Benchmarking Gradient Descent On Large Data

Darshan Mehta
darshanm@uw.edu

1 Introduction

Gradient Descent is an iterative optimization algorithm used for finding the minimum of a function [1]. It is widely used to find the optimal parameters of machine learning models which otherwise would require complex matrix decomposition and inverse. These heavy matrix operations are computationally infeasible and hence we use an approximate iterative algorithm to look for the ideal parameters. Gradient descent is based on the observation that if the multi-variable function $F(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $F(\mathbf{x})$ decreases fastest if one goes from $\mathbf{a}$ in the direction of the negative gradient of F at $\mathbf{a}$, $-\nabla F(\mathbf{a})$ [1]. At each step of this iterative algorithm, we take a small step δ in the direction of this negative gradient. This enables us to reach a point in the multi-dimensional parameter space that minimizes the function F.

While this approach seems fairly easy to implement and execute, the issue arises from the large number of data points that we need to process in each iteration over. The implementations of these algorithms usually benefit from extensive vectorization which performs the computation steps in parallel. However, with large datasets, this option is not feasible as it leads to an out of memory issue, especially on devices with low memory capacities such as a regular desktop. While there are ways to perform the same operation iteratively instead of vectorizing, it takes a toll in terms of time since iterating through such a large data might take more time. Hence, people have started to explore alternative systems such as Spark and Snowflake for performing these computations on datasets of large sizes due to the scalability of these systems.

In this project, I analyze the performance of Spark, Snowflake and a locally-executed Python instance on performing a few iterations of a simple gradient descent algorithm with a linear regression objective. I aim to keep my implementation very similar across all the platforms to make a fair comparison. Through these experiments we see how Snowflake has an effortless scale-up strategy that provides almost constant execution time no matter the data size. I also discuss some challenges of working with Snowflake and show how Spark provides a good compromise.

2 Evaluated Systems

For this analysis, I examine the performance of Spark, Snowflake and a locally-executed Python instance on performing the steps of the gradient descent algorithm. In this section, we will briefly discuss some key architectural elements of those systems.

2.1 Local Python Instance

This system represents a whole plethora of low-powered devices which are challenged in at least one of the following ways: low RAM, slow IO, no support for multiprocessing, etc. In order to study the efficiency of these devices in executing the gradient descent algorithm, I run the python program on my laptop computer in a somewhat constrained fashion. The system it is executed is a MacBook Pro Mid 2014 model which is equipped with a 2.6 GHz Dual-Core Intel Core i5 processor, 8 GB 1600 MHz DDR3 RAM and a 256 GB flash storage device.

The code is run on a Python 3.7.0 installation provided via the Anaconda distribution. While Python is more than capable of parallelizing and vectorizing operations by using libraries such as Numpy which

utilize the BLAS or the Apple Accelerate library underneath, I avoid using Numpy for processing large matrices in order to understand to estimate the potential of the low-powered system. Moreover, since we assume that the data is too large to fit in memory, so we never read the entire file and hold it in memory. Instead, we stream from the file one line at a time multiple times and perform operations on the fly. I, however, do use Numpy to perform the on the fly operations as it provides functions that make array processing very simple. Since size of the data in each line is small, Numpy chooses not to create additional threads since the initialization of new threads would have a higher overhead cost that executing them in a serial fashion on one thread. Note: the python code provided along with the project uses the wc shell command to count the lines in the file. This would need to be replaced with a Windows equivalent if run on a Windows machine or by traversing the input file another time and counting the number of lines.

2.2 Spark

Apache Spark is an open-source distributed cluster-computing framework designed to battle the limitations of MapReduce frameworks. It allows for in-memory computations on large clusters in a fault-tolerant manner by providing a distributed memory abstraction called Resilient Distributed Datasets (RDDs) [4]. These RDDs are immutable and provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. Spark is suitable for both: iterative algorithms and interactive/exploratory data analysis. They allow the users to perform two kinds of operations on the RDD namely, transformations and actions. Transformations are lazy operations that define how a new RDD is created from an old RDD and Actions launch a computation to return a value to the program or write data to external storage. the laziness of transformations allows Spark to chain them together to create a lineage of operations to track operations history that created the RDD, thereby simplifying its fault-tolerance mechanism. Spark also offers users to force persist an RDD in memory, but it can spill them to disk if there is not enough RAM.

For our analysis, we work with a Python API provided by Spark called 'PySpark'. We launch a cluster instance on the Amazon AWS EMR which comes with a pre-installed Spark image. We create one master node and 8 worker nodes. Both, the master node and the worker nodes are m5.xlarge instances which have a 4 vCore CPU, 16 GB RAM and a 64 GB AWS Elastic Block Storage (EBS). For the sake of a fair comparison, we do not use the MLLib package provided by Spark for our study, rather, we implement the gradient descent algorithm in a similar fashion as the other systems in comparison. We also use the interactive style provided by PySpark via an IPython Notebook interface to run the programs and monitor the outputs.

2.3 Snowflake

Snowflake is an enterprise-ready elastic data warehousing solution [2]. It offers a pure Software-as-a-Service experience for database systems where users can upload the data to the cloud and immediately start querying the system in a familiar SQL interface. Snowflake also offers built-in extensions for semi-structured and schema-less data. It allows users to choose the platform they want the service to run on and the region in which the servers should reside. Currently supported platforms are AWS, Azure and GCP. Snowflake separates storage and compute. Compute is offered via a shared-nothing architecture which allows seamless scaling and in an attempt to keep up with their style of compute, the storage is designed to not allow edits to the exiting data. Any changes must be over-written to the existing file, as a result making the update operations very expensive. A very interesting thing about Snowflake is their pricing strategy. Instead of having a complex billing options based on CPU node hours, they sell compute in terms of t-shirt sizes such as Small, Medium, Large, etc. where each t-shirt size is associated with a number of credits per hour and the cost of the credits is decided based on the kinds of services required from Snowflake.

Since Snowflake is a relatively new offering, they don't have features which other data warehousing systems offer, such as looping structures, support for PL-SQL, etc. They, however, do offer a Python Connector API which allows users to execute SQL commands from a remote system using Python. While this is associated with a network communication overhead, it makes the execution and analysis process extremely simple. In our analysis, we work with this Python Connector API provided by Snowflake. We use the 'Small' size configuration.

3 Problem Statement and Method

3.1 Problem Statement

The goal of this project is to analyze the scalability of some of the commonly-used systems on performing gradient descent iterations on a linear regression objective. The systems under consideration as explained in Section 2, are a local Python Instance, Spark, and Snowflake. In our analysis, we perform average the time of each of these systems over 50 iterations for different sizes of data. We also compare the platforms on the basis of ease-of-use. In essence, the two main questions we aim to answer are:

1. How do the systems scale with an increase in data size?
2. Which of the systems is more user-friendly?

3.2 Dataset

For this project, we use an EEG dataset available on Kaggle [3]. This dataset is the 12^{th} part of the multipartite massive dataset of EEG readings provided by the author.

The overall size of this data is 19 GB which is split across many CSVs in many folders. After appending all the CSVs together, we get a total of 14 million data points. In order to make the task more challenging, we append a copy of this file to itself thereby doubling the number of rows to 28 million. We then take the columns 1 through 3 to use as features which we then use to predict the 4th column. Since the goal of this project is to understand how the systems scale with increasing size of the dataset, the few extra columns were hence unnecessary and could be discarded. The new dataset hence has 28 million data points and 4 columns. Let us call the features $X1$, $X2$, and $X3$ and the label as Y.

Finally, we create 5 additional copies of this file, each time reducing the number of data points to half of the previous file. This way we have a total of 6 datasets each with different sizes for us to measure the scale-up of the systems.

3.3 Method

In order to make a fair comparison of the systems, the algorithm is designed to be very similar across all the platform while still keeping in consideration the system-specific limitations. I share below the basic high level algorithm used in all the platforms. More detailed implementation is available in the Appendix section.

1. Initialize the initial weights to 0.
2. Compute the objective function using the weights and the data points.
3. Use the weights to make a prediction for each data point.
4. Calculate the error made by the prediction for each data point.
5. Calculate the gradient step for the weights using the errors calculated in the previous step.
6. Update the weights by a small amount in the direction of the negative gradient.
7. Repeat steps 3 through 6 for 50 times.
8. Note the time taken for the 50 steps and recalculate the objective function to ensure the algorithm has indeed reduced the error.

There are slight modification to meet the limitations of each of the systems (see Appendix for details). When working with a local python instance, we do not load the entire dataset in memory, rather, we iterate over the file one line at a time and keep a track of the partial gradients along the way and repeat this for each iteration. When working with Spark, we focus on reducing the number of map and reduce jobs. We combine all the mapping tasks into the single map function, much like we did when working with one line at a time in Python. Then we finish with one reduce to pull the gradient from the workers. The weights are held in memory in the master node and shipped to the worker node before each iteration. We also try to cache the RDD representing the dataset on the worker nodes before beginning the iterations in order to remove the data ingestion time from consideration. When

working with Snowflake, we need to avoid update operations as they are very expensive because of the system design. I had initially implemented the algorithm using updates and the results were quite bad. So to improve the implementation, I use Common Table Expression (CTE) to create a temporary table during each iteration and then perform select operations and aggregations on this CTE. Since I use the Python Connector API, the weights are held in memory on the Python instance running locally on my laptop and are sent to Snowflake with the queries.

4 Results

Table 1 shows the time taken (in seconds) per iteration on average by each of the systems on different sizes of the dataset. The figure 1 shows a visual comparison of the data present in table 1. Note that the Y-axis of the figure 1 is on a $\log_{10}$ scale.

Number of data points	Local Python Instance	Spark	Snowflake (warm cache)	Snowflake (cold cache)
880,023	9.417	0.4	0.101	0.176
1,760,046	16.484	0.72	0.106	0.202
3,520,091	35.557	1.396	0.11	0.215
7,040,181	67.865	2.597	0.105	0.223
14,080,361	132.609	2.744	0.102	0.272
28,160,720	268.522	5.167	0.106	0.313

Table 1: Average time taken (in seconds) per iteration by each of the systems.

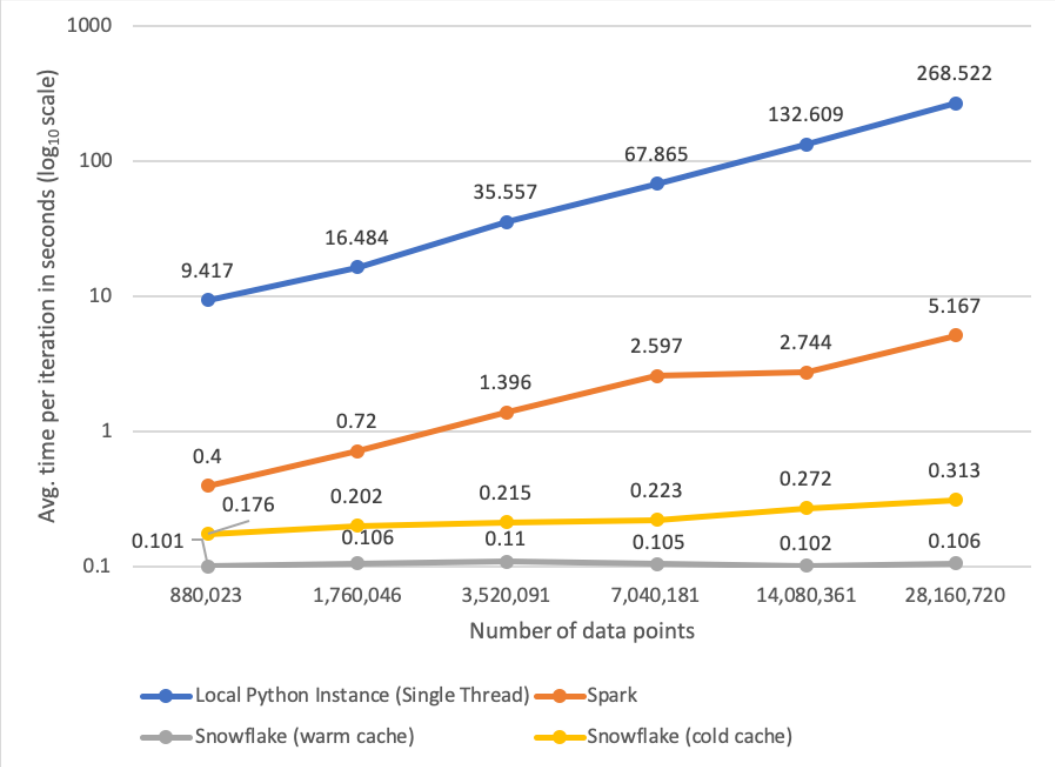

Figure 1: Comparison of the runtime per iteration of each of the systems across various data sizes.

The most conspicuous pattern which we notice in figure 1 is that Snowflake takes almost constant time no matter the size of the dataset. This effect is visible in both warm and cold cache observations. Spark and Python on the other hand have a linear scale-up with increase in data size. Spark is almost $25\times$ to $50\times$ faster than a local Python instance and similarly Snowflake is $17\times$ to $50\times$ faster than Spark for large data sizes. Clearly, Snowflake seems to be the winner of this battle and the answer to the first question in section 3.1, but there's a catch to all this.

We know that Snowflake has one of the best user experiences when it comes to pricing and getting started with the data warehouse, just as they claim in [2]. You pay for a t-shirt size and do not have to worry about the auto scaling or efficient data partitioning. As it is evident from figure 1, Snowflake does a great job at scaling resources with increase in data size. The challenge in working with Snowflake arises from the following two major reasons:

- Lack of support for basic programming concepts such as looping structures, PL-SQL, etc.
- Extremely expensive update operations which limit the ability to perform complex tasks.

Moreover, we notice that Snowflake starts to perform slightly poorly when the warm cache is not available. Speaking practically, it is often not possible for us to rely on the availability of warm cache on a multi-tenant platform where each user runs queries on very different and extremely large tables.

Working on the local Python instance was the simplest to use in comparison with the other systems. This is because of the programming constructs and data structures available in the language which makes it very expressive. Moreover, prior experience with the language biases my choice of platform, and the availability of great machine learning libraries gives Python a huge vantage point. The biggest downfall of using Python is the poor scalability, and the need for increasingly powerful systems for keeping up with larger data loads. As we see from figure 1, each iteration takes a lot of time on Python.

From figure 1, we see that Spark falls in between Snowflake and Local Python Instance in terms of compute time. In my opinion, it is indeed a good intermediate between the other systems in many ways. It offers a decently linear scale up with increasing data sizes. While the programming style is more constrained when compared to a regular Python program, it makes up for this limited expressiveness by providing powerful parallelism constructs that allow the end user to issue tasks that run in parallel in a very simple and effortless manner. Spark also offers great libraries on top of its core framework which make machine learning tasks a breeze. It also offers a SQL like interface to query data from the distributed systems. All in all, it has the flexibility of Python and the power and fault-tolerance capabilities of Snowflake.

5 Conclusion

Considering all the points discussed in section 4, I feel that Spark offers a great between ease of use and power and would be the system of choice for me. That said, Snowflake does show great potential. With its user-friendly transition/setup procedure and pricing strategies, it makes it a breeze to spin up a whole new system without refactoring much of the existing code-base. If it begins to provide support for some of the common programming features, it would become a great competitor to even more systems. Python alone, however, is not as powerful for large data loads. But by providing API for Python gives the other systems a great edge due to the large existing user base of this language.

Acknowledgements

A sincere thanks to Professor Dan Suciu and the TAs Brandon Haynes and Deepanshu Gupta for teaching us about the systems and for facilitating the trial versions of the systems.

References

[1] W. Contributors. Gradient descent, 2019. Retrieved on Nov 30, 2019 from https://en.wikipedia.org/wiki/Gradient_descent.

[2] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226. ACM, 2016.

[3] E. Guttmann-Flury. Eeg file part 12, 2019. Retrieved on Nov 20, 2019 from https://www.kaggle.com/qinxinlaneva/eeg-file-part-12/.

[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

Appendix

Below are the relevant excerpts from the implementation code from each of the sytems compared.

Local Python Instance Implementation

```
# calculate objective function
def calc_obj(file, weights, line_count):
    sum_ = 0.0
    with open(file_name, 'r') as infile:
        for row in infile:
            row_arr = np.array([float(num) for num in row.split(',')])
            sum_ += (row_arr[-1] - (row_arr[:-1].dot(weights))) ** 2

    return sum_ / line_count

# start the clock
start = time()

for i in range(num_iters):
    sum_arr = np.array([0.0, 0.0, 0.0]).reshape(-1,)
    with open(file_name, 'r') as infile:
        for row in infile:
            row_arr = np.array([float(num) for num in row.split(',')])
            x, y = row_arr[:-1], row_arr[-1]
            y_hat = y - (x.dot(weights))
            sum_arr += (x * y_hat)
    weights -= (sum_arr * alpha)

end = time()

print("Time taken per iteration", (end - start) / num_iters, "s.")
```

Listing 1: Local Python Implementation

Spark Implementation

```
# Read the dataset
raw_data = sc.textFile(dataset_path)

# Split each rdd row by comma and persist this new rdd
processed_data = raw_data.map(lambda d: [float(val) for val in d.split
    (',')])
processed_data = processed_data.persist()

# define step function
def takeStep(data_row):
    size = len(w)
    y_hat = data_row[-1] - sum([data_row[i] * w[i] for i in range(size
    )])
    return [data_row[i] * y_hat for i in range(size)]
```

```
13
14 # define objective function
15 def calc_obj(data_row):
16     size = len(w)
17     return ((data_row[-1] - sum([data_row[i] * w[i] for i in range(
   size)])) ** 2, 1)
18
19 # Get the number of rows and initialize alpha
20 num_rows = processed_data.count()
21 alpha = -2 * lr / num_rows
22
23 # Calculate objective value
24 temp = processed_data.map(calc_obj).reduce(lambda a, b: (a[0] + b[0],
    a[1] + b[1]))
25
26 # Perform the iterations
27 start = time()
28 for iter in range(num_iters):
29     stepped_rdd = processed_data.map(takeStep)
30     reduced_rdd = stepped_rdd.reduce(lambda a, b: [a[i] + b[i] for i
   in range(len(b))])
31     w = list(map(sub, w, map((alpha).__mul__, reduced_rdd)))
32 end = time()
```

Listing 2: Spark Implementation

Snowflake Implementation

```
1 ctx = snowflake.connector.connect(
2     user=SNFLK_USERNAME,
3     password=SNFLK_PASSWORD,
4     account=SNFLK_ACCOUNT
5     )
6 cs = ctx.cursor()
7
8 try:
9     # create some update srtings to speed up processing
10     cmd_take_step = \
11         "WITH TEMP_QUERY AS " + \
12         "(SELECT X1, X2, X3, Y - ({w1} * X1 + {w2} * X2 + {w3} * X3)
   AS Y_HAT FROM " + \
13         table_name + ") SELECT SUM(Y_HAT * X1), SUM(Y_HAT * X2), SUM(
   Y_HAT * X3) FROM TEMP_QUERY;"
14
15     cmd_calc_objective = \
16         "select avg(power(({w1} * x1) + ({w2} * x2) + ({w3} * x3) - y,
    2)) from " \
17         + table_name
18
19     # begin the iterations
20     start = time()
21     for iter in range(num_iters):
22         # perform one iteration
23         cs.execute(cmd_take_step.format(w1=w1, w2=w2, w3=w3))
24
25         # update the weights
26         result = cs.fetchone()
27         w1 -= (result[0] * alpha)
28         w2 -= (result[1] * alpha)
29         w3 -= (result[2] * alpha)
30
31     end = time()
```

Listing 3: Snowflake Implementation