TÜBİTAK

Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2016) 24: 1560 – 1570
© TÜBİTAK
doi:10.3906/elk-1311-128

Scalable sentiment analytics

Aslan BAKİROV, Kevser Nur ÇOĞALMIŞ*, Ahmet BULUT
Department of Computer Science, İstanbul Şehir University, İstanbul, Turkey

Received: 15.11.2013 • **Accepted/Published Online:** 03.04.2014 • **Final Version:** 23.03.2016

Abstract: Spark has become a widely popular analytics framework that provides an implementation of the equally popular MapReduce programming model. Hadoop is an Apache foundation framework that can be used for processing large datasets on a cluster of computers using the MapReduce programming model. Mahout is an Apache foundation project developed for building scalable machine learning libraries, which includes built-in machine learning classifiers. In this paper, we show how to build a simple text classifier on Spark, Apache Hadoop, and Apache Mahout for extracting out sentiments from a text collection containing millions of text documents. Using a collection of 7 million movie reviews taken from IMDB, a Bayesian classifier was learned to predict sentiments for test reviews. Separate classifiers were learned on both Spark and Hadoop, i.e. our contenders for scalable sentiment analytics. Our empirical results showed that the sentiment learning task on Spark ran almost 10 times faster than the learning task on Hadoop.

Key words: Sentiment analysis, MapReduce, Spark, Hadoop, Apache Mahout

1. Introduction

With the advent of social networks, forums, and blogs, the amount of data on the Web has increased rapidly, resulting in an information explosion. Internet users make purchases online, listen to music, or watch a movie, and later on they make comments about their purchases, indicate their musical preferences, and write their opinions about the movies they recently watched. Raw user data do not provide much information unless the information is explicitly cultivated. Data mining broadly refers to the process of extracting information from raw data. Data mining is used for a variety of information discovery tasks such as classification, clustering, and regression. Since actual task implementations analyze the entire set of data in order to find a pattern, the run time of these algorithms depends on the size of the dataset. Handling large amounts of data requires the use of special-purpose compute clouds. The MapReduce (M/R) programming model provides one such solution. Hadoop was one of the first platforms to provide an implementation of M/R on a compute cluster. A relatively new implementation of M/R called Spark has been shown to offer performance benefits of up to ten times compared to Hadoop on certain machine learning tasks except Bayesian classification [1]. In this paper, we study the efficiency of naive Bayes classification on Spark compared to the alternative platforms. We chose to study how to extract sentiment from review data, and we used a Bayesian classifier for this purpose.

The rest of the paper is organized as follows: the related work is discussed in Section 2. In Section 3, we describe the methodology in multiple steps: preprocessing of the data and the basics of the classifier learned. In Section 4, we provide the details of our Spark implementation and present our experimental results. In Section 5, we compare our Spark implementation with Hadoop and Mahout. Finally, we give avenues for future work and emphasize key takeaways from our study in Sections 6 and 7, respectively.

*Correspondence: nurcogalmis@std.sehir.edu.tr

2. Related work

Apache Mahout [2] is an open source library for performing classification, clustering, and recommendation using Hadoop. MLbase is yet another alternative on the same front [3]. Pang et al. [4] used the IMDB movie dataset for sentiment analysis. They used naive Bayes, maximum entropy, and support vector machine (SVM) approaches for classification. However, their study was done on a single compute node and did not address how to scale computation as the data themselves scale. The text features extracted included bag of words, bigrams, and part of speech tags. Their study showed that SVM with unigram features had the best performance.

Elsayed et al. [5] proposed an M/R algorithm for finding pairwise document similarity for large document collections. They used a cluster of 19 worker machines with a dual-core, 4 GB of memory, and 100 GB of disk space each. Their algorithm was implemented as two M/R jobs: the first job was used to index documents for finding out a list of document IDs that contain a given term and its associated term weight. The second job was used to calculate pairwise similarity scores. The experiments showed that the running time of their approach scaled linearly with the number of documents.

Khuc et al. [6] provided a method for analyzing sentiment in Twitter data using Hadoop. The authors created their own lexicon suitable for tweets, which included emoticons. They used the lexicon as the first classifier and logistic regression as the second classifier. Their experiment was done on 5 nodes in the Amazon EC2 cluster with 2 virtual cores and 1.7 GB of memory. An experiment to create the lexicon was carried out on 100K, 200K, and 300K tweets. For the 300K-scenario, it took 600 min to build a lexicon on 5 machines. Their lexicon-and-learning-based classifier took 20 min to analyze the sentiment of 2.5 million tweets.

Hunter et al. [7] migrated their millennium traffic project from a single machine to multiple machines. They used Spark as it allows M/R and iterative algorithms at the same time. A special-purpose data structure called resilient distributed data is used to share large data items between cluster machines for collaboration. The migration to Spark improved the run time of their system by 2.8 times.

3. Methodology

3.1. Preprocessing of the data

For downstream tasks that expect meaningful and cleaned-up data, our dataset is preprocessed as follows:

- All text is lowercased.
- Punctuation symbols are removed.
- Hyphenated word groups are separated.
- Stop-words such as "a, an, the, they, so, much" are removed using the natural language toolkit called NLTK [8].
- HTML tags are removed.
- In the original dataset, the score information is numeric and is in the range of [1.0, 5.0]. As a preprocessing step, a review is categorized as negative if its review score is less than 3.0; otherwise (if its review score is greater than or equal to 3.0) it is categorized as positive.

3.2. Naive Bayes classifier

Naive Bayes is one of the simplest machine learning algorithms used for text classification. The Bayesian formula [9] is:

$$P(C = c_k|X = x) = P(C = c_k) \times \frac{P\,(X = x\,|\,C = c_k)}{P\,(x)}, \tag{1}$$

where C is a class, X is a feature, and $P(C = c_k|X = x)$ is the probability of the text that has feature value of x for X being in class c_k. For each text, two probability values are computed, i.e. one per class. Each text consists of a set of words w_i as shown in Table 1.

Table 1. The depiction of raw reviews in the IMDB dataset.

Review	Words in review	Score
R_1	$w_1 w_2 w_{27} w_{4509} w_{22} w_{509}$	2.0
R_2	$w_{17765} w_{112} w_{2000} w_{4509}$	5.0
...	$w_{15} w_{112} w_{3329} w_{422} w_1$	4.0

During the preprocessing, every review is assigned to a class. A review having a score of less than 3.0 is tagged as negative; otherwise, it is assigned to the positive class. Therefore, the reviews are converted into new structures as shown in Table 2.

Table 2. The depiction of reviews in the IMDB dataset with class assignment.

Review	Words in review	Sentiment
R_1	$w_1 w_2 w_{27} w_{4509} w_{22} w_{509}$	Negative
R_2	$w_{17765} w_{112} w_{2000} w_{4509}$	Positive
...	$w_{15} w_{112} w_{3329} w_{422} w_1$	Positive

The training set consists of 4 million texts. Out of this 4 million, there are 2.4 million reviews in the positive class and 1.6 million reviews in the negative class. For each word, two counts are computed and stored: the first count represents the number of positive reviews that contain the word, and the second count represents the number of negative reviews that contain the word as in Table 3.

Table 3. Words and their occurrence numbers in each class.

Word	Positive review count	Negative review count
w_1	45,600	120,000
w_2	72,250	50,000
w_3	22,500	69,900
w_{43}	90,400	23,220

The formula in Eq. (1) is applied to each sentence in the test dataset in order to predict whether the sentence belongs to the positive or to the negative class. Suppose that a review R in the test dataset corresponds to "w_1, w_2, w_{43}". For R, the probability of being in the positive class or negative is computed as follows respectively in Eqs. (2) and (3).

$$\begin{aligned} p\,(+|R) &= p\,(C = +) \times p\,(w_1\,|\,C = +) \times p\,(w_2\,|\,C = +) \times p(w_{43}|C = +) \\ &= \frac{2,400,000}{4,000,000} \times \frac{45,600}{2,400,000} \times \frac{72,250}{2,400,000} \times \frac{90,400}{2,400,000} = 0.00001288 \end{aligned} \tag{2}$$

$$\begin{aligned} p(-|R) &= P\left(C=-\right) \times P\left(w_1 \,|\, C=-\right) \times P\left(w_2 \,|\, C=-\right) \times p(w_{43}|C=-) \\ &= \frac{1,600,000}{4,000,000} \times \frac{120,000}{1,600,000} \times \frac{50,000}{1,600,000} \times \frac{23,220}{1,600,000} = 0.00002024 \end{aligned} \tag{3}$$

The text sentiment is classified as the class that has a greater probability value. In the above example, the text sentiment for R is classified as negative.

4. Experimental setup

In Spark, jobs are submitted for processing a large dataset. Each job first loads its working data into memory for enabling rapid data access. The main component of Spark is the construction of a resilient distributed dataset (RDD).

4.1. Resilient distributed dataset

The RDD provides granular fault tolerance and distribution of work. Input data are sliced into multiple chunks so that parallel jobs can be executed on each chunk. Storing lineage information in the framework per RDD provides fault tolerance. Each compute step in the compute flow can be reexecuted linearly to enable recovery in case of failures.

Parallelized collections and Hadoop datasets are two ways to create an RDD. Parallelized collection is a wrapper on the Scala programming language's collection, which also supports parallel operations. It can be created by calling the parallelize method of Spark context on an existing Scala collection.

```
val data = Array(1, 2, 3, 4, 5)        // data is a Scala collection.
val distData = sc.parallelize(data)    // distData is an RDD.
```

An input that resides in a Hadoop distributed file system (HDFS) can be used to create an RDD by calling the textFile method of Spark context as follows:

```
val distFile = sc.textFile("hdfs://.../data.txt").
```

Two types of operations can be done on RDDs: transformations and actions. An RDD can be transformed into another RDD by using a mapper. An action corresponds to an aggregation used during reduction.

Spark currently provides three APIs, one each for Scala, Java, and Python programming languages. We used the Java API. We have a dictionary containing the number of occurrences of each word in each of the positive and negative classes. These "read-only" data are used to compute the sentiment of a given review as explained in Section 3.2. Since Spark is a distributed environment, each node must be able to do a lookup in this read-only dictionary. Spark's default behavior is to send the required data within the compute cluster before each iteration. The default behavior results in a bottleneck in the master node and its available bandwidth, and therefore limits scalability. Our solution to this problem is to use the broadcast variables of the Spark framework.

4.2. Broadcast variables

Broadcast enables us to send a map to worker nodes only once at the beginning of the job execution. We can share the maps that hold occurrence counts per category with the use of the broadcast feature in Spark. In order to test this feature, we implemented two methods in order to perform data lookups:

1) When any worker needs data and if the data reside in another node in the cluster, the data owner sends the requested data to the requestor. This operation consumes less random access memory (RAM), but requires high IO and CPU operations.

2) We can store lookup tables in full in all workers. This approach requires more RAM for data storage, but it needs less IO. This method can be accomplished by broadcasting lookup dictionaries to all worker nodes as follows:

$$\boldsymbol{Broadcast{<}JavaPairRDD{<}String,\ Double{>}\ {>}}$$

$$\boldsymbol{posMapBroadcast = sc.broadcast(positiveDataMapRDD)}.$$

Here, "positiveDataMapRDD" is the original data structure and "posMapBroadcast" is the new data version that will be broadcasted to each node. By using broadcast variables, we optimized our computation time by 1.15 times.

4.3. The movie review dataset

In experiments, Amazon movie reviews [10] were used. There were 7,911,684 reviews, which were extracted from 889,176 reviews for 253,059 products. Some reviews contain a single sentence, while some others contain more than 10 sentences. The median number of words per review is 101. All reviews have information about product ID, user ID, time, score, summary, and text. An example review is given below:

product/productId: B00006HAXW

review/userId: A1RSDE90N6RSZF

review/profileName: Joseph M. Kotow

review/helpfulness: 9/9

review/score: 5.0

review/time: 1042502400

review/summary: "Pittsburgh - Home of the OLDIES"

review/text: "I have all of the doo wop DVD's and this one is as good or better than the 1st ones. Remember once these performers are gone, we'll never get to see them again. Rhino did an excellent job and if you like or love doo wop and Rock n Roll you'll LOVE this DVD !!" [10]

Only the score and summary information above were used in our system. After the preprocessing, the sentiment label for each review was added to the end of each entry separated by a comma. The entire dataset was separated into two parts as the training set and the test set. The training set consisted of 4 million reviews, which had about 349,993,900 words. The rest of the dataset was used as the test set. Five different test configurations were constructed with 100,000, 250,000, 500,000, 750,000, and 1 million reviews, respectively. Table 4 gives detailed information about the test data.

Table 4. The description of the test data.

	# of reviews	# of words	size (MB)
100,000	100,000	8,880,070	62 MB
250,000	250,000	22,106,912	155 MB
500,000	500,000	44,566,580	313 MB
750,000	750,000	66,811,887	470 MB
1 million	1,000,000	88,930,249	625 MB

4.4. Cluster configuration

Two different clusters were used for performance comparison. The first cluster, called the Şehir cluster, has one master and 8 workers with 4-core CPU, 8 GB of RAM, and 100 GB of disk space. The Şehir cluster is depicted in Figure 1. The second cluster, called the Amazon cluster, is hosted in Amazon EC2 and has one master and 4 workers with 4-core CPU and 15 GB of RAM. The Java Development Kit version 1.7.0_03 was installed on each node for a Java runtime environment in both clusters. Since the dataset is large, the HDFS was chosen to store the data. HDFS version 1.0.4 was installed in the clusters. On top of the HDFS, we built Spark version 0.7.0. Figure 1 shows our cluster hierarchy.

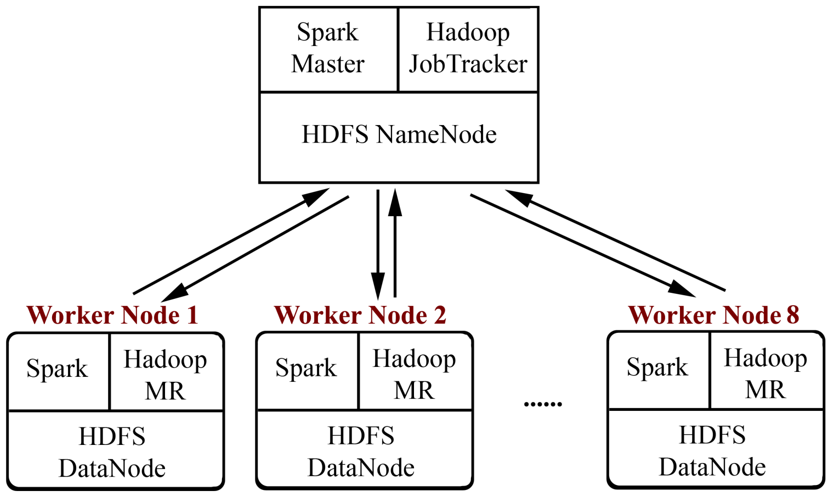

Figure 1. The depiction of the Şehir cluster.

4.5. Model building

For training, two map jobs and two reduce jobs were created: one M/R job pair was created for the positive class and the other M/R pair was created for the negative class. In the mapping stage, each review was mapped to either the negative class or the positive class. For both classes, a separate word-count dictionary was created. This action resulted in a positiveMap and a negativeMap. Each sentence of a review was split by whitespace into individual words. Every word was mapped into a value of “1” keyed by the word itself. In the reduce step, all values were summed up per key. After the job was completed, information as to how many positive and how many negative reviews a given word occurred in was obtained.

In order to outline how our M/R works, let us demonstrate how to compute the positive class probability for a given review R, which consists of words $w_1, w_2, \ldots, w_n$ (note that the computation of the negative class probability is very similar).

Map (M):

i. Each word is mapped to $(R,1)$ as its value, i.e. $(w_i, (R,1))$ where $i = 1, \ldots, n$.

ii. This (key, value) tuple is joined with a positive lookup map, which has words as keys and number occurrences of those words in positive reviews as the corresponding values. In the positive lookup map, an entry looks like (w_i, X), where X is a positive integer that holds the total number of occurrences of w_i in positive reviews.

iii. After the join, the interim results map has the following (key, value) pairs: $(w_i, ((R,1), X_i))$ where $i = 1, \ldots, n$.

iv. We swap the places of R and w_i in each of these tuples to finally get $(R, ((w_i,1), X_i))$ where $i = 1, \ldots, n$.

Reduce (R):

i. Each entry is then reduced by using reduceByKey function as follows:
$(R, \frac{X_1}{NumPos} \times \frac{X_2}{NumPos} \times \ldots \times \frac{X_i}{NumPos}) = Y$

ii. The tuple $(R, Y \times NumPos \times NumTotalDocuments)$ represents the final result (K, V). The key K is the review R itself, and value V corresponds to the probability of R belonging to the positive class.

5. Experimental evaluation

We compared our solution with two alternatives. Both of the alternative approaches were based on Hadoop: 1) a naive Bayes classifier built using Hadoop M/R and 2) another naive Bayes classifier built using Apache Mahout. We describe these two frameworks next before presenting our empirical findings.

5.1. Apache Hadoop

Hadoop is an Apache foundation framework that can be used for processing large datasets on a cluster of computers using the M/R programming model [11]. The two main projects of Hadoop are the HDFS and Hadoop M/R. The HDFS is a fault-tolerant, scalable, and highly configurable distributed file system written in Java. An HDFS cluster has a master name node that manages synchronization and coordination among data nodes and stores metadata for the cluster. Multiple data nodes store the actual user data. An HDFS client contacts the name node for file operations such as select, insert, and delete. The HDFS also has support for failing over to a secondary name node to avoid the single master being the single point of failure.

The Hadoop M/R enables programmers to write applications in order to process large datasets in parallel on a cluster of machines. An M/R job has two main components: 1) map and 2) reduce. The framework splits input data into multiple chunks so that multiple map tasks can process these individual data partitions in parallel. Outputs of the map tasks are collected and processed by the subsequent reduce tasks. The inputs and the outputs of each job are stored in the HDFS. Since the map and the reduce tasks operate on <key, value> pairs, the input and output format will also be <key, value> pairs.

5.2. Apache Mahout

Mahout is an Apache foundation project developed for building scalable machine learning libraries [2]. Mahout has support for building classifiers, clustering items, genetic programming, constructing random forests, and recommending items. All these end-user products are implemented on top of Hadoop.

Since Mahout has naive Bayes classifier support, we included it in our tests. During training, Mahout created a handful of feature vector output files and built a final model from these interim output files. The whole process took almost 3 h. During testing, the model built was used on the same test dataset that was used in the other competing approaches.

5.3. Empirical results

5.3.1. Broadcasting vs. not broadcasting

In order to see the effect of broadcasting vs. relying on the framework to shuffle data when needed, we conducted an experiment on five test scenarios. This test was done on the Amazon cluster. As shown in the results in Table 5 and Figure 2, the broadcast method took less time to go through the testing phase. For one million reviews, the broadcasting completed 1.3 min faster than the method without broadcasting. The reason for this improvement is that the application driver did not waste time trying to share the required data among the compute nodes as the alternative approach did. The gap in running time widened between the two methods as the size of the data increased. This is because the increased data size led to the increased data delivery between the compute nodes.

Table 5. The running time of the testing step on Spark (in minutes) hosted in the Amazon EC2 cluster.

	100K	250K	500K	750K	1M
Spark without broadcast	1.5	2	3.3	5.4	9.6
Spark with broadcast	1.4	1.6	2.8	4.7	8.3

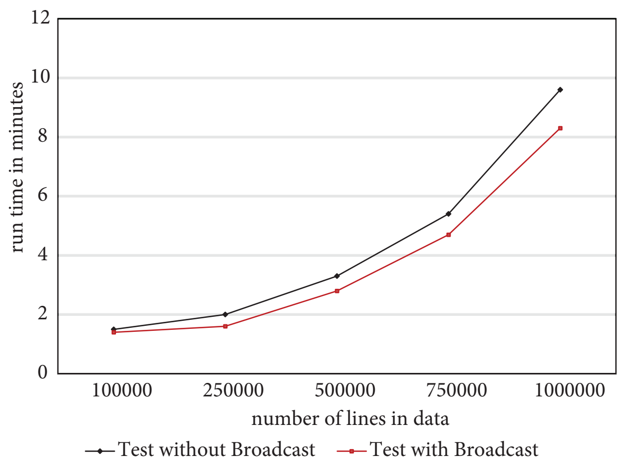

Figure 2. The running time of the testing step for the case of with broadcast and without broadcast.

5.3.2. Time required for training

On the Şehir cluster, we conducted two tests: one using 4 workers and the other using 8 workers. Table 6 shows how long it took to train on Spark vs. Hadoop with different numbers of workers. The training time in the case of Hadoop was in the order of minutes, while training using Spark was in the order of seconds.

Table 6. The running time of the training step on the Şehir cluster.

	4 workers	8 workers
Hadoop	12 min	10 min
Spark	73 s	62 s

5.3.3. Time required for testing

Table 7 and Figure 3 show the run time of the testing step on Hadoop and Spark on the Şehir cluster. Compared to Hadoop on all test scenarios, Spark implementation was up to 10 times faster in crunching data. For example, for 1 million reviews with 8 workers, Spark completed the testing in 7.9 min while Hadoop implementation required 70 min to complete. The benefits of using the broadcast variables were even more apparent in the Şehir cluster. Using 4 workers only, 750K reviews were digested in 8.6 min with broadcasting compared to 11 min without it.

Table 7. The runtime comparison of the testing step on Hadoop and Spark hosted in the Şehir cluster (in minutes).

	4 workers			8 workers		
	Spark		Hadoop	Spark		Hadoop
	w/oB[1]	wB[2]	Dist.[3]	w/oB	wB	Dist.
100K	1.8	1.6	13.1	1.9	1.6	15.4
250K	2.6	2.2	22.5	2.4	2.1	24.2
500K	4.6	3.5	33.3	3.3	2.6	36
750K	11.0	8.6	58.3	5.3	4.0	50
1 million	OoM[4]	OoM	78.5	8.9	7.9	70

[1] Without broadcasting.

[2] With broadcasting.

[3] Distributed cache.

[4] Out of memory exception: program runs out of memory. When a task starts running, the working data are loaded into the cache. For 1M reviews, since join operation is costly due to the Cartesian product with lookup map, this test case runs out of memory.

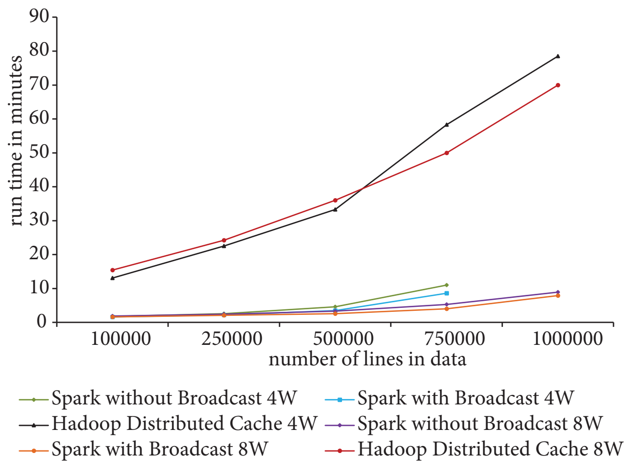

Figure 3. The running time of the testing step with broadcast and without broadcast.

Results for Mahout are shown in Table 8. For a small size dataset, e.g., 100K reviews, when the cluster was upsized from 4 workers to 8 workers, the computation time increased by 3 s due to the coordination overhead in the cluster. The advantage of a high number of compute nodes in a cluster did not justify itself, because the dataset size was not large enough. For larger data sizes, the benefit of using a higher number of workers was more apparent. For example, it took 112 s to digest 1 million reviews with 8 workers, while it took 150 s with 4 workers.

Table 8. The running time of the testing step for Mahout's naive Bayes classifier on the Şehir cluster (in seconds).

Number of reviews	4 workers	8 workers
100K	63	66
250K	75	69
500K	101	77
750K	112	82
1 million	150	112

6. Future work

In this paper, we showed how to build a scalable sentiment analyzer on Spark. We used HDFS to store the movie reviews data. Tachyon [12] is an in-memory distributed file system, which enables rapid file sharing across cluster frameworks. We can use Tachyon as an intermediate data storage layer between HDFS and Spark to speed up the sentiment learning and testing. Since we have used the same cluster for multiple frameworks (Spark, Hadoop, and Apache Mahout), we can increase physical resource utilization by using Apache Mesos [13]. Mesos is a cluster manager for efficient resource sharing between different frameworks. Alternatively, Hadoop Yarn [14] can also be used. Yarn manages resources and provides an efficient scheduling through a global resource manager (RM) and many local application managers (AM). Finally, we are planning to synthesize our implementation such that it can easily be deployed to a host cluster.

7. Conclusions

Machine learning algorithms are time-consuming when the dataset to analyze is large. In this paper, we showed how to build a naive Bayes classifier for millions of movie reviews in a matter of seconds using Spark. We compared our implementation to that of the state-of-the-art competitors: 1) a custom Hadoop-based implementation and 2) Apache Mahout-based implementation. The results showed that the classifier built on Spark ran almost 10 times faster compared to the Hadoop implementation. The classifier built on Apache Mahout took almost 3 h to build the classifier, while it took 73 s to build the model on Spark, and 12 min on Hadoop. The digestion of the test dataset showed that the performance of Mahout was comparable to that of Spark.

Acknowledgment

This work was supported by Türk Telekom Argela under Grant Number 6401-01.

References

[1] Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing; June 2010; Berkeley, CA, USA. p. 10.

[2] Team AM. Apache Mahout: Scalable Machine-Learning and Data-Mining Library. Forest Hill, MD, USA: Apache Software Foundation, 2011.

[3] Kraska T, Talwalkar A, Duchi JC, Griffith R, Franklin MJ, Jordan MI. MLbase: A distributed machine-learning system. In: CIDR; 6–9 January 2013; Asilomar, CA, USA.

[4] Pang B, Lee L, Vaithyanathan S. Thumbs up?: sentiment classification using machine learning techniques. In: Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing, Vol. 10; 2002. Stroudsburg, PA, USA: Association for Computational Linguistics. pp. 79–86.

[5] Elsayed T, Lin J, Oard DW. Pairwise document similarity in large collections with MapReduce. In: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers; 2008. Stroudsburg, PA, USA: Association for Computational Linguistics. pp. 265–268.

[6] Khuc VN, Shivade C, Ramnath R, Ramanathan J. Towards building large-scale distributed systems for twitter sentiment analysis. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing; 26–30 March 2012; Riva, Italy. New York, NY, USA: ACM. pp. 459–464.

[7] Hunter T, Moldovan T, Zaharia M, Merzgui S, Ma J, Franklin MJ, Abbeel P, Bayen AM. Scaling the mobile millennium system in the cloud. In: Proceedings of the 2nd ACM Symposium on Cloud Computing; 26–28 October 2011; Cascais, Portugal. New York, NY, USA: ACM. p. 28.

[8] Bird S. Nltk: the natural language toolkit. In: Proceedings of the COLING/ACL on Interactive Presentation Sessions; 2006. Stroudsburg, PA, USA: Association for Computational Linguistics. pp. 69–72.

[9] Lewis DD. Naive (Bayes) at forty: The independence assumption in information retrieval. In: Machine Learning: ECML-98; 1998. Berlin, Germany: Springer. pp. 4–15.

[10] McAuley J, Leskovec J. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. In: Proceedings of the 22nd International Conference on World Wide Web; 13–17 May 2013; Rio de Janeiro, Brazil. pp. 897–908.

[11] Lam CK. Hadoop in Action. 1st ed. Greenwich, CT, USA: Manning Publications Co., 2010.

[12] Li H, Ghodsi A, Zaharia M, Baldeschwieler E, Shenker S, Stoica I. Tachyon: memory throughput I/O for cluster computing frameworks. In: SOSP 2013 Workshop LADIS; 2013.

[13] Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz R, Shener S, Stoica I. Mesos: A platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation; 30 March–1 April 2011; Boston, MA, USA. p. 22.

[14] Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S et al. Apache Hadoop yarn: Yet another resource negotiator. In: Proceedings of the Fourth ACM Symposium on Cloud Computing; 1–3 October 2013; Santa Clara, CA, USA. p. 5.