBase and Storm. In order to show the results obtained by the architecture, a GUI has been implemented through the usage of JavaFX.

1. Introduction

In modern social networks thousands of posts/tweets are generated in every moment, making the analysis of their content challenging to be performed. In such scenario it is not possible to perform a real-time analysis in a single computer, in fact memory requirements make the task unfeasible. A distributed cluster of computers needs to be used, in order to deal with such ever-growing, massive, amount of data.

Typically, when multiple computing units are used, batch processing through MapReduce can achieve good speedup values, but, as it is, this framework is not able to cope with real-time necessities. We would need the real-time responsiveness of stream processing, still achieving a scalable and fault-tolerant solution.

Lambda Architecture[6], originally proposed by Nathan Marz, is designed to achieve this goal: it allows to take advantage of both batch processing and stream processing, in order to reduce latency and to maintain good throughput. This solution is also easy to scale, and fault-tolerant.

This architecture consists of three layers:

- **Batch Layer** manages a ever-growing and immutable database, on which it computes the functions needed to preform our analysis. These computations are batch processed, hence with broad latency and large throughput, usually MapReduce framework is used to perform this. The output of this layer consists in batch views, which summarize the results obtained by this layer.
- **Speed Layer** allows to mitigate the high latency of the batch layer through the usage of stream-processing, in order to offer real-time views on the data. Its aim is to *fill the gap* given by the batch layer, as this cannot consider real-time data, but only what is available at the beginning of its computation. Once a new batch view is available, the expired speed layer results need to be discarded, in order to avoid overlap among the two when presenting the results. This mechanism is schematized in Figure 1.
- **Serving Layer** offers the access to tables in which both batch views and speed views are stored. In such a way this layer can be used by views, such as GUIs, to access real-time data. Moreover this layer can be used to communicate with the speed layer and the batch layer in order to start/stop the computation, or to set some initialization parameters.

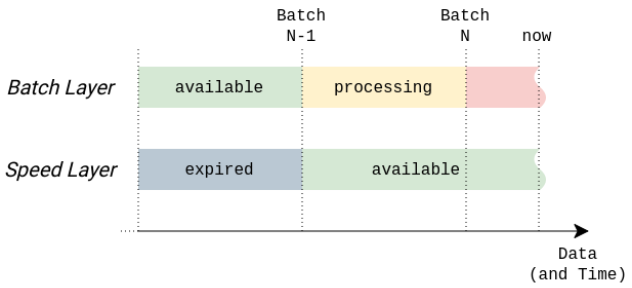

Figure 1. Data expiration mechanism. Allows to avoid overlap between the results in the speed view and in the batch view. Data in green represents the results that are currently used to provide the real-time outcome.

In this work Lambda Architecture is being used to perform Sentiment Analysis [4] on large amount of Tweets in real-time. In particular our aim is to classify as positive or negative the opinions expressed about a certain keyword in each tweet in the stream of uploaded tweets. In such a way we are able to quickly gather the average opinions of many distinct users over a set of given keywords, such as brands or companies.

In order to perform this analysis a classifier is required. This will be used by both the speed layer and the batch layer to estimate the sentiment of each tweet. In Section 2 the classifier is described in detail, whilst in Section 3 are reported the characteristics of each layer of the architecture. Finally, in order to show the user real-time results obtained by the architecture, a GUI was implemented, as described in Section 4.

2. Sentiment Classifier

The sentiment classifier, given the content of a tweet, estimates whether it expresses a positive or a negative opinion. The implementation of such module is made with the usage of LingPipe [1], a toolkit for text processing using computation linguistic.

In this implementation we make use of a language-model classifier, which performs joint probability-based classification of n-grams sequences over the positive/negative classes. In order to achieve this behavior the model needs to learn the multivariate distribution over the two classes.

The dataset Sentiment140 [3] was used in order to train the model. In particular a $90|10$ split was used in order to get a train-set and a test-set. The resulting model achieves an accuracy of 76.71% when using $n = 5$.

This classifier is being used by both the speed layer and the batch layer, in order to estimate the sentiment of the analyzed tweets.

3. The Implementation

As outlined in Section 1, Lambda Architecture consists of three layers: the *batch layer*, the *speed layer* and the *serving layer*. In this implementation each layer is realized through the usage of a specific technology offered by Apache software foundation.

In particular, as shown in Figure 2, the *speed layer* makes use of Apache Storm, the *batch layer* makes use of Apache Hadoop, whilst the *serving layer* makes use of Apache HBase. These systems were executed in pseudo-distributed mode on a local cluster.

The user interface, implemented through the usage of JavaFX, interacts with the *serving layer* in order to access the speed view and the batch view, thus offering the user updated information on the outcome of the analysis.

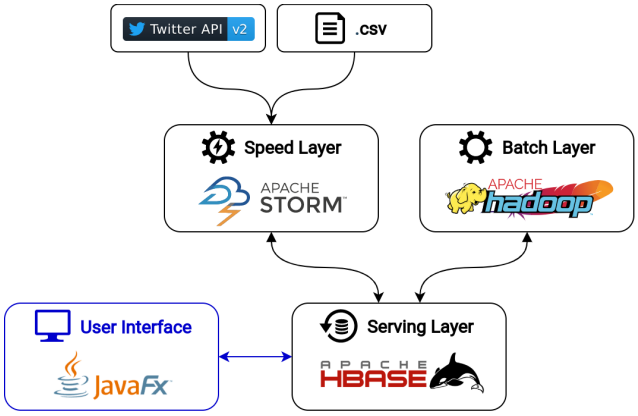

Figure 2. Utilized technologies and structure of the Lambda Architecture for sentiment analysis on tweets.

In order to connect to the stream of uploaded tweets the endpoints offered by Twitter API v2 [5] were used. With this API we are able to set filters

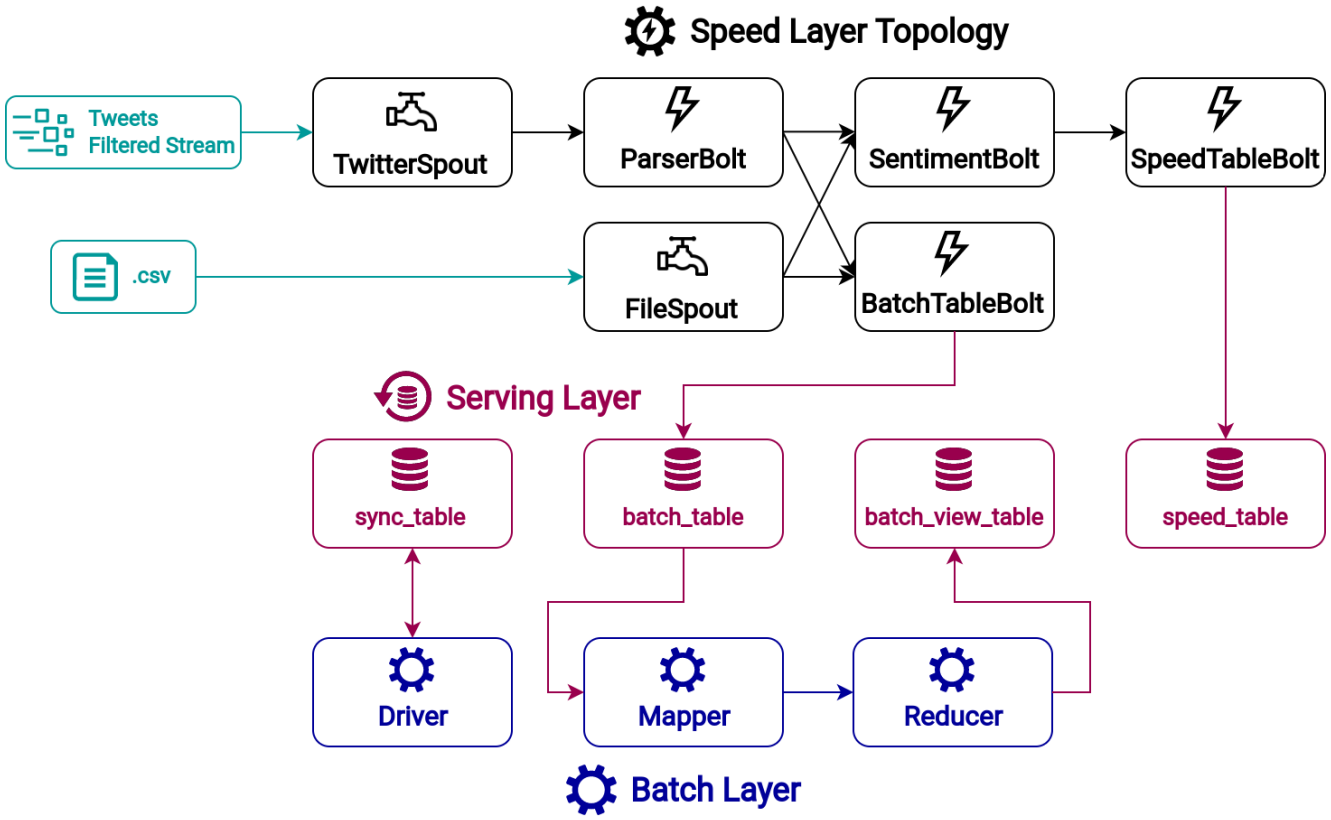

Figure 3. Main components of the three layers of the lambda architecture.

on the stream, in particular we are able to consider tweets which contain the keywords we want to analyze. Moreover, in order to test the architecture with a larger amount of data, we considered tweets from the test-set of the classifier.

The resulting architecture is reported in details in Figure 3, together with the interactions between each component. In the following subsections are reported the implementation details of each layer of the architecture.

3.1. Speed Layer

The speed layer of the architecture was implemented making use of Apache Storm: a distributed stream processing framework, suitable for real-time analysis.

A Storm application processes streams of named tuples. The structure of such application consists of *Spouts*, which generate tuples to be processed, and *Bolts*, that handle and manipulate the tuples. Spouts and Bolts are vertices of a directed acyclic graph named *Topology*, that describes the pipeline of transformations applied to data. In Figure 3 is reported in black the topology in this implementation.

The implemented topology consists of the following spouts and bolts:

- **TwitterSpout**
  This *Spout* connects to the Filtered Stream of tweets making use of Twitter API v2 [5] and emits the tweets that are received. In particular, making use of a dedicated API's endpoint, we are able to set a rule to the stream, gathering only English tweets, containing any of the keywords analyzed. Single tweets, in `.json` format, are emitted to the ParserBolt.

- **ParserBolt**
  This *Bolt* performs parsing of the `.json` tweet, extracting tweet-id, its content and the keywords that are contained, separated by commas. Resulting tuples are sent to the SentimentBolt and to the BatchTableBolt.

- **FileSpout**
  This *Spout* reads the input `.csv` file and emits the tweets that contain the keywords analyzed. In particular, output tuples consist of the tweet-id, its content and the keywords contained in the tweet, separated by commas. These tuples are sent to the SentimentBolt and to the BatchTableBolt.

- **SentimentBolt**
  This *Bolt* makes use of the trained Sentiment Classifier introduced in Section 2 to perform text-based positive/negative classification of the tweet. The emitted tuples contain the tweet-id, the keyword considered and the sentiment estimation, in particular 0 is used for the negative class and 1 for the positive one. Such tuples are sent to the SpeedTableBolt.

- **SpeedTableBolt**
  This *Bolt* writes the tuples produced by the SentimentBolt into the `speed_table`, building the speed view. In particular, the row-key used in the table is the concatenation of the tweet-id and the keyword considered, whilst the row-value is the sentiment estimation.

- **BatchTableBolt**
  This *Bolt* writes raw tweets into the `batch_table` in order to offer these to the batch layer of the architecture. In particular tweets are stored as key-value pairs in which tweet-ids are keys and tweets' contents are values.

It's important to highlight that the speed sayer is the only entry point for raw data (tweets) in the whole architecture. Each tweet, whether it was read from the input file or it was received from the stream, is written into the `batch_table` by the BatchTableBolt.

The synchronization of the speed layer with the batch layer is performed by the serving layer, that is aware of the batch layer status. Such layer, looking at timestamps, offers as a speed view only tweets that are not yet processed by the batch layer.

3.2. Batch Layer

The batch layer of the architecture was implemented making use of Apache Hadoop: a framework for big data processing based on MapReduce [2]. This programming model, introduced by Google, is particularly suitable for big data processing in batch mode over multiple distributed computing units.

An Hadoop program consists of *mappers*, that process the raw data input producing key-value pairs, of *reducers*, that are given multiple key-value pairs that are summarized into new pairs, and of a *driver*, that manages the execution of the MapReduce job.

In this implementation the aim of the batch layer is to process tweets stored in the `batch_table`, in order to offer the overall sentiment results for each keyword. This is done through the pipeline shown in Figure 3 in blue, and here described:

- **Mapper**
  The mapper considers tweets memorized in the `batch_table` and, for each one, it classifies its content through the Sentiment Classifier introduced in Section 2. Each input tweet will eventually contain one or more keywords, as a result the output of the mapper will be, for each tweet, a set of pairs where the key is a distinct keyword and the value is the sentiment estimation.

- **Reducer**
  The reducer receives all pairs with the same key, consequently it is able to aggregate all the sentiment estimations for a given keyword. Just by counting the positive and negative sentiment occurrences, the reducer can write to the `batch_table_view` the overall sentiment for the keyword analyzed.

- **Driver**
  The driver manages the MapReduce job and the synchronization with the other layers of the lambda architecture. In particular it interacts with the `synchronization_table` writing the time-stamps of the beginning and

the end of the batch computation. These values are then used by the serving layer in order to offer updated views of the results.

The batch layer performs its work continuously on all the data that is available at the moment of the beginning of the MapReduce job. As a result the batch views produced are always overwritten in the `batch_table_view`.

3.3. Serving Layer

The serving layer of the architecture was implemented making use of Apache HBase: a non-relational distributed database running on top of HDFS. This layer allows both the speed and the batch layer to store their output views, moreover allows the user to access real-time results, taking care of the synchronization between the two.

As shown in Figure 3 in red, four tables are used in this layer, with the following purposes:

- **`synchronization_table`**
  This table stores the beginning and end time-stamps of batch computations in order to perform synchronization between batch and speed views. In detail, here are stored the start-time of the batch computation, the end-time and the start-time of the previous computation. In such a way we are able to gather only not-expired results when querying the `speed_table`, thus avoiding overlap between the two views.

- **`speed_table`**
  This table stores the output of the speed layer, i.e. tweet-id and keyword as a key and the sentiment estimation as a value. When this table is queried, the serving layer provides the aggregation of the not-expired results. This behavior is achieved looking at the time-stamps stored in the `synchronization_table` and comparing them to the time-stamps of the rows memorized in this table. Doing so we are able to avoid overlap between the speed view and the batch view.

- **`batch_table`**
  This table stores the tweets to be analyzed by the batch layer. These are memorized as key-value pairs in which tweet-ids are keys and their content constitutes values.

- **`batch_view_table`**
  This table stores the latest batch view computed by the batch layer. In particular, each keyword in the analysis is used as a key, and the overall number of positive and negative sentiments are stored as values.

The serving layer in this implementation encapsulates the queries for updating the two views, together with the utilities to get the current views. The User Interface, described in the following section, makes use of the latter in order to keep the visualization updated, through a polling mechanism.

4. User Interface

In order to present the results obtained by the architecture, a GUI was implemented making use of JavaFX.

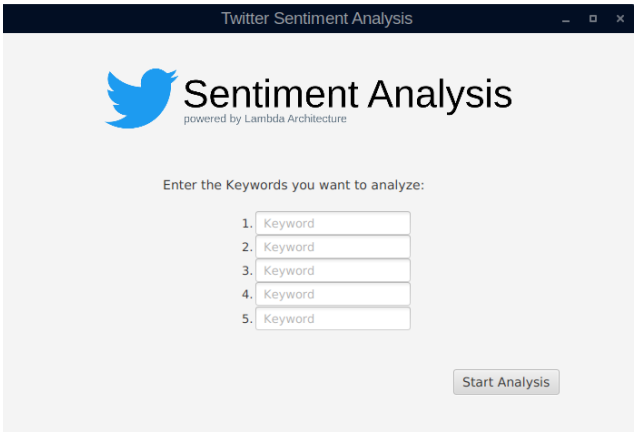

Figure 4. Welcome page of the application. It allows to enter the keywords the user wishes to analyze.

When launching the application the welcome page in Figure 4 is displayed. Here the user is able to enter the keywords to be analyzed. When clicking enter the lambda architecture is started asynchronously through a dedicated method of the serving layer. The keywords are passed to

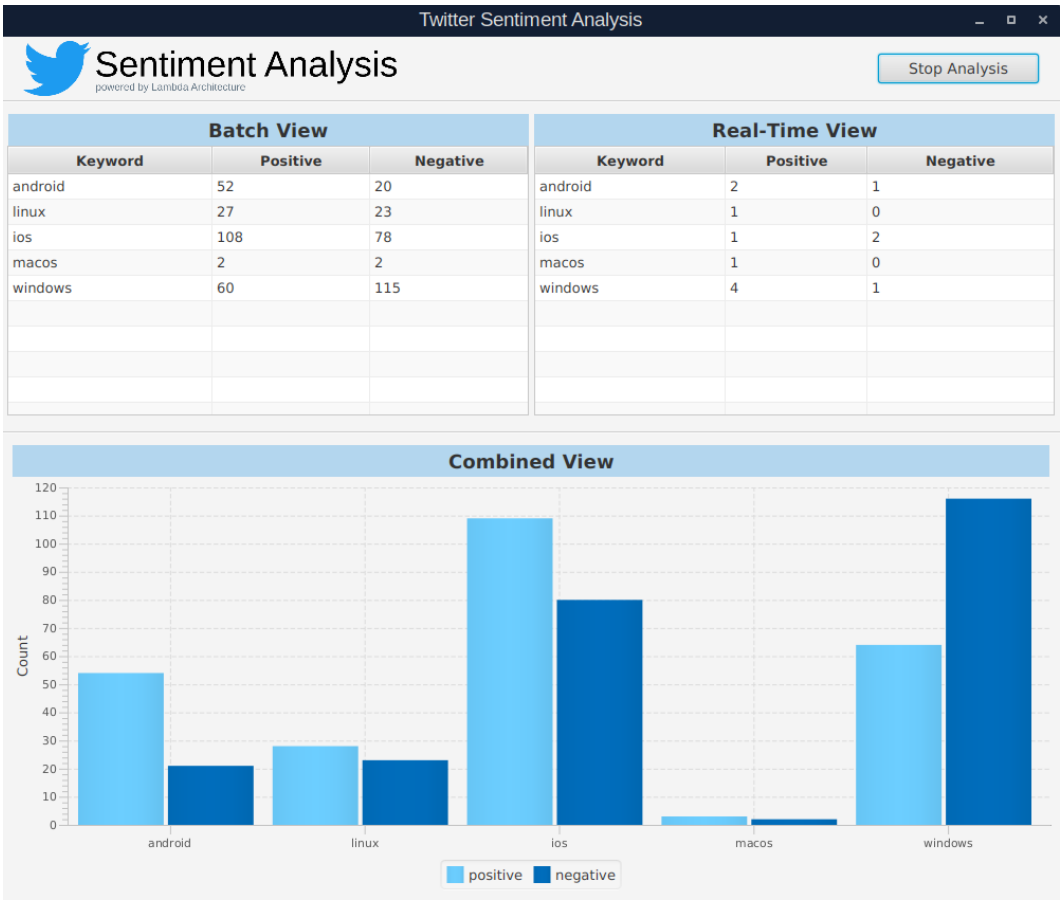

Figure 5. Main page of the application. Here are reported the real-time results of the analysis.

both the speed layer and the batch layer, so that the analysis can begin. In the meantime the main page of the application reported in Figure 5 is displayed to the user.

In order to keep the tables and the charts updated, the `speed_table` and the `batch_view_table` are queried periodically, making use of the functions offered by the serving layer.

On the top-right corner of the main page a dedicated button allows to stop the architecture at any time.

5. Conclusion and further developments

The developed architecture allows to perform real-time sentiment analysis on Tweets making use of a cluster of computing units.

A possible development of the proposed implementation could be to include a more detailed classification of tweets, adding the neutral sentiment class. Moreover, when Tweets contain more than one keyword, a more elaborate classifier could be developed to understand the sentiment with respect to each keyword, thus not considering the overall sentiment.

Finally, in order to perform a better synchronization between the speed layer and the batch layer, the insert operation on the `batch_table` and on the `speed_table` could be synchronized. In such a way overlap between views will never occur.

References

[1] Alias-i. Lingpipe 4.1.2, http://alias-i.com/lingpipe, 2008.

[2] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107113, jan 2008.

[3] A. Go, R. Bhayani, and L. Huang. Twitter sentiment classification using distant supervision. *Processing*, pages 1–6, 2009.

[4] W. Medhat, A. Hassan, and H. Korashy. Sentiment analysis algorithms and applications: A survey. *Ain Shams Engineering Journal*, 5(4):1093–1113, 2014.

[5] Twitter. Api. v2 https://developer.twitter.com/en/docs/twitter-api.

[6] J. Warren and N. Marz. *Big Data: Principles and best practices of scalable realtime data systems*. Simon and Schuster, 2015.