CISC 849 Program Heterogeneous Systems

Project Report

Build A Markov chain model with Hadoop and Spark

Weiyang Chen

1. Abstract

We live in a world where trust can not be given to everyone. It is always necessary to hide important information from untrusted people. More important issue here is how to identify the bad one from the good efficiently. In this report I will address how to build a Markov chain model for real time fraud detection in credit card transaction and implement this technique with Hadoop and Spark respectively. I will give a performance summarize of this case study. In general, Spark take less time than Hadoop to process a 2.4G credit transaction dataset in a common linux laptop, partly because of the less disk I/O in Spark.

2. Introduction

Fraud detection is a particular application for a general problem known as outlier detection. An outlier detection is to identify an item which do not follow an expected pattern in a dataset. Fraud detection is widely used in variety of fields such as credit card companies, insurance and retail. To minimize the lost of stakeholders, people should identify a malicious action as soon as possible. This requires detection system to be real time. There are a lot of outlier detection algorithms exist. Some of them need to scan the whole dataset. Time taken in this approach usually beyond our acceptable scope. To build a real time detection system, we need to let our algorithm generate a predict model which can be used as a real time detector.

One solution is to build a predict model based on Markov chain. The advantage of using Markov chain is that a sequence of actions being observed in a small time window is enough to identify if a specific action is bad. This feature makes a real time fraud detection possible. I choose Hadoop and Spark as its implementation. Hadoop and Spark are the most common big data process engine. Although they have their own expertise area, they all good at batch processing. In my project they are used to process a 2.4G log file which contain credit card transaction history from many users. A Markov chain model, which is a transition matrix, is generated after this process. This matrix can be used to depict a set of normal transactions of a user. If choose appropriate metric, we can get a complete real time fraud detection model. This is not part of my project. Detail discussion of several metrics can be found in [1].

3. Background

Markov Chain

Markov chain is a stochastic process. Given a sequence of random variables x_1, x_2, x_3 …… called states, it tries to answer the probability of being in a specific state after certain amount of time has elapsed. Time can be continuous or discrete, and number of events can be infinite. I am interested in discrete-time Markov chain with finite number of states. Markov chain is memoryless. For a certain state x_i at a certain time n; the probability of moving to next state x_j only depends on the current state x_i not its history.

To organize the state's transitions probabilities a transition matrix is used (Table 1). Each columns and rows represent a state in Markov chain. Each cells contain the probability of moving from the "row" state to the "column" state.

	x_1	**x_2**	…	**x_n**
x_1	$P_{(1,1)}$	$P_{(1,2)}$	…	$P_{(1,n)}$
x_2	$P_{(2,1)}$	$P_{(2,2)}$	…	$P_{(2,n)}$
…	…	…	…	…
x_n	$P_{(n,1)}$	$P_{(n,2)}$	…	$P_{(n,n)}$

Table 1. Transition Matrix

Hadoop & Spark

Hadoop provide a framework and infrastructure to process large dataset on commodity hardware. Hadoop family have a variety of products, from distributed database to high-performance coordination service. The typical use of Hadoop involve three components: Hadoop Distributed File System(HDFS) which provides high-throughput data access, Hadoop YARN which is a resource negotiator responsible for allocate computing and storage resource among different tasks and applications, Hadoop MapReduce which provides a parallel programming library based on map-reduce framework. Hadoop has been the most popular big data process engine for a long time.

Spark is cluster computing framework. "It provides a distributed memory abstraction that lets programmers per- form in-memory computations on large clusters in a fault-tolerant manner[2]." Spark does not provide infrastructure such as distributed file system and resource allocator. However it can be used in solo with its standalone mode. The typical environment for Spark involve a distributed file system such as HDFS and a resource negotiator such Hadoop YARN.

4. Application

The application in my project is to analyze a credit card transaction dataset which comes from many users. For each user a transition matrix is built based on Markov chain. The dataset is a set of transaction log files, the format of the log record is:

```
Customer_id, Transaction_id, Transaction_type
```

The `transaction_id` values are unique and have the semantics of positioning a transaction along a time line. Each log file will contain some transactions of different customers identified by `customer_id`. For example(Table 2):

customer_id	customer_id	customer_id
TJS41P2U9A	97351659821043	LHS
R8NYNNV4NP	97358894512859	MHN
…	…	…

Table 2. Log File Sample

Transaction_type is encoded by the following three quantities and expressed as a 3-letter token.

Amount spent: **L**ow, **N**ormal, or **H**igh
Whether the transaction includes high price ticket item: **N**ormal or **H**igh
Time elapsed since the last transaction: **L**arge, **N**ormal, or **S**mall

There are total 18 possible `transaction_type`. The goal is to build a 18*18 transition matrix for each `customer_id`. Each cell in a transition matrix tells us given the current `transaction_type` of type x_i the probability of the next `transaction_type` being type of x_j is $P_{(i,j)}$. Table 3 shows a small piece of the expected transition matrix belong to one `customer_id`. We should have one such a transition matrix for each `customer_id`.

	LNL	LNN	…..	HHS
LNL	0.09550118389897395	0.13338595106550907	……	0.02762430939226519
LNN	0.09809932556713673	0.11955855303494789	……	……
……	……	……	……	……
HHS	0.09986859395532194	0.12746386333771353	……	0.00433232312746627

Table 3. Output Sample

5. Hadoop Implementation

To build transition matrixes, there are two Hadoop jobs involve. The first one called `getSequenceBuilderJob` which is used to group records by `customer_id` and sort them by `transaction_id` in decrease order. The second one called `getMarkovChainTrainingJob` which is used to calculate probability and build transition matrixes. The first job read the log file from HDFS and then write the intermediate data to HDFS. The second job read the intermediate dat from HDFS and write the final results to HDFS.

In the first job, mapper generates a key-value pair with `(<Customer_id, Transaction_id>, Transaction_type)` for each record. `<Customer_id, Transaction_id>` is a composite key used to performed the secondary sort. By default, Hadoop MapReduce framework sorts mapper's output records and reducer's input records. In my application reducer should only use `Customer_id` as key for input records. However, I still need to sort records by `transaction_id` which has the semantics of positioning a transaction along a time line. In this case I need to performed the secondary sort at mapper side. This action ensure that the records received by reducer are grouped by `Customer_id` and sort by `Transaction_id`.

Here is a sample of the first job's output. A complete transaction sequence is build for each customer. `transaction_id` is removed because its primary job, sorting records by time, is complete. Reducer emit "1" for for each record.

```
TJS41P2U9A   LHS    1
TJS41P2U9A   MHN    1
TJS41P2U9A   LNS    1
……
R8NYNNV4NP   MNN    1
……
```

In the second job, for every two records next to each other, belong to the same customer, mapper generates a key-value pair with `(<Customer_id, present, future>, 1)`. `present` is the `Transaction_type` from the first record and `future` is the one from second. A hash map is used to search a transaction matrix for a specific customer. Reducer perform a counting for each key and store the result in transition matrix at position of `(present, future)`. Finally a Laplace smoothing is performed for each matrix in order to eliminate the zero.

6. Spark Implementation

"Formally, an RDD is a read-only, partitioned collection of records[2]." RDD can only be created from stable storage or other RDDs. Operation on RDD basically has two kinds: transformation and action. Transformation will generate a new RDD. Action will return a value from RDD or export data to a storage system. A Spark job can be divided into different stages. Shuffle defined the boundary between stages and it happen in both stages. In my application, there are two stages and five RDDs.

Stage zero just read data from HDFS and extract the necessary information. This stage only include one RDD, RDD$_0$. RDD$_0$ will not be computed until shuffle happen. Shuffle groups records by `Customer_id`. Stage zero write the shuffle output into HDFS.

Stage one has four RDDs. The shuffle operation on this stage generate the RDD$_1$. In RDD$_1$, records are grouped by `Customer_id`. Then, a sort operation applied on RDD$_1$. This lead to the RDD$_2$. In RDD$_2$, all records with the same `Customer_id` are sorted by `transaction_id`. Then, a matrix-build function applied on RDD$_2$. For each `Customer_id` a matrix will be built. This is RDD$_3$. Finally in RDD$_4$, I just convert the matrix from doubt array to string. Following graph show the complete process.

```
Read file from HDFS
JavaPairRDD < Customer_id, Tuple2<transaction_id, Transaction_type>>
     ⇩
Group RDDs by Customer_id
JavaPairRDD < Customer_id, Iterable<Tuple2<transaction_id, Transaction_type>>>
     ⇩
Sort RDDs by transaction_id
JavaPairRDD < Customer_id, Iterable<Tuple2<transaction_id, Transaction_type>>>
     ⇩
Build matrix for each customer
JavaPairRDD < Customer_id, double[][]>
     ⇩
Convert matrix to string
JavaPairRDD < Customer_id, String>
```

Figure 1

7. Result

The hardware environment for this project is MacBook Pro, 2.6 GHz Intel Core i5 with L3 shared cache, 8 GB 1600 MHz DDR3 memory and 128G SSD. Application run in the virtual machine with CentOS 7. Virtual machine configured as follow: 2 cores enabled with 4G memory and 20G storage space.

Both Hadoop and Spark implementation use OpenJDK 1.8.0 and Hadoop 2.6.0. They all built by Maven 3.3.9. However, in Hadoop implementation YARN is enabled. For Spark implementation, only HDFS is used as a shared storage system. Spark use its own standalone mode for task scheduling and job monitoring. Both Hadoop and Spark implementation run HDFS name node and data node in two separate machines. Hadoop job is launched directly in data node. Spark job is submitted in name node and run in data node. For Spark, name node also sever as master, data node serve as slave.

Time taken

Figure 2 show the time taken for Hadoop implementation and Spark implementation.

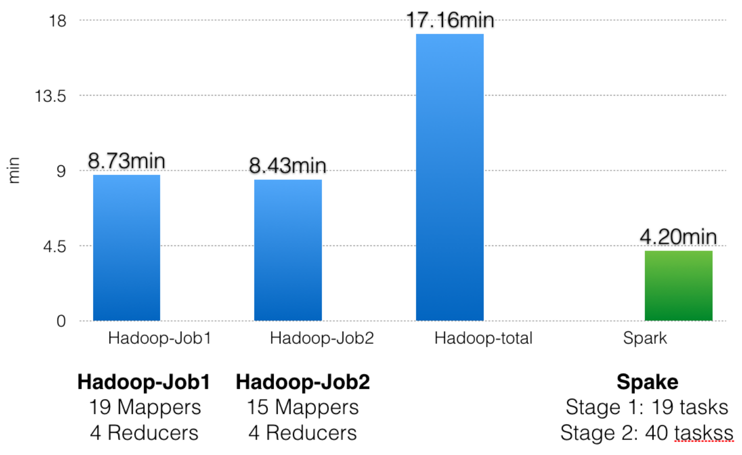

Figure 2

This figure clearly show that Spark take much less time compare with Hadoop. Potential reason for this will be analyzed in next section.

The number of mappers in Hadoop and number of tasks in Spark/stage 1 are defined by input data size and file system's block size. By default, HDFS block size is 128 MB. I use this default configuration. The number of reducers in Hadoop sets by user and it is optimized. The number of tasks in Spark/stage 2 also sets by user. When it reach to 40, performance do not increase and keep stable.

8. Analysis

The huge performance difference in Hadoop and Spark may due to many reason. I will not cover all reasons. Instead, I will focus on one reason which has huge impact on performance. As showed in figure 2, disk I/O in two implementation has a huge difference. This difference is consistent with the one in time taken. More disk I/O means more time taken.

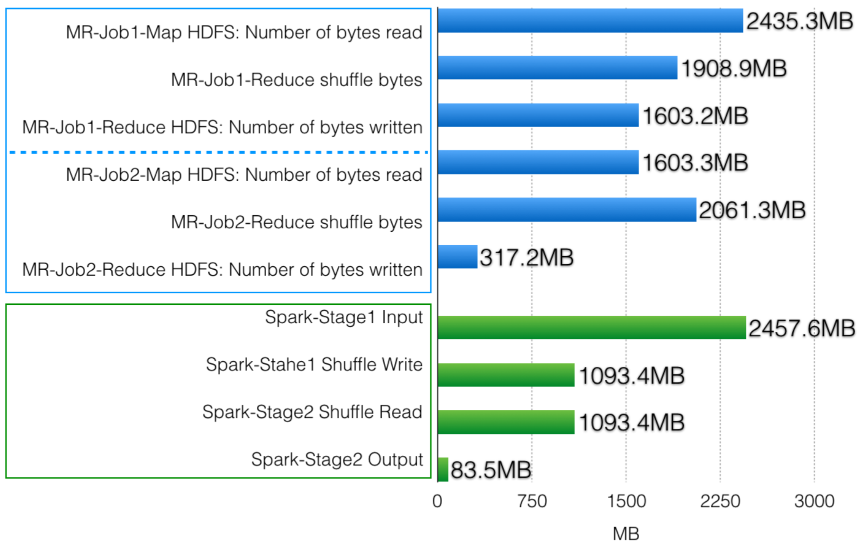

Figure 3

Both Hadoop and Spark has shuffle phase which involve disk I/O. However, they have different meaning. In Hadoop shuffle servers as the bridge between mapper and reducer. Each Hadoop map-reduce job needs a shuffle. If you can not complete your application in one job, more shuffle will involve. In Spark shuffle caused by some specific operations such `groupby()` and `reduceby()`.These operations are not necessary. By careful design, we can avoid them.

Less time taken in Spark also due to the appropriate task numbers in stage 2. In Spark, one node only have one JVM process for a given application. Parallelism achieved by several threads run in this process. Each thread corresponding to a task. The number of cores in this node usually defined the number of tasks can be executed in parallel. All these threads share the same JVM heap size. So, when shuffle read occurs we should ensure that each task is small enough so that they can all fit in the JVM heap size(in Spark1.6.0, memory restrict for shuffle operation is JVM heap size*0.75*0.5 by default) in order to avoid disk spill. To reduce the task size means to increase the number of task. This is why total 40 tasks in stage 2.

Conclusion

In this report, I addressed why fraud detection is important to us and why I chose Markov chain to build a predict model. Also, I showed how to implement this technique with two most popular big data process engines: Hadoop and Spark. The application is to analyze a 2.4G credit card transaction log file and generate a transition matrix for each customers. Finally I summarized the performance for Hadoop and Spark in a common linux laptop. I explained some reason for their performance. I also addressed the basic execution model in Spark. Although this is not a compare study, I want to mention several points Spark better than Hadoop from my opinion.

One advantage in is that Spark does not have to follow the rigid “map-reduce framework”, one map-reduce after another. By careful design, RDD can be stored in memory for future use and shuffle only happen when necessary. This avoid the unnecessary disk I/O which has huge performance impact. In order to store RDD in memory task size should be small enough.

Another advantage in Spark is that Spark is more easy to program. We can clearly see the lineage of RDDs transformation in program. From my opinion, Spark is a programming model combine the advantage of object-oriented programming and process oriented programming.

Attachment List

Spark

Source code:	SparkBuildMatrix.tar.gz
Spark Report:	Simple Application - Spark Jobs
	Simple Application - Details for Job 0
	Simple Application - Details for Stage 0 (Attempt 0)
	Simple Application - Details for Stage 1 (Attempt 0)
	Simple Application - Environment
	Simple Application - Executors (2)

Spark reports are saved a offline pages. To open Spark reports, please use Chrome.

Spark output:	part-00000

Due to the limit of space, only attach one task's output.

Hadoop

Source code:	HadoopBuildMatrix.zip
Hadoop Report:	Counters for job_1463060166153_0001
	MapReduce Job job_1463060166153_0001
	Counters for job_1463060166153_0002
	MapReduce Job job_1463060166153_0002

Due to the limit of space, do not attach Hadoop output. The smallest output file is 78MB.

Reference

[1] Jha, Somesh, Kymie Tan, and Roy A. Maxion. "Markov chains, classifiers, and intrusion detection." csfw. IEEE, 2001.

[2] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

[3] https://www.siam.org/meetings/sdm10/tutorial3.pdf