JKIICE Journal of the Korea Institute of Information and Communication Engineering

한국정보통신학회논문지(J. Korea Inst. Inf. Commun. Eng.) Vol. 21, No. 9 : 1731~1737 Sep. 2017

Spark 기반 빅데이터 처리를 위한 K-최근접 이웃 연결

기가기(纪佳琪)[1,2] · 정영지[1*]

K Nearest Neighbor Joins for Big Data Processing based on Spark

JI JIAQI[1,2] · Yeongjee Chung[1*]

[1*]Department of Computer Engineering, Wonkwang University, Iksan 54538, Korea
[2]Department of Information Center, Hebei Normal University for Nationalities, ChengDe 067000, China

요 약

K-최근접 이웃 연결(KNN 연결) 알고리즘은 기계학습에서 매우 효과적인 방법으로, 작은 데이터군에 대해서 널리 사용되어 왔다. 데이터의 수가 증가함에 따라, 단일 컴퓨터에서는 메모리와 수행시간의 제약으로 실제적인 응용 프로그램에서는 실행하기에 적합하지 못하였다. 최근에는 대규모 데이터 처리를 위해서, 많은 수의 컴퓨터로 이루어진 클러스터에서 실행될 수 있는 맵리듀스 (MapReduce)로 알려진 알고리즘이 널리 사용되고 있다. 하둡은 맵리듀스 알고리즘을 구현한 프레임워크이지만 스파크라고 하는 새로운 프레임워크에 의하여 그 성능이 월등히 개선되었다. 본 논문에서는, 스파크에 기반하여 구현된 KNN 연결 알고리즘을 제안하였으며, 이는 인메모리(In-Memory) 연산 기능의 장점으로 하둡보다 빠르고 보다 효율적일 것으로 기대한다. 실험을 통하여, 수행시간에 영향을 주는 요소들에 관하여 조사하였으며, 제안한 접근 방식의 우수성과 효율성을 확인하였다.

ABSTRACT

K Nearest Neighbor Join (KNN Join) is a simple yet effective method in machine learning. It is widely used in small dataset of the past time. As the number of data increases, it is infeasible to run this model on an actual application by a single machine due to memory and time restrictions. Nowadays a popular batch process model called MapReduce which can run on a cluster with a large number of computers is widely used for large-scale data processing. Hadoop is a framework to implement MapReduce, but its performance can be further improved by a new framework named Spark. In the present study, we will provide a KNN Join implement based on Spark. With the advantage of its in-memory calculation capability, it will be faster and more effective than Hadoop. In our experiments, we study the influence of different factors on running time and demonstrate robustness and efficiency of our approach.

키워드 : 빅데이터, K-최근접 이웃 연결, 스파크

Key word : Big data, KNN Join, Spark

Received 22 May 2017, **Revised** 25 May 2017, **Accepted** 30 June 2017

* **Corresponding Author** Chung Yeongjee (E-mail: yeongjee@gmail.com, Tel:+82-63-850-6887)
Department of Computer Engineering, Wonkwang University, Iksan 54538, Korea

Open Access https://doi.org/10.6109/jkiice.2017.21.9.1731 print ISSN: 2234-4772 online ISSN: 2288-4165

Ⅰ. Introduction

K Nearest Neighbor Join (KNN Join) is a kind of machine learning algorithm. The core idea of the algorithm is that given a test dataset R and a training dataset S, for each data r in R, find k nearest neighbor data in S. In the past it was usually used in small dataset. As the volume of data increases, how to deal with large and high dimension data by KNN Join is a meaningful topic.

KNN Join is a computational intensive task. The common implement approach is for each r in R, required to scan whole dataset S, compute the distance between r and every s in S(in this paper, we use Euclidean distance). Therefore, the complexity of KNN Join is O(|R|×|S|×|d|), where d is the dimension of the data. In this experiment, if only the dimension of the dataset is 2 and the order of magnitude is 10^5 for both R and S, the computation time is intolerable. There is a lot of related works to solve this problem [1,2], but as the scale of data increases it is also significantly limited to run this algorithm on a single machine. The most effective method is to distribute and parallel on a cluster.

MapReduce is a programming model for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner. It was firstly proposed by google and implemented by Hadoop, an open source software framework. MapReduce programs read input data from disk, map a function across the data, reduce the results of the map, and store reduction results on disk. So Hadoop MapReduce is a batch computing model which can't adapt to real-time calculation.

Originally developed at the University of California, Berkeley's AMPLab, Spark is a fast and general-purpose cluster computing system. Spark provides programmers with an application programming interface centered on a data structure called the resilient distributed dataset (RDD). Spark's RDDs function as a working set for distributed programs that offers a (deliberately) restricted form of distributed shared memory. So it is based on memory computing that runs programs up to 100x faster than hadoop MapReduce in memory, or 10x faster on disk and can process real-time request as well. Spark also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming. But there is no KNN implement based on Spark MLlib until now.

In this paper we propose a MapReduce-based approach for KNN Join run on Spark. We split the dataset R to many partitions and S to many cache blocks in order to adapt very big data. For each r in R calculate the distance with very cache block of S, and then join them together to find k nearest neighbor.

The remainder of the paper is organized as follows. Section 2 gives background about relative knowledge, Section 3 presents our algorithm implement on Spark, Section 4 gives the experiment results. Second 5 describe the conclusion and further work.

Ⅱ. Background knowledge

In this paper R is used to represent test datasets and r is a data of R, S is used to represent training datasets and s is a data of S.

2.1. Euclidean distance definition

We consider that the r in R and s in S are n-dimensional data. The r is represented as $(r_1,r_2,\cdots,r_i)$ and the s is represented as $(s_1,s_2,\cdots,s_i)$.The distance between r and s is defined as:

$$|r,s|=\sqrt{\sum_{i=1}^{n}(r_i-s_i)^2}$$

2.2. KNN algorithm definition

Given a data r (called a query object), a training dataset S, and an integer k, the k nearest neighbors of r from S, denoted as kNN(r, S), are a set of k objects from S such that: $\forall o \in kNN(q, S)$, $\forall s \in \{S - kNN(q, S)\}$, |o,

$q| \leq |s, q|$.

KNN algorithm is widely used in machine learning and data mining because of its stability, robustness and high accuracy [3]. But it also has following drwabacks: (1) The choice of k value affects the accuracy of results, for different dataset we should do experiment to get the right k value. (2) If samples distribute unevenly, it may affect the classification results.

2.3. KNN Join algorithm definition

Given two datasets R and S (where S is a training dataset) and an integer k, the kNN join of R and S is defined as: kNNjoin(R, S) = {(r, s)| $\forall r \in R$, $\forall s \in$ kNN(r, S)}Basically, this combines each object $r \in R$ with its k nearest neighbors from S.

2.4. An overview of KNN Join using MapReduce based on Hadoop

The solution of KNN join are divided into two categories: Exact solutions and Approximate solutions.

In the literature [4] the researcher provides an exact solution called H-BNLJ(Hadoop Block Nested Loop Join).The main idea is partition each R and S to subset $\{R_1,R_2,\cdots,R_n\}$ and $\{S_1,S_2,..,S_n\}$, every pair of subset(one value from R and another from S) is partitioned into a bucket(each bucket is comprised by a pair like (R_i,S_i)) in the map phase. In each bucket the distance between r in R_i and s in S_i is calculated, then the record order by distance like (r_id,s_id,distance(r,s)) is stored in reduce phase. In this MapReduce procedure, n^2 bucket written into files will be generated. In second MapReduce phase it will read all the files that last MapReduce generates. For each r_id gets the top-k minimum distance and determine the classification. The total cpu cost is O(|R||S|+|R|nklogk).In order to improve the performance, the above research provides another method called H_BRJ(Hadoop Block R-tree Join) that uses the R-tree.

The approximate solutions, H-zkNNJ in literature [4], RankReduce in literature [5] and Voronoi diagram-based partitioning method in [6] are proposed.

In literature [7],the researcher carries out a theoretical and experimental analysis based on different exact and approximate solutions.

2.5. An overview of KNN Join using MapReduce based on Spark

Although Spark comes with a wealth of machine learning code library, but unfortunately there is no KNN Join algorithm now. In the book [8], the author provides a method based on Cartesian product which is very simple to implement. But when the data are very large, this method will produce a large number of intermediate data that are unacceptable. In the literature [9], the researcher provides the KNN implement based on Spark, however, there is no further step to implement KNN Join algorithm on Spark. In the next section the KNN Join implement based on Spark will be provided.

Ⅲ. Handling KNN Join using spark

In this section we present a parallel and distributed KNN Join implement for big data classification based on spark [10]. Data preprocessing is not considered in this paper. All dataset we used in this paper have been formatted. In dataset S each record format as a tuple like $(s\_id,c,s_1,s_2,\cdots,s_n)$, in which s_id is the identifier of the record, c is the classification and $s_1,s_2,\cdots,s_n$ is the attribute. In dataset R each record format as a tuple like $(r\_id,r_1,r_2\cdots,r_n)$, in which r_id is the identifier of the record, $r_1,r_2\cdots,r_n$ is the attribute.

Because both training set and test set may be high dimension and big dataset, how to partition them in order to KNN run on Spark effective is a challenging task. There are multiple other key-points to be considered such as the number of maps or the number of cache block In S. We introduce the main idea and give the core algorithm.

(1) Read the big test dataset(R) into m partitions. It can be simply implemented by Spark api(using TextFile method). This means that there will be m tasks start and

the parallelism is m as well.

(2) Read training dataset(S) into Spark broadcast variable. The value of the broadcast variable will be distributed to each slave node memory. So if the memory can't hold all the data, we should split S into n subset(represented by variable s_subset in algorithm 1). First get one subset(calculate on line 3 of algorithm 1) into broadcast variable for use, then repeat until all subsets are used.

Algorithm1: split S into Broadcast variable

```
1: val s_RDD = textFile(SPath)
2: val sTotalNum = S.count() //the total record of trainging
dataset
3: val s_subset = sTotalNum/sCacheBlockNum //the record num
per cache block
4: val s_broadcast = broadcast(get s_subset record from S)
5: for(i=1 to sCacheBlockNum){
6:    other operation using s_broadcast
7:    s_broadcast = broadcast(get another s_subset record from S)
8: }
```

(3) We illustrate the operation on line 6 of algorithm1. Map function on R can be used to calculate the distance with the subset i of S for each record. For the same key r_id, the value will constitute a distance-classification map(the distance as the key of the map and the classification as the value of the map). Then the key-value pair is output as (r_id,map<distance, classification>).In order to sort elements in map by distance, we use TreeMap as implement of the map. Algorithm 2 provides the detail of the process.

Algorithm2: map R and output key-value pair

```
1: R.map{r=>
2: var map = new TreeMap(distance,classification)
3: foreach(s in subset of S){
4:    calculate distance between r and s
5:    then put into the map(distance,classification)
6: }
7: return (r_id, map(distance,classification))
8: }
```

(4) An RDD will be generated for each iteration, and we join all the RDD to form a final RDD. The key of the final RDD is r_id and the value is the list of map represented as List<Map<distance,classification>>.

(5) Reduce function is used on the final RDD. We merge all the map in the list, as the Map is TreeMap, and all the elements will be sorted by distance. Then we get the first k values from the map and determine the classification. The output is also a key-value pair as (r_id,classification).

The above steps can be illustrated in Fig. 1.

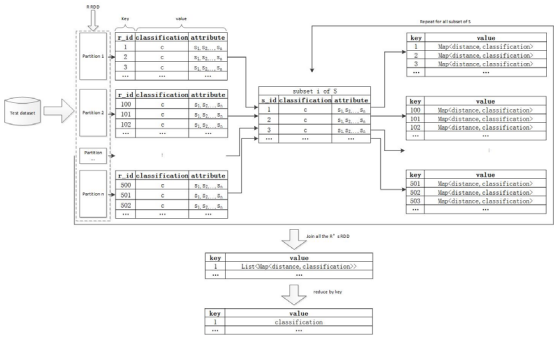

Fig. 1 KNN Join using Spark

Ⅳ. Experimental evaluation

4.1. Configuration

In order to evaluate the performance of the algorithm and the influence of different parameters on the algorithm, we build an experimental environment. It has 7 virtual machines(called node as well) installed on the VMware, among which 1 machine is regarded as a master node and other 6 machines as slave nodes. The VMware is run on a physical machine. The detail configurations of physical machine and virtual machine are showed in table 1 and table 2.

Table. 1 Physical machine configuration

CPU	Intel(R) Xeon(R) CPU E5-2430 0 @ 2.20GHz. 2 Processor, each of which has 6 cores.
Memory	32GB
Hard Disk	500GB
Ethernet	Gigabit ethernet

Table. 2 Virtual machine configuration

CPU	1 processor 4 cores
Memory	3GB
Hard Disk	20GB

We select one node as the master node and the rest are slave nodes. On each node install CentOS 7 operating system, Java1.8 64-bit, Hadoop 2.7.3 and Spark 2.1.

4.2. Datasets

In this experiment we use two datasets:

Susy: it is from the UCI machine learning repository. It is a classification problem to distinguish between a signal process which produces supersymmetric particles and a background process which does not. This dataset has 1 label and 18 features.

Random generated data: it is random generated by a java program. We write a java program to generate different dimensions, different sizes and well-formated data. Compared with UCI datasets，Random generated data has their advantage of being no need to normalize the data before the experiment.

In addition to the impact of the size of data on the algorithm running time, for other all experiments we utilize two datasets of equal size(10^6) for R and S.

4.3. Experiment

We will measure the performance of the algorithm from different dimensions which include the number of R's partitions, data sizes, data dimensions, number of nodes, cache blocks number of S and k value. In the following experiment, we give the default input parameters in table 3 unless there is specification, and then draw a figure according to the result.

Table. 3 Default input parameter for experiment

Number of nodes	3
K value	2
Data dimension	2
Partition number of R	24
Cache blocks number of S	1
Number of test set(R)	105
Number of training set(S)	105

(1) The influence of the R's partition number

Partition is a computing unit in RDD internal parallel computation. The dataset of RDD is logically divided into multiple splits, and each split is called partition. The number of partitions determines the granularity of parallel computing, and each computing in partition is carried out in a task, so the number of tasks is equal to the number of partitions. As we can see from Fig. 2, when the partition number is 2, the algorithm running time is very long because the number of partitions is small and the degree of parallelism is low, at this point only 2 nodes are computing and each node uses only one core of cpu. With the increase of partition number, the parallelism degree of cluster increases, therefore the execution time decreases. But the execution time won't infinitely reduce, when the number of partitions reach a certain range, the running time gradually stabilizes. The best number of the partitions is the total cores of the cluster. In this experiment, we have 3 computation nodes in cluster, each node's cpu is quad cored, so the total cores of the cluster are 3*4=12. It can be seen in Fig. 2 that the computing time is the shortest when the partition number is 12. If we set the partition number(here is 12) several times of total cores in cluster, we will get the maximum degree of parallelism as well, only a little scheduling time will increase.

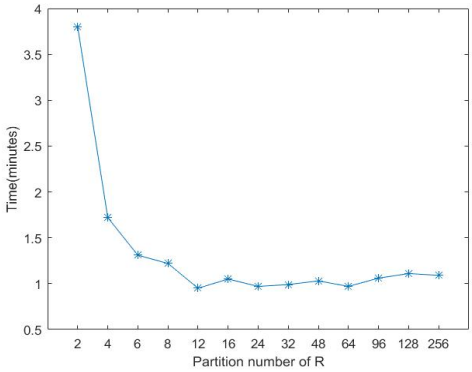

Fig. 2 Running time in different partition number

(2) The influence of data size

Obviously, as the data size increases, the total computation time will increase as is shown in Fig. 3. We fixed the size of S equal 10^6, then change size of R from $0.1*10^6$ to 10^7.

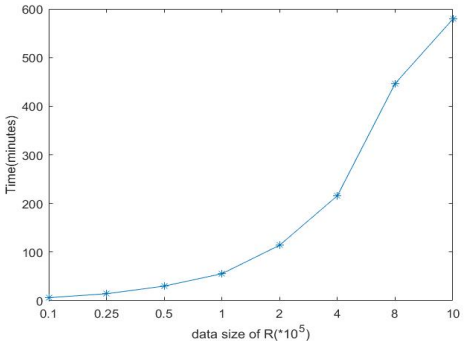

Fig. 3 Running time for different data size

(3) The influence of data dimension

With the increase of data dimension, thecomputational complexity will increase. Therefore the total running time will increase. In this experiment, we use 2 different sizes of dataset ($|R|=|S|=\{10^5,2*10^5\}$), each dataset changes dimension from 2 to 18. The results are shown in Fig. 4.

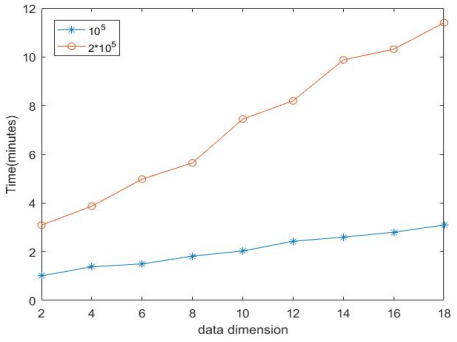

Fig. 4 Running time for different dimension

(4) The influence of the number of nodes

We divide this experiment into two stages. First, we change the number of node from 3 to 7 but fix the partition number to 32. From Fig. 5 we can see that when the number of node is 4, more running time is needed than 3 nodes. The reason is that the number of partitions is not reasonable as we have illustrated in (1). As is described in (1) the best partition number is the node number multiplied each node cores. So in our second experiment, different nodes will set different partition number shown in table 4.

Table. 4 Partition numbers in different nodes

Node number	3	4	5	6
Partition number	3*4=12	4*4=16	5*4=20	6*4=24

The result showed in Fig. 5, at this time we can see as the number of nodes increase, the running time decreases all the time.

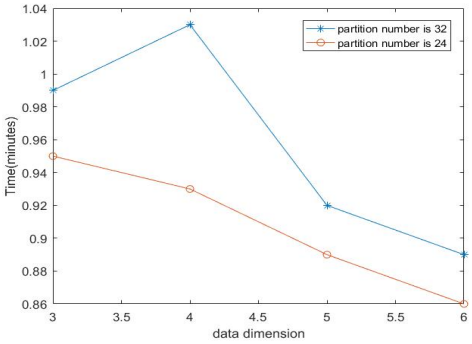

Fig. 5 Running time for different number of nodes

(5) The influence of the cache blocks number of S

The purpose of cache blocks of S is that when S is very large, the capacity of memory can't hold all data, so we load data into memory in batches. The number of S won't increase the computational complexity, so it will not have a significant impact on running time. This is also confirmed in our experiments that we set the cache blocks number of S from 1 to 30, the running time only increases less than 2 times shown in Fig. 6.

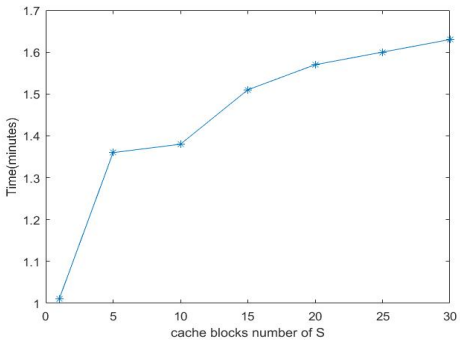

Fig. 6 Running time for different cache blocks of S

Ⅴ. Conclusion

In this paper, we have reviewed KNN Join implement on Hadoop and Spark. Then we propose an algorithm and develop a program for KNN Join based on Spark. This is an exact solution for KNN Join implement. We carry out experiments on different dimensions of different datasets to prove that the algorithm is effective. It is also can be extended to a variety of applications like classification, prediction analysis and clustering analysis. In our future study we will evaluate our approach on a real cluster with the same data and parameter used in this paper. And we plan to research an approximate solution to KNN Join implement based on Spark.

ACKNOWLEDGMENTS

This paper was supported by Wonkwang University in 2016

REFERENCES

[1] C. Yu et al, “High-dimensional knn joins with incremental updates,” *Geoinformatica*, vol.14, no.1, pp.55-82, Jan. 2010.

[2] T. Emrich et al, “On reverse-k-nearest-neighbor joins,” *GeoInformatica*, vol.19, no.2, pp.299-330, Apr. 2015.

[3] J. D. Kim, “A Method for Continuous k Nearest Neighbor Search With Partial Order,” *Journal of the Korea Institute of Information and Communication Engineering*, vol.15, no.1, pp.126-132, Jan. 2011.

[4] C. Zhang, F. Li, and J. Jestes, “Efficient parallel kNN joins for large data in MapReduce,” *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT 2012 - Berlin, ACM, pp. 38-49, 2012.

[5] A. Stupar, S. Michel, and R. Schenkel, “RankReduce processing k-nearest neighbor queries on top of MapReduce,” *Proceedings of the 8th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR‘10)*, pp.13-18, 2010.

[6] C. Ji et al. (2016, August). Inverted Voronoi-Based KNN Query Processing with MapReduce, *2016 IEEE Trust Com-BigDataSE-ISPA* [Online]. pp.2263-2268. Available: http://ieeexplore.ieee.org/document/7847232/.

[7] G. Song et al, “K Nearest Neighbour Joins for Big Data on MapReduce: A Theoretical and Experimental Analysis,” *IEEE Transactions on Knowledge and Data Engineering*, vol.28, no.9, pp.2376-2392, Sep. 2016.

[8] M. Parsian, *Data Algorithms: Recipes for Scaling Up with Hadoop and Spark*, 1st ed. Sebastopol, CA: O'Reilly Media, Inc., Jul. 2015.

[9] Z. Sun et al. (2016, July). Migrating GIS big data computing from Hadoop to Spark: an exemplary study Using Twitter. *2016 IEEE 9th International Conference on Cloud Computing(CLOUD 2016), IEEE* [Online]. pp.351-358. Available: http://ieeexplore.ieee.org/document/7820291/.

[10] K. S. Park, J. H. Choi, “Design and Implementation of a Search Engine based on Apache Spark,” *Journal of the Korea Institute of Information and Communication Engineering*, vol.21, no.1, pp.17-28, Jan. 2017.

기가기(JI JIAQI)

2007.7 Nanchang university of computer science and technology, bachelor, china
2010.1 Nanchang university of computer application technology, master, china
2016 - Wonkwang university of computer Engineering, Ph.D student, korean
※관심분야 : Big Data Processing

정영지(Chung Yeongjee)

1995 ~ 원광대학교 컴퓨터 · 소프트웨어공학과
1993 ~1995 한국전자통신연구원
1987 ~1993 삼성종합기술원
연세대학교 전기공학과 통신공학 (박사)
※관심분야 : 컴퓨터 네트워크,빅데이터 정보처리, 모바일 컴퓨팅