

# Implement MapReduce Function Using Python(multiprocessing, pyspark)

Dayoung Kang  
Smart Farm Department  
Jeonbuk National University  
Jeonju, Korea  
[kallzero1008@jbnu.ac.kr](mailto:kallzero1008@jbnu.ac.kr)

**Abstract**—Advances in technology and the explosion of data have made it possible to generate and collect much larger amounts of data than before. Known as big data, which are too large to be analyzed immediately. This study presents a method for distributing data by applying MapReduce to the linear regression algorithm using. In this study, we implemented the Map and Reduce function using Pyspark and compared its performance with MapReduce implemented in local(M1) and server(DIONE) environments developed with multiprocessing. The experimental results showed that distributed processing in the DIONE server environment (48cores), which has many CPU cores, is more efficient than the M1 local environment (8cores). And the performance of MapReduce implemented with Pyspark was better than multiprocessing.

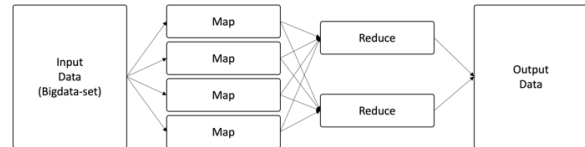
**Key Words:** Bigdata, MapReduce, Linear Regression, Multiprocessing, Pyspark

## I. INTRODUCTION

Advances in technology and the exponential growth of data have revolutionized vast amounts of data, which is referred to as big data. Bigdata is any dataset that contains volumes of information and complex data. It has 4 characteristics called 4V: Volume, Velocity, Variety, Veracity. Volume is the quantity of data. Velocity is the speed of the data during handling and generating. Variety refers to the range of data types and sources. And Veracity is related to the truth of data, which is important for precision in analysis. Additionally, “Value” is the importance of the data, and this is a very significant feature recently.[1] Various approaches have been developed to address the processing efficiency issues caused by big data, one of which is the utilization of the MapReduce framework. This study deals with the implementation of distributed processing of big data using MapReduce. Section 2 describes MapReduce and its working principle. Section 3 applies Ordinary Least Squares (OLS)-based linear regression model using MapReduce to show the code and working principle.

## II. METHOD

### A. MapReduce



**Figure 1.** Overview of MapReduce

MapReduce is a “distributed” computing framework for Bigdata (large scale data) processing that splits data into small pieces, called chunks, and stores them on multiple data nodes. It is called because it is composed of Map and Reduce operations. Map is the first step in a MapReduce job, taking the key-value pairs of an input chunk and outputting them as a list of new key-value pairs, meaning that for each input key-value pair, the map function can generate one or more new key-value pairs. Reduce is the second step in a MapReduce operation and deals with grouping values that have the same key. It takes an input key and a list of values for that key, and outputs a list of new values. In other words, the reduce function can take values for the same key and generate one or more new values[2].

### B. Ordinary Least Squares

Ordinary Least Squares, called OLS[3], is a technique to estimate coefficients in linear regression equations. It describes the relationship between independent quantitative variables and dependent variables. This OLS method is composed to minimize the sum of square differences between the observed and predicted values. To obtain the optimal set of parameters, the derivatives of the error for each parameter must be zero. If we optimize the formula, we can use this “ $b = (X'X)^{-1}X'y$ ” equation.

### C. Pre-Processing

We used “Loan Prediction Based on Customer Behavior” dataset by Kaggle. It aims to predict who possible defaulters are for the consumer loans product.

Index	Column	Dtype
1	Income	Int64
2	Age	Int64
3	Experience	Int64
4	Married/Single	Object
5	House Ownership	Object
6	Car Ownership	Object

7	Profession	Object
8	City	Object
9	State	Object
10	Current Job years	Object
11	Current House years	Int64
12	Risk Flag	Int64

**Table1.** Data columns

Using 3 steps of process pre-processing the data following this.

#### Step1. Label Encoding

The Dtype of dataset(Table.1) indicates that there is categorical data. Since the model only needs numeric data, we fit the train data with a label encoder and transform the validation and test data.

#### Step2. Down Sampling

Our target “Risk Flag” features have imbalanced class. Class 0 has 221,004 and Class 1 has 30,996 samples. This different number of data samples per class can lead to biased learning and not a good model. We thought we had enough samples to fit class 0 with the number of samples in class 1, so we downsampled all classes to 30,996.

#### Step3. MinMax Scaling

The reasons for using Min-Max Scaling in a linear regression model to normalizes the range of variables, reduce the impact of outliers and improve convergence speed.

### D. System Model

In Map operation, splits the input data into the specified number of partitions, each consisting of a tuple of the form (X\_partition, y\_partition). Takes a data partition and the current model parameters as input to compute the gradient. A data partition consists of the input attribute data (X) and the actual values for that attribute (y). The function that computes the gradient runs in parallel on each data partition. In Reduce operation, takes the intermediate gradients and the learning rate as input, combines the gradients, and updates the model parameters. It then sums the intermediate gradients, multiplies them by the learning rate, and returns the updated model parameters.

```
def split_data_into_partitions(X, y, num_partitions):
    data_partitions = []
    chunk_size = len(X) // num_partitions

    for i in range(num_partitions):
        start_idx = i * chunk_size
        end_idx = (i + 1) * chunk_size
        X_partition = X[start_idx:end_idx]
        y_partition = y[start_idx:end_idx]
        data_partitions.append((X_partition, y_partition))

    return data_partitions
```

**Figure2.** Algorithm of Data Partitions

Define the size of each partition, chunk\_size,

calculated by dividing dataset into the number of partitions. Each partition consists of a tuple of the form (X\_partition, y\_partition) and is returned by storing it in the list data\_partitions.

```
def map_function(data_partition, params):
    X, y = data_partition
    gradients = np.dot(X.T, np.dot(X, params) - y)

    return gradients
```

**Figure3.** Algorithm of Map

Function to compute a gradient on a data partition. Computes a gradient vector utilizing the formula from linear regression using OLS(Ordinary Least Squares)[3]. The reason for using OLS is that when updating parameters in linear regression, we proceed in the direction that minimizes the error. The gradient is a value that indicates in which direction the parameters should be updated to reduce error to adjust the parameters of a linear regression model.

```
def reduce_function(intermediate_results, learning_rate):
    total_gradients = np.sum(intermediate_results, axis=0)
    updated_params = learning_rate * total_gradients

    return updated_params
```

**Figure4.** Algorithm of Reduce

Compute a total gradient based on the gradients computed from each map function and the learning rate. Return the updated parameters by summing the intermediate results and multiplying by the learning rate.

```
def main():
    pool = multiprocessing.Pool()

    # Map step
    intermediate_results = pool.starmap(map_function,
                                       [(data_partition, params) for data_partition in data_partitions])

    # Reduce step
    params = reduce_function(intermediate_results, learning_rate)

    pool.close()
    pool.join()
```

**Figure5.** Algorithm of Multiprocessing

Create a Pool object using multiprocessing as a function for parallelization[4]. Then, in the map step, execute the map\_function(figure.3) in parallel using "pool.starmap" and get the intermediate result. In the reduce step, we call reduce\_function(figure.4) to get the final parameters. Finally, we end the parallelization.

```
def main():
    params = spark.sparkContext.broadcast(np.zeros(len(X_transposed)))

    for _ in tqdm(range(num_iteations)):
        rdd = spark.sparkContext.parallelize(data_partitions)

        # Map step
        intermediate_results = rdd.map(lambda x: map_function(x, params)).collect()

        # Reduce step
        params = spark.sparkContext.broadcast(
            reduce_function(intermediate_results, learning_rate))
```

**Figure6.** Algorithm of Pyspark

As multiprocessing, using algorithm of map

function(figure.3) and reduce function(figure.4). Param is a variable representing the initialized model parameters, which should be unchanged and shared across all worker nodes. Therefore, we use broadcast to efficiently replicate params to all nodes in the cluster. Spark uses a data structure called a Resilient Distributed Dataset(RDD) to distribute data.[5] When we using parallelize method to convert “data\_partitions” to an RDD, the data is distributed in parallel across multiple nodes in the cluster. Each node acts as a worker, performing computations on its own partitions. In the map steps, performs the task of applying “map\_function” for each data partition. “map\_function” is a function that computes the gradient for each data point, using the current model parameters params to perform the computation. Then map applies the function to the elements in each partition and converts the result of each partition to a new RDD. In the reduce steps, combine the gradients and update the model parameter based on the learning rate. This iterative learning process allows to handle large amounts of data and train the model in parallel in a distributed environment.

### III. RESULTS

Environment	CPU	Times
M1	8 cores	4min 2sec
DIONE	48 cores	1min 4sec
PySpark	8 cores	41sec

**Table2.** Results

With multiprocessing, we implemented the map and reduce operations in Python and compared the parallel results using different CPU cores. We used total 60,000 samples of data. Locally (pycharm with M1) it took about 4 minutes and 2.028 seconds with 8 CPU cores, and on jupyterlab (DIONE) it took about 1 minute and 3.902 seconds with 48 CPU cores. Using PySpark, it takes more less elapsed time than works with multiprocessing. We can see that the operations we implemented perform well in parallel and get faster as the number of CPU cores increases and Spark’s distributed data processing capabilities and data structures such as memeory based RDD are optimized for handle big data.

### IV. CONCLUSION

First, we compared the efficiency of mapping and reduce operations built with Python in local, server environments and spark. The results showed that mapping and reduce operations implemented with PySpark can handle big data more efficiently and flexibly due to parallel processing. In future research, we plan to test the developed model on big data that is larger than the dataset used in this study and make various type of model such as SVM.

### REFERENCES

- [1] Hiba Basim Alwan, Ku Ruhana Ku-Mahamud. Big data: definition, characteristics, life cycle, applications, and challenges. IOP Conferences Series: Materials Science and Engineering 769. 2019. 3-4
- [2] Ferhat Özgür Çatak, Mehmet Erdal Balaban. A MapReduce based distributed SVM algorithm for binary classification. Turkish Journal of Electrical Engineering and Computer Science 24(3). 2015. 6-7
- [3] Walter Krämer. Finite Sample Efficiency of Ordinary Least Squares in the Linear Regression Model with Autocorrelated Errors. Journal of the American Statistical Association 75. 1980. 1005-1009
- [4] Zena A. Aziz, Diler Naseradeen Abdulqader, Amira B. Sallow, Herman Khalid Omer. Python Parallel Processing and Multiprocessing: A Review. Academic Journal of Nawroz University (AJNU), Vol.10, No.3, 2021. 1-10
- [5] PySpark, Hight-performance data processing without learning Scala. IBM Data Science Experience. 2016. 2-5