Information Systems, Analysis and Design

Project D4c

Paralikas Ilias
AM 03116605
National Technical University of Athens
Athens, Greece
paralikasilias@gmail.com

Ioannis Fardelas
AM 03113190
National Technical University of Athens
Athens, Greece
farg245@gmail.com

***Abstract*—Machine learning operators executed within big data analytics runtimes, such as Apache Spark and Apache Flink, often involve complex code and consume a significant amount of time when processing large volumes of data. This paper presents a project aimed at developing predictive models to estimate the performance of these operators without the need for actual execution. Specifically, the project focuses on MLLib Operators within Apache Spark, including Decision Tree Classifier, FP-Growth, and PCA.**

***Index Terms*—Mllib, pyspark, performance**

I. Introduction

Big data refers to the massive volumes of structured and unstructured data generated at an unprecedented rate in today's digital world. This data holds valuable information that, when properly analyzed, can offer insights, patterns, and trends that were previously difficult or impossible to discern. The significance of big data lies in its potential to drive informed decision-making, uncover new business opportunities, enhance operational efficiency, and enable innovations in various industries [6].

Apache Spark is an open-source distributed computing system that gained prominence for its high-speed data processing capabilities and versatile analytics functions.It plays a pivotal role in handling and processing big data effectively. It is an open-source distributed computing system that provides a powerful platform for processing and analyzing large datasets in real-time or near-real-time [7].

As more and more organizations use big data analytics to learn from large and complicated sets of data, Apache Spark has become one of the most popular tools because it can handle large amounts of data and supports many different methods for understanding it. However, using machine learning operations in Apache Spark can be tricky and take a lot of time, which can slow down the whole process.

Rather than executing these operations directly, we propose the utilization of a computational model to forecast their execution duration and computational resource requirements. This involves the systematic collection of historical performance data associated with these operations. Subsequently, we employ specialized computer algorithms to generate predictive models.

The primary objective of this study is to present a practical solution aimed at enhancing the efficiency of big data analytics. Our central aim is to establish the capability to predict the performance characteristics of machine learning operations within Apache Spark accurately. In this paper, we will explain the procedures involved in data collection and processing, describe the specific computer programs employed for predictive modeling, and provide a comprehensive assessment of the effectiveness of our proposed approach.

¡**might change The ensuing sections of this paper are structured as follows: Section 2 describes the setup of our computational environment. Section 3 provides an overview of the computer systems and software utilized in our study. Section 4 offers insights into the data collection and preprocessing methodologies. Section 5 presents the datasets generated for our analysis. Section 6 expounds on the techniques employed in training our predictive models. Finally, in Section 7, we present the results and evaluation of our approach, encompassing the predictive accuracy of our models.¿

II. Installation and setup

Given that Machine Learning operations can be computationally intensive and time-consuming, especially when utilizing resources like Graphics Processing Units (GPUs), which were unavailable on both of our local computers, local execution was deemed impractical. Thus, we chose to leverage Google Colaboratory due to our positive past experiences with it. The primary advantages included convenient access to both the source code and files, as well as the availability of on-demand hardware resources, encompassing GPU processing and storage within Google Drive.

Furthermore, spark is very easy to use of colab. No complicated set up is required, other than installing pyspark.For our pyspark cluster, we chose to use the default settings.The most important ones are the number of cores, which was set to 2, driver and executor memory, 1GB each.

III. Software Description

A. Google Colab for Cloud-Based Computing

In this study, we harnessed the capabilities of Google Colab, a cloud-based Jupyter notebook platform. Google Colab

provided us with a versatile and scalable computing environment for our research experiments. This cloud-based platform allowed us to seamlessly integrate and utilize various libraries and services, making it an ideal choice for our computational needs.

B. Apache Spark for Distributed Computing

Our distributed computing engine was Apache Spark, which played a central role in our research. Within the Spark ecosystem, we leveraged PySpark, the Python API for Apache Spark. PySpark gave us access to the full potential of Spark's distributed computing capabilities, enabling efficient data processing and machine learning model development.

C. MLlib Library for Machine Learning

In our research, we made use of the MLlib library, a core component of the Spark ecosystem. MLlib is a resource for implementing and executing a range of machine learning algorithms, including those associated with the Decision Tree Classifier, FP-Growth, and Principal Component Analysis (PCA) operators whose perfomance we wanted to predict .

D. scikit-learn (sklearn) for Data Preparation

For critical data preprocessing tasks, such as data splitting into training and testing sets and standardization using the StandardScaler, we incorporated the scikit-learn (sklearn) library into our workflow. Scikit-learn's user-friendly APIs and robust functionality enhanced our ability to prepare and preprocess data effectively.

E. SparkMeasure Library for Performance Measurement

To monitor and measure the performance of our experiments and trials, we integrated the SparkMeasure library. This library allowed us to collect essential performance metrics, enabling us to assess the efficiency and effectiveness of our machine learning models during various experimental trials. Key features of the SparkMeasure library and the types of information it can return include:

Execution Time Metrics: SparkMeasure allows us to precisely measure the execution time of our machine learning operators. It provides insights into how long each operator takes to complete, enabling us to identify potential bottlenecks and optimize our workflows.

CPU Utilization Metrics: The library captures detailed information about CPU utilization during the execution of our operators. We can assess CPU usage patterns, including minimum, maximum, and average utilization, which aids in optimizing resource allocation.

Memory Utilization Metrics: SparkMeasure records memory usage data, including main memory cluster utilization. This information is crucial for understanding how our operators utilize memory resources and optimizing memory configurations.

IV. Operators

A. Decision Tree Classifier

Among the three operators we examined in this study—PCA, FP-Growth, and Decision Tree Classifier—the Decision Tree Classifier stands out as a versatile and interpretable machine learning algorithm employed for multistage classification strategies [1]. The process consists of classifying the data base on the value of each dimension separately.

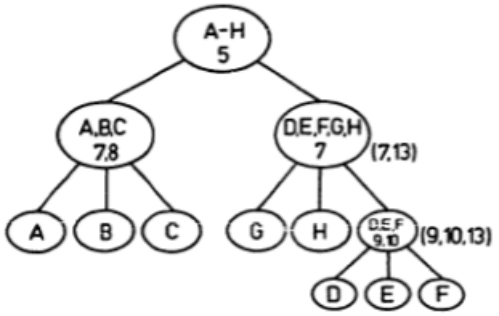

Fig. 1. Sample Decision Tree [1]

The Decision Tree Classifier constructs a tree-like structure, as illustrated in Fig. 1, where each node serves as a decision point. At these nodes, instances are directed to either the left or right subtree based on the value of a specific feature, resulting in a hierarchical decision-making process.

The essence of the Decision Tree Classifier resides in its training process, which revolves around identifying optimal thresholds at each decision node. These thresholds are meticulously selected to maximize information gain, a critical criterion in the construction of decision trees.

Information Gain: Information gain quantifies the reduction in uncertainty (or entropy) achieved by dividing a node into its child nodes. The formula for information gain is defined as:

$$InformationGain = S_{parent} - \sum w_i * S_{children}$$

Where entropy S is given by the formula

$$S = \sum -p_i * log_2(p_i)$$

with p_i the probability of getting a certain class on the node, before the split happens and w_i the relative size of the node compared to the to elements.

In our research, we explroed the Decision Tree Classifier as it is one of the key machine learning operators. Its utility extends to a wide range of applications, including classification tasks where interpretability and the ability to visualize decision-making processes are of paramount importance. This operator played a vital role in our analysis, and its performance was rigorously evaluated using the methodologies outlined in this study.

B. Principal Component Analysis

Principal Component Analysis (PCA) is one of the three key operators investigated in our study, alongside FP-Growth and Decision Tree Classifier. PCA serves as a powerful dimensionality reduction method frequently employed in data analysis, with the primary objective of capturing dominant patterns present in data matrices [4] as well as represent the important information as a set of new orthogonal variables called principal components [5].PCA accomplishes this by generating a complementary set of score and loading plots, which provide insights into the underlying structure of the data . Additionally, PCA represents crucial information through a set of new orthogonal variables referred to as principal components . Each principal component encapsulates a portion of the original data's variance, facilitating the extraction of essential features. The PCA algorithm operates by identifying the best-fit line, which minimizes the sum of distances from the data points. This line represents the first principal component. Subsequently, PCA ensures that each subsequent principal component is orthogonal to the preceding one, thereby avoiding redundancy in information representation.

PCA offers the flexibility to reduce dimensions as needed, preserving a significant portion of the information. This makes it a valuable tool for data preprocessing and dimensionality reduction. Furthermore, PCA can be leveraged to visualize high-dimensional data in lower-dimensional spaces, such as 2D or 3D plots. This visualization simplifies the interpretation and comprehension of complex datasets, aiding in pattern recognition and understanding.

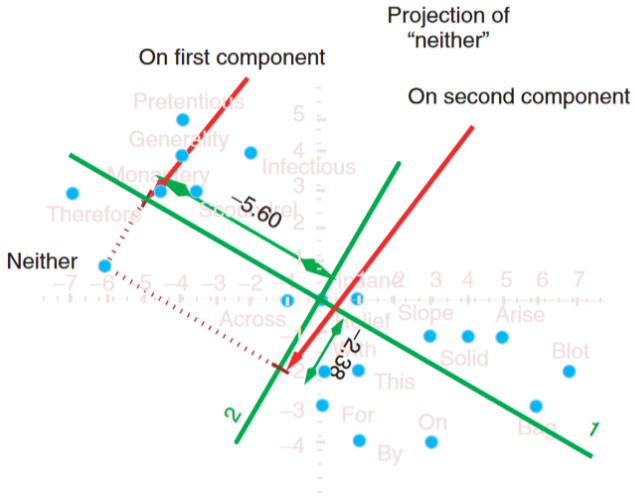

Fig. 2. Two proncipal components [5]

Minimum Support and Minimum Confidence

In the context of association rule mining, such as the Apriori algorithm, PCA can also be applied in tandem with setting minimum support and minimum confidence thresholds. These thresholds play a crucial role in determining the strength of association rules derived from the data.

- **Minimum Support:** Minimum support establishes the threshold for the minimum frequency at which an itemset (a set of items) must appear in the dataset to be considered "frequent." Itemsets that meet this support threshold are candidates for further analysis.
- **Minimum Confidence:** Minimum confidence specifies the minimum level of confidence required for association rules to be considered interesting or significant. Confidence measures the likelihood that an itemset or combination of items will lead to another item or set of items in a transaction.

C. Frequent Pattern Growth

The Frequent Pattern Growth algorithm, often abbreviated as FP-Growth, operates within a database of transactions, where each transaction represents a collection of items. In practical applications, such as retail, a transaction might correspond to a customer's shopping basket, with items denoting the products they have purchased. FP-Growth is a crucial operator in our study, alongside PCA and Decision Tree Classifier.The core steps of the FP-Growth algorithm involve:

Compression into FP-Tree: The initial phase entails compressing the database, emphasizing frequent items, into a structure known as a frequent-pattern tree or FP-tree. This tree preserves the association information among item sets while condensing the database [2].

Creation of Conditional Databases: Subsequently, the compressed database is partitioned into a set of conditional databases, each associated with a frequent item. These are special databases known as projected databases. Each conditional database is created to facilitate the mining of frequent item sets related to a particular item [2].

a d f	d 8	d a
a c d e	b 7	d c a e
b d	c 5	d b
b c d	a 4	d b c
b c	e 3	b c
a b d	f 2	d b a
b d e	g 1	d b e
b c e g		b c e
c d f		d c
a b d		d b a

Fig. 3. Transaction database (left), item frequencies (middle), and reduced transaction database with items in transactions sorted discerningly with respect to their frequency (right). [3]

Mining Frequent Item Sets: Mining is conducted independently on each of these conditional databases. The goal is to identify frequent item sets within each database, which are collections of items that frequently co-occur in transactions [2].

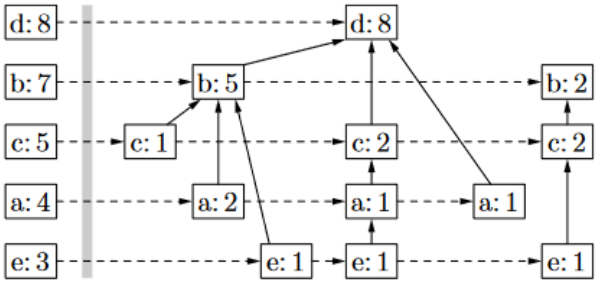

Fig. 4. FP -tree for the item set in Fig. 3. [3]

Derivation of Association Rules: Once the frequent item sets are established, the FP-Growth algorithm allows the derivation of association rules from them. These association rules offer insights into item co-occurrences and can be invaluable in applications such as market basket analysis [8].

Illustrations of the transaction database, item frequencies, and the reduced transaction database, as well as the FP-tree structure, are depicted in Figs. 3 and 4 .

V. Data Generation

The data that the operators were trained on were randomly generated, as the actual results of our runs were irrelevant. What we cared for were the size and type of data, as well as the time it took to process them. Decision Tree Classifier and Principal Component Analysis, as excepted can process numerical, as well as categorical data [10].

Our first attempt was to generate the data in the notebook that we planned on executing the machine operators and store the results in google drive, so we could preserve them through different runtimes. It quickly became apparent that python is a very slow language. That would be a problem, as the size of the data that we wanted to create, by the nature of the problem that we were trying to solve, had to be quite big, meaning we would have a lot of idle time.

That's why, we decided to generate the data in our local machines, through a program written in c++ which is much faster [11]. The difference in speed was tremendous and the data was generated in much less than a day.

Moreover, in order to have access to a variety of sizes, without having to create,store and load vast amounts of data, we decided that we should be able to combine different files with each other, or even read the same file multiple times. For this purpose we created a parameter, dataset repetition, which will become of importance later on.

A. Decision Tree & PCA Data Generation

The numerical data were systematically generated in a cyclic fashion to ensure non-linearity while retaining some semblance of real-world structural patterns. In contrast, the categorical data, rather than being represented as textual labels denoting distinct categories, were initially generated using a one-hot encoding technique. This approach was chosen primarily to expedite the data generation process, given that the labels themselves lacked any inherent physical significance at the outset.

It is imperative to note that, despite the possibility of employing Principal Component Analysis (PCA) for processing categorical data, such an approach is discouraged due to its potential to yield inaccurate results without modifying the algorithm [12]. In light of this, and considering that the results lack tangible real-world meaning, the decision was made to maintain the categorical data in its original one-hot encoding format as inputs for the analysis.

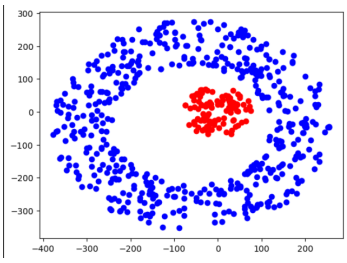

Fig. 5. cyclical data in 2 dimension

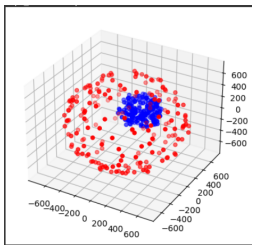

Fig. 6. cyclical data in 3 dimension

0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0

Fig. 7. Categorical data One Hot Encoding

The objective of this research is to assess the performance of various operators across datasets of diverse sizes and characteristics. To facilitate this evaluation, the generated datasets were categorized into three distinct size categories. Within each category, each dataset file contains a single class and featured 100 columns of numerical data. As for to the categorical data, each dataset comprised ten fields, within which the number of mutually exclusive columns in one-hot encoding format ranged from 2 to 10, as previously detailed.There were 3 different sizes, small, medium and big, with 1e5, 1e6 and 5e6 rows each.

Finally, when the data was read, there was a combination of these classes, of the same category, that were then processed, by dropping a random number of columns, as to have variety both vertically, meaning the number of rows, as well as horizontally, the number of columns.This would be the final information that would be used for the classifier, in order to estimate the desired metrics.

- Number Of Classes
- Number of rows
- Numerical data columns
- Categorical data columns

B. Frequent Pattern Growth Data Generation

As described in the previous section, the data that this algorithm is designed to generate are a list of lists, containing items designated by an integer. The values themselves do not contain any specific meaning and the items are not ordered, the numbers serve the purpose of distinguishing them from each other.

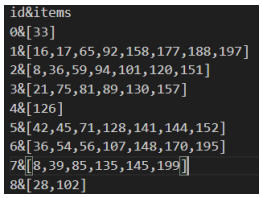

Fig. 8. FP-data (the & characters help us separate the list from the id later)

The parameters we had to tune in this case were less, by the nature of the operator. Apart from the number of rows, there was a variety in the number of unique items, as well ad the max items per row. That being said, when combined, these numbers would be mixed up. In order to calculate the mean item per list, we supposed that, given that there is a adequate number of samples, in every file , the mean number of items is half the max number of items per list.

$$MeanItemsInFile = \frac{MaxItemsInList}{2}$$

That would mean, that the mean item in the whole of the dataset would be a weighted average of the mean item per file and the number of rows per file.

$$MeanItemsInDataset = \frac{\sum MeanItemsInFile_i * RowsInFile_i}{\sum RowsInFile_i}$$

The metrics that will be used to evaluate are

- Number of rows
- Mean items in dataset

VI. Operators Execution

In order to measure the values that we are interested in, we used the library sparkmeasure [13]. More specifically, we made use of the sparkmeasure.runandmeasure function provided by said library, which requires as inputs the command that you want to measure. Afterwards, the metrics are printed int the standard output. In our case, is the output of the colab cell. That was inconvenient, as we wanted to store the metrics in a way that would be processed through code,so that it could be later processed and used in a machine learning model. For that purpose, we utilised the magic functions that colab provides. More specifically, the *%%capture cap* functionality, which, instead of printing the output in the cell, stores it in a variable called *cap*. This variable, in a latter cell is stored in a *txt* file, located in a shared folder of google drive.

One of the challenges that we faced with this approach was that, since the output was produced when the cell containing the operator execution was finished and variable cap was accessible after the cell was run. Given that google colab tends to disconnect when left idle, we could not leave the code unattended to run during the night, as the interruption of the cell mid execution would result in all of the work being lost, due to the fact that it could only be stored after the first cell had finished. That meant that we couldn't fully automate the process of generating trials, as we had to manually change the parameters, instead of listing them and iterating through them in a big for loop, which would be ideal. Additionally, we had to restart colab every now and then, losing the work that it was currently processing. This was problematic, especially in big datasets, that would be more often interrupted.

Another tricky part was that of the scope of the variables inside our functions. The function used to measure the values, as mentioned before, got the command to be measured as a string input. As a result, when passing a variable inside the string, it did not get the value contained in it but only the name. When sparkmeasure function tried to execute the command, it started outside of the scope of the function. This resulted in it reading the global variables first, thus getting an input that was not intended. That's why, instead of creating elegant functions, the execution and measuring process is rather ungracefully written, in terms of appearance.

The parameters that were tuned, for each operator, in the trial generation process where the following.

A. Decision Tree parameters

- *Number classes* , the number of different files to be read
- *Dataset size* , the magniture of each class, which could be small, medium or big, with the aforementioned rows each.
- *Vertical cut* , which indicated the range of the columns that had to be dropped. This value also expected values of small, medium or big.

Dataset repetition was kept to 1, as the data was big enough as they were.

B. Principal Component Analysis

In our PCA trials, as we had the same type of data, *Number classes*, *Dataset size* ,*Vertical cut* were present. Additionally, we tuned the *Dataset Repetition* parameter as one iteration over the data was not enough to cause a memory spill. That being said, even with larger numbers we did not manage to replicate a memory spill with the PCA algorithm, even when the data was bigger than the memory itself, with 16.8GB read in one instance.

C. Frequent Component Analysis

Even if the FP algorithm has different data, we tried to keep the same structure for all our trials. Since the data have a pretty simple form, we tried to have a couple more parameters , in order to have some actual meaning behind our results,These parameters were.

- *Number of Files*
- *Dataset Repetition*
- *Minumum Confidence*
- *Minumum Support*

VII. Results Predictions

A. Parsing The Trial Files

After running said trials, the only thing left was to extract the information from the txt files. The first step was easy, as we have already marked the file with specific symbols, such as '&' or '#' to indicate where the important information was, regarding the inputs of the models. After that, sparkmeasure prints the measured values in a simple fashion

$$nameOfValue => valueInms(ValueInReadableFormat)$$

Out of the provided values, we decided that the most important ones were

- elapsedTime
- executorRunTime
- executorCpuTime
- resultSize
- diskBytesSpilled
- memoryBytesSpilled
- peakExecutionMemory
- bytesRead

We just had to parse every line of the file and look for the specific strings and extract the value.

B. Preprocessing

In our initial attempt, we encountered significant challenges when training our neural network. The primary issue was that the data we were working with contained exceptionally large values, both in the input features and especially in the output. This presented a problem because the neural network's weights were randomly initialized, and the resulting outputs were nowhere near the actual labels. Consequently, our loss function produced astronomical values, rendering the training process ineffective. Regardless of the loss function we tried, this issue persisted, making it impossible to make meaningful progress.

To address this problem, we turned to a technique known as Standardization, which plays a crucial role in preprocessing data for neural networks. Standardization, often implemented using a tool like the StandardScaler, transforms the data to have a mean of zero and a standard deviation of one. This process brings the data into a "normal" range, ensuring that the values are neither too large nor too small. By standardizing our data, we were able to mitigate the issue of values that could cause our loss function to explode during training. This resulted in more stable training, as the loss values no longer reached extreme levels that hindered learning [14].

Furthermore, we made an adjustment related to the activation function used in our neural network, particularly the Rectified Linear Unit (ReLU) activation function [15]. ReLU is one of the most popular activation functions in deep learning, known for its ability to set negative values to zero while leaving positive values unchanged. To ensure that our computations only involved non-negative numbers, we added the minimum values from our data to both the input and output. This approach helped prevent negative values from causing problems during training and contributed to more reliable convergence during the optimization process.

In summary, by standardizing our data using techniques like StandardScaler, we brought our input and output values into a reasonable range for neural network training. Additionally, by adjusting the input and output values to be non-negative, we ensured compatibility with the ReLU activation function, which is widely used in neural networks for its ability to handle non-linear relationships and alleviate the vanishing gradient problem, which can occur with other activation functions like sigmoid or tanh. These preprocessing steps collectively improved the stability and effectiveness of our neural network training process.

C. Data Correlation Visualisation

In order to get a deeper understanding of the impact that each variable has on the measured values, we decided to plot them, in case there is something that can be deduced with the naked eye.

1) Decision Tree: The decision tree values are as follows.

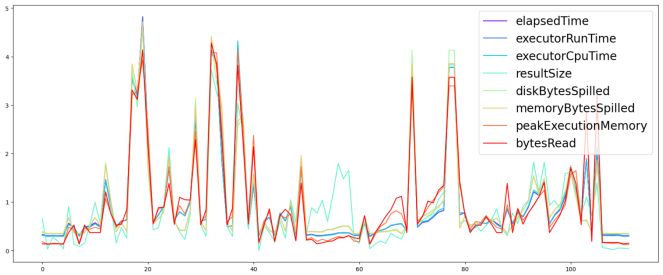

Fig. 9. Decision Tree output values

It quickly becomes apparent, that even if there is a lot of fluctuation in the values, they all follow a similar pattern, with the one that a Little bit being result size, just before the 60th trial. While plotting the output values against the inputs, we noticed a very high correlation between the numbers of rows of the input and every output value, which is no surprise since they are so similar.

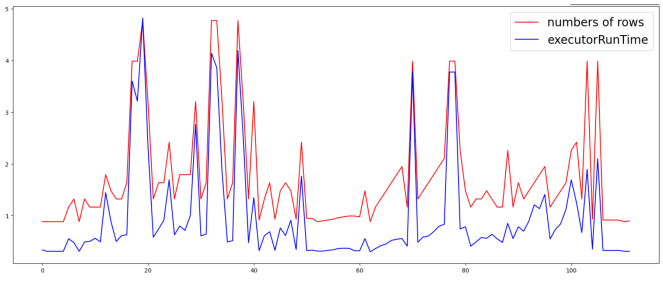

Fig. 10. DT : Number Of Rows & Executor Run Time

2) Principal Component Analysis: When it comes to PCA, a similar structure can be observed, where almost all values follow a similar patter, except bytes read, which is very similar to the number of rows of each trial. It is also worth noting that diskBytesSpilled as well as memoryBytesSpilled were not included as they are constantly zero in every trial.

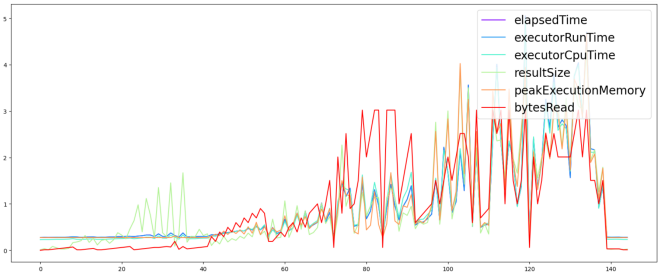

Fig. 11. PCA output values

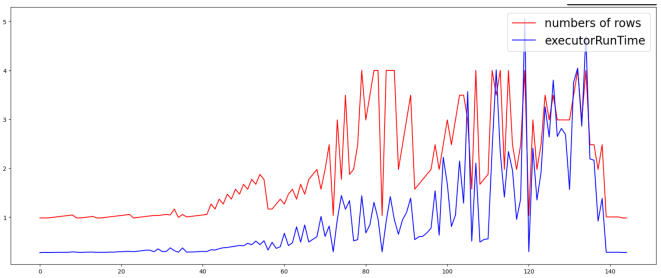

Fig. 12. PCA : Number Of Rows & Executor Run Time

3) Frequent Pattern: Finally, FP-Growth outputs are similar, except peakExcecutionMemory, while, again diskBytesSpilled and memoryBytesSpilled were zero.

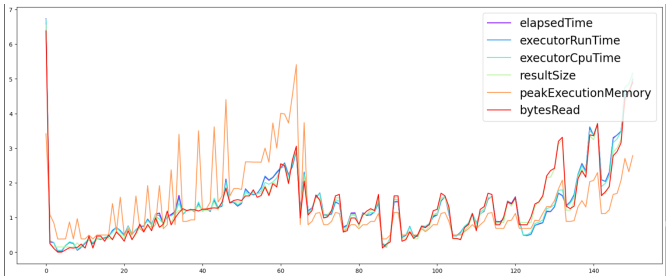

Fig. 13. FP output values

What differentiates FP from the other two is that both the input variables seem to follow the output values closely, with the total number of rows being a little more relevant.

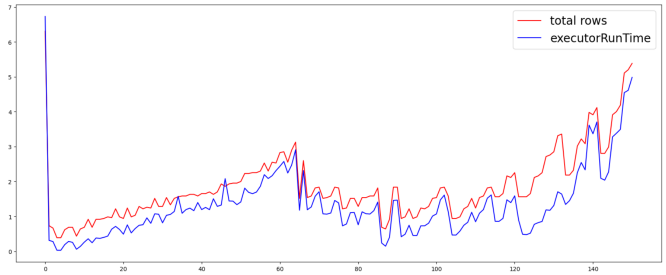

Fig. 14. PCA : Number Of Rows & Executor Run Time

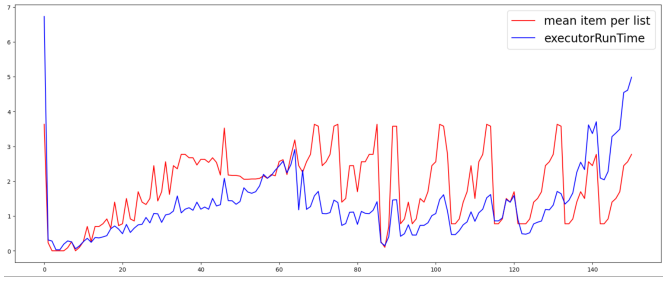

Fig. 15. PCA : Mean Item & Executor Run Time

D. Data Augmentation

As mentioned before, the process of generating loads of trials was time consuming, mainly because of the interuptions, since we were running our code on the free version of google colab. That is why we decided to create a custom function to produce more data, similar to those generated. We decided against using a function from a library such as sklearn or other machine learning libraries, as the goal was to add a small variation to all the values, up to 5%, while adding 3000 new inputs. This way we had more control of what was happening, instead of resorting to a black box.The networks were tested both with and without the augmented data and it yielded better results than without it.

E. Neural Networks

Out of the prediction models tested, the ones that had the best performance were neural networks.This in not a surprise as our outputs are relatively linear to some of our input data, as demonstrated before. The architecture and hyper-parameters were a product of a trial and error process. All three networks followed the same pattern. An input layer, and then hidden layers, with neurons following a pyramid like shape, with 8, 16,..64,..16,8 for Decision Tree, 8 to 128 and down to for PCA and 8 to 32 and down for FP. The activation functions chosen were ReLU for all hidden layers, and linear for the output layer. tanh, which is a very common one was also tested, but did not yield satisfactory results. The hyper parameters were the following

- **Optimiser: Adam**
- **Learning Rate : 0.005**
- **Loss : Mean Squared Error(MSE)**
- **Epochs :1000**

We also recorded the accuracy of the model, which in hindsight was not a classification task, but a value prediction one. Accuracy in such a case becomes ambiguous and is not a good metric. That's why we created our own metric function, which computes the percentage error between the test set and the predictions on the model.

As shown in the diagrams below, the error is generally confined between 5 and negative 5 % which is a very good result. There are however some outliers, which, when examined, turned out to be the biggest datasets, that as mentioned before, were the minority in our samples.

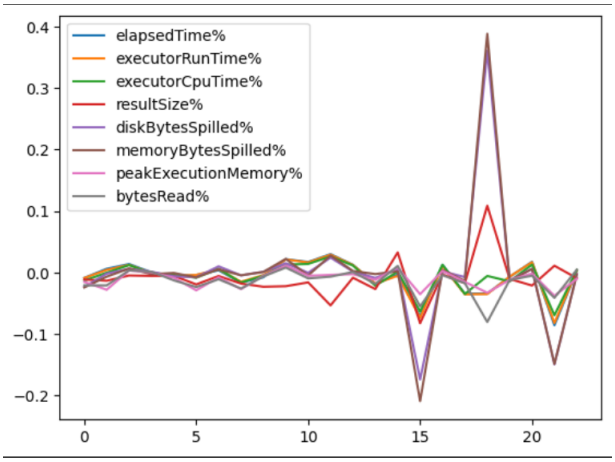

Fig. 16. DT Error plot

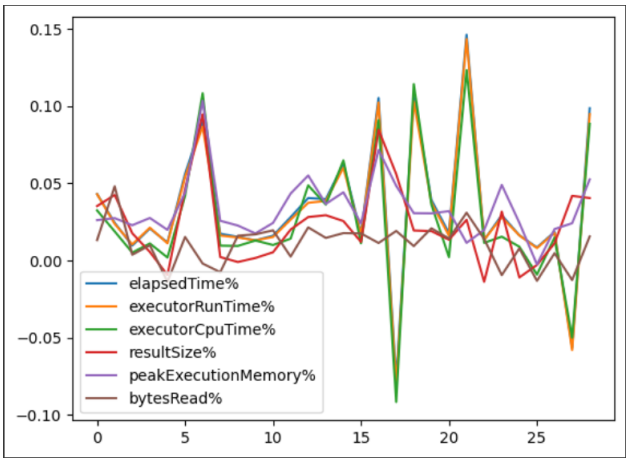

Fig. 17. PCA Error plot

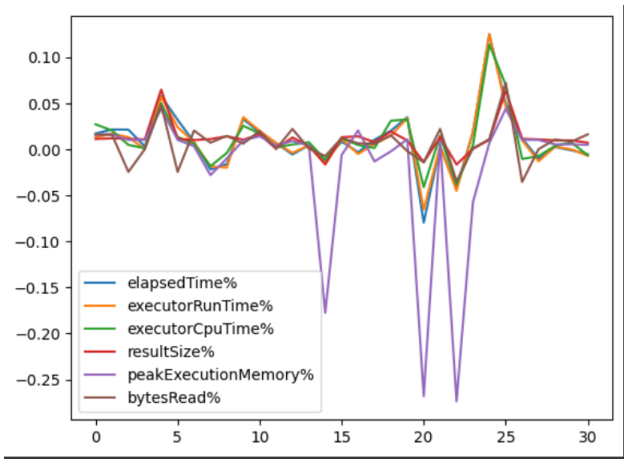

Fig. 18. FP Error plot

Another metric was the loss function, MSE, which was used during training as well.

VIII. Possible Expansions

Even if the results are quite satisfactory, there are some actions that we could take in order to fit some of the outliers. For starters, we could run more trials for bigger datasets. Additionally, we could employ a function that was designed to help in unbalanced datasets, some of which are provided by libraries such as sklearn.

IX. Conclusion

Machine learning has become a huge part of our everyday life. From our cell phones, computers, cars, to our fridges and devices that we wouldn't even consider would become "smart". With the expansion of such techniques, the computational power required has become more and more vital. Additionally, since we have evolved from the "all purpose" computer, to a more distributed model, the distribution of the computational power itself is of great importance. This study provides a useful insight on how to measure the performance of such operators.

As far as we are concerned, we faced a variety of problems, ranging from very practical, such as the location in which we run the code, to algorithmic and theoretical, such as the form of the data to be generated. We had to change environments for the sake of performance, visualize and analyse the data.

References

[1] Swain, Philip H., and Hans Hauska. "The decision tree classifier: Design and potential." IEEE Transactions on Geoscience Electronics 15.3 (1977): 142-147.

[2] Said, Aiman Moyaid, P. D. D. Dominic, and Azween B. Abdullah. "A comparative study of fp-growth variations." International journal of computer science and network security 9.5 (2009): 266-272.

[3] Borgelt, Christian. "An Implementation of the FP-growth Algorithm." Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations. 2005.

[4] Wold, Svante, Kim Esbensen, and Paul Geladi. "Principal component analysis." Chemometrics and intelligent laboratory systems 2.1-3 (1987): 37-52.

[5] Ringnér, Markus. "What is principal component analysis?." Nature biotechnology 26.3 (2008): 303-304.

[6] Pence, H. E. (2014). What is Big Data and Why is it Important? Journal of Educational Technology Systems, 43(2), 159–171.

[7] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., ... & Talwalkar, A. (2016). Mllib: Machine learning in apache spark. The journal of machine learning research, 17(1), 1235-1241.

[8] Liu, Y., & Guan, Y. (2008, November). Fp-growth algorithm for application in research of market basket analysis. In 2008 IEEE International Conference on Computational Cybernetics (pp. 269-272). IEEE.

[9] Scholz, M. (2012). Validation of nonlinear PCA. Neural processing letters, 36, 21-30

[10] Günlük, O., Kalagnanam, J., Li, M., Menickelly, M., & Scheinberg, K. (2021). Optimal decision trees for categorical data via integer programming. Journal of global optimization, 81, 233-260.

[11] Lion, D., Chiu, A., Stumm, M., & Yuan, D. (2022). Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster?. In 2022 USENIX Annual Technical Conference (USENIX ATC 22) (pp. 835-852).

[12] Niitsuma, H., & Okada, T. (2005, May). Covariance and PCA for categorical variables. In Pacific-Asia Conference on Knowledge Discovery and Data Mining (pp. 523-528). Berlin, Heidelberg: Springer Berlin Heidelberg.

[13] https://github.com/LucaCanali/sparkMeasure

[14] Shanker, M., Hu, M. Y., & Hung, M. S. (1996). Effect of data standardization on neural network training. Omega, 24(4), 385-397.

[15] He, J., Li, L., Xu, J., & Zheng, C. (2018). ReLU deep neural networks and linear finite elements. arXiv preprint arXiv:1807.03973.