Liverpool John Moores University

Analysis of Tabular Data Using Local Deployment vs. Cloud Services, with Data Pipelines Optimisation for Cloud Deployment.

A final project submitted in satisfaction of the requirements
for the degree Master of Science.

in

Data Science

by

Marcin Majeran

Committee in charge:

Professor Iain Steele,
Dr. Joao Da Silva Bento

2024

1. TABLE OF CONTENTS

2. ABSTRACT (to be changed)

Organizations across various sectors often grapple with the challenges of managing, analyzing, and deriving insights from vast amounts of numerical data. The increasing size and complexity of datasets strain local computing infrastructures, leading to performance and scalability issues. To address these challenges, this dissertation explores the integration of cloud computing technologies, applied through different types of cloud deployments, with data science methodologies.

The primary aim of this research is to evaluate the feasibility and effectiveness of deploying data science tools for tabular data analysis in both local and cloud environments. Through a comparative analysis focused on Google Cloud Platform (GCP), this study assesses key performance metrics such as computational time and efficiency, memory usage, processing speed, cost-effectiveness, and data security and privacy.

The results obtained through my work described in this document indicate that cloud-based solutions, particularly those utilizing GCP, offer significant advantages in terms of resources (memory & CPU) management, scalability and accessibility of each cloud deployment, compared to their local equivalent. However, the study also highlights the trade-offs in cost, time, and potential privacy concerns that organizations must consider when migrating to cloud-based infrastructures. Ultimately, this research provides valuable insights into the practical

implications of adopting cloud-based data analysis tools, helping organizations make informed decisions about their data management strategies.

3. INTRODUCTION

3.1. Context

The initial idea behind this project's motivation was to observe, whether a direct 'translation' of a locally built solution to the exemplary data science problem, gains any advantages across different cloud-based deployment types. The general setup of this project's structure envisioned a progressive rise in the deployment's 'cloud contained' factor – which can be explained as the estimate of what part of the solution utilizes available cloud tools and services, to what extent, and how 'deep' in the cloud infrastructure the deployment is. To further elaborate on this aspect – as an example, the basic cloud infrastructure, using the GCP's Cloud Shell as the working environment is considered very 'shallow' as a cloud solution. On the other side of this spectrum is a Vertex deployment, which is built within Google's Vertex AI instance, using its own, dedicated cores and memory – this solution will be considered as a 'deep', integrated and self-contained cloud deployment. Each solution has been setup in a way that allows measurements of key metrics, needed for a comparative analysis, to be easily taken during the program run.

3.2. Literature Review

The shift towards cloud computing has become increasingly prevalent across various sectors, with a significant percentage of businesses now managing their data in the cloud. According to the Colorlib report (Rok Krivec, 2024) - **94% of enterprises** worldwide use cloud computing to perform the data-oriented operations and practices, such as data storage, manipulation or machine learning. These adopted strategies can be categorized as IaaS cloud

architectures (Infrastructure as a Service), which are the most flexible and comprehensive among other types of cloud services (examples are: AWS, Microsoft Azure, GCP) (Patel Hiral B, 2021). This widespread adoption is driven by the numerous advantages that cloud computing offers, particularly in the context of data science. Among all versatile metrics that can assess to how beneficial the cloud adoption is, project's focuses on identifying potential **computational** advantages by using benchmark metrics such as memory and CPU usage.

One of the most obvious benefits of using cloud deployments would be the cost reduction. It has been proven that maintaining an on-premises private cloud computing platform, in early stages (for Dataproc Spark jobs) is more expensive than migrating the same workload to the cloud system made available by one of the established cloud providers (Per Bondenson 2021). This has been further confirmed by another study, which expands this conclusion, showing that despite initial costs of setup/maintaining can be higher for a locally distributed platform, in the long run the accumulative nature of cloud computing pricing makes the on-premises solutions cheaper than cloud (Cameron Fisher, 2018). Nevertheless, the scalability is another important factor worth taking into the account in all local vs cloud comparisons, as the ease with which the user can scale up (or down) deployments on the cloud platform is far superior to the struggle one can experience when trying to expand the local solution, in order to accommodate more memory or processing power.

Despite these benefits, there are challenges associated with cloud adoption, particularly concerning data security and governance. While cloud providers offer robust security features, such as encryption and identity management, the complexity of managing data across distributed cloud environments can lead to increased risks. According to researchers **48% of cloud-stored data** is sensitive (Rok Krivec, 2024). However, many businesses are willing to accept these risks in exchange for the scalability and flexibility that cloud computing provides, especially given the cost and performance advantages.

In conclusion, the literature indicates that cloud computing is not only a viable but often a superior option for data science applications, particularly for organizations seeking to improve efficiency and reduce costs. However, it is crucial for businesses to carefully assess their specific needs and potential risks to fully leverage the benefits of cloud-based solutions.

3.3. Data

To choose the dataset suitable for the analysis in this project, few factors needed to be considered. Firstly, the data had to allow predictions/classifications to be made, as supervised machine learning task will allow to precisely assess whether the algorithm performed well or not, by tracking training and test accuracies. Secondly, data needed to consist of significant number of features and rows, that would be sufficient to make the model's training complex enough to benchmark performance of various deployments, and capture the potential differences between them. Lastly, chosen dataset had to be easily processed by Google's Vertex AI service, which can process Image, Video, Text and Tabular data. After prior research and some tries with different datasets, the HAR (Human Activity Recognition) data, precisely **Human Activity Recognition with Smartphones** by UCI dataset turned out to be a perfect fit . Combined training and test data contribute to **10299** unique rows of **561-feature vectors with time and frequency domain variables** (+ labels). Each of those vectors represents the measurement taken on one of 30 volunteers, within age bracket of 19-48 years. Data has been collected through the app installed on **Samsung Galaxy S II** worn on the waist and consist of 6 different activities (labels): WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING, LAYING. Accelerometer and gyroscope data has been processed beforehand including Fourier Transform with 50% overlap, Butterworth low-pass filter and cutoff filter, to prepare the data to be used after download.

4. METHODOLOGY

4.1. Design

4.1.1. Deployments

This study follows the design made of 4 distinct parts, where each part is a separate, standalone environment, with purpose of solving the same data science problem. To ensure that this project captivates the difference between truly unrelated environments, where each one is on a different 'level' of how 'deep' in the cloud infrastructure it is, these themselves have to be thoroughly

independent. In compliance with this rule, 4 separate deployment types have been derived. In this section, the general, high-level description of those deployments and their properties can be found. Technicalities and more details are provided in the **4.2 Implementation** section.

- Local – on-premise solution, code is executed on local machine using available hardware. For this solution data is stored in locally hosted object storage server – to emulate the general, very simple data pipeline that pulls the data from cloud storage service.
- Cloud – the solution contained and executed on Google Cloud Platform in a Cloud Shell. Data is stored in a Cloud Storage bucket.
- Containerized – this deployment is built as a Docker container image, deployed and executed as a Job in GCP Cloud Run service. Data is stored in a Cloud Storage bucket.
- Vertex – solution which utilizies Google's proprietary ML platform, Vertex AI. Deployment uses Vertex instance to run the program and employs a 'mounted bucket' feature, which allows to directly connect an instance to the GCP Cloud Storage bucket and access it with minimal latency.

This set of 4 deployments serves as an entire project structure, remotely managed and stored in a (private) Github repository. Each solution consists of the same code, only altered to correctly communicate with a corresponding data source. This will bring additional information on how different settings for data storage affect the execution performance.

4.1.2. Functions

In general, each deployment has two instructions, that need to be fulfilled to successfully finish the program run with a solution (list of classified labels). These instructions are contained in python functions:

- *data_grab()* – responsible for:
  - connecting to the data source (this will vary based on the deployment type)
  - downloading train.csv and test.csv files from the data source
  - saving both files as Pandas DataFrames, concatenating them and returning as a single DataFrame 'df_merged'

This function has been designed to test multiple aspects of each run i.e. how well will different solutions handle various protocols for getting the data from a remote data source, downloading data from previously connected data source, and finally reading and saving it as a single data structure.

The second function is:

- *evaluate()* – which takes care of:
    - fitting the train data to the provided model
    - predicting labels for the test data
    - calculating the best cross validation test accuracy score and returning model fitted with data

This python function simply tests machine learning performance of each solution. Even if CPU usage is being measured for both functions, it's the *evaluate()* where this metric can be tested sufficiently, under high demand. Such set of two main instructions will serve as a 'base' for taking measurements. Additionally, each deployment type shares some additional instructions other than *data_grab()* and *evaluate().* These additional instructions can be summarized as:
After *data_grab(),* before *evaluate()*:

- Separating explanatory/response variables, and saving them as separate DataFrames
- 80:20 train-test-split
- Defining Random Forest Classifier model
- Setting up ad Grid Search to be used as a default model

The idea behind selecting Random Forest Classifier for the model used in the testing, is that with conjunction with Grid Search, (designed to search through 1-9 random states of the model by using 'random_state' parameter and passing a python list) it ensures that the computational task is sufficiently difficult to test the machines' memory and CPU usage. Because it builds several decision trees, Random Forest is computationally intensive, and Grid Search adds to this load by trying different configurations. By utilizing 'random_state', the program can maintain its high level of complexity without requiring the testing of extra hyperparameters, guaranteeing a thorough evaluation of system performance in various deployment scenarios, while preserving an acceptable runtime.

4.1.3. Measurements – performance indicators

To quantify the results of this study 3 main measurements, also called performance indicators, have been selected:

- Execution time – measured in seconds (s), helps to determine whether runs from one deployment are faster (or slower) than runs executed using different solution.
- Memory used – expressed in Mega Bytes (MB), tells how much from available memory, the program run ended up using.
- CPU used – this performance indicator is using average value of percentage of total CPU usage during execution time, measured every 0.1 second, where only 'ticks' with positive values are taken into the account (to avoid including zeroes when CPU is not in use).
- (OPTIONAL) Memory increment – xyz

In as single run, each of these 3 parameters will be tracked while executing *data_grab()* and *evaluate()* function. This setup ensures that separate results are stored for both actions (collecting data & machine learning). Every successful run should generate 7 unique values structured as a row of data, with each column (variable) being:

- label – 'name'/type of deployment, timestamp,
- data_time,
- data_memory,
- data_cpu_usage,
- ml_time,
- ml_memory,
- ml_cpu_usage.

For meaningful results, each deployment will be instructed to run a program for at least a hundred of times, with additional hundred when upgrading the instance's machine hardware is available (this is the case with **containerized** and **vertex** solutions). Upgraded runs will carry an additional '-boost' postfix flag in 'label' variable.

4.2. Implementation

Each implementation is coded in Python and its structure follows similar schema, having two folders: 'results' for storing measurements in csv files and 'src' which contains:

- config.py – where deployment's global variables like labels, keys, or bucket names are stored,
- HAR.py – the main script which executes both *data_grab()* and *evaluate()* functions. It uses *@profile* function decorators from *memory_profiler* library to document execution time and memory usage. Every program run generates a 'mprofile.dat' file which contains timestamps and MB of memory used per every tick during the **function** run (so it collectes data separately for every function with *@profile* decorator). The actual measurements are extracted from additional rows, added to the .dat file after each function finishes its run, these rows contain summarized values for function start, stop time and memory used.
- get_results.py – supporting script, which accesses 'mprofile.dat' generated file and pulls the data into corresponding results file. The necessity for this file comes from the nature of *memory_profiler* which creates a complete 'mprofile.dat' file only after entire program run has finished, meaning it couldn't be done in HAR.py without loosing some of the information.
- HAR_cpu_read.py – 2nd main script which had to be introduced due to the conflict of *mempory_profiler's @profile* decorators and *multiprocessing* package, which is used to estimate average CPU usage per each **function** run. Unfortunately, *memory_profiler* doesn't work in separate processes, which in this project, are used to measure CPU use. The general idea behind this method of measuring is to create a separate process every time one of the main functions is called. In this separate process a new function *cpu_reader()* is ran. This function measures CPU usage every 0.1s tick and saves it in a globally accessible multiprocessing.Manager.Queue object (saves only positive values to omit zeroes when there's no CPU operation – so mean isn't artificially lowered).
- command.sh – instruction script used to execute all code in the correct order. Additionally, it ensures that each 'mprofile.dat' file is removed after data is extracted, to avoid confusing these files in next runs.

Because of *memory_profiler* and *multiprocessing* libraries conflict, the second main script – HAR_cpu_read.py file had to be created. This fundamentally means that each measured run is, in

reality, two different runs – one for which execution time and memory used are measured – and second one during which an average CPU usage is recorded. This, unfortunately, introducesd unwanted complexity into the project and forced solutions as described above. Also, due to this complication CPU usage will be treated as an independent variable from execution time and memory used. Moreover, potential interference cases for CPU usage, potentially affecting other measurements will be addressed later in **5. CONCLUSIONS**.
Apart from the general file and folder schema, every deployment has its own unique properties and storage solutions.

4.2.1. Local

On-premise deployment utilizes MinIO object storage system which emulates cloud storage (MinIO is mainly compatible with AWS S3 solution but for purposes of project it's being compared to Google Storage which is also supported). MinIO allows its users to create buckets which store data as objects – in this project train.csv and test.csv files are stored in a single bucket deployed on a standalone MinIO server. This data can be retrieved, as an object type, from server utilizing minio Python package.
Specifications: CPU: Apple M1 3.2GHz 8 cores, RAM: 16GB

4.2.2. Cloud

This solution is entirely stored (except for the data) within GCP's Cloud Shell. Cloud Storage has been used to create a bucket (equivalent of local solution's MinIO bucket), within which the data is stored. By using storage.Client object from *google.cloud* library and previously setting up valid credentials for GCP project, script can connect to the bucket and pull the data. Retrieved information is read as BLOBs (Binary Large OBjects), downloaded as text and then casted into csv format with StringIO function from io packaged – allowing the data to be loaded into pandas dataframe and further processed by program.
Specifications: CPU: Intel(R) Xeon(R) 2.2GHz 4 cores, RAM: 16GB
What is worth noting at this point is that every GCP service used in this project is ran on server using **US-west1 (Oregon)** as a Region (no specific Zone). This choice ensures that all Free-Tier services can be utilized to their fullest (available in Free-Tier). It's also important as, in this case, Cloud Shell specification is determined by the region we're hosting our project in. However, as

there is no way of manually setting up a specific Region for Cloud Shell, GCP will assign a geographically closest, stable Region instead. Because of that, each run has been monitored to ensure that the same specification is used per every run.

4.2.3. Containerized

Similarly to Cloud deployment, this one is also stored within Cloud Shell, primarily for the ease of building a Docker container image using *gcloud builds* command which automatically saves that image in a specified GCP's Artifact Registry – which serves as a place where user can create container images' repositories, from where these can be easily accessed (within GCP) and managed. Image for this deployment is built within Cloud Shell, then sent to the Artifact Registry, making it finally accessible for GCP Cloud Run service where it can be executed as a Job. This deployment uses the same Cloud Storage bucket as the Cloud deployment. Due to the enclosed nature of this solution, all operations allowing the program to successfully pull and process data, and save the results, had to be put in the Dockerfile. Within that file, the container's builder is instructed to activate GCP's service account, using credentials passed in a separate my_key.json file, to allow container communicating with Cloud Storage system. After setting this access, instructions from Dockerfile ensure that correct work directory is selected from which command.sh file can be run. Another difference of this deployment is the results storage – to allow access to the measurements taken within the container, the program will send results.csv and results_cpu.csv to the 'results' folder in the same Cloud Storage bucket.

Specifications: Intel(R) Xeon(R) 2.0GHz 4 cores, RAM: 2GB

Boost: Intel(R) Xeon(R) 2.0GHz 8 cores, RAM: 16GB

This deployment has added 'Boost' line of specification, as after collecting the data for the default machine the entire process has been repeated using the upgraded version. This approach will allow for more thorough analysis, by comparing the same solution type with different hardware specification. Initially, every default version of a deployment was run on the 'recommended' machine's hardware, when boost version of a deployment is available, the configuration chosen will be trying to be as close as possible to the Local specification.

4.2.4. Vertex

Solution utilizing Google's proprietary Vertex AI service. This deployment runs on a Vertex AI instance, within which the default structure of the project is initialized. Storage is handled by Google's in-house solution – bucket mounting. GCP allows users to mount a bucket onto the running instance (Vertex AI instance in this case) and access its contents directly. This significantly reduces latency and allows the program to execute *data_grab()* faster. Since Vertex AI has been created with machine learning in mind, its hardware has been precisely selected to process ML requests. Unfortunately, GPU's offered as part of Vertex AI available processing units couldn't be applied in this project as *scikit-learn.RandomForestClassifier* isn't directly supported by Vertex AI (which supports Tensorflow and Keras models), so in this case it can only utilize the pure higher processing power offered by the service.

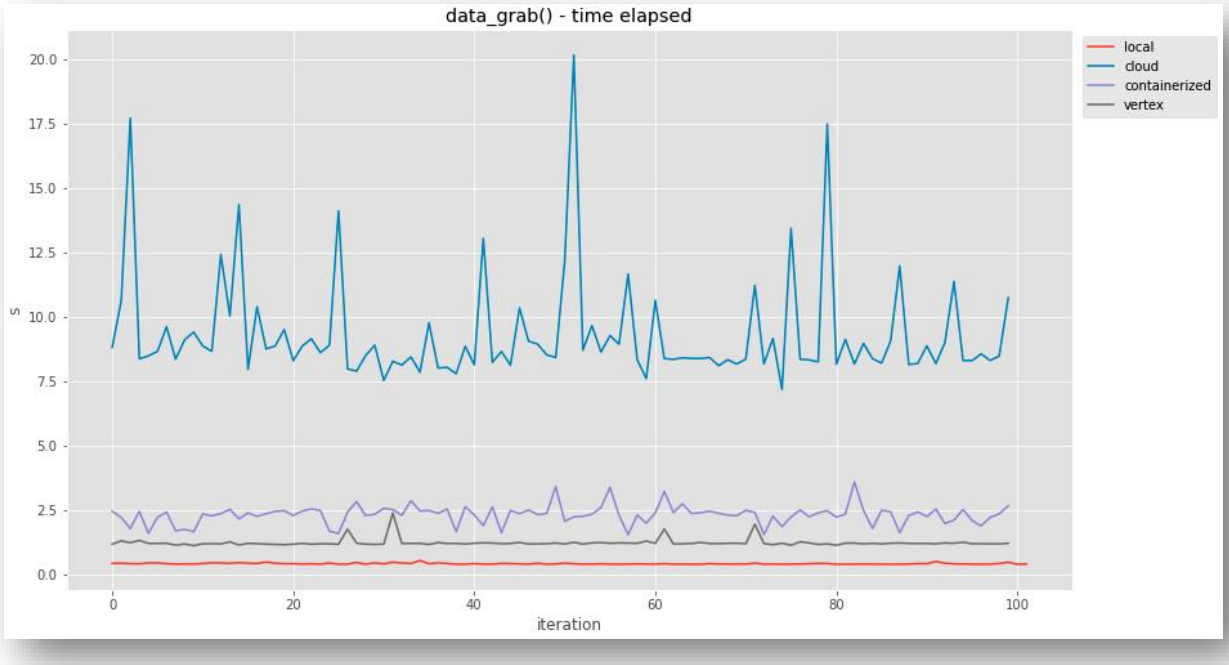
Specifications: Intel(R) Xeon(R) 2.20GHz 4 cores, RAM: 16GB

Boost: AMD EPYC 7B12 2.25GHz 8 cores, RAM: 32GB

5. RESULTS

5.1. Data pull

5.1.1. Execution time



<line plot> <mean, std, quantiles table>

5.1.2. Memory used

<line plot> <mean, std, quantiles table>

5.1.3. Average CPU usage

<line plot> <mean, std, quantiles table>

<box plot>

5.2. ML

5.2.1. Execution time

<line plot> <mean, std, quantiles table>

5.2.2. Memory used

<line plot> <mean, std, quantiles table>

<box plot>

5.2.3. Average CPU usage

<line plot> <mean, std, quantiles table>

<box plot>

5.3. Boost versions

<line plots> <mean, std, quantile tables>

6. CONCLUSIONS

6.1. General conclusions

6.2. Further work

7. SELF-EVALUATION

8. REFERENCES

Marcin Majeran