Random Forest with Distributed Computing System

ENGR-E599

Ronak Shah
School of Informatics, Computing and Engineering
ronashah@iu.edu

Shyam Narasimhan
School of Informatics, Computing and Engineering
shynaras@iu.edu

INTRODUCTION

Data pre-processing and Machine Learning are like the bread and butter for a Data Scientist and Python has some amazing traditional libraries such as numpy, pandas, scikit-learn which becomes best friends for an aspiring Data Scientist.

But, as we are progressing in our endeavor of becoming a proficient Data Scientist, we realized that these libraries have certain limitations when it comes to handling large datasets and working with restricted computational resources.

Thus, we wanted to learn different techniques and tools that could help us with combining our existing knowledge of machine learning and scale our algorithms for parallel processing on much computationally and memory efficient clusters under different HPC settings and hence in this project we are working on a popular machine algorithm of Random Forest and under different settings and tools such as jobblib, spark and dask to explore and compare the performances and gain insights into different parallelism tasks such as multiprocessing, multithreading, data level parallelism etc**..**

KEYWORDS

Random forest, distributed computing system, machine learning, spark, joblib, scikit-learn, dask, hyper-parameter tuning, grid search, multi processing, multi threading.

PREVIOUS WORK

Significant research has been done in the area of distributed data processing. Perhaps the most notable and relevant contribution is the MapReduce programming model [1], which applies the map and reduce functions from functional programming to large datasets spread over a cluster of machines. Since their introduction, the MapReduce concepts have been implemented in several projects for highly parallel computing, such as Apache Hadoop [2]. Chu et al. [3] show how ten popular machine learning algorithms can be written in a “summation form” in which parallelization is straightforward. The authors implemented these algorithms on a MapReduce-like framework and ran them on multicore machines. Also, research such as in [4] and [5] specifically focuses on applying parallel computing on Random Forest in particular and it is also one of the best algorithms considered for parallelizing due to its inherent nature of building trees in parallel.

DATASET

The data used in this study were acquired from PAMAP2 (Physical Activity Monitoring in the Ageing Population), a publicly available dataset containing physical activity data.

Physical activity is widely known to be one of the key elements of a healthy life. The many benefits of physical activity described in the medical literature include weight loss and reductions in the risk factors for chronic diseases. With the recent advances in wearable devices, such as smartwatches or physical activity wristbands, motion tracking sensors are becoming pervasive, which has led to an impressive growth in the amount of physical activity data available and an increasing interest in recognizing which specific activity a user is performing. Physical activity recognition has numerous applications in fields, such as medical monitoring, healthy aging, active living and rehabilitation, and so on. For many years, the medical research literature has extensively explored the positive effects of physical activity on health.

ARCHITECTURE AND IMPLEMENTATION

Random Forest:

(a) How Decision Tree works:

Decision Tree is an algorithm which recursively splits the data in different branches in a node, so as to decrease the entropy for the classification. It forms the decision tree in greedy approach. For every node, the decision tree takes the feature which decreases entropy the most and splits the data accordingly for the next depth.

The categorical feature is used as it is in the node, and the numerical feature is binarized, that is, it's split across a threshold

value of that feature. So, to use a numeric feature in decision tree, you would have to find the best threshold of that feature which would decrease the entropy most in that split.

(b) How Random Forest works:
Random Forest is an aggregation of multiple decision trees. It follows the bagging method (sampling with replacement) to get the sample, selects features randomly into consideration as candidates for splitting the node, and chooses the best feature among these candidates for splitting the node and making decision tree. In case the target variable is numeric, then the results of multiple decision trees are averaged. If the target variable is numeric, the majority of the results of multiple decision trees is taken into consideration.

(c) How Random Forest can be parallelized:
If we sequentially construct trees, the random forest has high running time complexity. **(O(t*k*n*logn))**, where t is number of trees, K is number of chosen features, and n is size of training set. Increasing the number of trees would increase the accuracy, but also would increase the running time. Training the subset of trees across different processors would save lot of running time. We can build the trees of random forest across multiple cores or multiple nodes in a cluster. If we use only single core, then the algorithm would build decision tree one after another serially, thus consuming lot of time. But if we parallelize the process and distribute the creation of trees among the cores, then at the same time multiple decision trees will be built. In terms of Mapper and Reducer, the mapper would split the samples across the cores and build respective decision trees, and the reducer would combine the results of these multiple decision trees and aggregate the results.

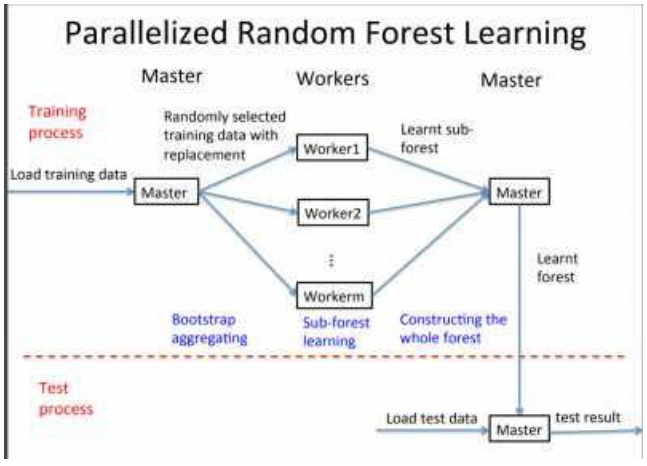


Figure 1. Parallelizing Random Forest

Basics of Hyperthreading, Multithreading and distributed systems

Distributed Computing is the ability to perform multiple tasks at the same time across different computers / modes. This would require sharing of data across the nodes.

We exploit the ability of distributed systems to perform heavy tasks in parallel.

This can also be done within one node if one computer has multiple processors, which in turn has multiple cores.

How parallelism happens in a single Node

In context of parallelism for Random Forests, it's possible to do parallelism in multi-threading mode or multi-processing mode.

Multi-threading – A single process or a program is created, which in turn divides the task in multiple light weighted tasks. These light weighted tasks or processes are called threads.

These threads are executed across multiple cores of the computer. All these threads share common **memory space**, **address space** and **data space**, since they are a part of a single process / program.

While running a multi-threaded programming, we need to ensure **thread safety**, which basically means one thread should not be modifying the data to be used by the other thread. This can be achieved by Synchronization.

Multi-Processing – Multi-Processing happens when multiple programs are running at the same time. When multiple programs / processes run at the same time, each process has its **own** memory, address and data space.

If the data is being used and modified by multiple processes, then there is a need for message passing if there are dependencies between the processes.

The message passing may consume time in the form of communication time.

Hyper threading – Generally a single thread can run in a core at a time. But sometimes multiple threads can run at the same time in the same core. This may be achieved by the effective scheduling and resource allocation of the threads.

This is called hyperthreading.

If two threads can run parallelly in a core by Hyper threading, the speed effectively increases by roughly 1.25 times.

Some limitations of Python in Multi-threading:

Python has a very good memory management system, which they achieve by using reference counting for each object created in Python. When the reference count becomes 0 for an object, at that time the memory allocated for that object is released.

In case of multi-threaded program, there would be lot of referencing of the objects by the multiple threads, and if each

thread is executed in parallel, it would be difficult to maintain the reference count of the object, and the program may crash.

Lots of Python libraries are built in C language, which uses multi-threading. To overcome this problem of multi-threading in Python and to use the libraries effectively, the developers introduced the concept of **Global Interpreter Lock** (GIL), which allows only single thread to be executed at a time in CPU bound programs.

With GIL not released for a process, we cannot do effective multi-threading.

To overcome this limitation, multiprocessing is an effective way, since each process has its own address and memory space.

If we write a thread safe code with effective scheduling for each thread in accessing / modifying the data, then for each thread we can temporarily release GIL and perform multi-threading. Some frameworks of **Cython** like Joblib, Openmp etc. provide this ability.

System Specifications of Future systems Node and our local machine:

By running the command **lscpu** in the linux terminal, we can come to know the specifications of the node.

We have following details given by the command.

Number of Sockets = 2

Cores per Socket = 12

Number of threads per core = 2

Number of CPUs = 48

Number of Sockets equals the number of processors in the node.

We have 12 cores per processor. In total, the node consists of **24 cores**

Each core can work on 2 threads at the same time by hyper-threading.

So that's why we have 48 CPUs (2*12*2). But as noted above in Hyperthreading, the effective speedup of a core across the two threads would be 1.25. The speedup effectively would be 24*1.25 = 30 times.

In our local machine, we have following specifications.

Number of sockets = 1

Cores per socket = 2

Number of threads per core = 2

Number of CPUs = 4

So, the effective speedup in local machine would be 2*1.25 = **2.5 times**.

Scikit-learn:

The Scikit-Learn package builds multiple decision trees of Random Forest by multi-threading.

This means each thread would be sharing the same memory, address and data space in the same node.

To achieve multi-threading, we use joblib library, whose base functions are written in Cython. The loops used in the joblib package are thread safe, and so python internally releases GIL while running the commands across multiple threads.

The parallelism in Scikit-learn is used for fitting multiple decision trees.

Within a single decision tree, parallelism is used to distribute across multiple nodes.

Scikit-learn also uses parallelism to apply each decision tree to the test data, and for feature importance calculation.

Spark:

Apache Spark is an open-source powerful distributed querying and processing engine. Spark allows the user to read, transform, and aggregate data, as well as train and deploy sophisticated statistical models with ease. Spark uses Resilient Distributed Datasets (RDDs) which are a distributed collection of immutable JVM objects that allow you to perform calculations very quickly, and they are the backbone of Apache Spark. As the name suggests, the dataset is distributed; it is split into chunks based on some key and distributed to executor nodes. Doing so allows for running calculations against such datasets very quickly. RDDs operate in parallel leading to an enormous increase in speed.

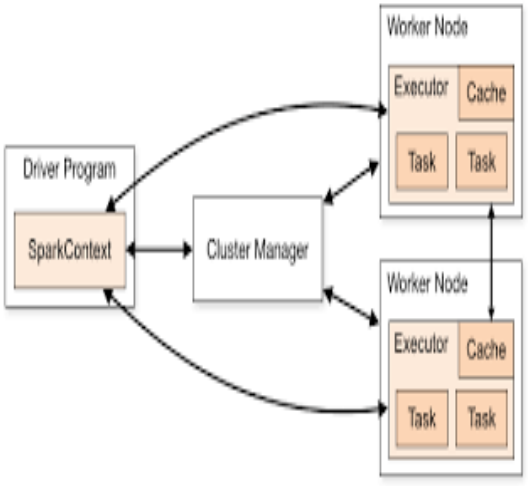


Figure 2. Spark in distributed mode

At a high level, every Spark application consists of a driver program that launches various parallel operations on a cluster. Typical driver program could be the Spark shell itself, and

you could just type in the operations you wanted to run. Driver program access Spark through a SparkContext object, which represents a connection to a computing cluster. In the shell, a SparkContext is automatically created for you as the variable called 'sc'.

Dask:

Dask is available within the Python's ecosystem and its dataframe functionalities are much similar to pandas allowing us in preprocessing data with much ease. Thus, dask natively scales Python and it provides advanced parallelism for analytics, enabling performance at scale for tools you love.
Dask is composed of two parts:

1. Dynamic task scheduling optimized for computation. This is similar to Airflow, Luigi, Celery, or Make, but optimized for interactive computational workloads.
2. "Big Data" collections like parallel arrays, dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

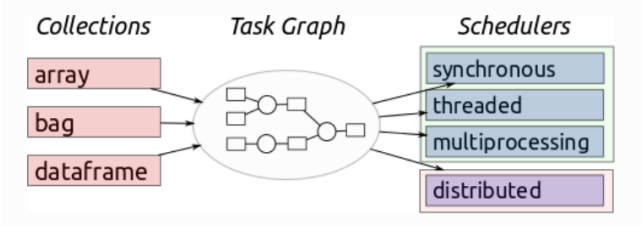
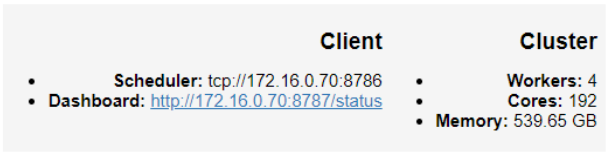


Figure 3. Dask Distributed Computing

Dask setup on Juliet cluster:



METHODS AND EXPERIMENTS

Experiments on local machine:
We have done the data preprocessing using Pyspark, which includes exploratory data analysis, feature transformation and engineering, feature selection, and replacing missing values and all these tasks are parallelized by spark allowing for fast computations. We have implemented the algorithm using Scikit Learn and Spark and have compared the results and running time of both the implementations.

We have done the following processes on our local machine:

(a) Preprocessing:

We have aggregated the data across different users and predicted for the activity ID which is being performed, using the sensors' data. In all we have 2,872,533 rows of the dataset. As of now, we have dropped few columns containing more than 50% null values, and dropped rows containing Null in other columns. This way, we end up with 2,844,868 rows.

(b) Scikit-Learn Implementation:

Using pandas dataframe and Scikit Learn Package, we see the results of Random Forest implementation.

(c) PySpark Implementation:
After preprocessing the data, we implemented the algorithm fitting using the in-built function of PySpark.

Results:

Platform	Preprocessing Time	Training Time
Scikit Learn	1 minute 23 seconds	3 minutes 30 seconds
PySpark	21 seconds	2 minutes 20 seconds

Analysis of results:

Clearly, we see that the performance in distributed systems by spark is much faster which is because it makes optimal utilization of all the available resources and performs true parallelization of all the tasks.

Figure 4. Snapshot of different spark jobs executed in runtime controlled by the scheduler

Experiments on server/cluster:

Scikit-Learn:

In the node of future systems, we ran the scikit-learn module of random forest, which made effective use of multi-threading for the training of random forest classifier.

For running 10 estimators across 40 jobs, it took 40 seconds.

The memory profiler module of Python showed the memory usage increment of 1214 MB for the process triggered while fitting the random forest classifier in this setting.

There was only one process created in this setting, running across multiple threads but consuming more than 300% of CPU memory.

Following is the snapshot of the result.

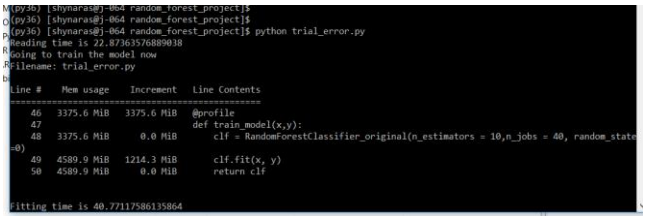


Figure 5. Joblib Multi-Threading Time and Memory Profiling

Cython joblib (Multi-processing):

For fitting the trees, we used multi-processing setting of joblib module, which would then employ serialization / deserialization by pickle.

We modified the forest file from scikit-learning ensemble directory [7] to enable multi-processing with joblib [8]

For running 10 estimators across 40 jobs, it took 53 seconds.

In different settings, the multiprocessing mode was slower than multi-threading mode, and as the number of threads would increase, random forest would be even slower for multi-processing, because of the communication time lost in serialization/deserialization.

The memory profiler module of Python showed the memory usage increment of 200 MB for the process triggered while fitting the random forest classifier. However, we are unsure on how much memory would be consumed by the child sub processes created.

There were multiple processes running concurrently in the node, and all of them were consuming less CPU memory.

Following is the snapshot of the results:

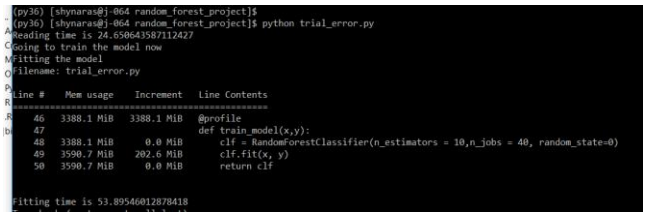


Figure 6. Joblib Multi-Processing Time and Memory Profiling

We ran Random Forest in multi-threading and multi-processing mode for 20 estimators for 10, 20, 30, 40, 50 … 100 jobs.

Following is the snapshot of the time performance in Multi-threading vs multi-processing.

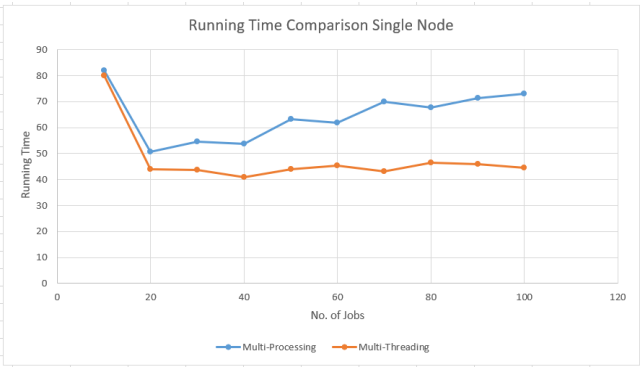


Figure 7. Multi-Threading vs Multi-Processing Time Consumed

We see that as the number of jobs increase, the serialization overhead in case of multi-processing mode increases, resulting in higher communication time.

We don't have this con in case of multi-processing because of shared memory space.

After doing some experiments with Cython's joblib library and scikit-learn which helped us better understand the concepts of multiprocessing and multithreading on a single machine across multiple cores and we now wanted to experiment with running machine learning models on cluster mode across different nodes and having already done that with Spark during our lab exercises we went ahead with dask.

We have done the following experiments on the cluster

(a) Preprocessing:

The preprocessing steps were almost the same as the ones done on local machine but were done with much more efficiency due to higher availability of resources.

Data loading with dask:

Dask works very well with large datasets, in which it scales the data loading process across the clusters and the parts of the data are stored across these machines referenced by its index in the original data. The operations are then performed by the workers on these small chunks of data, thereby parallelism and increasing performance.

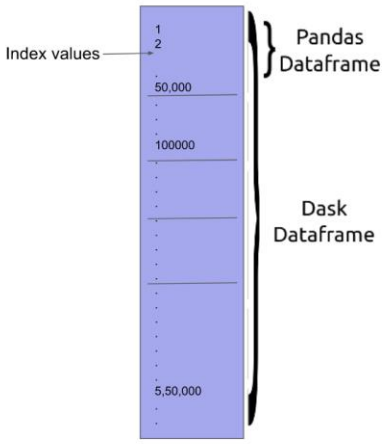


Figure 8. How Dask breaks Dataframe

As can be seen from the above image, dask breaks the standard pandas dataframe into chunks of smaller data-frames and are then stored across different machines on the cluster allowing many jobs to be computed in parallel resulting in much more efficiency while dealing with large datasets.

We loaded our data using dask dataframe and the explored its memory efficiency which can be seen from the below image:

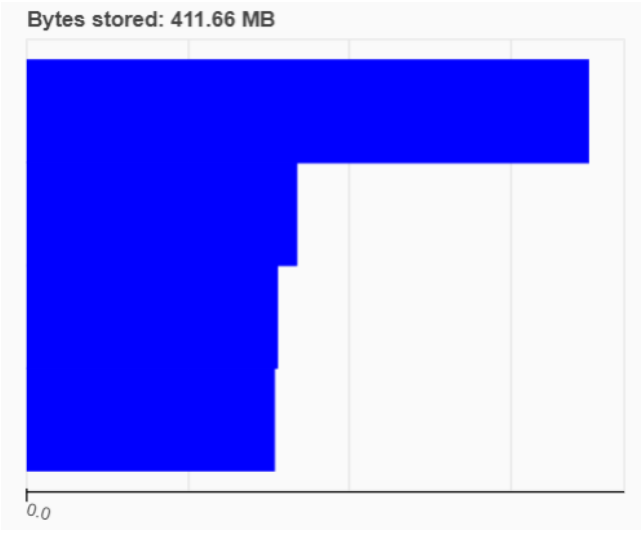


Figure 9. Memory across 4 workers before data loading

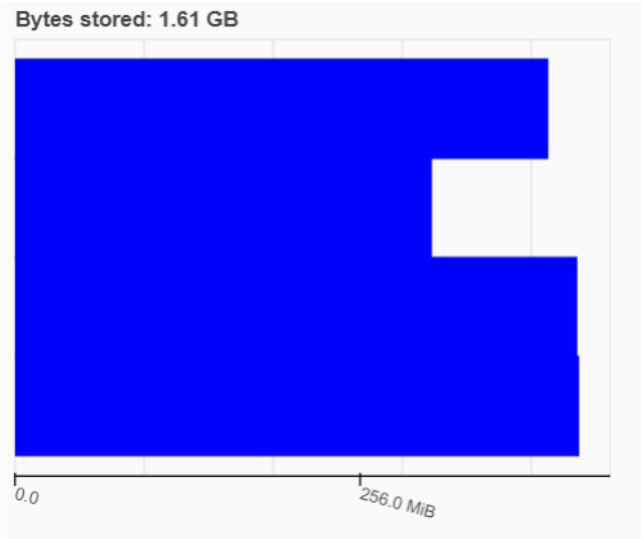


Figure 10. Memory across 4 workers after data loading

The images above show the memory of 4 workers across the cluster before and after loading the dataset.

We can see that the loaded data is spread across all these workers indicating that the data is indeed distributed parallelly.

Parallel computation of operations on data:

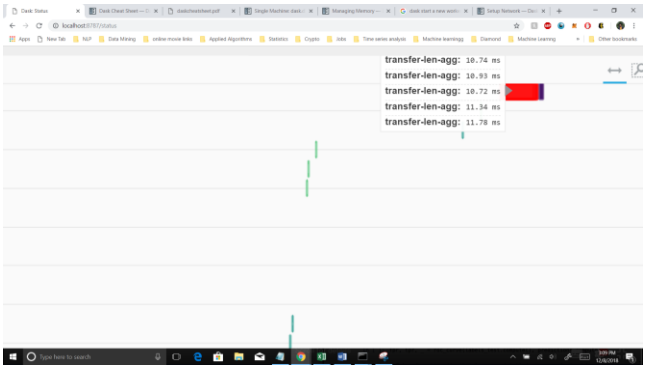


Figure 11. Calculating length of the dask dataframe

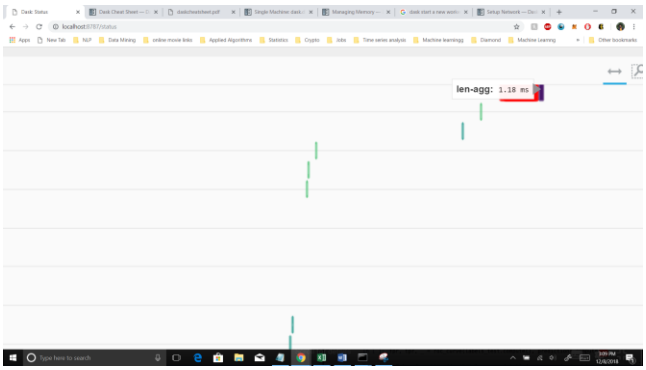


Figure 12. Calculating length of the dask dataframe

The above images show the status of the tasks while performing 'length' calculation operations on the dask's dataframe object. The red lines indicate that the workers across the nodes are performing the length calculating operation on the part of the dataframe local to its node and then the blue line shows the computation of the length of the overall dataframe which combines the results obtained from all the above parallel operations.

(b) Hyperparameter-tuning:

Hyperparameter tuning is an important step in model building and can greatly affect the performance of your model. Machine learning models have multiple hyperparameters and it is not easy to figure out which parameter would work best for a particular case. Performing this task manually is generally a tedious process. In order to simplify the process, sklearn provides Gridsearch for hyperparameter tuning. The user is required to give the values for parameters and Gridsearch gives you the best combination of these parameters.

sklearn Gridsearch:

For each combination of the parameters, sklearn Gridsearch executes the tasks, sometimes ending up repeating a single task multiple time. As you can see from the below graph, this is not exactly the most efficient method:

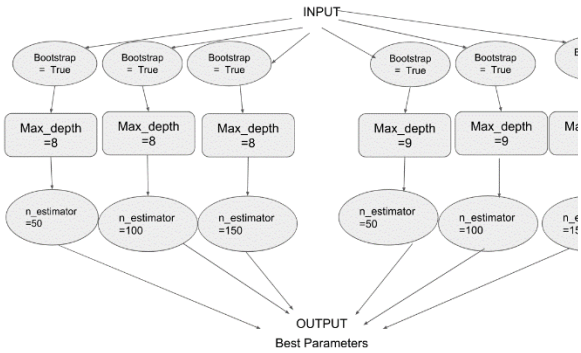


Figure 13. Grid Search Using Dask

Dask-Search CV:

Dask performs grid-search, wherein it merges steps so that there are less repetitions.

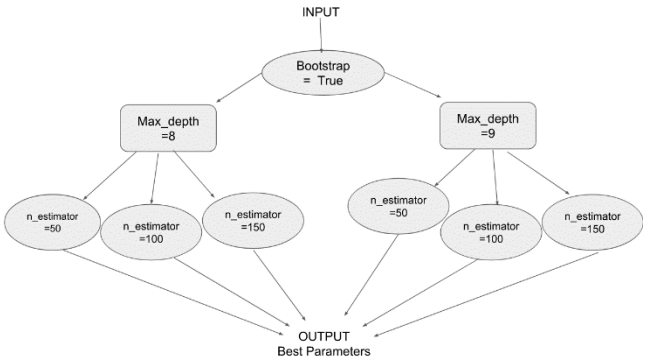


Figure 14. Grid Search Using Scikit-Learn

We have applied grid search using scikit-learn and dask for parameters: 'bootstrap': [True], 'max_depth': [8, 9], 'n_estimators': [10, 20].

Results:

Dataset	Dask	Scikit-learn
PAMPA2	2min 33 sec	4 min 3 sec

Figure 15. Hyper-parameter tuning performances

Analysis of results:

As we can see from the results above there is huge performance improvement using dask for grid search as it combines many repeated tasks and performs them in parallel across the cluster as shown in the figure.

Model Fitting:

Figure 16. Scikit-Learn in Parallel Mode

Figure 17. Dask in Parallel Mode

As we can see from the above images, Dask makes full utilization of all the machines or nodes available on a cluster. Dask has a central task scheduler and a set of workers. The scheduler assigns tasks to the workers. Each worker is assigned a number of cores on which it can perform computations. The workers provide two functions: compute tasks as assigned by the scheduler

serve results to other workers on demand. The central task scheduler sends jobs (python functions) to lots of worker processes, either on the same machine or on a cluster.

We experimented with using scikit-learn and Dask (4 workers) on the server:

Results:

Below are the results of running scikit-learn's joblib parallelism and dask's distributed parallelism across cluster of 2 nodes:

Dataset	Scikit-learn	Dask
PAMPA2(2844868, 52)	1min 29s	1min 18s
1 million data points	35.3	40.9
15 million datapoints	11 min 55 secs	7 min 45 secs

Figure 18. Scikit-learn vs Dask

Analysis of results:

We can see that the time taken by Dask is just few seconds less than the scikit-learn on the PAMPA2 dataset and we cannot see much increase in the performance here even though we are distributing the jobs across 2 nodes in a cluster.

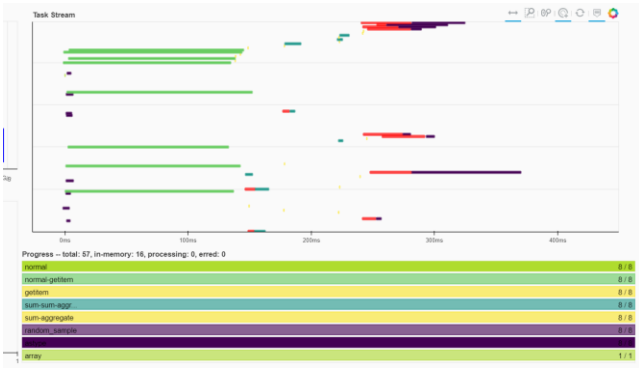


Figure 19. Dask's tasks in action for model fitting

While dwelling deeper looking into the dask's performance UI we can see from the above graph which shows different tasks performed by Dask in runtime while fitting the model. We could see that they are many white spaces between tasks which indicates the time for which our cluster is idle, and the red lines indicates the time spent in communication. Thus, due to the size of the dataset the cluster is not utilized to its full capacity.

We then created a custom classification dataset of 1 million datapoints for verifying our analysis and indeed we could see from the result that scikit-learn is faster than Dask, as Dask's performance is hampered by communication and parallelism overheads.

We then created a bigger custom classification dataset of 15 million data points and ran sklearn and Dask over this data. As we could see from the results, we got a huge speedup while working on a big dataset by parallelizing the jobs across the nodes in cluster as dask is making optimal utilization of resources.

Predictions on the PAMPA2 dataset:

Predictions made by both the scikit-learn and by using dask were almost the same at around 99%.

Conclusion:

Achievements:

We have successfully preprocessed and implemented Random Forest using scikit-learn and Spark on a local machine and compare the performances.
We have implemented Radom Forest using different settings for multithreading and multiprocessing in joblib on the server.
We were successful in setting up dask on cluster across 2 nodes and initialize workers on the same and were able to read and store data in a distributed environment and have applied Random Forest using Dask on datasets with varying datapoints and evaluated its performance with the standard scikit-learn implementation.

Findings:

While working on cluster using dask we realized the importance of the size of the dataset for it to truly take advantage of the entire cluster environment as there are many communication overheads that comes into picture while working on cluster and were able to analyze some of the distributed task being performed in runtime.

While working on parallel computing in single node, we realized that implementing Random Forest in multi-threading setting is more efficient than multi-processing setting because of the time consumed in serialization and deserialization.

Recommendations:

For truly utilizing the abilities of dask's distributing, the data should also be distributed across workers based on the application so that the workers can work on small chunks of data in parallel. Dask, Spark and other such distributed frameworks should be used keeping in mind the size of the dataset and its application in mind.

Future Work:

To improve parallelism, we need to have a good idea about the hyperparameters to be used for training machine learning algorithm, like if we want to run it in multi-processing or multi-threading modes, or if we want to distribute computations across multiple nodes, how many jobs we want the task to be divided in, etc. The optimum hyperparameters would hugely depend on the type of our problem (Random Forest in our case), the input data size, number of features, etc.

We can develop a good heuristic function to get the optimum set of hyperparameters, so that we would have good insight on what distributed system we would want to use and in what situation.

Dask scales very well with the standard python ecosystem we would like to implement it with other machine learning algorithms and also we would like to try many different low level settings provided by dask to get a better control of the communication and memory management by the desk's scheduler. We would also like to implement spark across nodes in cluster and compare it with the dask's implementation.

ACKNOWLEDGMENTS

We would like to thank all Professor Judy Qui and Professor Bo Peng who have helped us immensely throughout the entire course and project always guiding us in the right direction and giving us the necessary knowledge making this project a success.

We would also like to thank Langshi Chen and Selahattin Akkas for guiding us throughout the lab sessions and clearing our concepts in Distributed Systems.

We would also like to thank Future Systems for providing us the infrastructure.

WHO DID What

Ronak Shah:
1. Implemented Random Forest by Py-Spark and Scikit-Learn on local machine.

2. Implemented Random Forest by Dask across 2 nodes and generated visual insights.

3. Experimented on Hyperparameter tuning the parameters for running in various settings.

4. Setup environment for distributed computing on the server.

Shyam Narasimhan:
1. Did literature review on distributed computing system and parallel computing within single node.

2. Preprocessed the data

3. Implemented Random Forest by Joblib in multi-threading and multi-processing settings on the server and gave comparable results

REFERENCES

[1] Jeffrey Dean and Sanjay Ghemawat, "Mapreduce: simplified data processing on large clusters," in Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation -Volume 6, Berkeley, CA, USA, 2004, pp. 10–10, USENIX Association.

[2] "Apache hadoop," http : / / hadoop . apache . org, Dec 2010.

(PDF) Parallelizing Machine Learning Algorithms. Available from: https://www.researchgate.net/publication/260300430_Parallelizing_Machine_Learning_Algorithms [accessed Dec 09 2018].

[3] C.T. Chu, S.K. Kim, Y.A. Lin, Y.Y. Yu, G. Bradski, A.Y. Ng,and K. Olukotun, "Map-reduce for machine learning on multi-core," in Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference. The MIT Press, 2007, p. 281.

(PDF) Parallelizing Machine Learning Algorithms. Available from: https://www.researchgate.net/publication/260300430_Parallelizing_Machine_Learning_Algorithms [accessed Dec 09 2018].

[4] Genuer, R., Poggi, J.-M., Tuleau-Malot, C., Villa-Vialaneix, N., 2015. Random Forests for Big Data. arXiv preprint arXiv:1511.08327.

[5] Wright, M.N., Ziegler, A., 2015. ranger: A fast implementation of random forests for high dimensional data in C++ and R. arXiv preprint arXiv:1508.04409.

[6] Jianguo Chen, Kenli Li, Zhuo Tang : A parallel random forest algorithm for big data in a Spark Cloud Computing Environment

[7] Forest file
https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/ensemble/forest.py

[8] Joblib file
https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/utils/_joblib.py

[9]
https://www.researchgate.net/publication/260300430_Parallelizing_Machine_Learning_Algorithms