Comparison between a sequential and a distributed version of a Twitter sentiment analysis tool

Emilio Cecchini
emilio.cecchini@stud.unifi.it

Abstract

In this paper I will analyze the advantages in speed and efficiency obtainable in implementing a Twitter sentiment analysis tool with Hadoop. The main goal of this study is not to build a perfect classifier, but it is instead to show the speedup that you can get using a distributed algorithm versus a sequential version. To build the sentiment classifier the library LingPipe is used, so the details of that classifier will be ignored. The distributed version is written in Hadoop with the MapReduce framework.

1. Introduction

Sentiment analysis involves classifying opinions in text into categories like *positive* or *negative* often with an implicit category of *neutral*. In this project, for simplicity, we will ignore the *neutral* category. All the entire part of building the classifier and getting results out of it is done with the LingPipe library, that is a Java library to perform machine learning tasks such as text classification and natural language processing.

First of all there is a *training* phase, where the model is trained to recognize a tweet as negative or positive. This phase is only implemented in a sequential version. Once the model is built and trained, there is the *classification* phase, where many unclassified tweets are passed as input to the model. This phase is implemented in two different versions: a sequential version and a distributed version in Hadoop with the MapReduce framework.

2. Training

The training process is done with a dataset containing tweets already classified. The model reads the tweet texts and their sentiment and try to learn how to perform the classification.

The process of training the model must be computed in a sequential way because every tweet and its associated sentiment must be processed by the same unique model. A parallel version could be hypothetical possible but in that case reading and processing the tweet must be encapsulated in a critical section and therefore all the advantages of parallelization would be lost.

Thanks to the LingPipe library, the algorithm of training the model is straightforward:

Algorithm 1 Training the model

```
m = initializeModel()
for each tweet t do
    handle(m, t)
end for
saveModel(m)
```

The function `saveModel()` save the model in a file so that it can be loaded later by a classifier to perform a classification.

3. Classification

The classification phase can be performed in a distributed way, so I have implemented a sequential version and a distributed version in Hadoop.

3.1. Sequential version

With the LingPipe library it is very easy to implement a sequential algorithm that classify tweets.

The first thing to do is to load the trained model. This process may take a long time if the model has been trained with a very large dataset. Loading the model can not be parallelized, so it is suggested to keep the file as small as possible to prevent it to become the bottleneck of the program.

After loading the model, the program iterates over the tweet texts and uses the classifier to predict their sentiment. During the iteration, the program keep tracks of the number of negative tweets as well as the number of positive tweets with two counters. The output of this algorithm are the two counters from which the general sentiment can be derived.

Algorithm 2 Sequential classification

```
m = loadModel()
p = 0
n = 0
for each tweet t do
    s = classify(m, t)
    if isPositiveSentiment(s) then
        p = p + 1
    else
        n = n + 1
    end if
end for
return p, n
```

3.2. Distributed version

The classification process can be improved by implementing a distributed algorithm. This project uses Hadoop with its *Hadoop Distributed File System* (HDFS) in order to better handle large amounts of data.

The distributed algorithm is implemented with the *MapReduce* framework, that is a programming model for processing and generating big data sets on a cluster of computers. A MapReduce program is composed by a *map* phase, which performs a filtering operation of the input data, and a *reduce* phase, which collects the results of the map function and generate an output.

In order to perform the classification of a tweet the model must first be loaded. This implementation exploits the *Distributed Cache* tool to make the file where the model is stored accessible to all the map tasks. The Distributed Cache is a read-only data structure from which each application in the MapReduce framework can read files. Each map task loads its own model so there are not synchronization issues such as race conditions.

Once the model is loaded from its file we can perform the map phase, which consists of mapping each tweet text to its classified sentiment.

Algorithm 3 Map function

```
m                                   ▷ Loaded model
t                                   ▷ Tweet text
k = null
s = classify(m, t)
if isPositiveSentiment(s) then
    k = positives
else
    k = negatives
end if
return k, 1
```

As you can see, the map function is very similar to a single iteration of the sequential algorithm. The input for the map function is a single tweet text, this text is classified with the previously loaded model and then the output key is generated depending on the classified sentiment. There could be only two different output keys: *positives* if the tweet has been classified as positive and *negatives* if the tweet has been classified as negative.

The reduce phase is quite straightforward. It takes as input the key, which as already explained can be *positives* or *negatives* and a list of as many ones as many tweets have been classified in that category.

Algorithm 4 Reduce function

```
k                          ▷ The key (positives or negatives)
v                                   ▷ List of ones values
c = 0
for each element in v do
    c = c + 1
end for
return k, c
```

The only thing that the reduce function has to do is to count the number of elements that are in the list associated with the input key.

Since the keys that the mapper passes to the reducer are always two (*positive* or *negatives*), the number of reducers can be specified at compilation time to two. Moreover, to optimize the operations of dispatching the mapper output, a *partition* function is implemented. The purpose of this function is to assign the output key of a map function to a specific reducer. Since we have set the number of reducers to two, the output of this function is 1 or 0 dependending if the key is *positives* or *negatives*.

Algorithm 5 Partion function

```
if k == "positives" then
    return 1
else if k == "negatives" then
    return 0
else
    return -1                                   ▷ Error
end if
```

In this way we have one reducer that counts the number of tweets classified as positives and another reducer which counts the number of tweets classified as negatives.

4. Speedup analysis

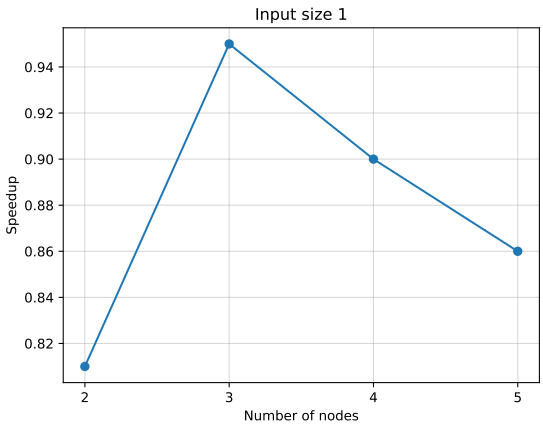
To compare the performance of a sequential algorithm with a parallel version, the main measure used is the *speedup*, that is the ration between the execution time of the sequential version and the execution time of the parallel version with the same input.

$$S = \frac{t_S}{t_P} \tag{1}$$

Ideally, the speedup should be the number of processor used to perform the parallel version. For a distributed algorithm, it should be the number of nodes in the cluster. In this section we report the results of some experiments in order to analyze the speedup of the classification algorithm in its two versions: sequential and distributed.

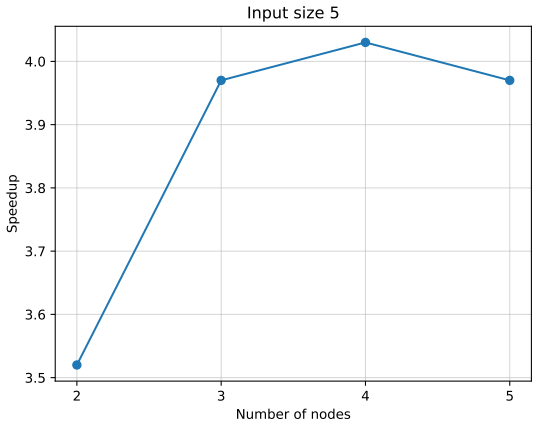
The inputs for these experiments are some text files containing the tweet texts. The size of a single text file is about the size of a HDFS block, that is 128 MB by default. This make it easier to control the number of splits into which the input will be divided. In order to see the trend of the speedup, we tested the distributed algorithm in a cluster with a different number of nodes, more precisely from two to five nodes. Each node has an Intel Haswell CPU with four cores and 8 GB of RAM.

In this plot there are the speedups obtained with an input consisting of a single text file, that is a single HDFS block.



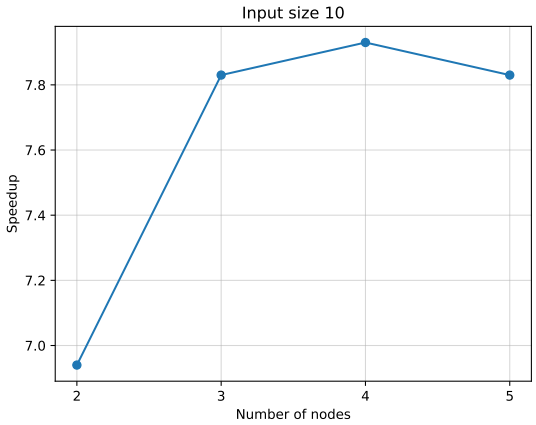
As you can see, with an input of only one file, the distributed version is slower than the sequential algorithm because the speedup is minor than one. The reason of that is because the time spent in managing the various tasks and moving the data in the distributed environment is too much to get an improvement in the overall execution time.

Here we report the speedups with an input of five files, that are five HDFS blocks.



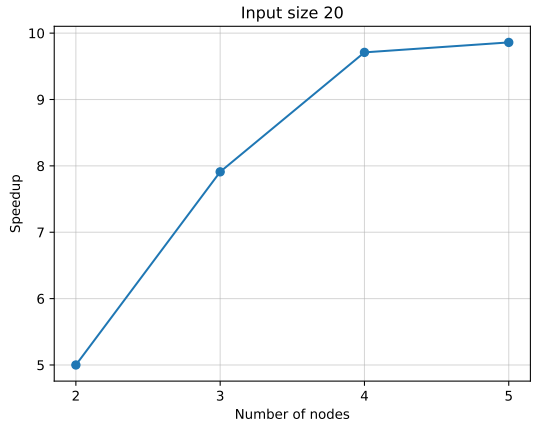
Increasing the size of the input we can see a big improvements in the performances. With only two nodes you have a speedup of 3.5, with three, four and five nodes the speedup stabilizes around 4. It means that with an input of that size, that is five HDFS blocks, until three nodes you get a *super-linear* speedup, with four nodes you get a *linear* speedup and with five nodes you get a *sub-linear* speedup.

Here we report the speedups with an input of ten files, that are ten HDFS blocks.



We can notice that the trends is the same as in the previous plot, but the obtained speedups are higher.

Here we report the speedups with an input of twenty files, that are twenty HDFS blocks.



With an input size of about 2.5 GB we get better performances with a larger number of nodes, such as four or five.

From these simple experiments we can confirm that Hadoop is designed to work with big data, because if the input size is too small we have a lot of overhead due to tasks management. Apart from the first case, where we passed only a single file to the classifier, the speedups are always super-linear, that means that the speedup is alway greater then the number of nodes in the clusters.