MAKING ONLINE RECOMMENDATIONS MADE EASY BY APACHE SPARK

Aslan Bakirov; İstanbul Şehir University, Data Science Lab

Kevser Nur Çoğalmış; İstanbul Şehir University, Data Science Lab

Ahmet Bulut; İstanbul Şehir University, Data Science Lab

Keywords: Apache Spark; Collaborative Filtering; Scalable Data Analytics.

1. ABSTRACT

With the advent of social networks, forums, and blogs, the amount of data on the Web has increased rapidly, resulting in an information explosion. Using the Internet, users make purchases, listen to music, or watch a movie; then later on, they make comments about the purchases they have recently made, indicate their musical preference, and write their opinions about the movies they have recently watched.

Raw user data as is does not provide much information unless the information is explicitly processed. Data mining refers to the process of extracting information from raw data. Data mining is used for a variety of knowledge discovery tasks such as classification, clustering, and regression. Since actual task implementations analyze the entire set of data in order to find a common pattern, the run time of these algorithms depends on the size of the dataset.

Handling large amounts of data requires the use of the MapReduce (M/R) programming model on special purpose compute clouds. Hadoop was one of the first platforms introduced that provides an implementation of M/R on a compute cluster. A more recent implementation of M/R called Apache Spark has been shown to provide ten-fold performance benefits compared to Hadoop on certain machine learning tasks.

In this study, we present our results in designing a scalable data analytics platform on top of Apache Spark. More specifically, we built an online collaborative filtering application for testing the scalability and usability of Spark's machine learning library on a cluster of 8 workers. Each worker contributes 4 cores, 8 GB RAM, and 100 GB of disk space to the compute pool. Our conclusion is that Spark is sufficiently advanced for deployment in production environment.

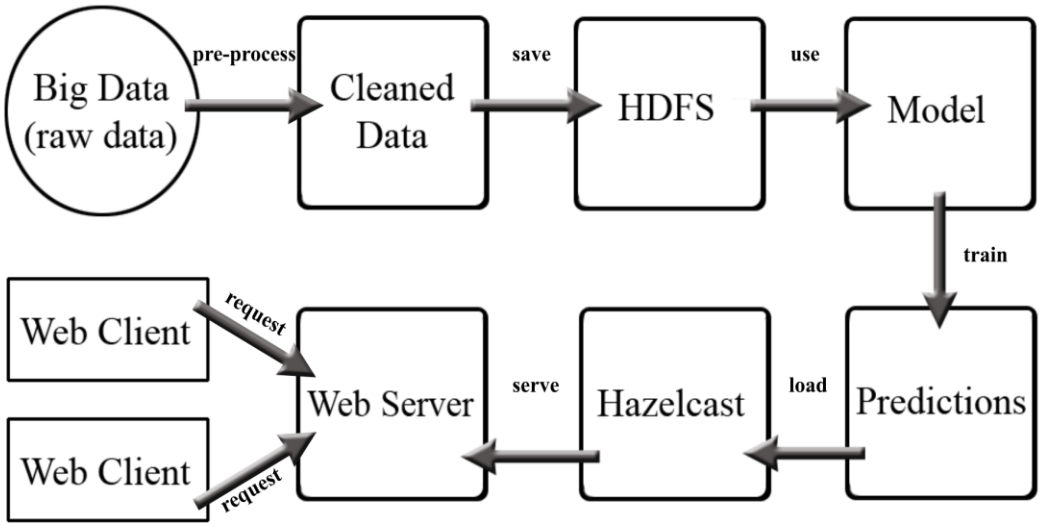

Figure 1.
System Workflow for Making Online Recommendations.

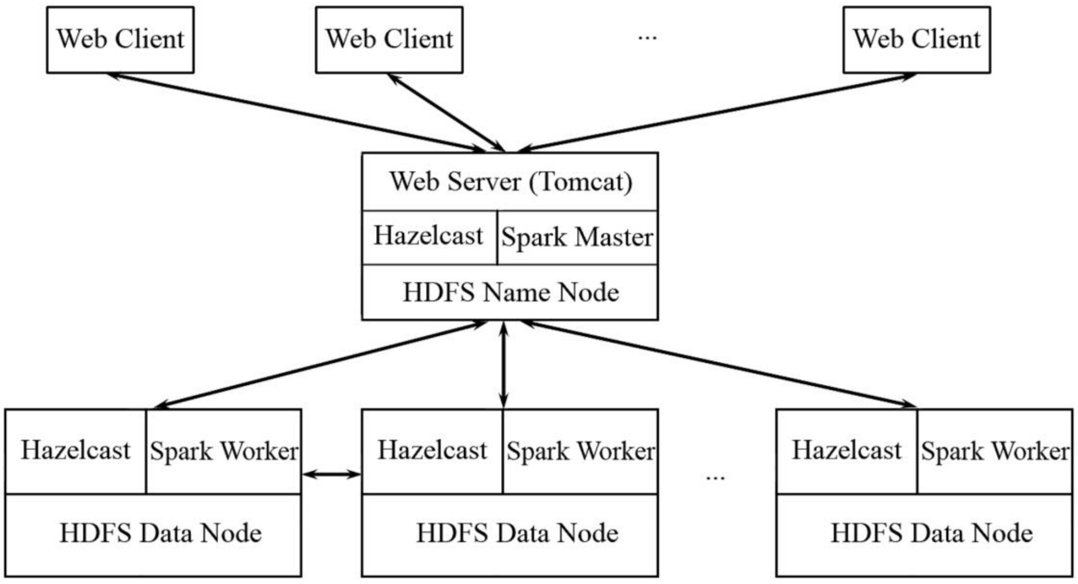

Figure 2.
System Architecture for Making Online Recommendations.

2. SYSTEM DESIGN AND IMPLEMENTATION

Here, we provide the details of our reference system that uses the recommendations computed using Spark's MLBase library [1]. The reference system denotes an entire Web 2.0 service, which supplies vital data from an in-memory cache layer. The cache layer is bootstrapped with recommendations computed a-priori by the backend batch-processing engine powered by Spark [2].

2.1. System Workflow

The complete system workflow is shown in Figure 1. We start with raw data. Spark's MLI interface accepts input data in a particular format for building any ML model. Hence, we have to first pre-process the raw data in order to convert it into the desired format and then persist it onto HDFS.

We build an alternative least squares based machine learning model and fill in the missing entries with data (the score matrix, which contains user-to-product score information) using this learnt model.

When the Web server is up, the Hazelcast cluster loads all data into the main memory.

The Web server contacts the cluster to look up product recommendations on the fly.

2.2. System Architecture

Here, we have a tiered architecture. At the base layer, the available data, which is used for training and testing, is stored in a Hadoop Distributed File System (HDFS).

We employed Spark's MLBase to build a score prediction model on top of this data. The learnt model is stored in a distributed cache, which is buit on Hazelcast [3]. Hazelcast is an open source in-memory data grid written in Java. Using Hazelcast, data can be distributed evenly across a clustered cache.

To avoid data loss due to node failures, multiple replicas of each data record are distributed among the cluster nodes. There are different use cases of Hazelcast. In practice, it is most widely used

- as a distributed cache in front of a persistent storage system,
- for clustering Web sessions,
- for in-memory big data processing and analytics.

Our use of Hazelcast corresponds to the first use case. The topmost layer in the architecture is our Web layer, which recommends movies to users in the system. The full system architecture is shown in Figure 2 above. The cluster consists of 1 master and 8 workers. Each worker contributes 4 cores, 8 GB RAM, and 100 GB of disk space to the compute pool. The number of Web clients may be arbitrarily large.

Spark's master node also serves as the HDFS Name Node and is not only part of the Hazelcast distributed cache (the cluster ring) but also runs a Web server. Multiple Web clients query this

Web server. The master node hands out tasks to workers in the cluster, which carry out the actual task executions.

Each Spark worker is responsible for storing data, i.e., acting as an HDFS Data Node, and from executing tasks assigned by the master. Furthermore, each worker belongs to the same Hazelcast cluster ring as the master.

2.3. MLBase

MLBase is a scalable machine learning (ML) framework that runs on Spark's core processing engine. MLbase consists of three components: (1) Machine Learning Library (MLlib), (2) Machine Learning Interface (MLI), and (3) Machine Learning Optimizer (ML Optimizer).

- ML Optimizer automates the task of ML pipeline construction. It optimizes algorithm parameters and data sampling at runtime. The optimizer estimates execution time and algorithm performance using statistical models built on data collected from previous job executions.
- MLI is an API for feature extraction and algorithm development that provides high-level ML programming abstractions.
- MLlib is a distributed machine learning library that runs on Spark.

Developing scalable machine learning algorithms using MLI is very simple. For instance, the following code block in Scala programming language builds a model for recommending movies to users using user to movie pairs and user ratings datasets.

```
var X = load("user_movie_pairs", 1 to 2)

var y = load("user_ratings", 1)

var (fn-model, summary) = doCollabFilter(X, y)
```

For a given user Alice, the scores Alice would give to movies that she has not seen yet can be estimated using the model learnt, i.e., **fn-model** above.

2.4. Building a Model

In order to learn a recommendation model, we used Amazon's IMDB movie reviews dataset [4,5]. There are 7,911,684 reviews, which were extracted from 889,176 reviews for 253,059 products. 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 !!"

In our case, we only used the following data attributes: user's identification (user id), product's identification (product id) and the score given by a user with a specific *user_id* to a product with a specific *product_id*. Table 1 shows sample data after basic pre-processing has been performed.

User ID	Product ID	Score
26256	208865	3.0
484047	208865	3.0
118779	208865	5.0

Table 1.
Training Data. Users score movies they watched on a scale of 0-5.

We used the alternative least squares algorithm (ALS) available in MLI in order to train a recommender from 500K user id, product id, and score entries [6]. The algorithm predicts the unfilled cells in the users x products score matrix shown in Table 2a using partial score data. An unfilled cell for a *user_id* to *product_id* pair means that user has not yet scored that product. Once the score matrix is filled up with score predictions as shown in Table 2b, the whole matrix is written onto the HDFS. In our implementation, this whole operation took 6 minutes. When the Web server is booted up, the Hazelcast cluster loads the score matrix stored in HDFS right into the cluster's distributed main memory. When a user logs in, top recommended products based on the scores computed are shown to the user in a matter of seconds.

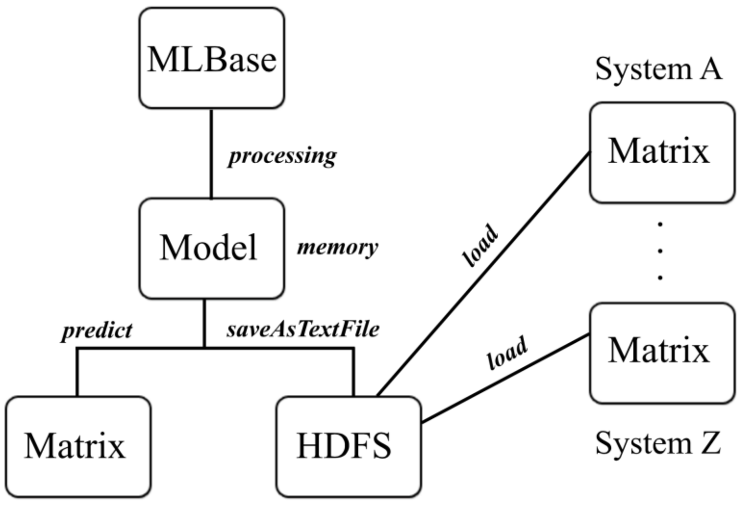

Figure 3.
MLBase Architecture.

	Product ID		
User ID	2		1
			5
		3	

(a) Before predictions made.

	Product ID		
User ID	2	x_1	1
	x_2	x_3	5
	x_4	3	x_6

(b) After predictions made using ALS.

Table 2.
Using the users' product scorings data available, the ALS algorithm is applied in order to fill in the missing entries in the score matrix.

The time it takes to build a model from scratch is typically a couple of minutes, but fortunately, the framework allows us to save the learnt model on to the HDFS. This enables us to use the model later on and also make it available for use by other applications. This capability is depicted visually in Figure 3.

3. CONCLUSIONS

We built an online system that uses recommendations made by a backend batch system developed on Apache Spark. Using the ALS algorithm available in Spark's MLI API, it is possible to craft a distributed and online recommendation system. Filling in a sparse prediction matrix that includes 500K entries, took 6 minutes in our cluster of 8 workers, each of which had 4 cores and 8 GB of RAM. Also, we noted that MLBase allows us to save the learnt model, which can be used by downstream systems and applications.

4. REFERENCES

[1] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLBase: A distributed machine-learning system. In CIDR, 2013.
[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, pages 10–10, 2010.
[3] Hazelcast. Hazelcast the leading in-memory data grid. http://hazelcast.com/
[4] A. Bakırov, K. N. Çoğalmış, and A. Bulut. Scalable sentiment analytics. Turkish Journal of Electrical Engineering & Computer Science, 2014.
[5] J. Leskovec and A. Krevl. SNAP Datasets: Stanford Large Network Dataset Collection at http://snap.stanford.edu/data. June, 2014.
[6] Y. Koren, R. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems. In Computer, v42 (8), pp 30—37, August 2009.