# Twitter sentiment analysis with Hadoop

Emilio Cecchini

June 13, 2019

Università degli Studi di Firenze
*emilio.cecchini@stud.unfi.it*

## Overview

## Sentiment analysis

## Sentiment analysis

- 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, so we will ignore the internal details of the classifier.

## Training phase

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.

## Classification phase

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.

## Training the model

## The training process

- 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.

## The training process

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

**Algorithm 1** Training the model

**Input:** a list of tweets $T$ with text and sentiment.

```
$m = $ initializeModel()
for each $t \in T$ 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.

## Sequential classification

## Sequential classification

- 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.

## Sequential classification

**Algorithm 2** Sequential classification

**Input:** a list of tweets $T$ with text only.
**Output:** two values $p, n \in \mathbb{N}$ that are the number of tweets classified as positives or negatives respectively.

```
$m$ = 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** $p$, $n$
```

## Hadoop and the MapReduce framework

## Hadoop

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.

## MapReduce framework

- A MapReduce program is composed by a *map* phase and by a *reduce* phase.
- The map function processes each element of the input and returns a value associated with a key.
- The reduce function takes as input the keys returned by the map function and their associated values and then it combines them in a single value.

## Distributed classification

## Distributed Cache

- 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.

## Map phase

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

```
Input: a trained model $m$, a tweet text $t$
Output: a key $k$ and a value $v = 1$

  s = classify($m$, $t$)
  if isPositiveSentiment($s$) then
      $k$ = *positives*
  else
      $k$ = *negatives*
  end if
  return $k$, 1
```

## Reduce phase

The reduce function takes as input the key and a list of as many ones as many tweets have been classified in that category.

**Algorithm 4** Reduce function

```
Input: a key $k$ and a list $V$ of values $v = 1$
Output: a key $k$ and a value $c$

  $c = 0$
  for each element in $v \in V$ 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.

## Partitioner

- 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.
- 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.

## Partitioner

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

**Input:** a key $k$

**Output:** a value $v$

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

## Speedup

## Speedup

The *speedup* is the ratio between the execution time of the sequential version and the execution time of the parallel version with the same input.

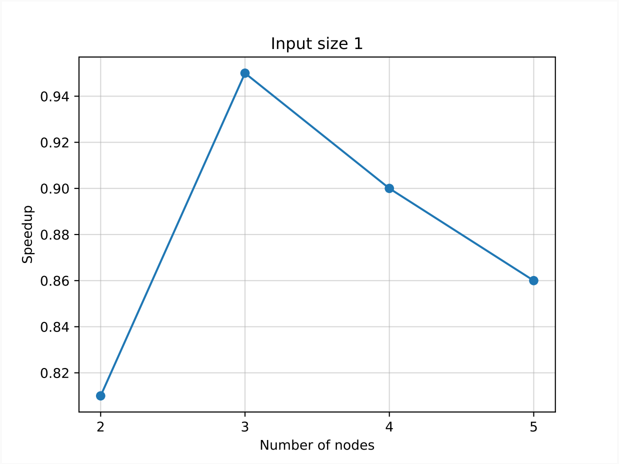
$$S = \frac{t_S}{t_P}$$

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.

## Experiments setup

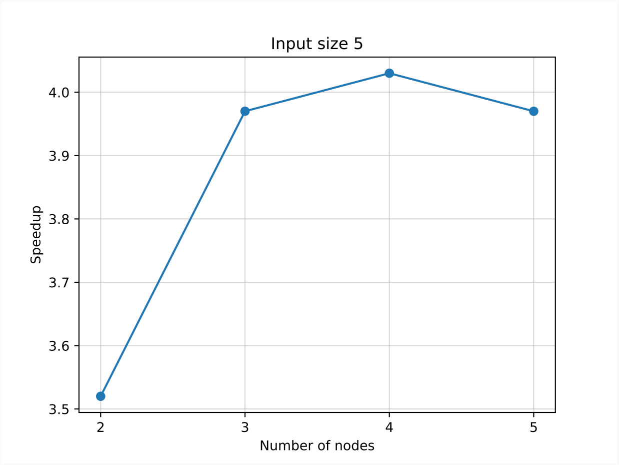
- The inputs for these experiments are some text files containing the tweet texts.
- The size of a single text file is about the size of a HDFS block, that is 128 MB by default. This make it easier to control the number of splits into which the input will be divided.
- In order to see the trend of the 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.

## Input size = 1

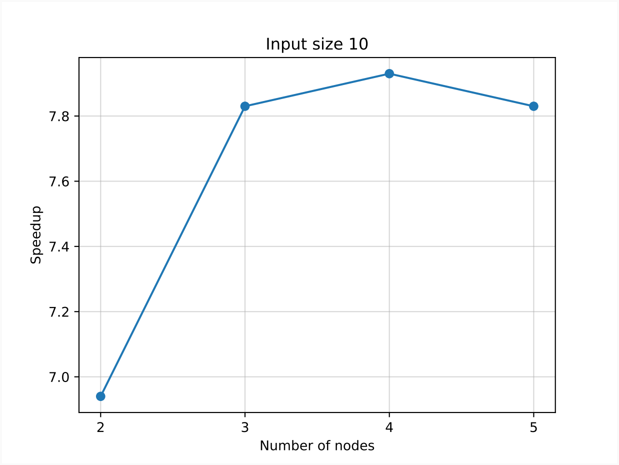


Input size 1
Speedup
Number of nodes
0.82
0.84
0.86
0.88
0.90
0.92
0.94
2
3
4
5

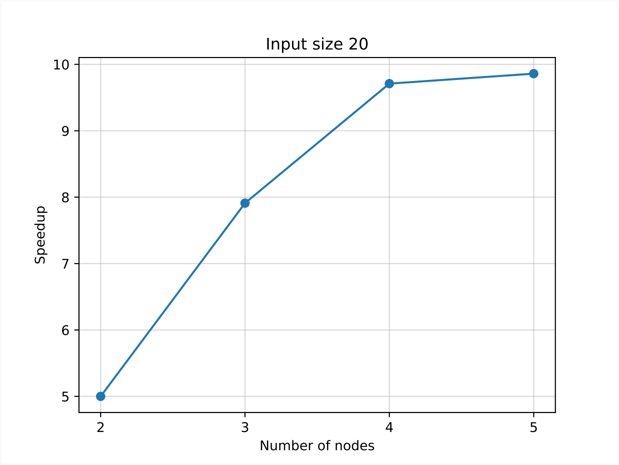
## Input size = 5



## Input size = 10



## Input size = 20



## Conclusion

- 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.