Twitter Sentiment Analysis with Lambda Architecture

Francesco Gradi
francesco.gradi@stud.unifi.it

Federico Màlato
federico.malato@stud.unifi.it

Abstract

A Lambda Architecture is a very reliable and fast way to process large amount of data. The aim of this work is to implement a reliable architecture to perform a sentiment analysis on Twitter, using the Apache suite in a pseudo-distributed mode and Java. After a configuration phase, a bunch of tweets are downloaded with Apache Flume to perform the batch analysis. While Apache Hadoop processes the fetched data, a real time analysis is performed on the same source using Apache Storm, and the results of both procedures are merged into a single table in order to be analyzed. We have also made some little experiments to show the functionality and to appreciate the opportunities and the advantages that such an approach gives.

1. Introduction

In the last decades, Internet has given us the access to an enormous amount of data, which can be processed through Social Media and a bunch of other methods. So, we need efficient ways to process those data. An example could be a distributed architecture: a distributed system is made of several parallel processes that can also run on different machines communicating to each other via messages.

Let's consider Twitter: every second 8500 new tweets are written all around the world [1]. A single PC can't process such an amount of data in order to extract some statistic or some sort of useful information.

There are two paradigms for parallel computation: *batch processing* and *real time processing*. Following the first paradigm means processing a great amount of data previously stored in a distributed file system, while the second uses a data stream and processes immediately the fetched data. Both approaches have pro and cons: batch may be processing old (and so potentially useless) data but has better computational power than real time processing. A good solution is to merge batch with real time to obtain a reasonably fast, fault-tolerant architecture with a high throughput: that is, to obtain a *lambda architecture* [2].

1.1. Lambda Architecture

A Lambda Architecture is made of several layers, and each of them completes a specific task:

- A *Batch Layer* is responsible for the storage of very big dataset and for the computation of those stored data using batch paradigm. In our project, this part has been implemented with Apache Hadoop [3].
- A *Speed Layer* processes in real time an input data stream, filling the time gap generated from the batch process' high latency.
- A *Serving Layer* stores the results in a distributed database that puts together the results of the two previous levels. The Serving Layer also create views based on a query. We achieved this goal using Apache HBase and Apache Drill.

1.2. Motivations

The final goal of the project is to perform a sentiment analysis on Twitter's tweets: due to the size of the dataset and the parallel nature of the problem, using a distributed approach (a lambda architecture is even better) is considered a good strategy.

Sentiment Analysis is a study which aim is to understand the sentiment of a group of people with respect to a specific topic, be it politics, economics, a product feedback and so on.

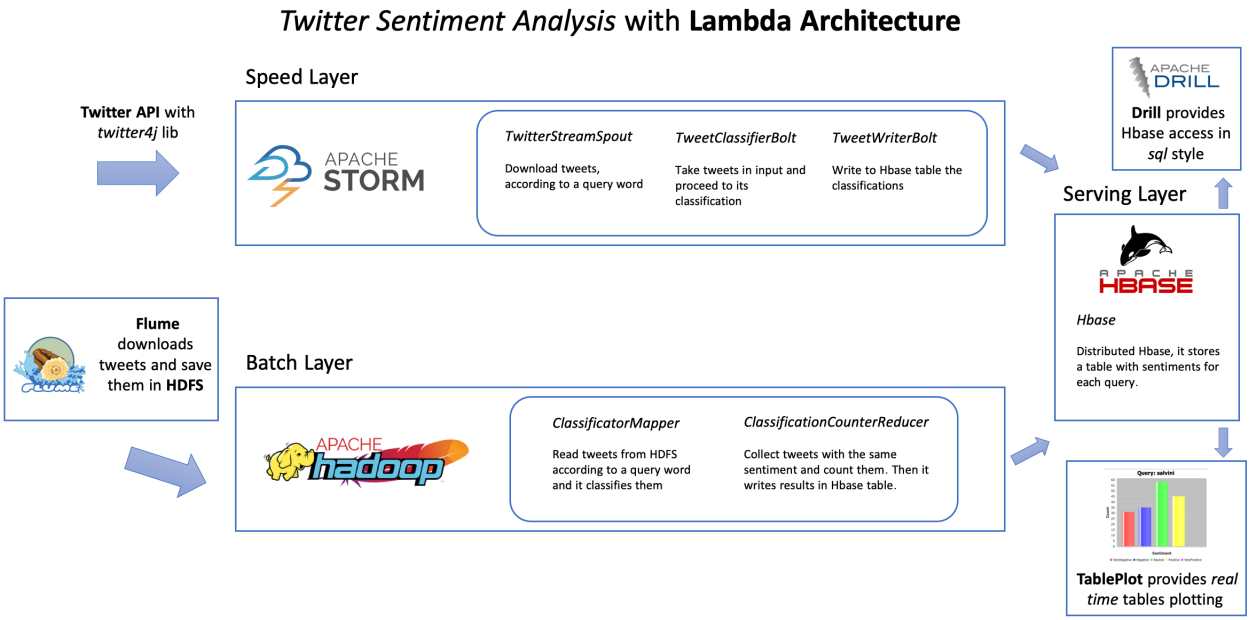


Figure 1. The Lambda Architecture scheme.

1.3. Classification

In this project, the classification is carried using *json* dictionaries. In these dictionaries, every word is associated with a sentiment value (in particular it is a list of 10 bits); adding positive and negative sentiments we obtained an integer sentiment value for each term. Repeating this process, we could classify whole sentences, in our case tweets. Clearly this method is language dependent, it is necessary a dictionary for each language: we have choosen to implement *English* and *Italian*.

2. Batch analysis

The program has been coded in Java, using some open source softwares from the Apache suite. Apache Flume fetch the data from the web, Apache Hadoop process them and Apache HBase and Apache Drill accomplish the storing task.

2.1. Apache Flume

As we have just said, Apache Flume's task is to retrieve great amount of data from a web server and store them on a distributed file system [4]. Flume has been configured to fetch data from Twitter using the *Twitter4j* [5]; we only had to sign in to the Twitter developers website and to ask the permission to download tweets for non-commercial use. We have been then granted some access tokens and an online app allowed to access all the tweets from every Twitter user. Finally, using a Cloudera *.jar* file [6], we have been able to download a subset of the tweets containing some keywords and to store them on the Hadoop's HDFS. This distributed file systems stores data on DataNodes, a sort of shared memory between different machines, and retrieves them efficiently through a NameNode. In order to increase reliability, it is possible to store several copies of the data.

At this point, we have a folder on the HDFS with a bunch of *.json* files, each one containing a certain number of tweets, one tweet per line, stored as a *JSON Object*.

2.2. Apache Hadoop

The core of the batch analysis is Hadoop: it reads tweets from the HDFS and processes them following a Map-Reduce paradigm [7]. The program needs two arguments: the folder in which the data are stored and a query.

After the initialization step in the *Run()* method, the Mapper object gets a line of a file in the folder as an input. Due to the fact that a single line represents a *JSON Object*, we've been using the *SimpleJSON* library to parse them and to extract the "text" field. Then the program classifies the tweet's text: if the query is not contained in a text, the corresponding tweet is not classified (i.e. it will produce an output *('Neutral', 0)*); otherwise, the Mapper compute the

score of the text. The tweet's text is classified in one of the five categories we chose (*VeryNegative, Negative, Neutral, Positive, VeryPositive*) following a simple rule: if the overall sentence score is below -3 the sentence is classified as *VeryNegative*; if the score is between -3 and -1 then it is a *Negative* sentence; else if the score is exactly 0 the sentence is *Neutral*; if the score is between 1 and 3 then the tweet belongs to the *Positive* class; finally, if the score is above 3 the sentence is *VeryPositive*. This way, each Mapper returns a double *(<belongingClass>, 1)*.

Each Reducer gets those tuples as input and sums the values for each tuple: after that, it reads the row corrisponding to the sentiment of an existing HBase Table (named after the query) and updates the value of the counter.

2.3. Apache Hbase

We have created the distributed database with Apache HBase [8]: during the Hadoop initialization there are also some calls to various HBase's methods, in order to create (or retrieve, if it exists already) a table according to the current query. This table is named after the query, each row is associated with a sentiment class and a value. Once the table has been created, it is read from the Reducer and continuously updated according to the tweet processing's results: a Reducer can call a *get()* method to retrieve the row's name that matches the belonging class of the tweet, get the current value, adding it to the input's value and overwrite the old table's value with the result.

2.4. Apache Drill

Since the HBase interface is somewhat primitive, we decided to use Apache Drill to make it more user-friendly. This way we could read from the HBase table through a simple and standard *SELECT* query, just like we would have done with a standard SQL language.

3. Real time analysis

3.1. Description

Another module of the Apache suite is Storm, which aim is to fetch data in the fastest way possible from the web and perform some kind of analysis on them in real time [10]. To achieve this, a Storm program defines a pipeline through which data flows, and it operates using two basic operators: *spouts* and *bolts*.

A spout is useful when the data still has to be fetched from the source (usually some website). Storm provides a basic interface class that must be implemented in order to perform a specific task, using the three core methods *open()*, *execute()* and *declareOutputFields()*. The first method, *open()*, configures the spout: you can think of it as a sort of constructor method for this kind of object. *execute()* is the operative method: at this time the data is fetched, can be parsed in some way (accordingly to the final goal) and is then emitted to the next operator in the pipeline. Finally, *declareOutputFields()* is a configuration method that tells the next object in the pipeline what is the format of the data that it will get as input. Similarly, a bolt is like the data processing object of the pipeline: it is necessary when the data has already been fetched and can receive input data either from a spout or another bolt. Like a spout, it is built around three core methods, *prepare()*, *execute()* and *declareOutputFields()*. From last to first, *declareOutputFields()* is the exactly same method described above and specifies the format of the propagated data. *execute()* also has the same role of the spout's *execute()* method, but this time it can be used to perform a much wider amount of tasks such as counting the words in a set of sentences, storing the input data to a database and so on. Last but not least, the *prepare()* method is used as the bolt's constructor and configuration method.

The key concept of Storm is the *topology*, that is, a description of the entire pipeline. Each Storm program relies on a topology which has to be really clear and precise in order to avoid mistakes and errors. In the *main()* function of the program a *TopologyBuilder* object is instantiated and, using the *setBolt()* and *setSpout()* methods, the topology is generated. These two methods also require another key concept of the Storm architecture: grouping. Whenever a spout or a bolt is instantiated in the topology, a grouping has to be specified. Grouping define the way the data are propagated through the pipeline's levels. There are several kind of grouping, but we will focus on the two main types: shuffle and field grouping. The difference between these two is the following: while shuffle grouping distributes the output tuple's elements to random operators of the next level in the pipeline, the field grouping allows to give a specific part (or *Field*) of the output tuple to one specific operator. For example, in a simple two-level topology with one spout and two bolts (let's call them "boltA" and "boltB") suppose a data format ['id', 'number', 'text']. Shuffle grouping will distribute randomly those three fields across the two bolts, while if we have that "boltA" requires an id to perform his task and "boltB" needs the whole tuple, we can say that "boltA" must receive the field 'id' of the spout's output tuple just by setting a field grouping between the spout and "boltA" on the 'id' field. For instance, we will also say that there is a field grouping between

spout and “boltB”, where the fields are ‘id’, ‘number’ and ‘text’.

3.2. Storm structure

As explained above, our goal was to fetch tweets from a Twitter stream and process them in order to perform a sentiment analysis. To achieve this, we’ve implemented the following topology:

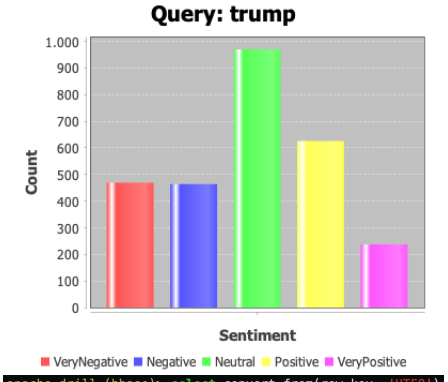
- *TwitterStreamSpout*: First, we’ve created a spout that actually open the Twitter stream and emits only the text of the parsed tweet. Since we’ve had some issues with the *Twitter4j* API, that allowed a maximum number of requests in a 15-minute time span, we’ve set a single spout that makes only one request every 5 seconds. If the fetched tweet is different from the previous one, then it is emitted to the bolts. Otherwise, the bolts will just wait for another 5 seconds period. In order to avoid unclassifiable tweets, we’ve also set a language filter through the method *setLang()*.

- *TweetClassifierBolt*: The first bolt of the architecture receives the current tweet’s text as input field from the spout. Its aim is to split the sentence into single words and classify them accordingly to a vocabulary. In the prepare() method, the *TweetClassifierBolt* reads the two *.json* vocabulary files and updates the corresponding attributes vocaublaryIt and vocabularyEng. After that, during the *execute()* call, the input sentence is splitted into words and to each word a score is assigned by comparison with the dictionaries entries. The total score of the sentence is given by the sum of the scores of the single words. This value is emitted to the next level of the pipeline.

- *TweetWriterBolt*: The last bolt in our topology uses the score to classify the tweet’s text in the same way we saw for the batch analysis. This bolt also creates (if it doesn’t exists already) an HBase table with the same name of the query and increments the table entry which reflects the belonging class of the current tweet. This way we can have a good approximation of the Twitter users’ sentiment with respect to a specific topic.

4. Limitations

The main limitation is given from the *Twitter4j* API, as already stated before: this API allows developers to perform a maximum of 255 requests to Twitter every 15 minutes. So we’ve had to deal with this limitation by




```
apache drill (hbase)> select convert_from(row_key, 'UTF8')
 as sentiment_id, convert_from(trump.number.value, 'UTF8')
 as number_id from trump;
+--------------+------------+
| sentiment_id | number_id  |
+--------------+------------+
| Negative     | 464        |
| Neutral      | 970        |
| Positive     | 626        |
| VeryNegative | 469        |
| VeryPositive | 237        |
+--------------+------------+
5 rows selected (0.401 seconds)
```

Figure 2. Results with histogram and Drill query on *"trump"*

introducing a delay in the requests: a single spout performs a request every 5 seconds. This limitation slows a lot the whole process and doesn’t let Storm, which is widely used because of its speed in completing the task, express its full potential. Anyway, since our goal was to implement a Lambda Architecture, this issue has only the effect to produce a lot less data and so doesn’t affect our work. Another limitation is given by the vocabulary: we’ve used a subset of the NRC Emotion Lexicon [11], which allows us to classify tweet in 105 different languages. Since the search and comparison in the classify task would have been mastodontic for all those idioms, we’ve chosen to keep only the English and the Italian lexicons, and for each language there are about 11000 different words. Also, these lexicons does not consider humor, so a humoristic tweet is likely to be classified as a tweet of the opposite class. For example, the word “tremendous” is associated with a negative score but some tweet could state that someone has generated “a tremendous hype” around some commercial product. If this tweet only contains the word “tremendous”, it will be considered a negative tweet, while considering the whole context would produce a positive result. Anyway, given those limitations, we’ve ensured that the overall behaviour of the program is a good approximation of the true sentiment by performing some experiments.

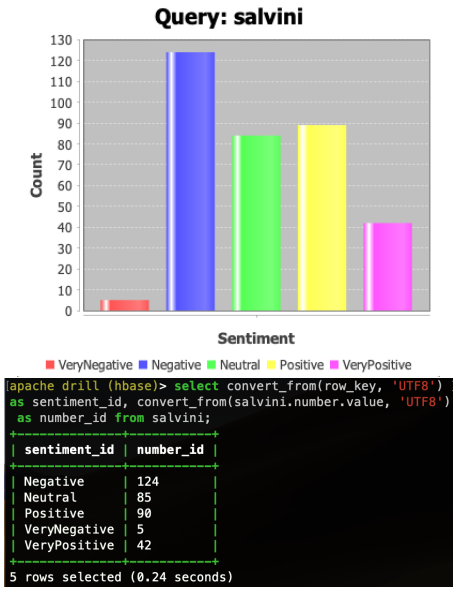


Figure 3. Results with histogram and Drill query on *"salvini"*

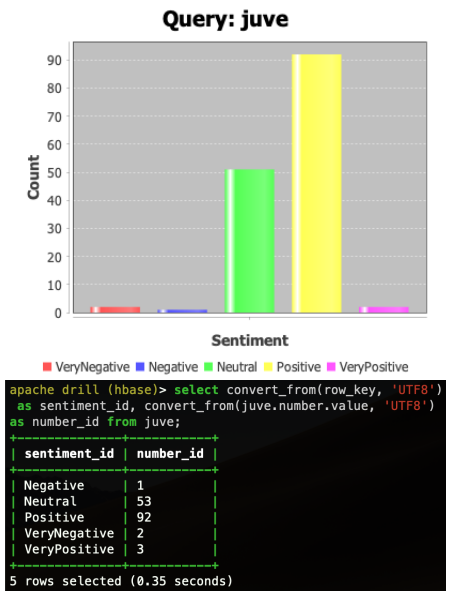


Figure 4. Results with histogram and Drill query on *"juve"*

5. Experiments

We've run our architecture on two very different topics, in order to test the spectrum of words in the dictionaries and to show that the program result doesn't depend on the topic of the query. We've encountered some problems while choosing the right queries: some words may be too general for either Hadoop or Storm (like "iPhone", which gives us tweets from all around the world with a consequent huge amount of *Neutral* tweets in the Hadoop classification) or too specific (like "computer", that retrieves one tweet every 2-3 minutes when limited to the two chosen languages in Storm). Since we needed quite a number of data in the shortest time possible, we focused our attention on today's hot topics. After some trials, we agreed that "salvini" and "juve" were the best results for the italian lexicon, while "trump" was a very good query for the english vocabulary. Results visualization was achieved through Apache Drill and a simple Java histogram that draws a graphic according to the data stored in the HBase table. For each query we let Flume run for about 30 minutes, and we got about 40MB of tweets for "salvini" and "juve", while "trump" gave us nearly 300MB of tweets to analyze. After that, we let a day pass and then started both Hadoop and Storm on the same query.

6. Results

Our results were extremely limited (especially due to the *Twitter4j* situation) and query driven, so we won't put the attention on the numbers, but on the general behaviour of the architecture. From a politics-related topic we could expect a 50/50 division between consensus and dissensus referred to a single politician, and that's exactly what happened: in both 2 and 3, despite the high number of *Neutral* tweets (the vocabularies were limited, as we said above, plus there are a lot of news channels talking about those personalities and they should be neutral) we can see that the sum of *VeryNegative* and *Negative* tweets is comparable to the sum of the opposite classes. Thinking of the football club Juventus, that currently is (with Francesco's heavy heart) a very important italian football club, we can still expect a high number of *Neutral* tweets due to the news channels, but also a great number of positive tweets since the club performed really well in the last season, as confirmed from the results shown in 4.

7. Future work and conclusions

In conclusion, we must say we felt a bit privileged to be able to see the general sentiment about some topic so close to us. Even if our architecture has some clear limitations, we understood completely the importance of retrieving data efficiently and extracting useful data from them. An interesting future work could be using a larger vocabulary and perhaps introducing a context-sensible text analyzer in order to recognize some figures of speech and maybe humor, leading to an even better result.

References

[1] *https://www.internetlivestats.com/one-second/*

[2] Marz, Nathan; Warren, James. Big Data: Principles and best practices of scalable realtime data systems. Manning Publications, 2013.

[3] Bijnens, Nathan. "A real-time architecture using Hadoop and Storm". 11 December 2013.

[4] *https://flume.apache.org/*

[5] *http://twitter4j.org/en/index.html*

[6] *https://github.com/cloudera/cdh-twitter-example*

[7] *https://hadoop.apache.org/*

[8] *https://hbase.apache.org/*

[9] *https://drill.apache.org/*

[10] *https://storm.apache.org/*

[11] *http://www.saifmohammad.com/WebDocs/NRCemotionlexicon.pdf*