Issues in Information Systems
Volume 22, Issue 2, pp. 239-249, 2021

DOI: https://doi.org/10.48009/2_iis_2021_252-263

Distributed KNN algorithm using virtual machines

Joshua Thomas, *Purdue University Northwest, thoma662@pnw.edu*
Yash Patel, *Purdue University Northwest, patel971@pnw.edu*
Zachary Nippert, *Purdue University Northwest, znippert@pnw.edu*
George Stefanek, *Purdue University Northwest, stefanek@pnw.edu*
Ronald C Rabello De Castro, *Purdue University Northwest, rdecastr@pnw.edu*

Abstract

This paper describes a technique for distributing a machine learning task between multiple computers to speed up processing of large data sets. The goal of the project was to create a program where a user submits data to a server, the data is then split and distributed to multiple computers for processing and the results sent back to the server. The system was tested using a virtual network of six virtual machines (VMs) to demonstrate performance when computer resources are limited for distributed processing. The data was spread across each of the nodes on a single computer with one master and five subservient VM nodes. The KNN (K-Nearest Neighbors) supervised learning algorithm was used to compute statistics on the data and return the results back to the server. The dataset that was used consisted of six features and preprocessing was done so it only contained integers for the KNN algorithm to process. The distributed system used the "Ray" framework with six nodes which was compared to processing of the dataset on a single node. The results showed that processing of the dataset across six nodes (virtual machines) reduced the time for processing by 35.1%. as compared to a single node.

Keywords: big data, distributed computing, machine learning, KNN

Introduction

Big data is becoming more common as there are more problems that require information to be extracted from this data. In order to analyze and process this data, machine learning techniques are often used. Large datasets and machine learning typically require multiple computers to assist in processing the large datasets to deliver results in a timely manner for quick decision making at an organization. To accomplish this, the distributed machine learning environment will typically use fast computers with specialized hardware (GPU processors) or parallel processing hardware to deliver fast results.

However, with all the underused processing power that is available within an organization, the opportunity exists to harness conventional computer resources for the purpose of big data analysis using machine learning. Instead of increasing the number of devices that are being used at an organization or buying specialized hardware, it may be more economical to use the underutilized computers that an organization already owns when building a distributed machine learning system (DL). However, using conventional hardware may increase the time to process big data.

The primary goal of using distributed computing to perform machine learning is to partition the computational workload and assign it to multiple computing resources. One of the challenges is the communication cost in the distributed computing environment. In particular, the iterative nature of many

machine learning algorithms along with the large size of the models and training data typically require a large amount of communication between different machines in the training process.

Previous art in the area has mostly focused on parallelizing the processing of machine learning on large datasets when processing on a single machine becomes impractical. For big data, such parallelized processing has been done using expensive parallel processing computers or very fast special purpose computers on a network. The processing of machine learning tasks across conventional computers on a local area network has also been explored by Provost and Hennessy [Provost & Hennessy, 1996] and provides an opportunity to do distributed machine learning using inexpensive hardware which is a focus of this paper.

A number of surveys on distributed machine learning exist [Verbraeken, et al., 2020; Liu, Chen, & Wang, 2017; Zhang, Alqahtani, & Demirbas, 2017, Peteiro-Barral, & Guijarro-Berdiñas, 2013] that cover: 1) the types of specialized, high speed, computer hardware such as programmable GPUs that can be used to speed up machine learning, 2) scaling up resources through High Performance Computers and distributed processing on a network, 3) training algorithms to train a machine learning model (e.g., supervised and unsupervised learning), 4) methods such as evolutionary and stochastic gradient descent algorithms that allow the machine learning algorithm to improve itself, 5) ensemble models where multiple models can be combined for ensemble learning, 6) hyperparameter optimization algorithms that can be used to improve performance by optimizing the parameters of the machine learning algorithm(s), 7) partitioning of dataset approaches to distribute processing across nodes, 8) techniques that enable the interleaving of parallel computation and inter-worker node communication (e.g., bulk synchronous parallel, state synchronous parallel, approximate synchronous parallel, and barrierless asynchronous parallel) and 9) various architectural topologies that can be used in distributed machine learning (e.g., trees, rings, peer-to-peer, and parameter server). Additionally, the surveys describe various machine learning frameworks such as Apache Spark MLlib, Google MapReduce, Ray, and many others including the Keras library that has a backend to run atop Google's Tensorflow.

However, little has been explored in leveraging existing resources even further to achieve efficiencies such as the use of multiple virtual machines running on a single computer or on computers on a local area network (LAN). The overarching goal of this project is to build a distributed machine learning system which will compute statistics based on big data using multiple virtual machines as the distributed processing network. Datasets are divided up for distributed processing and the nodes return statistical data to a master server which then will be used to create useful reports with the combined data. The project will also include a front-end that will enable the display of the result driven by a distributed machine learning system.

This paper will describe the process to set up and develop a distributed machine learning system which will process data across the system and develop a statistical report based on the data. The configuration of the proposed system is to have a master node that will be used to host the code and a web server. The webserver on the master node will be accessible from an intranet allowing users throughout an organization to upload data for processing using machine learning. After processing the data, results will be presented to the end user as a downloadable report. The report itself will consist of data including feature ranking, metrics computed with KNN, and a confusion matrix of the presented metrics; additionally, it will include a timestamp of the processing time of the data.

Background

Some previous research has focused on how to optimize various aspects of distributed machine learning tasks on multiple processing nodes.

Crotty, Galakratos, and Draska, [Crotty, Galakatos & Kraska, 2014] describe a system called Tupleware that they developed to split up execution of machine learning across a small cluster that can speed up processing by a unique technique to optimize processing by predicting execution behavior and processing specific functions of the overall task. The system defines machine learning workflows by supplying user-defined functions to API operators (e.g., map and reduce). The system attempts to predict execution behavior on each processing node. An Optimizer converts the workflow into the Function Analyzer that examines each user-defined function to gather statistics for predicting execution behavior and then forms an execution plan. The optimizer converts the workflow into a distributed program and applies low-level optimizations that specifically target the underlying hardware using the previously gathered user-defined function statistics. A Scheduler plans how to optimally deploy the distributed program on the cluster given the available resources. The program is automatically deployed on the cluster where each node has four hyperthreads. The defined workflow tasks are assigned to dedicated threads. It was found the this approach significantly outperformed Hadoop because of the substantial I/O overhead required for materializing intermediate results to HDFS between iterations. Tupleware was able to cache intermediate results in memory and perform hardware-level optimizations to improve CPU efficiency.

Li, et al. [Li, et al, 2014] describe scaling of machine learning using distributed machine learning on large data sets using a parameter server. In their system data and workloads are distributed over nodes, while the server nodes maintain globally shared parameters. The framework manages asynchronous data communication between nodes, and supports consistency models, elastic scalability, and continuous fault tolerance. It has the advantage of factoring out commonly required components of machine learning systems so that application-specific code can remain concise. The parameter server has key features of: 1) it has efficient communication where the asynchronous communication model does not block computation, 2) it optimizes machine learning tasks to reduce network traffic and overhead, 3) it uses flexible consistency models where relaxed consistency hides synchronization latency, 4) it has elastic scalability where new nodes can be added without restarting the running framework, 5) it has fault tolerance and durability that it can recover from and repair non-catastrophic machine failures within one second without interrupting computation, and 6) it has ease of use where the globally shared parameters are represented as vectors and matrices that facilitate development of machine learning applications. The system is novel in that it can relax a number of otherwise hard systems constraints since the associated machine learning algorithms are tolerant to perturbations making it capable of scaling to industrial scale sizes. The system was found to perform well where bulk communication reduced the communication cost, message compression reduced the average (key,value) size to approximately 50 bits, and when a server node was terminated, the parameter server was able to recover the failed node within one second.

Ulanov, simanovsky and Marwah [Ulanov, Simanovsky & Marwah, 2017] propose a system for estimating in advance how many nodes should be used for parallelizing a machine learning task using distributed computer resources. They measure the scalability by means of the speedup an algorithm achieves with more nodes. They propose a framework for distributed processing using time complexity models for gradient descent and graphical model inference (inference algorithms) to achieve this. They build

theoretical models and validate these experimentally. The framework enables one to build a speedup plot for a machine learning algorithm and use it to estimate an optimal number of nodes that will be needed. The framework views these algorithms as a combination of computation and communication steps. Parallelization techniques are algorithm independent. The framework uses input from algorithm-specific computation and communication time complexity formulas, but unlike previous work in the discipline, does not require any test runs or profiling of the algorithms. The results showed a close match between their models and empirical data.

Provost and Hennessy [Provost & Hennessy, 1996] describe a method for scaling up to perform distributed machine learning using PCs on local area networks. In their approach, they partition the data set and distribute it across a local network of heterogeneous workstations. They used a standard rule-learning algorithm, but modified it to allow cooperation between learners. The learner workstations communicate with each other by passing messages that facilitate sharing of learned knowledge. The distributed learner workstations cooperate to ensure that the group learns rules that only are satisfactory over the entire data set. Statistics from the training set are used to estimate the probability that a learned rule is correct. The results as compared to learning rules generated on a monolithic processor were essentially the same.

Lee, et. al. [Lee, et al., 2017] use codes that offer robustness against noise (e.g., straggler nodes, system failures, communication bottlenecks) to speed up distributed machine learning. Matrix multiplication and data shuffling distributed learning algorithms are used. For matrix multiplication, codes are used to alleviate straggler nodes and show that if the number of homogeneous worker nodes is n, and the runtime of each subtask has an exponential tail, then coded computation can speed up distributed matrix multiplication by a factor of log n. For data shuffling, communication bottlenecks can be reduced by caching data which can reduce the communication cost. It was found that distributed processing was significantly sped up by the introduction of redundancy through codes in the computation.

Zhang, et. al. [Zhang, et al., 2018] describes an adaptive synchronous parallel strategy for distributed machine learning. By using a performance monitoring model, the synchronization strategy of each node (each node may have different processing power in the cluster of computers) with the parameter server is adjusted adaptively by using the performance of each node to produce higher accuracy. This strategy prevents the machine learning model from being affected by irrelevant tasks running on other nodes in the same cluster. The results show that this strategy improves clustering performance and increases the accuracy and convergence speed of the model which increases model training speed.

Georgopoulos and Hasler [Georgopoulos & Hasler, 2013] propose an algorithm to learn from distributed data on a network of computers without exchange of the data-points. Parts of the dataset are processed locally at each machine, and then a consensus communication algorithm is used to consolidate results. An iterative two stage process is used to converge results. It was found that convergence is guaranteed when the learning algorithm imposes a contraction on the learning data. The results obtained on their dataset were superior to those obtained by using a centralized machine to do machine learning.

Seohko Kang [Kang, 2021] uses KNN in combination with graphing to make complex predictions. Called kNNGNN (k-Nearest Neighbor Learning with Graph Neural Networks), Kang's system is able to make complex predictions and display them in line-graph format. The initialization query begins with a search through supplied data, creates a kNN graph and sends it through the graph neural network.

Mai, Hong and Costa [Mai, Hong & Costa, 2015] designed a host-based communication layer that improves network performance of distributed machine learning by reducing network bottlenecks. They reduce network load in the core and at the edges of the network and traffic management to reduce average training time. The key feature of their system is that it is compatible with existing hardware and software infrastructure. They show that overall training time can be reduced by up to 78% using their system.

Chatzigeorgakidis, Karagiorgou, Athanasiou and Skiadopoulos [Chatzigeorgakidis, et al, 2018] used Flink Machine Learning utilizing probability and regression analysis. Their layout utilizes three separate processing platforms in one session. They created multiple sessions and unified them to decrease latency.

Statement of The Problem

Processing big data using supervised learning algorithms to discover relationships in the data are typically compute intensive tasks. To reduce processing time, fast computers are necessary. As big data demands grow, the demand for computational resources may increase to get timely results from processing of large datasets. Cloud resources may be used to give an organization more flexibility to meet demand, but an alternative approach is to use the distributed computer resources that already exist at an organization. In order to use those resources, techniques to split the data for distributed processing and the flexibility to configure distributed processing environments must be developed and tested. Techniques to speed up processing using individual conventional computers without specialized hardware need to also be investigated so that conventional resources can be optimally utilized for machine learning.

Significance of The Study

Previous work has focused on many different techniques to increase the efficiency of processing big data across multiple nodes on a network with specialized high-speed hardware. Others have looked at effective means of optimally scaling up processing using more conventional resources. This approach focuses on the use of conventional resources available for distributed processing of machine learning tasks on a network by looking at increased efficiencies when using VMs to scale up when fast hardware resources are limited. This approach includes using multiple VMs on a single computer to determine if there are efficiencies as compared to processing data using one node.

Based upon the background search, the significance of the study was to develop an agile environment utilizing pre-existing frameworks with limited hardware and software requirements. The solution provides a cost-effective method for processing large datasets which enables users to interact with the system remotely while providing fault tolerance in relation to the dataset. The proposed system is easy to deploy since there is minimum setup and configuration and conventional computer resources are used that are running virtual machines. A full stack system is provided because of the cost effectiveness of using a virtual environment. Also, an initial system can be built and instances of it copied to each node, rather than building each node from scratch.

Methodology

The first part of the project was to develop a network on which to test the distributed system. For ease of use and testing purposes this was done across a virtual network with a total of six virtual machines. The distributed machine learning system consisted of one master node and five subservient nodes. The master node was equipped with 15 GB of RAM and 6 processors on the CPU. Each additional node would be utilizing 7 GB of RAM and 4 processors on the CPU. This configuration was created to test if the environment would work and to determine any improvements in processing time using VMs because this setup would not capture and create delays due to network communication. After the network was built, the KNN machine learning algorithm was selected and the base code modified to allow parallel and multithreaded processing. The algorithm takes advantage of the "Ray" parallel framework to communicate with subservient nodes locally. This method applies a Ray remote object to a function embedded in KNN. This embedded function allows for processing of the Euclidean method on each subservient node in parallel with other nodes, thus increasing the processing time of each iteration of KNN. Once all of the iterations are processed on the nodes, the master node then performs metrics on the processed data and sends a detailed report as a PDF to the front-end for the user to download. To compare the processing time of the distributed system, the same KNN algorithm was run on both the entire six node system and on a single node of comparable processing speed. The front-end is hosted using the Django framework, allowing user interaction to upload large datasets, specify the column in which the labels are chosen from, and the type of machine learning algorithm to utilize in processing the dataset.

The reasoning for testing our methodology only with the KNN algorithm is because it is a relatively simple algorithm to write from scratch and to understand. To evaluate the effectiveness of the Ray framework to process a portion of the algorithm, a lightweight algorithm was used that could be modified for our study. The paper will describe how using Ray was a core component of this research. Future research can focus on the effectiveness of Ray with additional algorithms like Naive Bayes unsupervised learning, linear regression, Supported Vector Machines (SVM), and neural networks to determine if these algorithms can be similarly modified for use in a distributed environment as the one described in this paper.

Potential Benefits

The benefits of using a system of distributed computers to perform machine learning on big data are increased efficiency by reducing the time necessary to process the data. Given large enough datasets, it is anticipated that time delays produced from transferring data over a network could be overcome. While our test was conducted over a virtual network with no delay, we did a preliminary network test with the "iris" dataset, which increased the time to compute the algorithm over 600%. The iris data set is a few kilobytes in size and normally computes on an unmodified KNN algorithm within a minute; the time to run iris with our algorithm took about six minutes. From this information we can conclude that even with a small data set, traversing data across a network will significantly increase the time for the algorithm to process. Therefore, it is inefficient to use this algorithm to compute small to mid-size datasets. Further experimentation needs to be done to find the optimal size a dataset should be to achieve maximum performance over a distributed environment using this approach. Once an optimal size is found it will start to decrease the time it normally takes for a dataset to be processed. Additional benefits when computing

resources are very limited and budgets for purchasing specialized hardware are unavailable are to use VMs on a single computer to mimic a distributed machine learning environment to improve performance as compared to processing a dataset on a single node.

Overview

The initial step was to choose a host system that could handle a large virtual environment. Our host system was a windows machine that was equipped with an 8 core and 16 thread CPU, and 32 GB of RAM. This allowed our team to run 6 VMs simultaneously. Next, a virtualization environment was created using VMware. The virtual environment consisted of one master node (one VM) and five subservient nodes (5 VMs). The master node was equipped with 4 processors for the CPU and 12 GB of RAM, the subservient nodes all were equipped with 2 processors for the CPU and 4 GB for the RAM. The specification of the entire environment was 14 processors for the CPU and 32 GB for the RAM. Additionally, each of the nodes had an additional 20 GB of storage, were configured with NAT network adaptors, and had the Ubuntu 20.04 LTS distro operating system installed. Each node was updated, and python 3.9.4, pip3, and Ray software was installed. The master node additionally had Django, python-virtualenv, and multiple python libraries (numby, time, pandas, matplotlib, itertools, seaborn, csv, sklearn, fpdf, and scipy) installed.

In order to have the nodes communicate with each other and compute data simultaneously, Ray needed to be configured and installed. Ray is a python-based API for building distributed applications, it allows for functions to be computed on machines that are connected to a personal Ray server. Ray can be installed simply enough with the following command on the terminal:

```
pip install ray
```

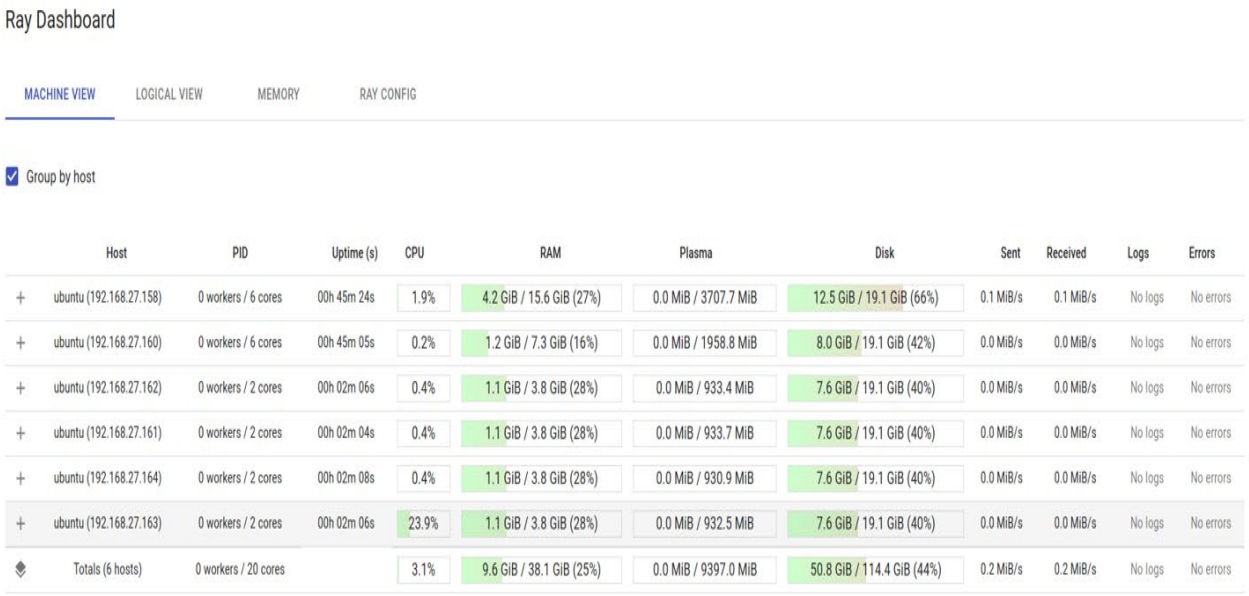
Once Ray was installed the server on the master node was initialized with the following command on the terminal:

```
ray start --head --dashboard-host=x.x.x.x
```

The command allows the server to start and access via the dashboard from other devices by placing the master node IP address with a port number of 8265 into a browser (replace x.x.x.x with address of the server). Even though the Ray server is started on the master node, you must input the following command on the subservient nodes in order for them to communicate with the master node.

```
ray start --address='x.x.x.x:6379' --redis-password='5241590000000000'
```

The redis-password shown is the default password and can be changed for a more secure connection. Once each device is connected to the master node the connection and resources of the network can be viewed by accessing the Ray dashboard as displayed in Figure 1.



	Host	PID	Uptime (s)	CPU	RAM	Plasma	Disk	Sent	Received	Logs	Errors
+	ubuntu (192.168.27.158)	0 workers / 6 cores	00h 45m 24s	1.9%	4.2 GiB / 15.6 GiB (27%)	0.0 MiB / 3707.7 MiB	12.5 GiB / 19.1 GiB (66%)	0.1 MiB/s	0.1 MiB/s	No logs	No errors
+	ubuntu (192.168.27.160)	0 workers / 6 cores	00h 45m 05s	0.2%	1.2 GiB / 7.3 GiB (16%)	0.0 MiB / 1958.8 MiB	8.0 GiB / 19.1 GiB (42%)	0.0 MiB/s	0.0 MiB/s	No logs	No errors
+	ubuntu (192.168.27.162)	0 workers / 2 cores	00h 02m 06s	0.4%	1.1 GiB / 3.8 GiB (28%)	0.0 MiB / 933.4 MiB	7.6 GiB / 19.1 GiB (40%)	0.0 MiB/s	0.0 MiB/s	No logs	No errors
+	ubuntu (192.168.27.161)	0 workers / 2 cores	00h 02m 04s	0.4%	1.1 GiB / 3.8 GiB (28%)	0.0 MiB / 933.7 MiB	7.6 GiB / 19.1 GiB (40%)	0.0 MiB/s	0.0 MiB/s	No logs	No errors
+	ubuntu (192.168.27.164)	0 workers / 2 cores	00h 02m 08s	0.4%	1.1 GiB / 3.8 GiB (28%)	0.0 MiB / 930.9 MiB	7.6 GiB / 19.1 GiB (40%)	0.0 MiB/s	0.0 MiB/s	No logs	No errors
+	ubuntu (192.168.27.163)	0 workers / 2 cores	00h 02m 06s	23.9%	1.1 GiB / 3.8 GiB (28%)	0.0 MiB / 932.5 MiB	7.6 GiB / 19.1 GiB (40%)	0.0 MiB/s	0.0 MiB/s	No logs	No errors
◆	Totals (6 hosts)	0 workers / 20 cores		3.1%	9.6 GiB / 38.1 GiB (25%)	0.0 MiB / 9397.0 MiB	50.8 GiB / 114.4 GiB (44%)	0.2 MiB/s	0.2 MiB/s	No logs	No errors

Figure 1: Visible connected nodes and network resource utilization viewable in Ray Dashboard.

Once the Ray connection was completed, a website was built to provide a user-interface to end users and to connect the algorithm (KNN or other algorithms). Because we needed the website to execute code on uploaded data, Django which is a python-based web framework was used. The Django framework was created in a python virtual environment and ran over port 8000. A website was designed to allow an end user to upload a csv file, enter the column the data was to process on, choose a machine learning algorithm, reset fields, and download report data as displayed in Figure 2.

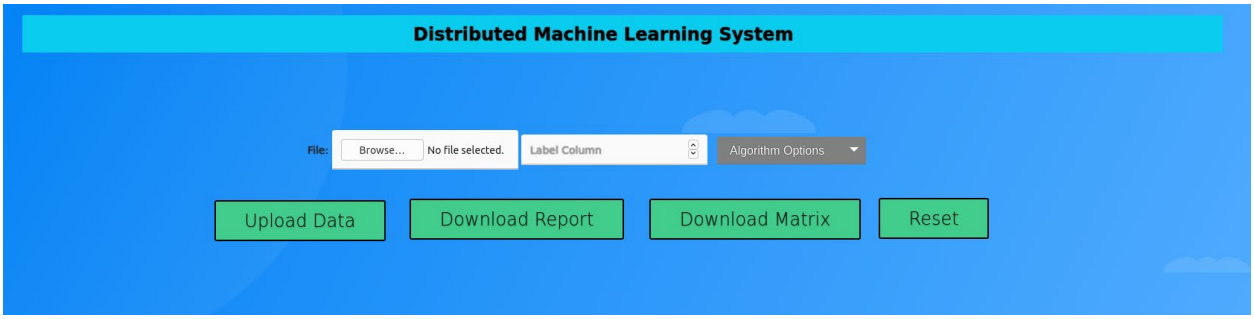


Figure 2: Font-end user selection of uploading csv file, label column, and algorithm type to upload data and download report and matrix.

Once the site was completed, the KNN algorithm was modified to add a Ray object that enables parallel and multi-threaded processing with it. Inside the KNN code there is a section of code that takes the training dataset and compares it to each row inside of the test dataset. It was determined that this was the best place to put the Ray object which would take each iteration of the loop and process it on a remote node. Thus, each row of testing data was compared to the entire training dataset on each remote node. Unfortunately, this meant that the entire training set had to be sent to each node across the network (which is 70% of our

data). The reasons why this is ineffective and future suggestions are discussed later. The implementation of KNN is shown in Figure 3:

```
# Euclidean Distance

def eucledian(p1, p2):
    dist = np.sqrt(np.sum((p1 - p2) ** 2))
     return dist
@ray.remote

def knnRemote(X_train, item):
```

Figure 3: Modification of KNN algorithm to enable parallel and multi-thread processing with Ray Object.

In addition to the algorithm, scripts to preclean the dataset had to be developed. Our implementation of the KNN technique only accepts integers as input so any dataset that is uploaded needs to have only integers as the data. Our actual test data after cleaning was a 3.3 MB file that consisted of COVID data and consisted of 97418 rows and 6 columns.

Results

Table 1 contains comparison data of the metrics from a single node system versus the six node, distributed system.

Table 1: Single System vs. Distributed System Metrics

System Type	Six Node Distributed System	Single Node
Accuracy	97.58%	97.58%
Processing Time	3110s	8849s
Data Lost	0%	0%

Discussion and Analysis

To start off with the baseline computation, when the modified KNN was run on a single system it had comparable specs to the distributed system - it took the system 8849.27 seconds or 2.46 hours to process. Using the same algorithm with the distributed system (6 VMs) took 3110.59 seconds or 51.84 minutes. There was a 62.59% increase in processing time for a single node, that is, the distributed VM system reduced the time for processing by 35.1%. Given the fault tolerance that the Ray framework incorporates, we noticed that there was no loss of accuracy.

In addition, it should be noted again that this test was performed over a virtual network all on the same pc using virtual machines. If we used a LAN, we may have been able to simulate a better accuracy on the amount of time it would have taken for the data to process across the network. The processing may have been affected as well because this was using virtual cores which aren't as efficient as real cores.

There are several recommendations and future work that should be addressed. First, we would like to test the system using a cluster with real end-user computers such as Windows or Linux, and Jetson nano (NVIDA). Also, it is recommended to create the Ray cluster on those devices as nodes and have a physical master server. After setting up nodes and master servers in one LAN network, the application should be tested to see how it performs well on big data. Additionally, it should be noted that originally Raspberry Pis were considered to be used as inexpensive processing platforms. However, using a Raspberry Pi would not work in this configuration since it uses an AMD processor which is not compatible with the Ray framework. As a result, the next alternative for an inexpensive platform was to use VMs as end nodes for the application – which is what we ended up using. Since we used a single computer to host multiple devices we were bound to using CPU processing. Future work will focus on using physical devices on a LAN which have GPU processors as fast and more optimal devices for testing this system.

Further recommendations would be to modify how the Ray object works with the KNN or other learning algorithm. Our configuration can be improved to reduce the amount of data being sent to each node. Currently, the implementation sends 70% of the data to each of the nodes in the network. This significantly increased the amount of data that has to be communicated through the network. If we were able to either modify the algorithm so that we only had to send the test data and train data once it is sent to the nodes or where only test data is sent (30%), efficiency could be improved.

Conclusions

In conclusion, we were able to determine that there is a benefit to having a distributed machine learning system using our custom configuration process big data. It was found that setting up a virtual distributed machine learning network of virtual machines on a single computer can improve performance over a single node when computing resources are limited. The next step using this configuration is to modify the implementation of the KNN and run it across a physical LAN of conventional, Commercial-Off-The-Shelf computers. When using the Ray framework it is best to replicate the environment using computers on the network running Linux since at the time of this research, Ray is not optimized for Windows. If tests are successful using a LAN of conventional computers, this system could be implemented inside of a WAN.

References

Brownlee, J. (2020, February 23). Develop k-Nearest Neighbors in Python From Scratch. Machine Learning Mastery. https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/

Chatzigeorgakidis, G., Karagiorgou, S., Athanasiou, S., & Skiadopoulos, S. (2018). FML-kNN: scalable machine learning on Big Data using k-nearest neighbor joins. *Journal of Big Data*, *5*(1), 1-27. https://doi.org/10.1186/s40537-018-0115-x

Crotty, A., Galakatos, A., & Kraska, T. (2014). Tupleware: Distributed Machine Learning on Small Clusters. IEEE Data Eng. Bull., 37(3), 63-76.

Georgopoulos, L., & Hasler, M. (2014). Distributed machine learning in networks by consensus. Neurocomputing, 124, 2-12.

Hegde, V., & Usmani, S. (2016). Parallel and distributed deep learning. In Tech. report, Stanford University.

Kang, S. (2021). k-Nearest Neighbor Learning with Graph Neural Networks. *Mathematics*, *9*(8), 830. https://doi.org/10.3390/math9080830

Lee, K., Lam, M., Pedarsani, R., Papailiopoulos, D., & Ramchandran, K. (2017). Speeding up distributed machine learning using codes. IEEE Transactions on Information Theory, 64(3), 1514-1529.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., ... & Su, B. Y. (2014). Scaling distributed machine learning with the parameter server. In 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14) (pp. 583-598).

Liu, T. Y., Chen, W., & Wang, T. (2017, April). Distributed machine learning: Foundations, trends, and practices. In Proceedings of the 26th International Conference on World Wide Web Companion (pp. 913-915).

Mai, L., Hong, C., & Costa, P. (2015). Optimizing network performance in distributed machine learning. In 7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15).

Peteiro-Barral, D., & Guijarro-Berdiñas, B. (2013). A survey of methods for distributed machine learning. Progress in Artificial Intelligence, 2(1), 1-11.

Provost, F. J., & Hennessy, D. N. (1996, August). Scaling up: Distributed machine learning with cooperation. In AAAI/IAAI, Vol. 1 (pp. 74-79).

Ulanov, A., Simanovsky, A., & Marwah, M. (2017, April). Modeling scalability of distributed machine learning. In 2017 IEEE 33rd International Conference on Data Engineering (ICDE) (pp. 1249-1254). IEEE.

Verbraeken, J., Wolting, M., Katzy, J., Kloppenburg, J., Verbelen, T., & Rellermeyer, J. S. (2020). A survey on distributed machine learning. ACM Computing Surveys (CSUR), 53(2), 1-33.

Zhang, J., Tu, H., Ren, Y., Wan, J., Zhou, L., Li, M., & Wang, J. (2018). An adaptive synchronous parallel strategy for distributed machine learning. IEEE Access, 6, 19222-19230.

Zhang, K., Alqahtani, S., & Demirbas, M. (2017, July). A comparison of distributed machine learning platforms. In 2017 26th International Conference on Computer Communication and Networks (ICCCN) (pp. 1-9). IEEE.