FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Workflow process definition in a Cyber Physical Production System

Gonçalo Henriques de Proença Pinho

Mestrado em Engenharia Eletrotécnica e de Computadores

Orientador: Gil Gonçalves

Co-orientador: João Reis

March 31, 2022

Abstract

As automatization and digitalization grows exponentially in the world we know, industry faces more and more challenges in order to keep up with the indexed demands. Workflows ease the processes definition that allow users to respond to those demands. In order to facilitate workflow definition for any given scenario, we propose a recommendation system based on user-defined specific and performance requirements that calculate and return optimal workflows. The system takes two performance metrics and the user is to choose the amount of preference each performance requirement has on the final workflow. This preference is chosen through a percentage value. The proposed system uses this input from the user and resorts to a function block database to estimate the best workflow for the desired requirements. The database has all the information of each function block performance as weights, according to the two defined metrics. Graph theory and dynamic programming strategies were used to implement this software. So as to automatize the definition of these weights, an optimizer system was also developed based on a genetic algorithm. This system takes history logs of workflows performance results and returns the most optimized set of weights to the user, so that they can be used to define each function block to be then used in the main recommending system. The test validation phase was done using a dataset from a real-life example. The used paper proposes a strategy to recommend workflows based on user-defined requirements for specific situations and resorting to a linear programming algorithm. Both systems were tested on their performance, scalability, and the ability of working as the complement of one another. All tests done showed good performance from both algorithms in different case scenarios. Afterwards we discussed on different possible approaches for each major step on the proposed implementation as well as some future work suggestions. Lastly we presented some overall conclusions on the work done.

Agradecimentos

Inicialmente gostava de agradecer aos meus dois orientadores. O professor Gil Gonçalves pela oportunidade de realizar este trabalho espetacular bem como toda a ajuda dada no processo. Um especial abraço ao meu co-orientador, o professor João Reis pela sua disponibilidade incansável, toda a orientação dada e a todas as trocas de ideias que possibilitou que este trabalho pudesse ter sucesso.

Quero também agradecer do fundo do coração à minha família, à minha mãe, ao meu pai e à pessoa mais importante da minha vida, a minha irmã pelo apoio incondicional que me deram não só durante a realização desta dissertação mas também ao longo destes últimos 5 anos e meio, apesar de todas as diversidades encontradas no caminho. Sem eles nada disto seria possível e devo-lhes tudo por esta oportunidade.

Um agradecimento especial também à Anabela, ao Luis, ao Gonçalo e à Lia que também são a minha família e que também mostraram apoio incondicional.

Outro agradecimento a mais um membro da família ainda não mencionado que é o Levi Peseta, por estarmos juntos a apoiar-nos não só nestes últimos 5 anos mas nos últimos 15 anos de irmandade.

Queria também agradecer a todas as pessoas que me ajudaram a chegar a este ponto, à Catarina e à Diana que me acompanharam de perto durante a realização deste trabalho, à Matilde por todo o apoio dado no último ano e meio, aos meus antigos colegas de casa Ana, Anze e João por terem ajudado bastante nas alturas críticas desta dissertação e a uma pessoa muito especial que me ajudou a manter a sanidade mental neste últimos meses de trabalho com todo o otimismo da sua parte, a Maria.

Por último quero agradecer bem lá do fundo às pessoas que me acompanharam de perto nesta jornada de 5 anos e meio e que pretendo levar comigo para o resto da vida. Sem eles isto não tinha tido metade da piada e possivelmente muito mais difícil de alcançar. E que apesar de todas as adversidades estiveram sempre do meu lado e prontos para me receber. Um grande abraço para o Coentrão, Imobicho, Suma, Barbot, Jynx, Quiche, Impressora, TAC, Eclair, Sirenes, Badass, Pro-V, Sequóia, Desaparecida, Capucho e Fi (acho que não me enganei na ordem). Isto sem vocês não valia nada,

Gonçalo Henriques de Proença Pinho

“Dai-me mais vinho, porque a vida é nada.”

Fernando Pessoa

Contents

List of Figures

List of Tables

Abbreviations

AutoML	Automated Machine Learning
BK	Basic Kernel
BRJ	Basic Record Join
BTO	Basic Token Ordering
COTS	Commercial Off-The-Shelf
CPS	Cyber-Physical System
CPPS	Cyber-Physical Production System
DAG	Directed Acyclic Graph
DFP	Data Fusion Pipeline
DS	Data Source
DT	Digital Twins
DW	Data Warehouse
EMCA	Enhanced Monte Carlo Algorithm
ETL	Extract-Transform-Load
FB	Function Block
GA	Genetic Algorithm
ICE	Intelligent Cyber-Enterprise
ICPS	Industrial Cyber-Physical Systems
IoT	Internet of Things
MILP	Mixed Integer Linear Programming
NoSQL	Not Only SQL
OPRK	One Phased Token Ordering
OPTO	One Phase Token Ordering
PK	Indexed Kernel
RJMCMC	Reversible-Jump Markov Chain Monte Carlo
TEMC	Tensor regression-based Extended Matrix Completion
SSJ-MR	Set-Similarity Joins using MapReduce
TOD	Time-Ordered Data
UDF	User-Defined Function
VMI	Virtual Machine Instances
WFaaS	Workflow as a Service
WfMR	Workflow Model Recommendation

Chapter 1

Introduction

1.1 Context

As times goes, the growing of the industry 4.0 revolution is more and more noticeable. This has brought us a lot of new, complex and widely known concepts that have been used by companies at an increasingly rate. Internet of Things (IoT), Smart Factory, Machine Learning, Cyber Physical Systems (CPS) and cloud systems are some examples of the most known names related to industry 4.0 [1]. The companies have been experiencing a dire need to implement the digitalization of their own factories as their biggest competitors have been doing it for years.

Digitalization of factories floors are, on one hand, very complex, though on the other hand have huge potential applications as they bring a full package of advantages such as Machine-to-Machine (M2M) communication, error tracking, energy and production efficiencies, anomalies prediction and much more.

Nowadays' factories floors are way more complex than ever before and that increases the need to develop tools that would give more flexibility, better communication between different machines and modularity for the systems. Given that, the digitalization becomes such an important aspect for the industry. This digitalization creates the so called digital twins. A digital twin may be seen as a simulated world where everything is connected to everything. It provides a digital version of a factory with real-time data. This data can be processed and used to improve the performance on several sectors of an industrial factory.

IEC 61499 is a standardization model based on function blocks to represent processes that might be part of a workflow. Even though IEC 61499 is a standard developed for distributed control systems, it is been used nowadays as a tool to implement Cyber-Physical Production Systems and create key workflows of Digital Twin orchestration. A workflow represents a sequence of processes that are supposed to meet some kind of requirements such as solving specific problems, data processing from sensor readings or production. Workflows are commonly seen in Cyber Physical Systems (CPS). CPS represent the objects of the physical layer of the industry on a digital twin. Cyber Physical Production Systems (CPPS) represent a CPS but with the add-on of production controlling features. The standardization of these workflow processes is very important as it makes possible to use a global approach to the adaption and creation of new workflows in a CPPS.

1.2 Motivation

The industry 3.0 revolution has brought us the automatization, computers and electronic devices in general, and hence the need of optimization and adaption of these machines has increased as well. Industry 4.0 has been making that both easier and possible through digitalization.

Workflows take a huge role on the automation processes, and for this work purposes, a workflow is deemed as a simple sequence of processes/activities. Therefore, throughout the rest of the document we refer to a workflow as a pipeline, thus using both terms.

Workflows can be tested and evaluated to give us a better understanding of how well it meets desired requirements. To have a better understanding on this, workflows are extremely well explained with a real life example in [2], as it is explained a real situation with the need to use workflows to solve a problem, as well as a talk on every aspect someone should be paying attention to when defining a workflow.

The need of adaption, optimization or even the creation of workflows is huge. A lot of data may be gathered within the execution of a specific workflow and several different goals or requirements can be set. Some examples of workflow requirements are mentioned in [3]. Those requirements are defined as either specific requirements or performance requirements. Specific requirements describe what the user needs the workflow to do. Performance requirements are the ones linked with the execution performance of the workflow.

To make it more clear, an example of a workflow in the given context may go from retrieving data from a working machine, reduce, and normalize that data. For this example, the type of data ingestion, data reduction, and data normalization are the specific requirements. Performance requirements could be execution time efficiency and execution cost.

1.3 Problem Definition

Having all this information, setting the problem definition is the next step to take. Defining a workflow may get really hard depending on the goals and requirements to meet, as well as on its complexity. Moreover, context adaptability is also a challenging point. Therefore, the plan is to develop a recommendation framework.

So for different scenarios, a user may want to build a pipeline that meets the desired requirements. In order to do this and for each stage of the workflow, he must choose an option that completes that stage. This may get harder and harder to be done manually when dealing with several stages and several options for each stage, and so the problem to solve is the automatization of this process.

Being in a user standpoint and having well defined requirements for a workflow to meet, it would make life easier to have a solution that could give feedback on how the workflow should be made or adapted, in case where the workflow already exists and needs performance improving.

In short, the engineering problem we have in hands to solve is the formulation of a workflow in a way that it is mathematically easy to deal with, and that allows going from an input to an output. The input shall be a set of blocks available to be chosen and a set of requirements for the workflow to meet, and the output a list of blocks in form of pipeline that maximizes the desired requirements. To sum up, we are dealing with an optimization problem, specifically maximization.

1.4 Objectives

This dissertation focuses on the study and development of a technology that would solve the problem of a user who needs workflow tweaking. A system that would receive user-defined specific and performance requirements for a certain workflow for a specific situation, and send an optimized workflow as output that meets those requirements, is a good example of a solution to the problem presented.

In first place, a deep study was done to the world around workflows, researching already existing ones, understanding the different types there are and examples for each one (e.g. predictive, optimization and parameters estimation, image and data acquisition and processing, data reduction, etc). For instance, in [4] it is shown an example of a workflow for a gas turbine maintenance and is explained why specific processes are used.

As far as the practical implementation concerns, the following topics represent the main goals of this dissertation:

- Understanding the different requirements for workflows in accordance with all the study done.
- Development of an adaptive recommending system that calculates the best possible workflow while meeting user-defined requirements. This is the core element of the dissertation.
- Understanding how to assess the system performance and use it for testing and validation.

Chapter 2

Bibliographic Review

This chapter encompasses all the research made as well as the methodology used during this same research. Research methods will be explained in the section 2.1. Several papers from different areas of interest will be addressed as well as a brief explanation of what information can be either useful or not for the dissertation in the section 2.2, thus representing the state-of-art.

2.1 Research Methodology

In this section research methods and the approach done on the research process will be explained.

2.1.1 Research Questions

The first phase of knowledge acquirement regarding this topic was in the first contact with the supervisors where the paper [5] was recommended for the students to read. The paper addressed the concept of research questions and their importance for a successful searching. These questions allow the researcher to figure out which questions were to be answered with the work.

There are two types of research questions mentioned in [5]. General research questions and specific research questions. The general questions approach the dissertation topic in a generalized manner while the specific questions focus on gathering much deeper knowledge in specific areas directly correlated with the dissertation.

The research questions used in this research are presented down below.

General Research Questions

- RQ1: What is a digital twin?
- RQ2: What is the Industrial 4.0 revolution?
- RQ3: What is a Cyber-Physical Production System?
- RQ4: What is a workflow in the CPS context?

Specific Research Questions

- RQ5: What types of workflows exist?
- RQ6: What are the existent methods to define workflows?
- RQ7: How can we split a pipeline into different phases?
- RQ8: What requirements should be defined for each type of workflow?
- RQ9: What would be the best evaluation metrics for a pipeline performance?
- RQ10: Are there any recommendation systems that recommend workflows?
- RQ11: How should a workflow be defined / formulated for recommendation systems?

2.1.2 Keywords

In order to ease the research, the second phase would be the definition of a few keywords to be used along with the research questions. The chosen keywords were the following:

- Industry 4.0.
- Digital Twin.
- Cyber-Physical System.
- Pipeline.
- Workflow.
- Predictive Maintenance.
- Data Reduction.
- Prescriptive System.
- Recommendation System.

2.1.3 Research Process

Once research questions and keywords are set, the next step is to find the best search engines and databases. Two seminars were carried out by a FEUP's team to enlight the students on this topic. Several engines were addressed during these seminars, such as *IEEE Explore*, *Web of Science*, *Scopus*, *Inspec*, *Google Scholar*, and more. *Google Scholar* was the prevailing engine for this research.

Besides the searching engines, some applications were also addressed to be used as libraries for the papers found. *Mendeley*[1] was the choice to serve this purpose.

The table 2.1 is used to clarify which keywords and search strings were used for each of the Research Questions referred above.

[1] https://www.mendeley.com/

Table 2.1: Table with the relation between each research question and respective keywords.

Research Question	Keywords
What is a digital twin?	Digital twin, Digital twins
What is the Industrial 4.0 revolution?	Industry 4.0, Industry 4.0 revolution
What is a Cyber-Physical Production System?	Cyber-Physical System, Cyber-Physical Production System
What is a workflow in the CPS context?	Workflow in a CPS, Workflow, Pipelines in a CPS, Pipeline
What types of workflows exist?	Predictive maintenance workflow, Data reduction workflow, Predictive maintenance pipeline, Data redcution pipeline
What are the existent methods for workflow definition?	Workflow definition, Pipeline definition
How can we split a pipeline into different phases?	Workflow definition, Pipeline definition
What should be the requirements for each type of workflow?	Workflow definition, Pipeline definition
What would be the best evaluation metrics for a pipeline performance?	Workflow definition, Pipeline definition
Are there any recommendation systems that recommend workflows?	Recommendation system, Prescriptive system Workflow recommendation, Pipeline recommendation
How should a workflow be formulated for recommendation systems?	Recommendation system, Workflow recommendation sytem, Pipeline recommendation system

The results from the research has returned a total of 24 papers which had the needed information to answer all the questions. The next section includes the obtained information and its analysis from all the papers found.

2.2 Literature Review

This section will address all the related work done in order to answer the best way possible to every research question presented in 2.1.1. The section will be divided in different sub-sections. In 2.2.1 RQ1, RQ2 and RQ3 will be addressed with some research on DT and CPPS. The sub-section Workflow definition is one of the most important of the state-of-art as well one of the most deep-studied as it focus on workflows definition, and how important it is for the industry, aiming at RQ4, RQ6, RQ7, RQ8, and RQ9. In sub-section 2.2.3 some types of workflow are going to be presented, and explained in detail, thus answering RQ5. Last but not least in 2.2.4 existing recommendation/prescriptive systems for workflows are studied, compared and correlated with the system this dissertation proposes to do, in order to answer RQ10 and RQ11.

2.2.1 Digital twins and Cyber-Pysical Production System

As already referred a couple of times both digital twins and Cyber-Physical Systems are two important concepts in order to have some general context. First of all, it is important to understand the definition, correlation and comparison of both.

In [6] all this points are explained. They refer to DT as the concept of using a copy of a physical system to perform real-time optimization. A CPS is seen as the integration of computational and physical processes. They allocate each of the concepts into two different categories. DT are in the engineering category and CPS are in the scientific category. On one hand, as the DT focus on representing a virtual physical model, it is important to have similarities of high detail between the two systems. It has a high accuracy on several aspects of the model, like geometry, structure, behavior, rules, and functional properties. It represents a specific physical object. On the other hand, the CPS focus more on controlling than on mirrored models. *"The relationship between the cyber and the physical worlds of a CPS is not one-to-one, but instead a one-to-many correspondence"*[6]. These two concepts not only have their differences but also their correlation as well as an integration aspect. The integration of both of them may help factories to get more precise and efficient operations. They also refer to DT as a critical base to implement a CPS. The figure 2.1 intends to illustrate this integration.

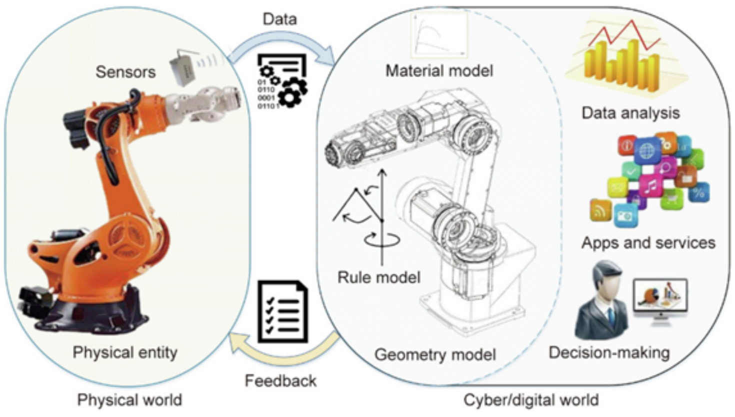


Figure 2.1: Integration of CPS and DT. From [6].

To sum up both definitions and applications in real world, the paper concludes that having all the data needed about environment, status, behavior, and properties of a physical system as well as a good model as its base then a DT can be used to help manufacturers. Those are the core elements of a digital twin. When it comes to a CPS, it has in account the sensors and actuators, these generating more and more data that will eventually be processed by the cyber system. Therefore sensors and actuators are its core elements. In this last paper they also make a small introduction to some elements of Industry 4.0 revolution, affirming that these two concept appeared along with it, stating that in Germany, the cyber-physical systems are considered to be the core and the creation of Industry 4.0 [7].

Another approach taken during the searching for more information about a CPS, was how it can be implemented and what the future of enterprises will be as an Intelligent CPS. This topic was studied in [8] as they give a small introduction to the large number of areas that cyber-physical systems can be implemented, those being medical and healthcare, energy, transportation, mobility, manufacturing, etc. They present some principles for CPS as to design complex systems, such as the interoperability and communication between sub-systems as well as approaches to develop tools to help reducing the energy consumption. These are connected with the expression ICE, this being an important concept for the implementation of CPS in the industry. Several other aspects about it are addressed in the paper.

2.2.2 Workflow definition

Being the core element of this dissertation, workflow definition becomes the most important concept of the research phase. Several papers were under analysis in order to get the best idea what a workflow is. These papers were [2, 9, 10, 11, 12].

To get a first visualization of what a workflow may be, [2] really helps as they talk about CPS that use a vast network of devices such as sensors, gateways, switches, routers, computing resources, applications and services to link both physical and cyber worlds. This application can be broken down in the form of workflows. A workflow, as explained back in section 1.2, is a sequence of very-well defined processes that aim to meet predefined requirements. In this paper they give a real life example of application of workflows. *"Considering Flood Disaster Management applications that may consist in a collection of workflow activities such as capturing and analyzing social media and sensor data as well as using more complex computational models to detect dangerous situations in real-time. A workflow activity like this may need to implement computation models for analyzing social media data, for instance, anomaly detection, clustering, classification. The workflow activity requires the real-time processing of the sensor data."*. They then address the different requirements deploying a workflow like this would have, such as cyber device types, cyber application services, social data sources, and ambient physical world. To finish, they refer 10 different papers that schedule different activities, in different areas of study, with different approaches ([13], [14], [15], [16], [17], [18], [19], [20], [21], [22].

The paper [10] refers to workflows and its importance as a service in the cloud. This paper gives some info on computational resources that scheduling a workflow would need. For instance, 3 conclusions are obtained from the studies of scheduling workflows as a service in a cloud, such as the performance of a WFaaS system will be better when the number of usable Virtual Machine Instances (VMIs) increases. Besides that, in [4] it is proposed a workflow model for a gas turbine maintenance. The workflow is shown in Figure 2.2.

One of the best methods of workflow scheduling was studied in [9]. They propose a scheduling method based on colored Petri nets. Firstly, it is presented the different methods of scheduling (static, dynamic, and phased). Static would be the scheduling happening in the beginning for the

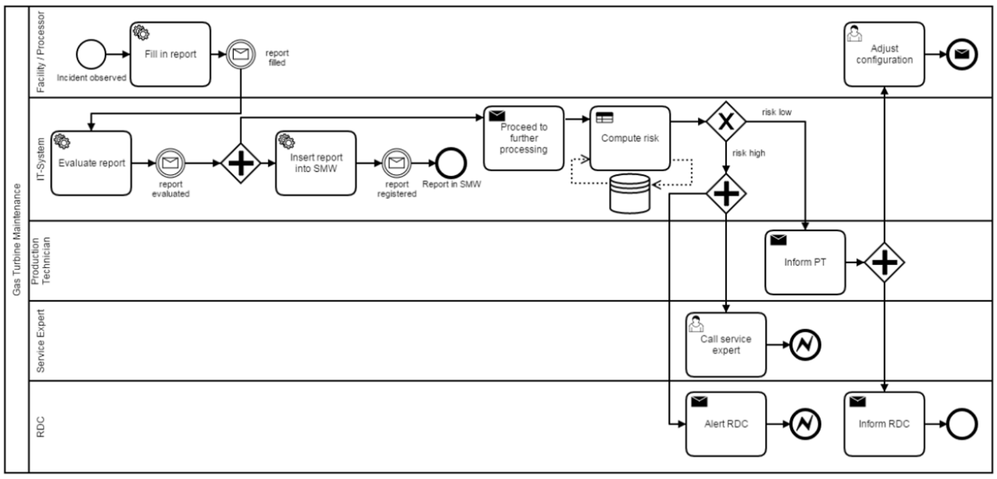


Figure 2.2: Gas turbine workflow model. From [4].

whole workflow and does not have in account the resources available during the execution of the pipeline. Dynamic would re-schedule the workflow every time a stage is completed, always having in attention the resources available. The phased approach is the proven to be the best for the case study in the paper as it is a mix of the other two as it is defined a group of stages to be scheduled all in once, having then the characteristics of a static scheduling that would be optimized for different resource availability during execution time. Another good information obtained from this paper is the evaluation metrics they use to test the performance of each method, these being **late tasks**, **total tardiness**, and **average throughput time of instances**.

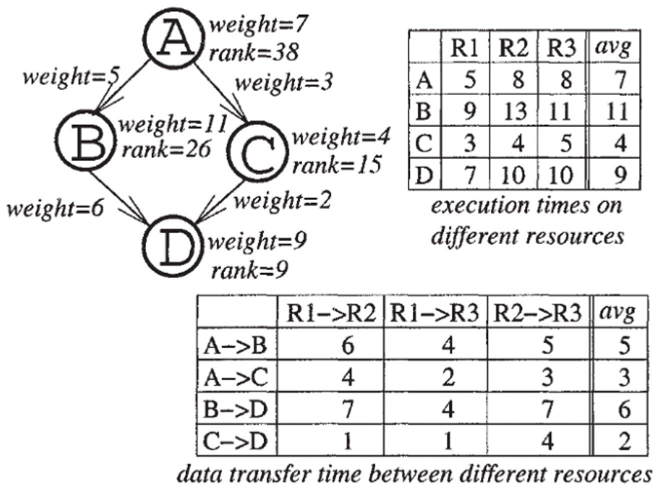
The other two papers, [11] and [12], were two very important source of knowledge. The first one proposes a very good combination of scheduling algorithm and scheduling strategy. They use a full-ahead scheduling with the HEFT algorithm [23]. The HEFT algorithm consists on 3 phases:

- ***Weighting*** - assigns the weights to the nodes and edges in the workflow.
- ***Ranking*** - creates a list of tasks, organized in the order how they should be executed.
- ***Mapping*** - assigns the tasks to the resources.

The figure 2.3 shows how to calculate the weights and ranks for each node.

In what scheduling strategy is concerned, it was said above that the best one to match with the HEFT was the full-ahead strategy though there is one other choice, *Just-in time strategy*. Just like it was explained above side-by-side with paper [9], they refer to full-ahead as a static method where the full graph scheduling is performed at the beginning of execution, and the just-in time strategy as a dynamic method that consists of mapping the tasks to the resources, always choosing the most appropriate solution for the current step. There are might be some strategies in between those, being the phased, just like partitioning [24]. Static seems to be the method that is most suitable for the dissertation.

In the last paper of this topic [12], they focus on how a pipeline should be split into different phases/categories such as data ingestion, communication, storage, analysis, and visualization.



	R1	R2	R3	*avg*
A	5	8	8	7
B	9	13	11	11
C	3	4	5	4
D	7	10	10	9

execution times on different resources

	R1->R2	R1->R3	R2->R3	*avg*
A->B	6	4	5	5
A->C	4	2	3	3
B->D	7	4	7	6
C->D	1	1	4	2

data transfer time between different resources

Figure 2.3: Weights and ranks calculated with HEFT algorithm. From [11].

They did a survey by analyzing a total of 38 papers of which would be the best software to develop each of the phases. This is one of the most completed papers when it comes to rich information as it addresses what is missing on related papers and proceeds to give input on that missing information. The tools proposed for each phase are the following:

- ***Ingestion*** - Custom tools.
- ***Communication*** - A COTS service.
- ***Storage*** - NoSQL databases.
- ***Analysis*** - Incorporating a suitable real-time processing tool.
- ***Visualization*** - This should be addressed in accordance to the requirements for the data to be analyzed.

In [4] the most useful piece of information is the visual representation of workflow that gives a better context on the dissertation main topic. The two most important were [11, 12]. Good scheduling algorithms as well as good scheduling strategies are presented that may be really helpful in future work.

2.2.3 Types of workflows

In this part, workflows will be divided and studied by their types. As shown in 2.2.2, pipelines can be applied in a big variety of situations. This research focused on three main types of application of workflows them being predictive maintenance, data reduction, and parameters estimation. Each individual type will be explained in the sub-sections down below.

2.2.3.1 Predictive maintenance

Maintenance is a very important process for the success of an industry. Machine downtime is one of the heaviest costs a manufacturing enterprise can have. To avoid this, maintenance and repair of machines are done following on of three methods, reactive, preventive, and predictive maintenance [25]. While each of strategies have their own advantages and disadvantages, in [26] it is explained the strong and weak points of each. While the reactive maintenance has the smallest costs on the repairing, it does cost a lot on downtime. Preventive does reduce the downtime to its minimum, though increasing the repair costs as a machine may go under maintenance with no need. Predictive maintenance offers the best of both worlds as it predicts when a machine would have a failure or breakdown causing downtime. Predictive maintenance works under heavy prediction algorithms that can use several history logs. Some papers were selected to study predictive maintenance in different applications. In [4] it is proposed the workflow model used for the maintenance of a gas turbine shown in 2.2. In [26] a few more details are addressed such as the requirements for this type of workflows. They distinguish data ingestion based on historical data and based on real-time sensor data. It is proposed a mix of strategies for the different data analysis that want to be done. The figure 2.4 the different approaches on predictive maintenance.

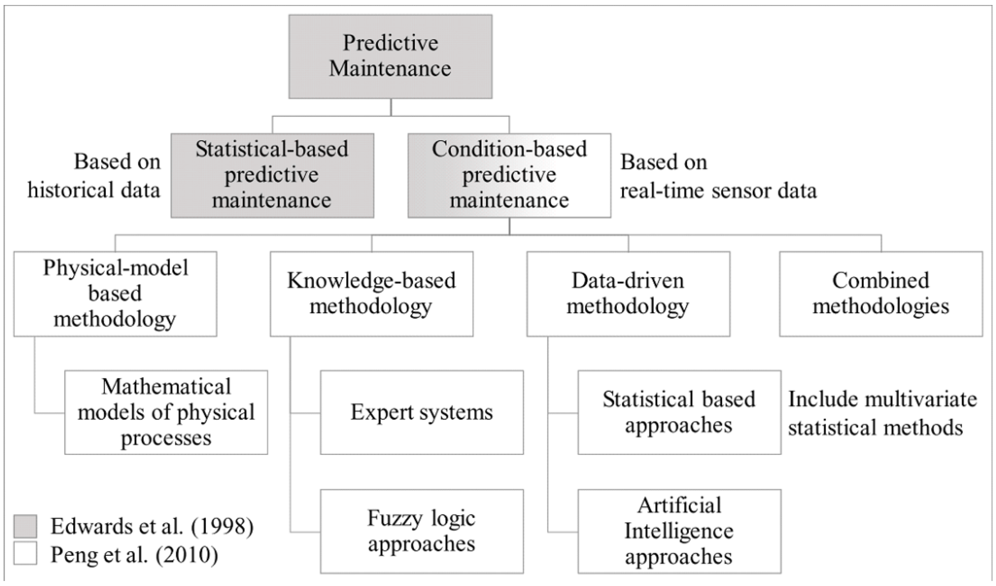

Predictive Maintenance
Based on historical data
Statistical-based predictive maintenance
Condition-based predictive maintenance
Based on real-time sensor data
Physical-model based methodology
Knowledge-based methodology
Data-driven methodology
Combined methodologies
Mathematical models of physical processes
Expert systems
Statistical based approaches
Include multivariate statistical methods
Fuzzy logic approaches
Artificial Intelligence approaches
Edwards et al. (1998)
Peng et al. (2010)

Figure 2.4: Systematizing predictive maintenance approaches. From [26]. Edwards et al. (1998) [27]. Peng et al. (2010) [28].

In the paper [3] not only a predictive maintenance pipeline is shown, but also discussed the problem domain and formulation, and evaluation metrics. The base concept of this paper is predicting equipment failures by just reading logs from the past. Log data is a collection of events recorded by various applications running on different machines. It has information about the timestamp, event code, message text, and event severity. The core element of the workflow is a central database, as data is processed from it. Analysis consist of data preparation, model building, model evaluation, and monitoring. The predictive maintenance workflow is shown in figure 2.5.

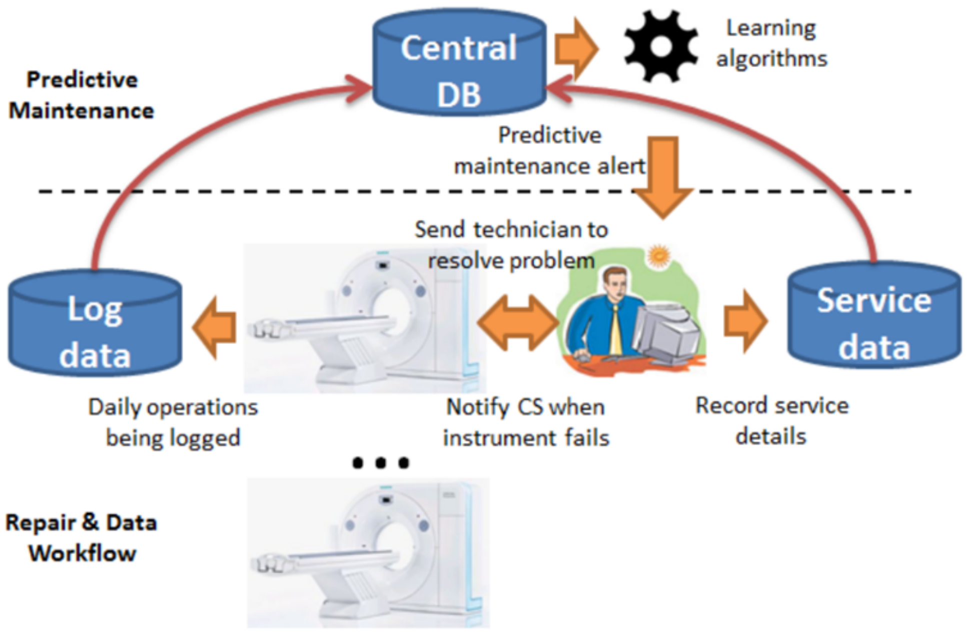

Predictive Maintenance
Central DB
Learning algorithms
Predictive maintenance alert
Send technician to resolve problem
Log data
Service data
Daily operations being logged
Notify CS when instrument fails
Record service details
Repair & Data Workflow

Figure 2.5: Predictive maintenance workflow. From [3].

The requirements for this workflow are:

- Predictive interval.
- Infected interval.
- Responsive duration.
- Interpretability.
- Efficiency.
- Handling class imbalance.

The chosen evaluation metrics were precision and recall given by the following expressions:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{2.1}$$

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{2.2}$$

The last paper that was under analysis concerning this topic was [29], having a huge amount of information regarding several workflow-related topics. It is pointed to major smart manufacturing facilities where the increasing rate of data production is a big concern. Smart manufacturing is thoroughly explained as well as the aspects it focus on them being the beneficial transformations it brings to an industry, the different phases of implementation, and its challenges and requirements. After addressing all those points, they also propose a data pipeline architecture. This pipeline is divided in six distinct stages. The stages are:

1. **Site Manager:** multiple functions such as scheduling and job assigning to ingestion engines based on their availability and location and decide how much data each node should ingest based on CPU and bandwidth availability.

2. **Inesgtion Process:** communicate location, bandwidth, CPU and memory availability to the site manager, interpret data collection tasks sent from the site manager and automatically extract time-series data in accordance with tasks parameters and transmit the acquired time-series data to the cloud.

3. **Message Queue:** notify the subscription service when new data has been received from the factory and persist the received data in a queue so it may be read by data processing components in the data pipeline.

4. **Subscription Service:** listen to the new message queue for new data and notify subscribers when new data is available for processing.

5. **Data Processing:** the data processing requirements vary from site-to-site and application-to-application. Daily average, monthly average, and annual average were the simple aggregation functions for time-series data.

6. **Data access:** ensure data is stored in the appropriate location/context and respond to requests for data that adhere to the convention.

The pipeline and the stages mentioned above are represented in figure 2.6

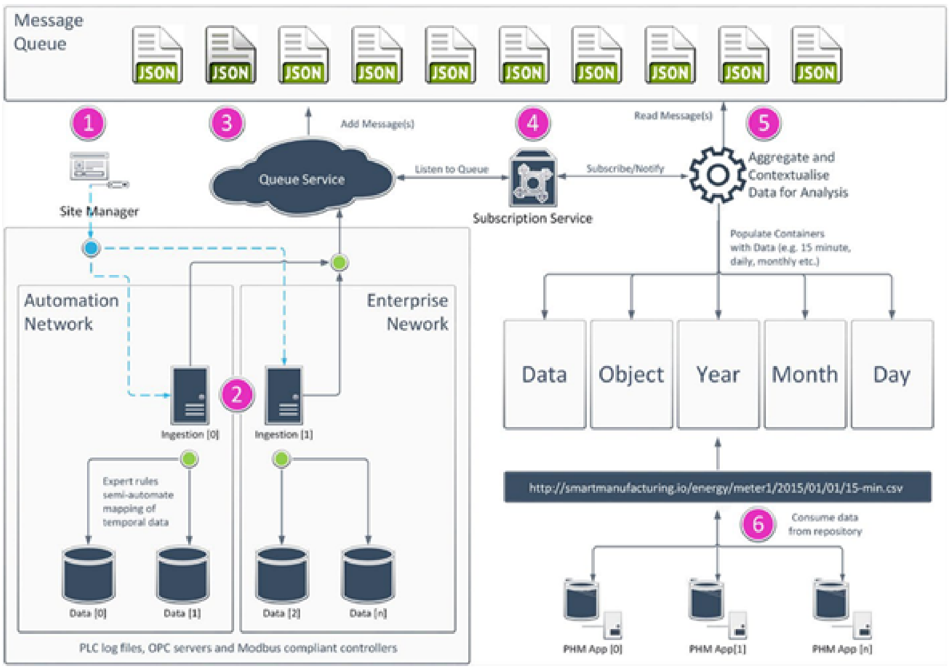

Message Queue
JSON
1
3
4
5
Add Message(s)
Read Message(s)
Site Manager
Queue Service
Listen to Queue
Subscription Service
Subscribe/Notify
Aggregate and Contextualise Data for Analysis
Populate Containers with Data (e.g. 15 minute, daily, monthly etc.)
Automation Network
Enterprise Nework
2
Ingestion [0]
Ingestion [1]
Expert rules semi-automate mapping of temporal data
Data [0]
Data [1]
Data [2]
Data [n]
PLC log files, OPC servers and Modbus compliant controllers
Data
Object
Year
Month
Day
http://smartmanufacturing.io/energy/meter1/2015/01/01/15-min.csv
6
Consume data from repository
PHM App [0]
PHM App[1]
PHM App [n]

Figure 2.6: Big data pipeline architecture and workflow. From [29].

The simulation decomposes the workflow into three distinct parts – data ingestion in the factory, data processing in the cloud, and feeding industrial analytical applications.

There are two main entities when it comes to data ingestion in the factory (ingestion engine and smart sensor). The ingestion engine is part of the pipeline and receives data collection instructions from the site manager, which returns an instruction to read the **Return Air Temperature (RAT)** for Air Handling Unit 1 (AHU1). The smart sensor is a third-party component and is programmed to read the **Set Point Temperature (SPT)** for the AHU1. See figure 2.7.

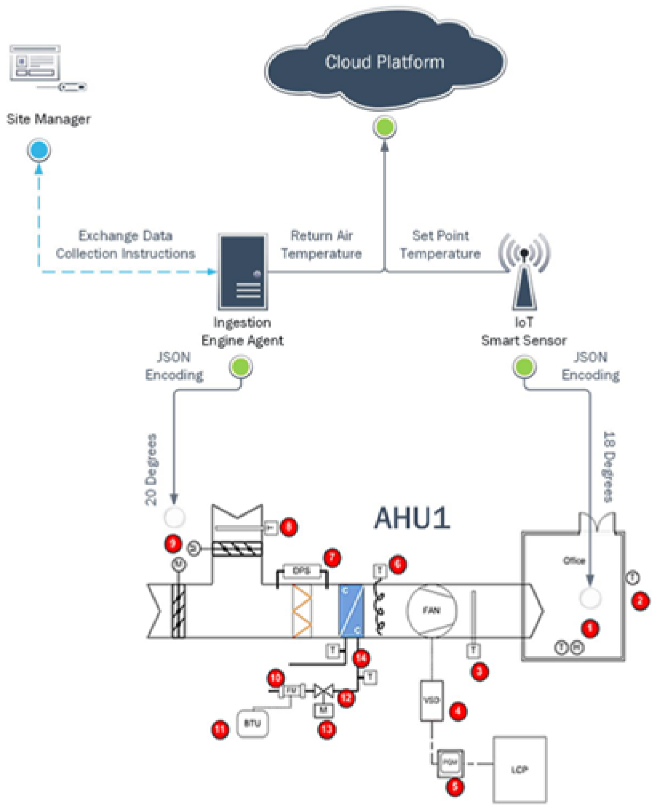


Figure 2.7: Simulation of data ingestion. From [29].

Processing of the data coming from the RAT and SPT measurements. This component reads both messages from the message queue and executes its routine which results in the aggregation of the results in, for instance, 15-min, hourly, daily and monthly intervals. It is located in folders that can be accessed by industrial analytic applications. See figure 2.8.

In what industrial analysis is concerned, the time-series data resides in a directory structure that gives context to the accessed data. Two applications were used. The dashboard uses the data pipeline to access pre-compiled aggregates of time-series data to eliminate the overhead of running this routine dynamically. The other application is a predictive maintenance model that identifies issues in AHU's. The predictive model requests 15-min data for both measurements (RAT & SPT) given its need for granular data. Both applications were able to access data using a common interface without having to engage low-level industrial protocols. See figure 2.9.

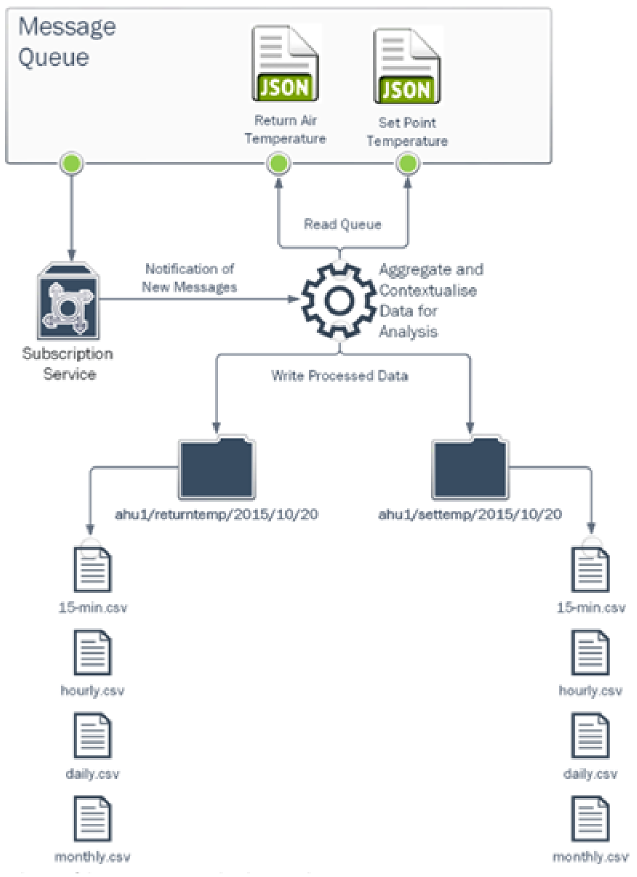


Figure 2.8: Simulation of data processing. From [29].

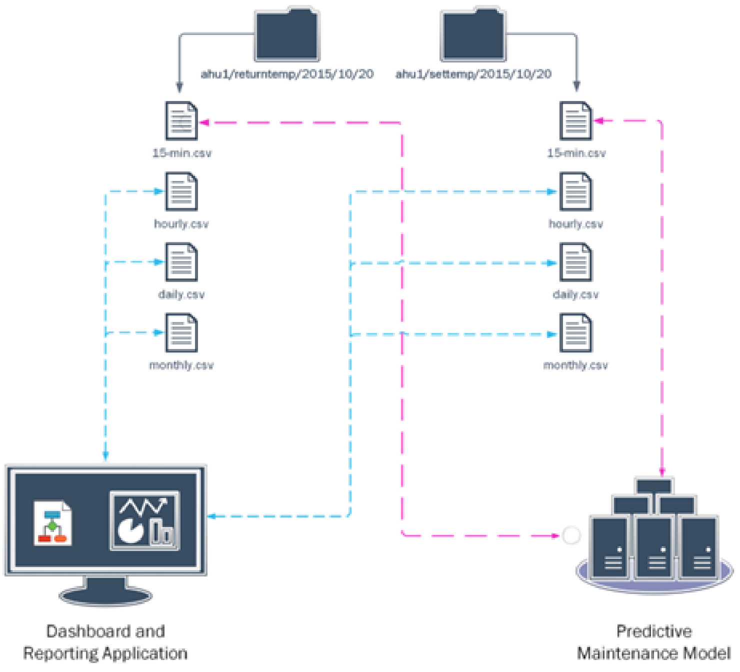


Figure 2.9: Simulation of data analysis. From [29].

From [29] a lot of useful information can be obtained like a pipeline example for predictive maintenance, sub-workflows for the different phases of a workflow, and good techniques to define very carefully what should be done in each of the phases. The most useful information in [3] for the dissertation are the requirements for the workflow and the evaluation metrics. The paper [26] doesn't quite fit on what it has been the main focus of this research, although some approaches on predictive maintenance can be useful in the implementation.

2.2.3.2 Data reduction

Data reduction is a good option in situations where there's a lot of information to be dealt with. Most of the pipelines found concerning data reduction purposes are linked to astronomy-related data. It makes sense as data collected from the space usually comes with a lot of noise, increasing exponentially the rate of data production.

The first paper under analysis is [30]. In the first instance they propose a couple of tools where data reduction pipelines may be used like MIDAS [31], IRAF [32], and IDL (trademark of Research Systems Inc.). They propose the first stage of the workflow to be the data organization which is described by the following steps: files sorting, target identification and group them into data sets that are incomplete and lastly, calibrations are added to the datasets. The figure 2.10 shows the flow chart for the data organizer. Note that if any step in a darker box fails for any dataset, then that dataset is labeled as *incomplete*.

The second stage following data organization is the data processing. Usually data processing is more complex than the data organization. Therefore it makes total sense to separate both stages. The final workflow model proposed uses the "Recipe flexible execution workbench" (Reflex design) implemented in Keppler [33] workflow engine. Figure 2.11 shows the example of a Reflex application in the Keppler engine.

In the rest of the paper they present some advantages on Reflex workflow design, such as rule-based data organiser in use, utlilizers may monitor the progress of data reduction, interact with the process if needed, and modify the pipeline, and also the ability Reflex has to re-execute steps concerned with changes in parameters in data.

The next paper to be discussed is [34], which addresses the rising and already touched subject of data production rate in the industrial context. In this case it is talked about the Oak Ridge National Laboratory (ORNL) and its neutron source facilities, responsible for a huge amount of data production. This massive quantity of data is stored using the standard schema "NeXus" [35]. This stored data that is then loaded for post-processing by various data reduction workflows using Mantid framework [36]. Figure 2.12 gives a general idea of how this framework works. This framework could be a useful tool for data reduction pipelines implementation.

More space-related data to be reduced is addressed in [37, 38]. In [37] it is presented a data reduction pipeline to reduce data coming from a satellite. This satellite receives data from the space. The pipeline features newly developed routines such as, accurate data culling, noise estimation, and minimum variance map-making made with the ROMAGAL algorithm, described as based on a generalized least squares approach [39]. It is also necessary to make noise estimation.

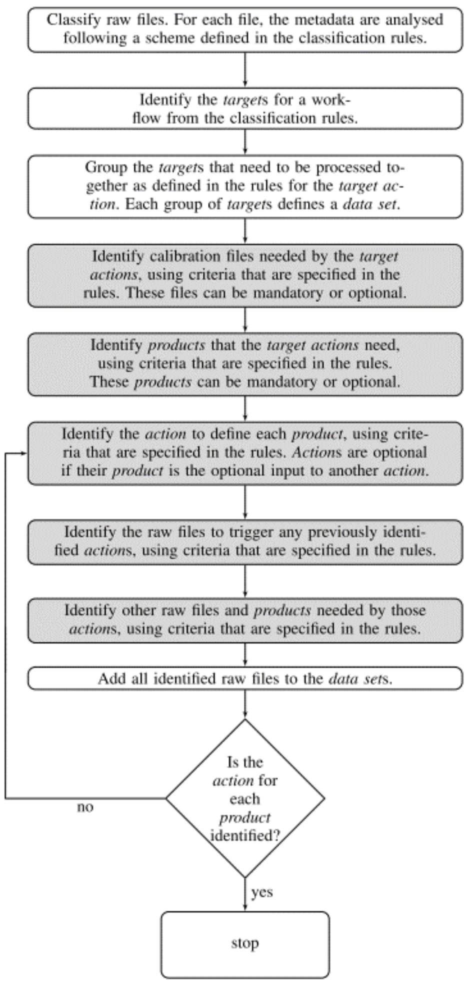

Classify raw files. For each file, the metadata are analysed following a scheme defined in the classification rules.
Identify the *targets* for a workflow from the classification rules.
Group the *targets* that need to be processed together as defined in the rules for the *target action*. Each group of *targets* defines a *data set*.
Identify calibration files needed by the *target actions*, using criteria that are specified in the rules. These files can be mandatory or optional.
Identify *products* that the *target actions* need, using criteria that are specified in the rules. These *products* can be mandatory or optional.
Identify the *action* to define each *product*, using criteria that are specified in the rules. *Actions* are optional if their *product* is the optional input to another *action*.
Identify the raw files to trigger any previously identified *actions*, using criteria that are specified in the rules.
Identify other raw files and *products* needed by those *actions*, using criteria that are specified in the rules.
Add all identified raw files to the *data sets*.
Is the *action* for each *product* identified?
no
yes
stop

Figure 2.10: High-level flow chart of a data organiser. From [30].

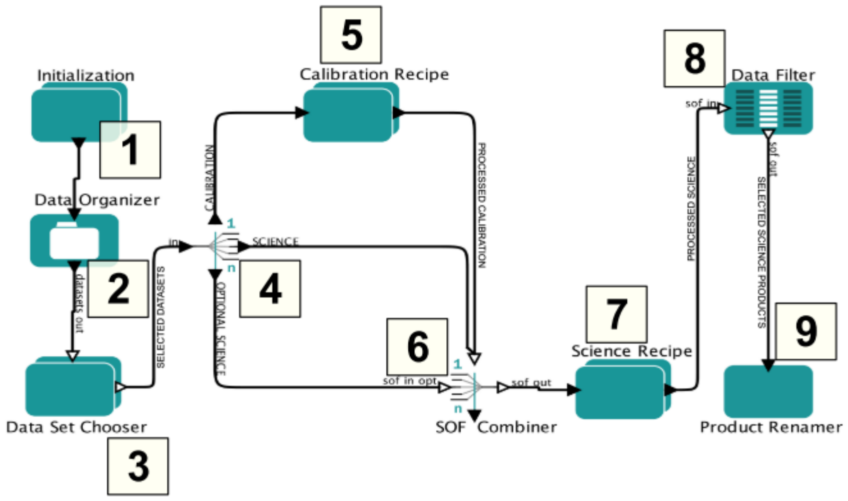


Figure 2.11: Example of a basic Reflex workflow. From [30].

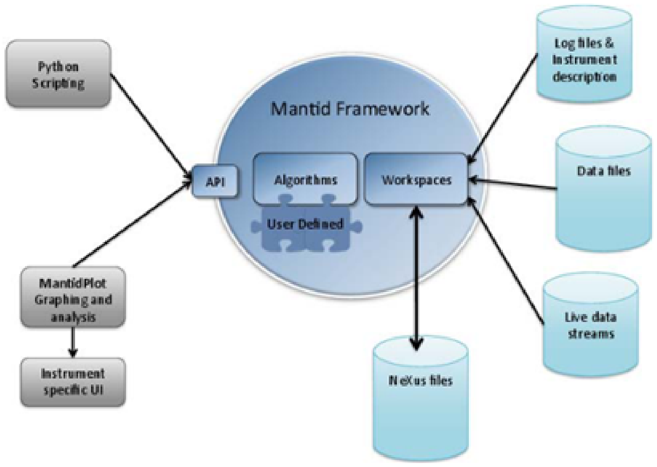


Figure 2.12: Overview of the central role of the Mantid framework in data reduction workflow. From [36].

This paper also provides some concept with acquisition strategy for the input data. The processing is carried out through a series of IDL and JYTHON tools. Preprocessing includes the identification of time-ordered data (TOD), drift removal, and deglitching. The figure 2.13 show the data reduction pipeline proposed.

At last [38] presents the infrastructure available to perform a data-reduction pipeline. As this paper doesn't show a pipeline, the main points to retain are linked with data processing, and the application to perform data analysis and processing. Since data come in different formats, a *CONVERT* class is used to convert all those formats into a format that can be understood by the algorithm.

In [30] some good interesting knowledge has been taken such as an interesting data reduction pipeline model (Reflex) as well as a workflow engine (Keppler). They also provide some useful information on data organization that is a core point of data reduction processes.

Paper [34] has some noticeable information on a possible framework to be explored for pipeline definition. Also the "NeXus" schema for data storing is a tool worth of reference.

From [37] another data reduction pipeline was studied. A map-making algorithm was also learned as well as some tools for data processing. A couple of stages for data preprocessing was also included.

Lastly [38] has some good tips on data processing, and the framework used for this purpose.

2.2.3.3 Parameters estimation

As far as parameter estimation pipelines are concerned three papers were studied. The first one being [40]. This paper uses captured experimental data as a reference to implement a parameter estimation algorithm. The algorithm used is an Enhanced Monte Carlo Algorithm (EMCA). After the several parameters to estimate are set, and the equations established, the parameters are estimated with the algorithm.

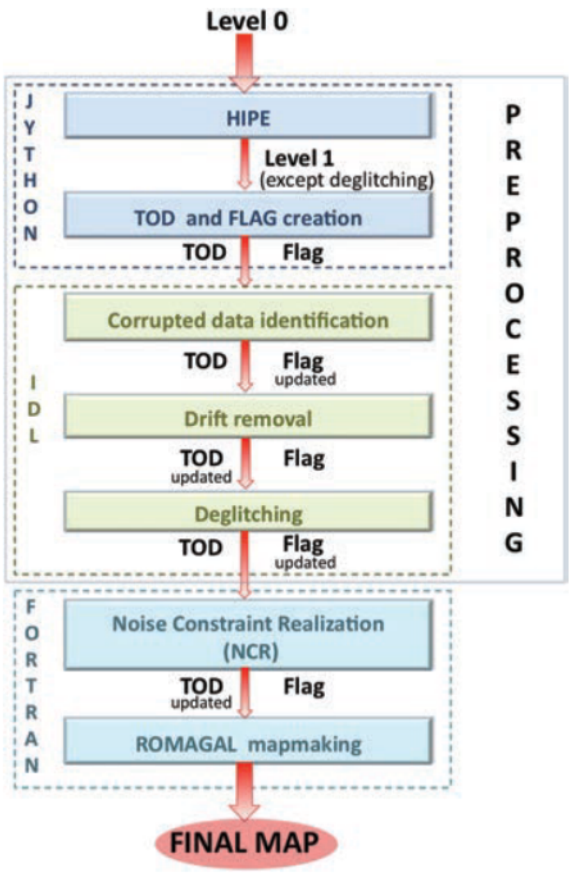


Figure 2.13: Schematic representation of the Hi-GAL data reduction pipeline. From [37].

In [41] it is proposed a workflow used for real-time parameter adaptions. The workflow is shown in figure 2.14 and it's split in 3 different categories: analysis step, restriction of optimization space, and model check. The evaluation metrics for this pipeline is the sensitivity analysis of the parameters. To finish, all the developed work was done in MATLAB[2].

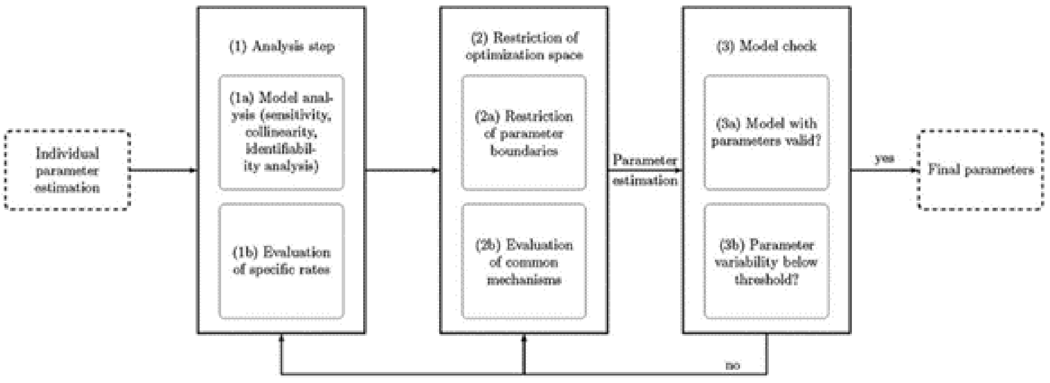

Individual parameter estimation
(1) Analysis step
(1a) Model analysis (sensitivity, collinearity, identifiability analysis)
(1b) Evaluation of specific rates
(2) Restriction of optimization space
(2a) Restriction of parameter boundaries
(2b) Evaluation of common mechanisms
Parameter estimation
(3) Model check
(3a) Model with parameters valid?
(3b) Parameter variability below threshold?
yes
Final parameters
no

Figure 2.14: Schematic representation of an enhanced workflow for model calibration. From [41].

The third paper [42] is a study on an already existent pipeline "BayesWave" [43]. Furthermore the workflow algorithm uses a tri-dimensional Reversible-Jump Markov Chain Monte Carlo (RJMCMC) [44].

In short, the most useful knowledge retrieved from the research on workflows for parameter estimation was the algorithm for parameter estimation EMCA in [40] and RJMCMC in [42, 43]. Moreover, the workflow used in [41] also shown in 2.14 along with the categorization of its different stages, and the already existent BayesWave pipeline in [43], is useful information for the work to be done.

2.2.4 Recommendation systems

The last part of the research done was addressing recommendation systems that would meet the solution for this dissertation. Some workflow recommendation systems were studied and analysed through a total of 6 papers. Those papers will be broken down next.

The first two papers to be mentioned [45, 46] are from the same authors and are related. The first one addresses several important topics about recommendation systems. Firstly it is introduced the concept of Automated Machine L earning (AutoML) as a good resource to help on finding the perfect pipelines for a certain context using deep learning methods. However, this methods can take hours or even days to find the perfect workflow. To prevent this from happening there are a couple of filtering strategies such as OBOE [47]. Figure 2.15 shows how the pipeline is divided in three steps, and each step divided with 3 options, having the best path highlighted in a blue line.

AdaPipe is a recommendation system for Adaptive computation Pipelines in ICPS computation services. The software uses a sparse response matrix where each row and column matches a

[2]https://www.mathworks.com/products/matlab.html

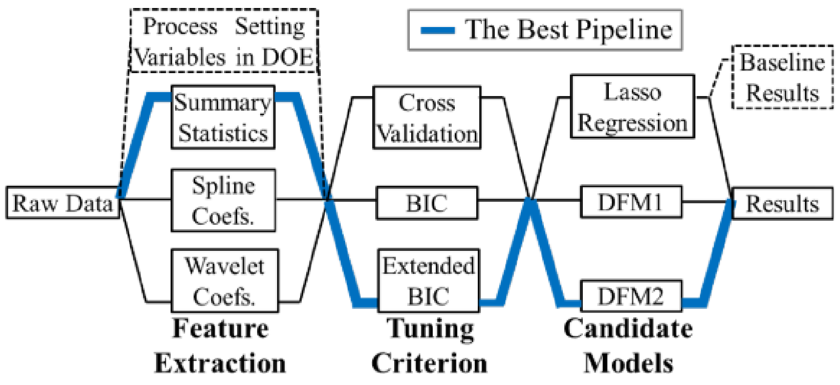


Figure 2.15: Computation pipelines for modeling and prediction. From [45].

dataset with a pipeline. AdaPipe consists of a TEMC model and two covariates generation machines. TEMC model takes the covariates tensor and the sparse response matrix as input to complete the response matrix and provide the ranking and subsequent recommendation of pipelines. An overview of the AdaPipe framework can be seen in figure 2.16. All the steps done by the algorithm to find the benchmark pipeline is clearly displayed on the schematic representation.

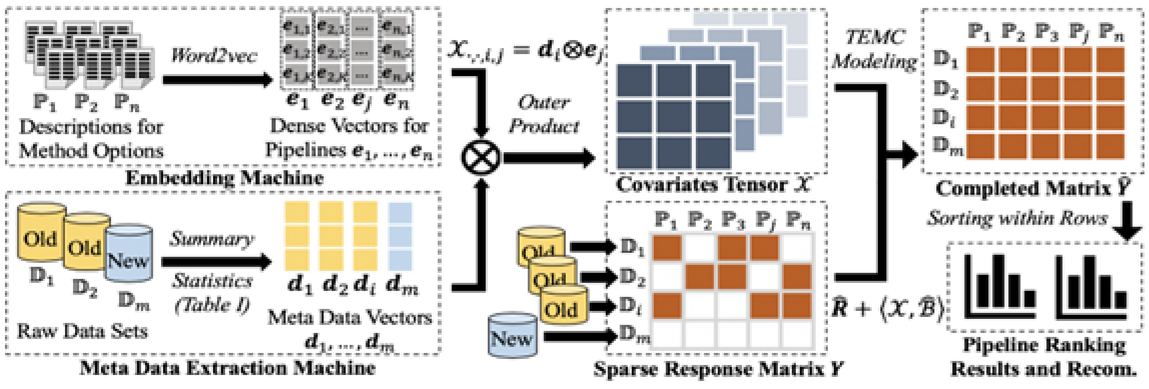

Word2vec
Descriptions for Method Options
Dense Vectors for Pipelines $e_1, ..., e_n$
Embedding Machine
$\mathcal{X}_{:,:,i,j} = d_i \otimes e_j$
Outer Product
Covariates Tensor $\mathcal{X}$
TEMC Modeling
Completed Matrix $\hat{Y}$
Sorting within Rows
Old
New
Raw Data Sets
Summary Statistics (Table I)
Meta Data Vectors $d_1, ..., d_m$
Meta Data Extraction Machine
$\hat{R} + \langle \mathcal{X}, \hat{\mathcal{B}} \rangle$
Sparse Response Matrix Y
Pipeline Ranking Results and Recom.

Figure 2.16: An overview of AdaPipe system. From [45].

Afterwards a case study is done. In order to rank the pipelines, it is proposed to use the minimal number of top-ranked pipelines to reach the best statistical accuracy. One final note on the three main reasons of why AdaPipe had the best ranking and recommendation performances:

1. Pairwise loss function provides the capability to rank the computation pipelines by comparing statistical prediction errors in pairs, but ignoring the alignment between predicted responses and true responses.

2. A low-rank matrix R and covariates X effectively quantify the implicit and explicit similarities, which jointly contributes to accurate ranking and recommendation performance.

3. The embedded dense vector representations are informative to identify the similarities and differences among computation pipelines.

The other related paper [46] also presents a pipeline based on machine learning. Figure 2.17 is very similar to the one in 2.15, though it has some disparities. Once again the best path is represented by the blue line.

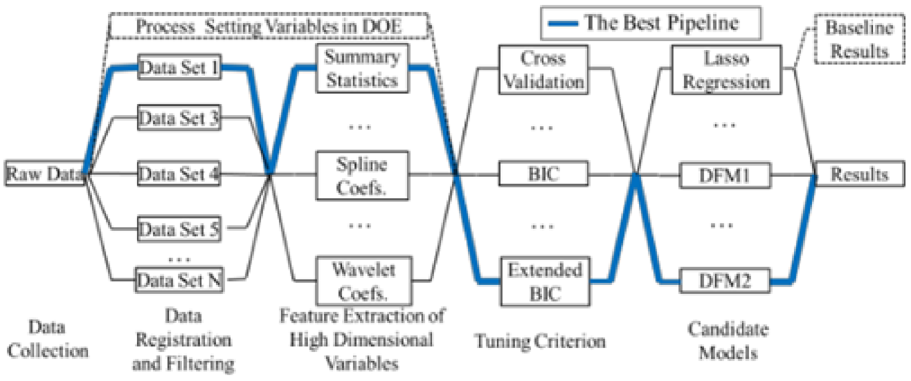
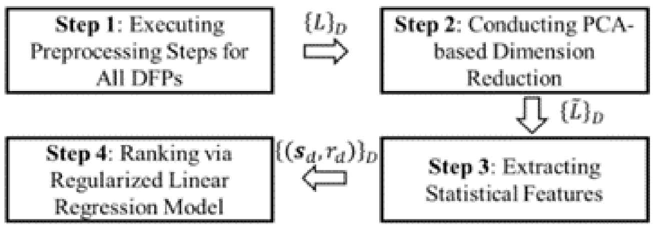


Figure 2.17: Data Fusion Pipelines for Variation Modeling. From [46].

As there are a lot of Data Fusion Pipelines (DFPs) options, it is wanted to get the best one. This should be retrieved from testing all the possible DFP's and then evaluate the performance of each one and then selecting the best. Just as told before, executing all DFP's would require a lot of resources and once again a learning-to-rank method is used. Figure 2.18 and 2.19 show the proposed learning-to-rank method and its implementation, respectively.



Step 1: Executing Preprocessing Steps for All DFPs
$\{L\}_D$
Step 2: Conducting PCA-based Dimension Reduction
$\{\tilde{L}\}_D$
Step 3: Extracting Statistical Features
$\{(s_d, r_d)\}_D$
Step 4: Ranking via Regularized Linear Regression Model

Figure 2.18: Data Fusion Pipelines for Variation Modeling. From [46].

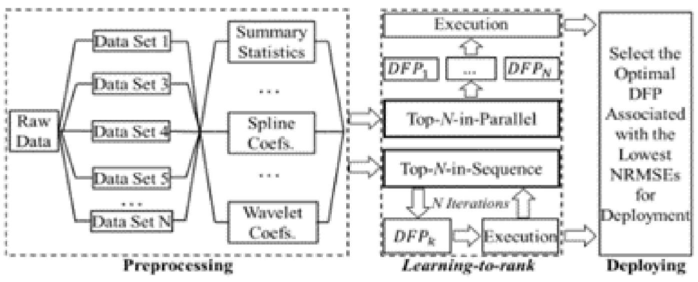


Figure 2.19: Data Fusion Pipelines for Variation Modeling. From [46].

The third paper found regarding this topic was [48]. This is a small paper that has some interesting references in the background and related work section. They introduce several recommender systems such as Linton and colleagues, Matejka and colleagues, and Murphy-Hill and colleagues. The paper explains that these systems focus on recommending single tools, which can be really useful for some contexts. However, when implementing with no context, this tools may not constitute a complete task. Afterwards in the paper a Top-K sequential pattern mining algorithm is used to perform workflow recommendations.

The next paper [52] proposes a workflow model recommendation approach based on a design information model for product design process recommendation. A Workflow Model Recommendation (WfMR) is divided in 4 stages: design document, design information, design process, and recommendation algorithm. In order to have a computer able to process the 4 stages above mentioned, 3 models are built: design information ontology model, workflow model, mathematical model for the WfMR algorithm. In figures 2.20 and 2.21, the framework used for the proposed approach, and the respective process are shown, respectively.

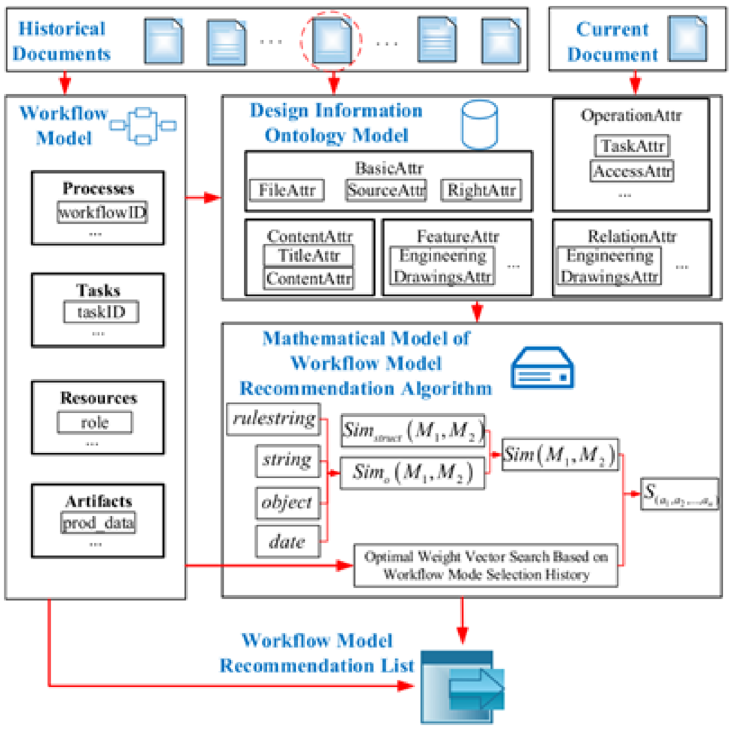


Figure 2.20: WfMR approach framework. From [52].

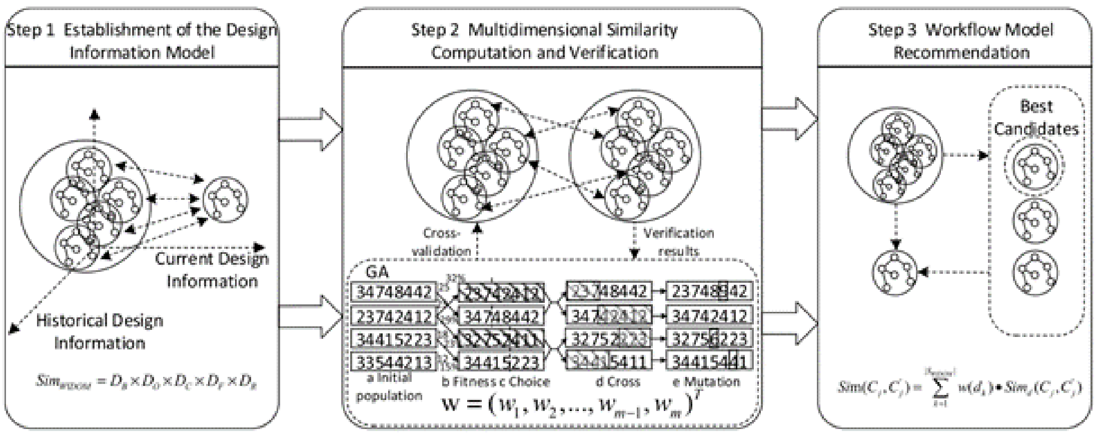

Step 1 Establishment of the Design Information Model
Current Design Information
Historical Design Information
$Sim_{WDOM} = D_B \times D_O \times D_C \times D_F \times D_R$
Step 2 Multidimensional Similarity Computation and Verification
Cross-validation
Verification results
GA
a Initial population
b Fitness c Choice
d Cross
e Mutation
$W = (w_1, w_2, \ldots, w_{m-1}, w_m)^T$
Step 3 Workflow Model Recommendation
Best Candidates

Figure 2.21: WfMR process. From [52].

It is used a genetic algorithm to distribute the weights between the nodes. The evaluation metrics proposed are Recall and Precision as previously mentioned in Predictive maintenance section with equations 2.1 and 2.2.

Next paper [53] proposes a hybrid framework that recommends workflows. This framework is shown in figure 2.22. It is split in different stages and each stage is explained in the paper.

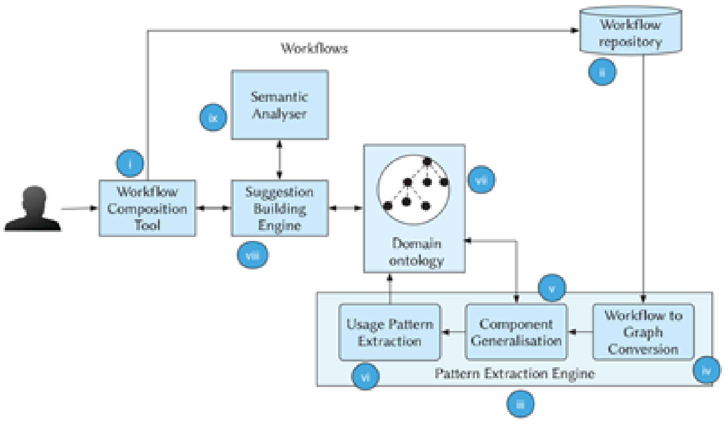


Figure 2.22: Hybrid framework architecture. From [53].

The generation algorithm is likewise suggested in the paper as shown:

1. Workflow is converted to a graph.
2. Components in the workflow are generalized.
3. The repository is searched for patterns that overlap with the partial workflow.
4. Once matching patterns are found, they are specialized.
5. The partial workflow is then semantically analyzed and enriched.
6. Compatible components are retrieved from the repository based on the enriched workflow semantics.

7. The results are sorted and returned.

This recommendation system was done to do a more dynamic workflow constructor which is not quiet what is desired for our work.

The last paper [54] refers to a recommender system that does not recommend workflows but instead recommends options for a user based on workflows. Although this system is not exactly what is intended to develop, the way the algorithm works can be used in future work. In addition to that, figure 2.23 represents some good models for data processing that might be used in the framework to be developed.

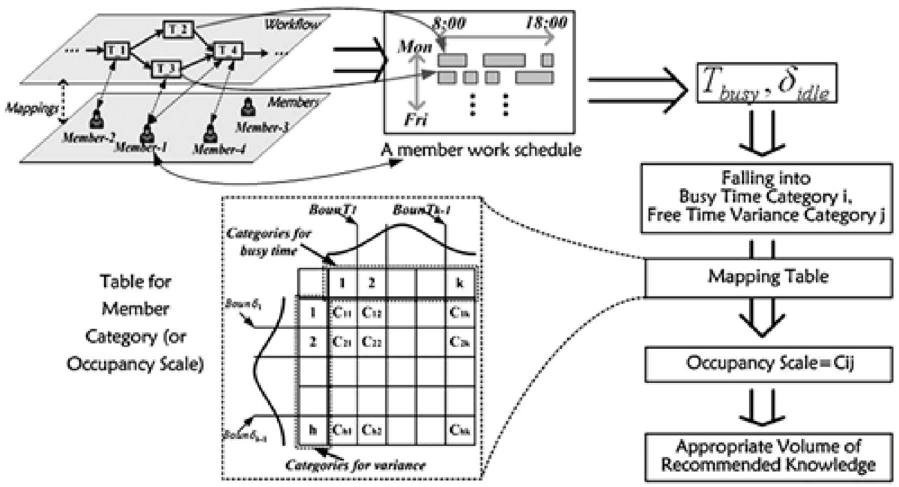


Figure 2.23: Statistical analysis of occupancy scale from schedules in workflow. From [54].

In both [45, 46] a very similar approach is used. Though this two papers base their work methodology in machine learning and that is not quite the path wanted to be taken for the future work, some really useful information can be taken. The way they think about how to get to a solution, the way they have divided the model in different stages and their options, mainly the learn-to-rank method, and the evaluation metrics proposed can be very useful for this dissertation.

The paper [48] has an interesting approach on the lack of available systems that can recommend complex and complete workflows in whatever context they are needed. Besides not only give a few examples of tools that fit on that description but also implement a simple algorithm to overcome those same difficulties.

In paper [52] a few important concepts are addressed, such as the parts a WfMR involves and respective models of implementation. Though it is a very specific application, the presented genetic algorithm as well as the evaluation metrics may become really useful in the future.

In [53] some visualization details may be useful for contextualization purposes, though it does not quite fit into what the research focus on.

Lastly and as said above, [54] does not recommend workflows, but uses a recommendation framework that follows a couple of processes that may become useful.

Chapter 3

Implementation

For the purpose of meeting the all the goals described in Objectives as well as give an answer to the Problem Definition, a recommending system for workflows was developed. This system works based on a set of user-defined requirements. These requirements are split into two different categories, as explained back in Motivation, specific requirements and performance requirements. In short, the user defines which specific requirements are desired, and chooses a ratio value that represents the priority given between two performance requirements. Having all that information, the system recommends the three best workflows, ranked.

The first major step onto the implementation of the project is the choice of the programming language. Some research was made with Python being the most highlighted option and eventually chosen due to some of its advantages as for meeting requirements for the project such as:

- It is an object-oriented language.
- Easy to use and learn.
- Tremendous amount of good and useful libraries to use for the purpose, such as NumPy[1], Scikit-Learn[2] and CSV[3].
- Good support community.

The implementation of the code was done in Visual Studio Code[4]. This choice was based on some points of advantage compared to other IDE options. Being light weight, having a robust architecture, freeware, helpful extensions, good connection with repositories like Git Hub are some of them as well as some previous working experience with the platform.

[1] `https://numpy.org/`
[2] `https://scikit-learn.org/stable/`
[3] `https://docs.python.org/3/library/csv.html`
[4] `https://code.visualstudio.com/`

3.1 Workflow Recommendation System

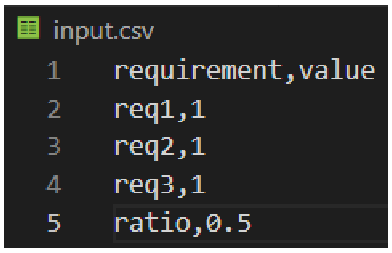
3.1.1 Input files

A set of requirements are defined by the user on a csv file named *input.csv*. Those requirements are subdivided into different categories. The categories were chosen according with what type of data process was associated to each one just as studied very carefully in section 2.2. This input file will be the starting point of the main algorithm.

The file has 2 columns, the first one being the name of the requirement and the second one the respective value. Every line is a different requirement. The associated values differ for the different requirements.

Every line but the last one of the csv file represents all the specific requirements. They will have associated a number 0 or number 1. Number 0 means the final workflow is not supposed to meet that specific requirement, this being not include a solution for that category while a number 1 means that requirement has to be taken in account when designing the workflow. So it works as a binary choice because every category will necessarily have to be either selected or not.

The very last line of the csv input file has the value for the two performance requirements chosen by the user. A value between 0 and 1 that represents the ratio value of preference between one of the requirements over the other one. A snippet of an example of the csv file and how it should be structured is shown in figure 3.1.



```
input.csv
requirement,value
req1,1
req2,1
req3,1
ratio,0.5
```

Figure 3.1: Requirements CSV snippet.

In order to help illustrating how this works, for the explaining example we use performance and time efficiency as the two performance requirements to classify a workflow. Performance and efficiency are two of the most used metrics to evaluate workflows [30, 37, 55, 8, 56, 57, 58]. **Performance** is used very commonly and provides good information about the pipeline. Efficiency may be a subjective concept since it can refer to a lot of variables such as cost, time, resources, and much more. In order to keep it simple for the explanation, we used **Time Efficiency**.

So this last value asks for a better performance for the workflow (when closer to 0) or a faster workflow (when closer to 1). This way, it is given to the user the ability to be more specific if he wants to either give priority to a workflow that is faster to be completed or to a workflow that has a better performance on the final results.

A very similar problem approach was used in [10] as they analyze each workflow on their performance and cost and also use a performance/price ratio as input.

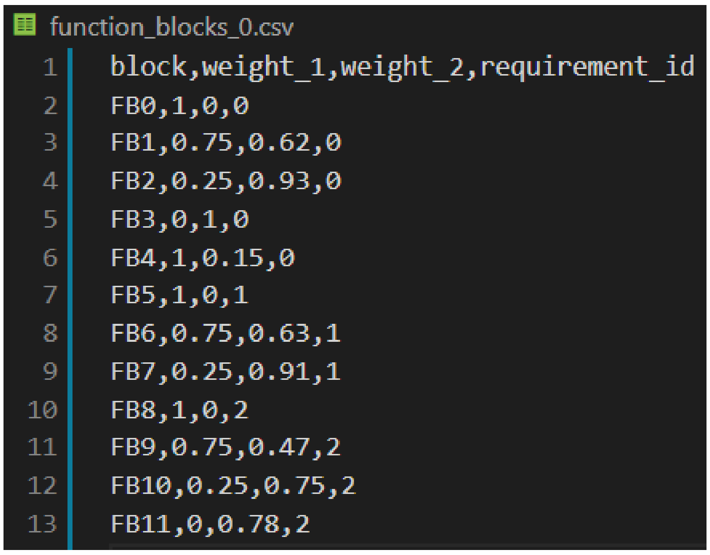
Another important file for the algorithm to work is the function blocks database csv file. This file contains all the function blocks as well as their information that will later be used in order to calculate the best workflow.

This csv is composed by 4 columns. Each line represents a function block and each one will have information on its name, time efficiency and performance values, and in which requirement it fits. Each one of this properties goes on a column.

They key point here are the values for the time efficiency and performance for each option. The main idea behind this is that each function block has a set of two weight values, representing the time efficiency and performance, that profiles itself. The values are between 0 and 1 and they represent how good a function block is in terms of time efficiency and performance. As opposed to the Time efficiency/Performance values in the requirements file, these are not mutually exclusive, this meaning a block can be both good on performance and time efficiency. Another important note to add on this is that the values are relative to the values of the other function blocks on the database for a specific category. For instance, if a block has the number 1 in performance it means it is probably the best one in terms of performance compared to every other block of that category that was taken in account, though it may not be the best one there is as some options may not be in the function blocks database. The theory behind these values that characterize each block is better explained in section 3.2, where it is thoroughly explained the genetic algorithm used to estimate these values. In some real-life situations these values may be relative between all the blocks of all the categories as we will see in section 4.1.

The last value is the *requirement_id* value. The whole code is programmed so each requirement category has an associated index number. This number is according with the line number on the requirements *input.csv* file explained above.

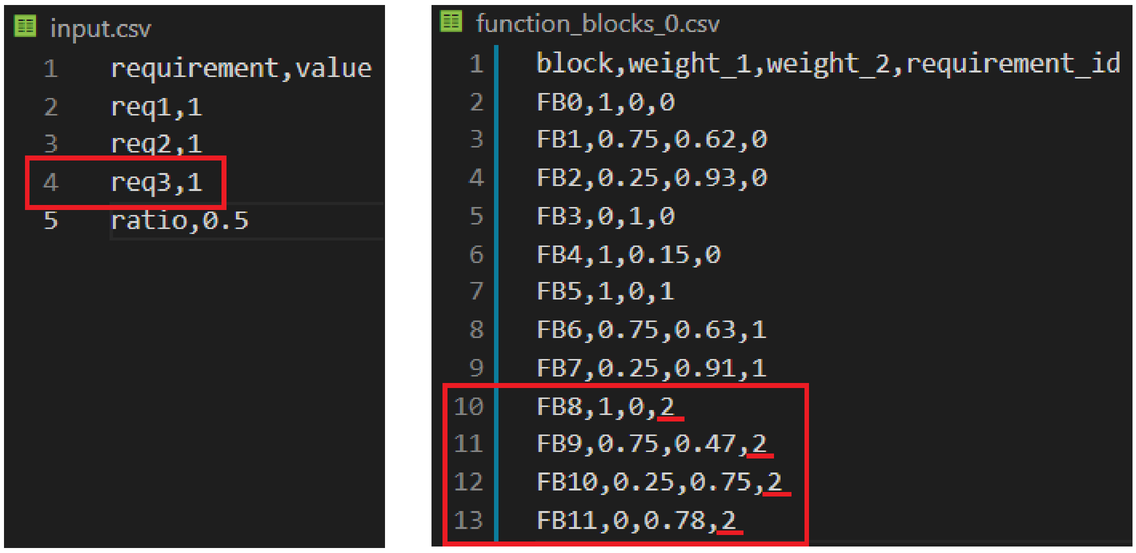
A suggestion of how the csv file for the function blocks database looks like for the example presented in figure 3.4, is shown in figure 3.2.


function_blocks_0.csv

```
block,weight_1,weight_2,requirement_id
FB0,1,0,0
FB1,0.75,0.62,0
FB2,0.25,0.93,0
FB3,0,1,0
FB4,1,0.15,0
FB5,1,0,1
FB6,0.75,0.63,1
FB7,0.25,0.91,1
FB8,1,0,2
FB9,0.75,0.47,2
FB10,0.25,0.75,2
FB11,0,0.78,2
```

Figure 3.2: Function blocks database CSV snippet.

There is an important dependence between both the files mentioned above. The requirements input file follows an order which is connected to the requirement value of each function block in the function block database. For instance if a specific requirement, for instance *req3*, is the third requirement on the input file, then every function block associated with *req3* in the function blocks database should have the number 2 as its *requirement_id* value as illustrated in figure 3.3. The function blocks *FB8, FB9, FB10, FB11* all belong to the requirement *req3* as it is the third on the requirements file. This guarantees that when the framework reads the requirements file, it knows exactly what function blocks he should be looking for.



```
requirement,value
req1,1
req2,1
req3,1
ratio,0.5
```

```
block,weight_1,weight_2,requirement_id
FB0,1,0,0
FB1,0.75,0.62,0
FB2,0.25,0.93,0
FB3,0,1,0
FB4,1,0.15,0
FB5,1,0,1
FB6,0.75,0.63,1
FB7,0.25,0.91,1
FB8,1,0,2
FB9,0.75,0.47,2
FB10,0.25,0.75,2
FB11,0,0.78,2
```

Figure 3.3: Relation of dependence between the requirements input file and the function blocks database file.

To sum up everything regarding these two files, the framework is scalable on the both requirements categories and function blocks. In order for a user to add categories or/and function blocks to a category, some updates on these files must be done and every step is explained in section 3.3 where it is detailed how both algorithms of weight optimization and workflow recommendation can work together.

The last additional input file is an optional file called by default *workflow.csv*. This csv file contains a workflow suggested by the user in order to receive some recommended changes to make on it in order to be more suitable with the user's performance requirements. The csv is composed by a column and each row has the name of the function block following the order of the original input requirement files. Two notes on this are that the blocks suggested in this file must be in the function blocks database, and contain function blocks only from desired categories selected in the requirements file.

3.1.2 Graph theory

In order for the program to represent all the blocks and the connection between them it is used graph theory. The choice behind graph theory comes after some research on graphs and how well it suits the context, being an appropriate approach for ranking/ordering problems. Graph theory has been used quite often with similar problems [53, 30, 59, 60, 61].

Graphs can be either directed or undirected. Directed graphs are the interesting ones here since the graph is supposed to show a sequence of function blocks as a workflow. Apart from being directed, the graph is also acyclic, whereas it is divided by categories it must be very well structured, for instance category 2 blocks can only be followed by blocks of category 3. So as in [10], workflows are described in Directed Acyclic Graphs (DAG). Another property of the graph is that its blocks will have a score associated since we want a path to have a fitness value and then the main problem becomes a route optimization situation. This fitness represents the sum of the scoring value that a set of function blocks would get by being chosen in a given scenario for a specific value for time efficiency/performance ratio as input.

Having an idea of how the graph is supposed to work, it is necessary to understand the best way to represent a graph in the working programming language, Python.

The main strategy is to have a graph implemented that is composed by vertices, each representing a function block, and edges representing the connection between function blocks of different categories. So the rule for the graph construction is about a vertex of a specific category can only connect with a vertex of other category following a specific category order (defined on the requirements file mentioned in section 3.1.1) and never between two vertices of the same category nor between two vertices of non-consecutive categories, following this way the rules of a DAG and never choosing more than one vertex from the same requirement category.

Usually graphs are implemented using either adjacency lists or adjacency matrices. The most common rule to decide which one to use is based on the number of edges the graph has. Given a graph $G = (E,V)$ composed by its edges E and its vertices V, if $\#E = \#V$, then we have a sparse graph. Else if $\#E = \#V^2$ the it's a dense graph. If a graph is more sparse then an adjacency list should be used, otherwise in case of a more dense graph, an adjacency matrix should be the option.

This makes total sense since a list representation would be more compressed and a bit harder to access and code, though it would only use the minimum necessary amount of resources while using a matrix would use the maximum resources. So having a graph with 2 edges only or a graph with 100 edges would use the same resources for the same number of vertices, being only advantageous for dense graphs with a lot of edges.

Estimating our number of edges with a realistic example, let's say we have two categories of requirements, the first one with 5 blocks and the second one with 10. So the total number of vertices is 15 and 225 is its square value. Since vertices can only be connected to blocks of the following category, then each of the 5 first blocks would have 10 edges to the 10 blocks of the second family of requirements. Adding this up, a total of 50 edges would be part of the graph. In short, if we use an adjacency matrix, the resources consumption would be the same as if we had the maximum of 225 edges. Using an adjacency list would take only the necessary memory so it justifies in our case to use a list since we will never have an approximate number of edges to the maximum possible according with graph theory.

Having decided the representation method, two classes were created, one for the graph called *Graph*, and one for each vertex called *Block*. The *Block* class has seven attributes with five of them being information about the block plus a list of neighbors of each vertex corresponding to the vertices it is connected to. The other two (*value* and *precedent*) are explained later in section 3.1.4 and they are used for scoring and path choosing purposes. Figure 3.5 shows how every function block is composed. The graph class contains a dictionary where all the blocks and corresponding edges will be saved as keys and values respectively as in figure 3.4

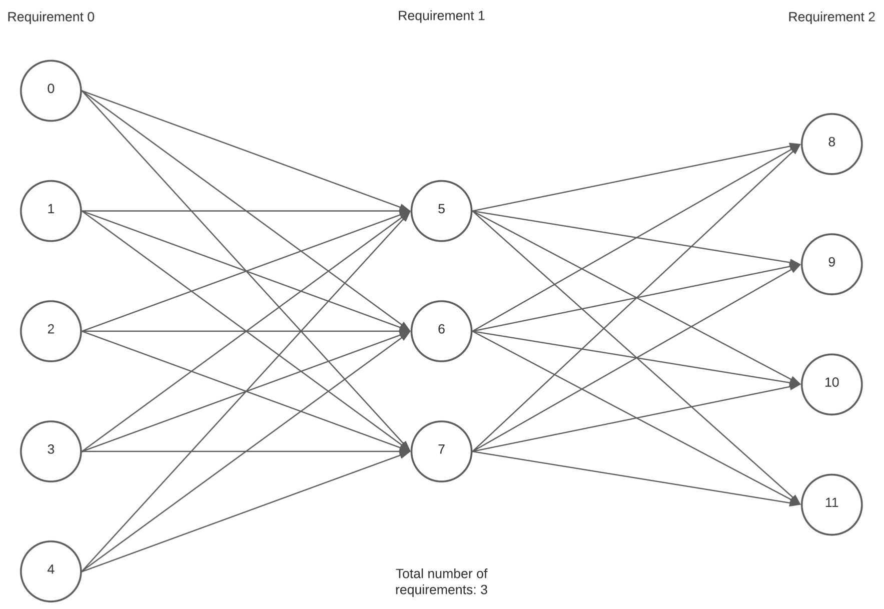


Figure 3.4: Graph model example

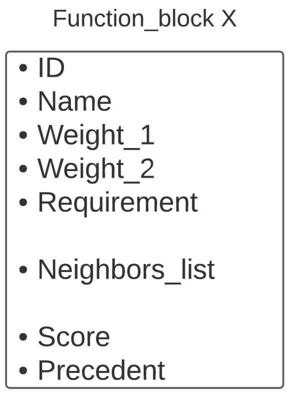


Figure 3.5: Function block model example

Knowing that DAG’s have been used previously to represent workflows [10], the problem in hands is to connect different blocks, each one with a value associated, its score, and build paths in the graph. Each path has a value which is the sum of every block score that is in the path. Several combinations of paths can be achieved, this number depending surely in the number of blocks and categories, the problem then became how to find the longest path in the graph, and after that finding the second longest and also the third longest. Finding longest paths in a DAG is a common problem and several methods are available through dynamic programming [62].

Usually longest path algorithms follow an order to process each vertex and calculate the best solution for each one starting from the end and moving backwards. This order is called the topological order. Topological order is obtained by doing the topological sorting of a DAG and only works in DAGs. Topological sort algorithm is a modification of the Depth First Search algorithm (DFS). The DFS loops over the graph starting by printing the first vertex, and then looping over its adjacent vertices, printing each one, and then looping over the adjacent vertices of each of those vertices and so on keeping doing it recursively, never doing this process more than once to the same vertex while the topological sort before printing a vertex calls recursively every of its adjacent vertex until it reaches the end when a vertex has no more adjacent vertices. So a vertex is never pushed into the order stack until all of its adjacent vertices are already in the stack. The stack is then reversed.

To illustrate this better an example of a simpler graph is shown in figure 3.6. The topological order for this graph would be $[1,0,3,2,4]$.

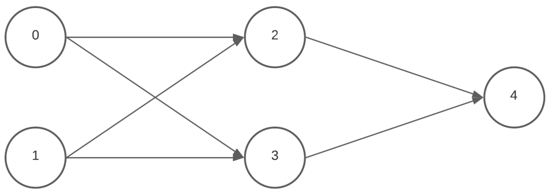


Figure 3.6: Simples DAG example

Following the longest path strategy, once we have the topological order the longest path algorithm follows this same order to check on every possible path, updating the values of the distance to each of the vertices everytime a higher value is reached. Using this strategy, it is avoided the overlapping in many repeating sub-paths since only the best ones are kept.

The algorithm is better explained in the section 3.1.4.

3.1.3 Solution design

The working and coding philosophy towards solving the proposed topics was to get a fully working program meeting the most important requirements as soon as possible even if lacking optimization. So the idea was to return the best workflow given certain function blocks of certain specific requirements with certain weights concerning its performance and time efficiency. Thereby, before the final optimal solution was reached, in section 3.1.4, several tweaks were made to a worse optimized version during the coding process.

Although the first raw version of the recommendation algorithm could do the essential of what it is expected to do, going from reading requirements from the user, and using a function block database, to build an optimal workflow that meet those requirements and return it to the user, it had several points missing optimization. First things first, the program would get all the function blocks from the database and build the graph based on all the function blocks available, thus not having in account the fact that some of them wouldn't be a possible solution due to the fact that the requirements by the user were read later. This way, some non-admissible solutions could be on the line and had to be taken care of later. This required more resources since the graph was always the most complex possible having extra vertices and edges that would never be needed.

Another bottleneck point in the first version of the algorithm was the fact that the several function blocks on the csv file for the function block database had to be in order of requirements index number. This made it harder to add new blocks to the document since they had to be put in a very specific order.

The second version of the code had this two points fixed not adding any feature to the final solution. This way, the code first read the requirements and based on them, would search the blocks database for the ones that would fit, and use them to build the graph. Apart from that feature, the reading of this same database was done differently, then being independent of the order the blocks were on the csv file, therefore making it way easier for the user to add new blocks. In short, the new version brought optimization in resource consumption, since the graph was filled with usable function blocks only, and taking all the non-admissible solutions out of the way, while making it easier for a user to add new information to the database.

The way the best workflows were calculated in the previous versions of the code also changed to the last version. Firstly, after having the graph all filled up with the vertices and corresponding edges, the scoring process was done. Note that a workflow class was used having two attributes, the workflow list and the total score of the workflow, in order to keep things cleaner. The score was then calculated bye doing the dot product of each block weight values and the input ratio value for the performance requirements. It checked each requirement at a time in the graph and for each block it is calculated the score and chosen the highest-scored one. Doing this for every requirement makes it select the best one for each requirement. This would surely return the top pipeline, however it would be impossible to return more than one workflow in an optimized way. Apart from that, we were not fully taking advantage of graph theory to grab the best paths, and so in the next sub-section the approach taken to solve this is carefully explained.

3.1.4 System breakdown

The algorithm starts by reading the values in the input file and translate them into a matrix of one line and N columns, being N the number of specific requirements in the file. These values are carefully explained in section 3.1.1. This way the algorithm will be able to know which function blocks are supposed to be used and calculate the workflow.

Having the requirements matrix, the next step is to import and create a list of all the function blocks from the database that are within categories of requirements selected by the user on the input file. This guarantees that only desired function blocks for one specific problem are being imported into the program ensuring an important resource optimization. Every optimization step and version of the code is deeply explained in the previous section 3.1.3. So the function starts by checking each specific requirement, its value and if the workflow is supposed to meet that specific requirement, then it checks the available function blocks that pertain to that requirement and appends them to the list. In the end of the importing phase, the function block list will have the blocks ordered by requirement index. This is an important note for how the graph will be designed, but also the database doesn't need to follow a necessary order for the importing to work properly since it only relies on the requirement property of each block from the database, even though it orders it correctly while importing them. The value for the time efficiency/performance in the requirement matrix isn't yet being evaluated.

The next step is to translate the just created function block list into a graph, so a graph is created following the class structure. The graph construction goes through two phases, creating all the vertices and then filling it with every corresponding edge connections. The first phase consists on reading the function block list imported from the database from the previous stage and create for each one a block from the class *Block*, explained in section 3.1.2, and then add it to the graph by using a class method called *add_block*. This ensures that every block, fulfilled with its information, is added as a vertex to the graph. The second phase comes after having the graph will all the vertices. The function will go through every vertex, check its requirement attribute, then loop over all the blocks from the graph that belong to the next requirement ID and add an edge between those two vertices. So this way every block that belongs to the requirement category 1 will connect to every block that has requirement 2 and so on, following therefore the structure presented in figure 3.4.

Before heading to the workflow generator part, the scores are calculated a priori, using a function called *scoring*. This function takes two arguments, the graph and the ratio value of the two performance requirements defined by the user in the input file. So it gets this input value and loops over all the vertices in the graph, calculating the score for each one resorting to the inner graph method *calculate_scores*. The value of each vertex is stored in the previously talked attribute of each block, *value* in section 3.1.2.

The final and most important step from our program is the workflow generator. This is firmly the core part of the code. The first note is that a class named *Workflow* is used. The first attribute is a list containing all the blocks chosen for the final workflow. The other one is the workflow score. Using the class made it easier to implement the feature of returning more than one workflow. In the solution design section 3.1.3 it was addressed the need of optimizing the code in order to return the top 3 workflows, since in previous versions the code would return the best workflow only, while not taking full advantage of what graph theory can offer.

As previously explained in the last part of section 3.1.2, topological sort followed by longest path approaches were used. Just like in [62], we first find the topological order and then use that same order to calculate the longest path. A default topological sort algorithm[5] is used as a method of the graph class. This algorithm is shown in Algorithm 1.

[5]https://www.geeksforgeeks.org/topological-sorting/

Algorithm 1 Topological sort algorithm

```
1:  $visited = [False] * num\_vertices$                               ▷ Creates a list of visited vertices
2:  $stack = [empty]$                                                 ▷ Creates an empty stack
3:  for $vertex$ in $num\_vertices$ do                                ▷ Loop over all the vertices in the graph
4:      if $visited[vertex] = False$ then                             ▷ Check if this vertex has been visited before
5:          procedure TOPOLOGICAL SORT UTIL($vertex, visited, stack$)  ▷ Recursive function
6:              $visited[vertex] = True$                              ▷ Vertex marked as visited
7:              for $a$ in $adj[v]$ do                                ▷ Checking vertex adjacency list
8:                  if $visited[a] = False$ then
9:                      $TopologicalSortUtil()$                       ▷ Call the recursive function
10:             $stack.append(v)$        ▷ Once no more adjacency vertices, it is added to the stack
11: $stack[:: -1]$                                                    ▷ Order of the stack is inverted
```

Having the stack ordered, it is time to run the longest_path function, using once again a standard algorithm[6] with a few changes to meet our graph design. The algorithm used is described in Algorithm 2. It starts by creating a list of distances for each block, initializing it with each block score. Then following the topological order it checks for every block the distances of each of its adjacent vertices. If the distance is lower than the sum of distance of the first block and the score of its adjacent, then the adjacent block's distance is updated to this value, and its precedent attribute is update with the first block. This is where the last attribute of class *Block* comes in play. Precedent will then be used to move backwards on the graph to get the longest path.

Doing this for every block in the stack will have us a distance list for each block representing the maximum score possible if that block is chosen as the last one of the pipeline. This way, we check on the distances list which of the blocks from the last requirement category on the graph has the highest value. This block is the last block of our longest path and its distance is our pipeline score, and so it is added to the workflow list. Then a while loop is used to go to the precedent block each iteration, and append them accordingly to the workflow list. Once we reach the condition of no precedent block, it means the first block of the pipeline was reached and the workflow is completed. The list is then inverted to the correct order.

Now that the best workflow is calculated, the second and third longest paths are to be calculated. Unfortunately, the longest path algorithm approach is not ready to calculate, for instance, the three longest paths without "physical" changing the graph. The definition of second longest path in a DAG is the longest path in the graph that doesn't share at least one vertex with the longest path. Following this thinking, the third longest path is the longest path that doesn't share more than one vertex with the longest path and the second longest path simultaneously.

In order to achieve this, the longest path function was run several times, each iteration a different block from the best workflow was "removed" from the graph and calculated the longest path of that graph. "Removed" from the graph doesn't mean actually removed from the graph but since the block class has the attribute of its score, this value was changed to large negative number that would make it never being picked for the longest path. After calculating it, the score value

[6] https://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/

Algorithm 2 Topological sort algorithm

```
1:  $dist[0] * num\_vertices$                          ▷ List of distances for each vertex
2:  for vertex in $graph$ do
3:      $dist[vertex] = vertex.value$                 ▷ Updates each distance with the vertex value
4:  for $i$ in $stack$ do                              ▷ Loops over the stack
5:      $u = stack[i]$                                ▷ u is the vertex in the position i of the stack
6:      for $v$ in $adj[u]$ do
7:          if $dist[v] <= dist[u] + v.value$ then    ▷ Compares the current dist of v with v+u
8:              $dist[v] = dist[u] + v.value$         ▷ Updates v dist if reaches a new maximum
9:              $v.precedent = u$                     ▷ Updates precedent attribute of v with the best option
10: Check what vertex of the last requirement category in the graph has the largest distance.
11: Runs from the end to the beginning following the precedent of every block
```

is re-added to the block and the loop moves to "remove" next block. Every iteration, the score is compared to the maximum achieved till then and everytime a new high score is set the second best pipeline is updated to the current longest. At the end of the loop, the graph is back to the original and the second best workflow is saved.

The same technique is used to calculate the third best option as well, but instead of "removing" one vertex at a time it's removed two, one from each of the two best workflows, never removing the same block twice in case they share at least one.

This method of ranking workflows by score has some steps to avoid bug situations when for instance a requirement only has one block, and that block has to be chosen forcibly and so is never removed from the graph. The same happens if a category only has two blocks and both were chosen in the best and second best workflow guaranteeing this way that every specific requirement is always met in the final workflows.

Another feature of the recommendation system is the option for the user to use a workflow as an input and ask for changes in order to make it more suitable to its needs. The user is asked at the beginning of the program if he wants to submit a workflow in order to receive suggestions to change it. If answered positively, then the program compares both the submitted pipeline by the user and the best workflow calculated for the current performance requirements, and suggests how blocks should change. If the submitted workflow is the benchmark workflow, then no changes are suggested.

The recommendation system also features the creation of automatic datasets to be used in the genetic algorithm as input, later explained in section 3.2.1.

Final note that all the values (ratio, weights) must have exactly two decimal digits in order to avoid ambiguous situations, later explained in chapter 4.

3.1.5 Architecture

The architecture of the recommendation system is shown in figure 3.7. It decomposes the system in the several phases of the code, going from the user and its information files, to the information processing stage where the requirements are read, used to create the function blocks list from the database, also reading the workflow submitted by the user, if desired, then proceeding to the graph building phase, where vertices are created, added to the graph, and then connected between themselves. The scores are then calculated for each block. The workflow generator stage starts by calculating the top 3 pipelines, then creating datasets for results analysis. Lastly, recommendations are sent to the user.

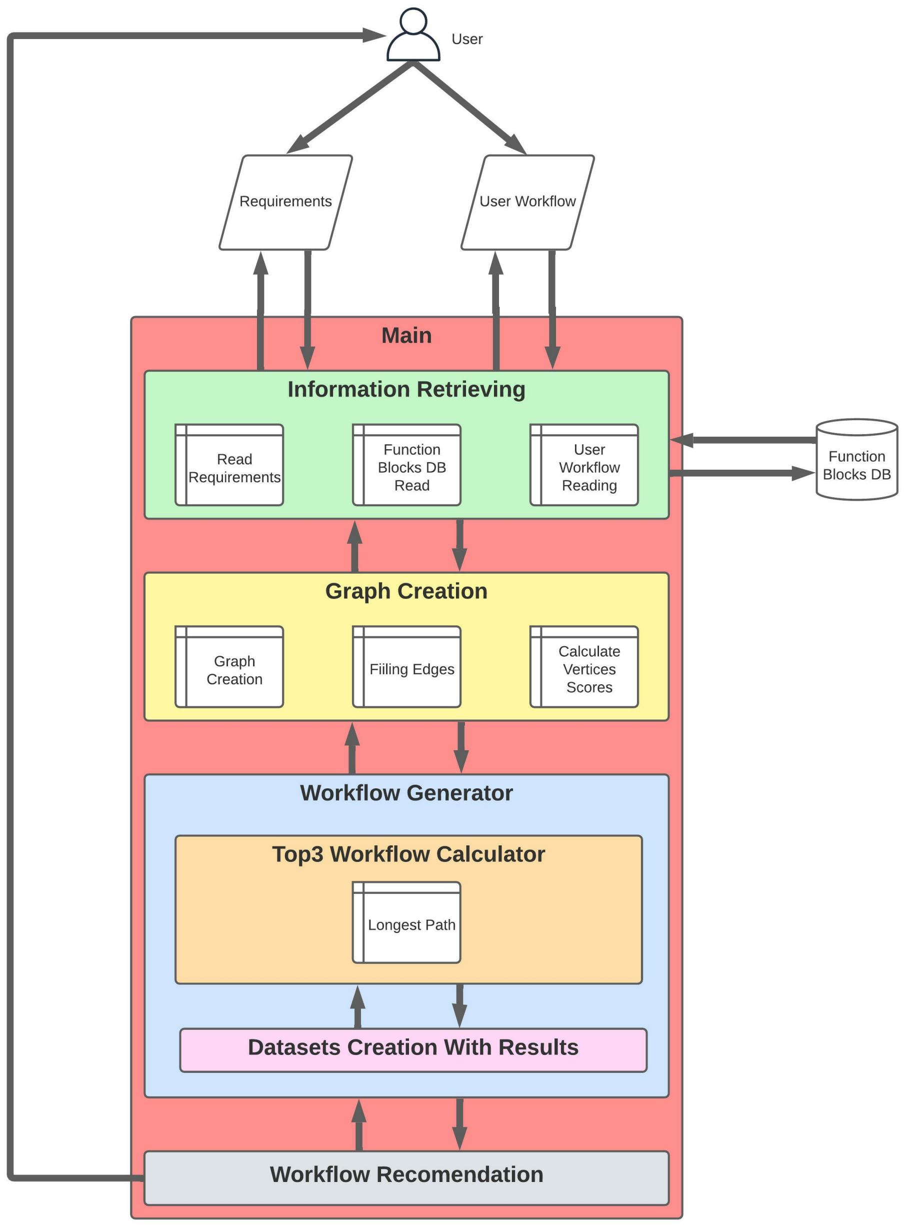


Figure 3.7: Recommending System Architecture.

3.2 Weights Optimization

One of the core and distinguishing factors of the work presented is the way the main algorithm classifies and scores different workflows options, and of course this is directly concerned with the set of two weights each function block has associated as a characterizing feature. Although this set of values works in a very objective way when it comes to algorithm and code running, its real meaning might be a bit more subjective, thus making their assignment a very difficult task. Knowing that translating how good a block is, concerning its time efficiency and performance to a range of values between 0 and 1, is a very hard work to do, resorting to a training/optimizing algorithm seemed like the best path to take. Genetic algorithms (GA) have been used for some time now in lookalike situations [52, 59]. In [9] genetic algorithms are deeply addressed, thoroughly explained, and evaluated as well as compared to some non-GA heuristics for optimizing purposes.

Knowing this, a genetic algorithm was implemented to optimize the weight values of a certain category of function blocks, using a dataset with several samples of a ratio value as an input, for instance the dataset that is automatically generated by the recommending system, and information on which function block is the targeted option for each case as a target (section 3.2.1).

Figure 3.8 shows how a genetic algorithm works, and the implementation used on this work is explained in section 3.2.3

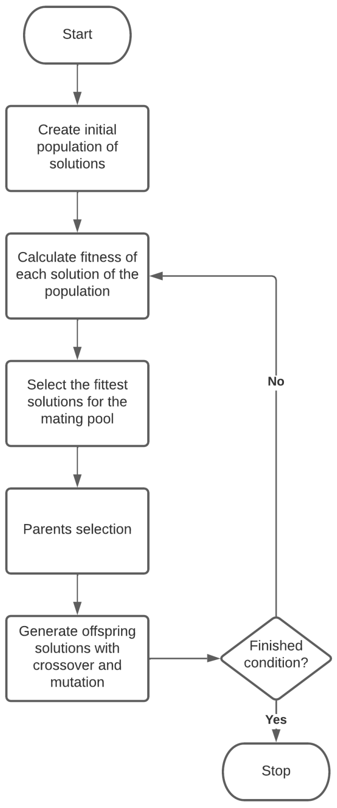

Start
Create initial population of solutions
Calculate fitness of each solution of the population
Select the fittest solutions for the mating pool
Parents selection
Generate offspring solutions with crossover and mutation
Finished condition?
No
Yes
Stop

Figure 3.8: Genetic Algorithm.

3.2.1 Input files

The main function of the genetic algorithm is to help the user to get the most optimized weights for function blocks in order for the recommendation system return the best results.

As explained before, defining weights of two performance requirements, for every function block, for instance performance and time efficiency, may get a little tricky and might entail hours and hours of researching and testing. To avoid this, this automatized feature is made available so in any case scenario the user can draw on a history log of different workflows performance and use it to get optimized weights for several function blocks.

The idea behind the GA is to start as an input a pair of ratio values for each of the two weights and optimize the weights of each function block within a category of a specific requirement that would return a set of two weights for every block for those input values scenario.

So a training dataset example to be run by the GA would be a csv file starting with two columns with the decimal percentage of the proportion chosen for each of the performance requirements, and the following columns fulfilled with zeros and one 1 each corresponding to a function block of a category, the target one having the value 1. An example of this dataset structure is shown in figure 3.9. Note that the values from the two first columns add up to 1 since they represent the ratio value for the performance requirements. For instance in the first sample on this case for a pair of values of 34% of importance to the performance requirement 1 (performance) and 66% of importance to the second performance requirement (time efficiency), the desired function block is the *FB3*.

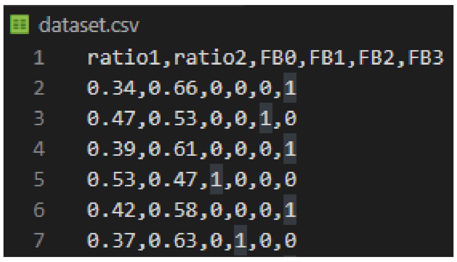

dataset.csv
```
ratio1,ratio2,FB0,FB1,FB2,FB3
0.34,0.66,0,0,0,1
0.47,0.53,0,0,1,0
0.39,0.61,0,0,0,1
0.53,0.47,1,0,0,0
0.42,0.58,0,0,0,1
0.37,0.63,0,1,0,0
```

Figure 3.9: Input dataset for the genetic algorithm.

Following this format is essential and the whole code is expandable being only subjected to changes in the hyperparameter values of the algorithm, explained next.

3.2.2 Solution design

So as explained in the beginning of this section, an optimizing algorithm seemed like the reasonable solution to take. Before implementing a genetic algorithm to estimate the desired values, deep learning was thought of as the best approach to take and was also implemented and tested.

Initially some basic concepts were reviewed concerning this topic, such as neural networks, methods for the weight update process, etc. The key point here was to find a good algorithm to update our weights the same way weights are updated when doing machine learning. Algorithms like least squares linear regression [63], backpropagation [64], and stochastic gradient descent [65] were the main solutions to take a look on. Since having a bit of experience in using backpropagation, as well as its simplicity to understand and implement, this was the first choice.

The idea was to use the backpropagation algorithm to train our weight values, and so a neural network was implemented following a reasonable structure for our problem.

Firstly it was tried to train all the function blocks within a category. A neural network with 2 layers, the first one with two neurons (the values of the performance/time efficiency ratio required by the user), and the second layer having the same number of neurons as the amount of function blocks to train. An output value between 0 and 1 was achieved in the last layer. Each connection would be the corresponding weights to train. These were initially random, and were then adjusted by the error of the output value. The input files for this algorithm were the same as for the genetic algorithm.

A few problems of convergence were found with this algorithm since both weights of performance and time efficiency for each function block would have to be adjusted using the same error value, since this value came from the error between the calculated score and the target value. This problem became bigger and bigger for more function blocks to be trained.

It was then thought of using a simpler neural network that would train only one function block at a time. This solution got better results, but only for function blocks that would be on either ends, too good in performance or too good in time efficiency, having very bad results in cases of middle term.

Having no conclusive and enough good results with the backpropagation method using deep learning, the genetic algorithm was the next solution to take. A good advantage of resorting to a genetic algorithm was that all function blocks of a category would have to be optimized at the same time.

A quick note on the fact that the same input files were used for every version of the genetic algorithm. A very standard algorithm was implemented in first place. A 10-solution population was used in the beginning. The fitness function consisted on calculating the average error of block choice on every sample compared to the target. Started by calculating the score of each function block for each dataset example, and then picked the highest-scored one as the choice and compared it to the target solution. If the right one was picked the error would be 0, otherwise it was 2.

The main problem was the fact that it wouldn't converge to an optimal solution of 0 errors, however whenever it did the results were great.

Not knowing that this problem was due to hyperparameter tuning, a few changes were made to the fitness function in the second version of the genetic algorithm using the same amount of solutions to the same population.

Rather than being an average number of times that the wrong function block was chosen, the new fitness function would calculate the average of the sum of absolute error for each case. For example, knowing that the target file had only one 1 for the supposed function block to be picked, a module subtraction between each function block score and its target value was done. This fitness function had a great advantage compared to the first one, which was the fact that it gave much more information about the set of actual weights, saying how far it is from the optimal solution in a non-discrete way.

The problem of this solution was simple to demystify. Since the sum of the modules of the errors were averaged, the algorithm would converge to a solution where all the weights were set to 0. This made total sense, because then the optimal point for the fitness was 1, since every score would be 0 and just one of the function blocks would have error 1. This fitness function undervalued the function blocks that were supposed to be picked and overvalued the ones that were not supposed, since they were 3 times more.

Having found this problem, the first thought was to multiply the error on the function block that needed to be chosen by 3, evening it compared to the other three. A similar result to the backpropagation was found with fitness function. It returned good values for function blocks that were in extreme points of performance or time efficiency but poor values for middle-term blocks.

Realizing that the first fitness function actually made total sense in the context, since the main goal was to get the right function blocks to be picked for every situation possible, it was brought back from version one. The first version had a problem on converging to a 0 error on the fitness function on every run, and not in the quality of the solution whenever it did. After some testing and reading on general genetic algorithms, not converging problems were linked to lack of variability. Using a population sized 10 solutions and using 5 parents to mate and crossover/mutate in order to optimize a set of 8 weights, for instance, not enough random solutions were being created to try to get to better results by going out of its convergence area.

Therefore, hyperparameteres were tuned differently by having a population of 100 solutions each one, using 30 parents, 30 offspring caused by crossovering and mutating the parents, and the resting 40 were randomly created each generation. The results were shockingly good. Even with bigger and more complex datasets with more and more cases, the algorithm would converge to an optimal solution of 0 errors with less than 200 generations of iteration.

This last version was the starting point of a working solution for weight optimization and there was just one last little problem: if iterated some generations, a lot of solutions would fit with an optimal solution having an error of 0. This created a problem of selection of the best optimal solution, since the fitness function gave the algorithm really limited but super important information. The way to go from here was to implement a more complex fitness solution that would draw a distinction on the optimal zero-error solutions but getting from this pool of optimal solutions, the best one. The full implementation of the GA is explained in the next section.

3.2.3 Genetic algorithm

Genetic algorithms are used for several applications specially in order to solve problems similar to this one, as already addressed in the start of this section.

Before explaining our algorithm step by step, just a quick note that a module was edited and imported into the main program. The functions on the concerned module were edited to meet the current problem making it a bit more specific for our algorithm. It is carefully explained in section 3.2.4 and referred in the rest of this section as *GA module*.

So the first step into genetic algorithms is to have a nice overview on an algorithm like such. Figure 3.8 shows an intuitive flowchart of how a genetic algorithm works. Having all the parameters set up, it starts a loop over generations and calculates the fitness, selects the mating pool, selects the parents, from the mating pool does the crossover and mutation process, and finally generates the offspring.

So the first stage of the code consists on the creation of the input information, from the dataset, into matrices, and the assignment of the number of weights we are looking to optimize. For instance, if we have four function blocks, the number of weights shall be eight (one for time efficiency and one for performance for each one of the blocks). The weights represent the genes in this context.

The next stage is to set up an initial population. A population is a group of solutions, each solution being an individual composed by a set of genes. Each individual will have a combination of values for the genes, a chromosome, that corresponds to the weights of the function blocks to be optimized. Each chromosome is organized by the number of function blocks, each one with two weights. Firstly, it is defined the number of solutions per population, that is, the number of individuals that make up a population. Then, the population size is calculated by getting the number of people in a population, and structuring it in chromosomes. The final step of this stage is to create the population, using the first function *create_population*, from *GA module*, thereby creating a full population of random values for every weight between 0 and 1. Larger populations guarantee a more variable set of solutions, making it easier for the algorithm to converge to an ideal solution since more combinations of weights are selected and so the chances of getting better values increase.

After preparing the first population, two more parameters are set before getting into the loop of generations, the desired number of generations and the number of parents mating. This last one refers to the number of solutions from a population that are selected for mating and to go directly into the next population. These are going to be the so named *parents*, and they represent the fittest people in a population. Having around 15-20% of a whole population as parents guarantees good results.

Having these two values defined and entering the loop of generations, thus starting the steps shown in the flowchart previously. First things first is to make the fitness calculations of the current population. This is done and returned by the second function from the *GA module*, *cal_pop_fitness*. The returned fitness is an array of the size of the population with the fitness of each individual of the current population. As explained in the next sub-section, the lower the fitness value the better the solution. This is due to the fact that the fitness consists on the average of an error and thus it is supposed to be minimized. So our fitness function is actually a cost function.

After calculating the fitness values for each individual, we need to pick the best of them as parents. This step is done by the function *select_mating_pool* from *GA module* into a mating pool.

Created the mating pool with the best individuals of the population, it is time to do the next step of the flowchart, the crossovers. Using the selected parents, the function *crossover* from the *GA module* will crossover the genes from the parents following a rule defined in the function, creating the offspring. The offspring consists on the resulting solutions from the crossover process. Before gathering the parents and the offspring chromosomes, the offspring individuals go through one last process from the GA variant which is mutation. Mutation is applied to the crossover results, offspring, and done in a uniform way to the random genes. The offspring size represents the same size of the parents, somewhere around 15-20% of the total population.

After the crossover and mutation processes we have two groups of individuals, the parents and the mutated offspring. These two are gathered together to form a new population. The two groups together shouldn't have enough solutions to fulfill a population as a whole, as they would represent a maximum of 40% of the next whole population. Therefore, for the rest of solutions are created random values, just like done in the beginning of the algorithm. When everything is merged a new population is born and goes as the actual population to the next generation of the loop.

To help visualize the algorithm, let's put in a practical example. If the population has 100 solutions/individuals and the chosen number of parents mating is 20, then the best 20 individuals go to the mating pool, it is done a crossover between this 20 individuals into the offspring and then they are mutated. So we have 20 parents, 20 solutions in the offspring, which come from crossovering and mutating the 20 parents, and 60 more are created randomly, making it 100. This new population is the next generation of individuals.

So this whole process of creating new generations of populations follows a rule of selecting at every generation the best solutions, parents, and use them to try to achieve even better solutions, through crossover and mutation, while guaranteeing that the best individuals from the last generation are kept to the next one. So if the offspring comes worse than the parents, then it wouldn't mess the population up since the best solutions are always kept. Adding a factor of randomness, with the creation of random individuals to fulfill the rest of the population after the offspring, allows the population to get better solutions out of the current best solutions. This way if a generation is good but not the greatest, it helps the population to bring new people out of the population tendency and increasing the chances of bringing a new better solution, thus exploiting good solutions out of the convergence area.

The tricky part of creating a good genetic algorithm is to define the best fitness function possible. The main goal of this algorithm is to get a solution of a population the right values so they go through a score calculation process and the right function blocks end up being the highest ones in all samples of the dataset. This way, the fitness function calculates the average of times a solution got a function block wrong. Minimizing the fitness value, the algorithm will try to return the solution with less error. Since more than one solution may end up having 0 errors, these all are optimal solutions, though there is a way of getting the best ones from the optimal solutions. So a bit more complex fitness function was developed. Instead of only returning the number of errors throughout all the samples of the input dataset, it also takes in account absolute error, just as done in the second version referred in section 3.2.2, making a more enriched fitness value for each of the solutions.

Coefficients were assigned to each of these two values. 0.99 goes for the number of errors, and 0.01 for the absolute error. This values are justified by the fact that we still want the algorithm to converge to solutions with 0 errors in all the samples, while also wanting that a small difference may be detected within optimal solutions, that being the least absolute error. This fitness function is better explained in the next section.

After iterating and generating better and better solutions, a termination condition was set in order to stop the generations loop whenever the same best and optimal solution stabilizes for a defined number of iterations.

After getting the best optimal solution, the weight values may not be exactly how they are supposed to. The GA may actually get an optimal solution, and so the values returned work for the samples, but they may not work properly in the recommendation system since they may not be normalized within the function blocks of the category whose weights were just optimized. For instance all the values, returned may have an order of magnitude of 10^{-4} and still give the best workflow possible, but it is desired to have representative values between 0 and 1 of how good a function block actually is in two performance metrics.

In order to perform this, after gathering the best solution within the last population, a function called *norm* is called. This function takes the best solution and the input dataset. Then a feature scaling normalization between 0 and 1 is done to weights. This procedure doesn't change the optimal results, because the values are always relative between function blocks, and so this normalization method shouldn't affect the results. In the end a checkup of the new normalized weights is done with the dataset, to make sure it still is an optimal solution with no errors. The final values are finally returned to the user.

3.2.4 Library

After some research on how a genetic algorithm works as well as on the best libraries for python implementation, some good reading material came out in the open.

One of the most suitable modules in python for genetic algorithms is the *PyGAD*[7] library [66, 67]. It features some really good functions for every common step of a genetic algorithm, and follows a strong structure of hyperparameters which allows the code to be expandable, thus making it more reliable.

The main functions from the module were already addressed in the last sub-section, and now it is explained each one and how it works in the given context.

The first function is the one that generates a population, *create_population*. It is used in the beginning to create a new whole initial population and then in every loop of each generation to create the new random individuals to fulfill the next generation population. This function takes one argument which is the size of the population. The size of the population is a two dimensional number, the first one being the number of solutions it is supposed to generate, and the other one the number of values to generate for each solution. The number of values for each individual of a population is called a chromosome and is composed by genes that in this context are the weights. So receiving the total number of weights, the program knows that each function blocks

[7] https://pygad.readthedocs.io/en/latest/

will have two weights, one for time efficiency and one for performance. The function is prepared to recognize this and therefore it structures each solution in a matrix of two columns by the number of function blocks rows.

For instance, if you send 10 solutions per population and 8 weights, then the function will generate for each of the 10 solutions a chromosome with four rows and two columns, being each row one function block and the two rows its time efficiency and performance values. In short, it returns an array sized (10, 4, 2) from a (10, 8) argument. This is all done by using the *random.uniform* function from the *NumPy* module, and gets values between 0 and 1.

The next function of this module is *cal_pop_fitness*. This is the most important function in the algorithm, calculating a population fitness, and so the way it makes the calculation is the key for the main algorithm to converge to an optimal solution. The main idea is to calculate an average error between the scores that the current weights get from the input value of the dataset and its target values.

The function takes four arguments, the array with the ratio values, the targets array, a population, and a variable for the number of errors. The fitness is a 1-dimensional array with a length equaled to the number of solutions within a population. The function starts by looping over every person in the given population, and for each person it loops through each sample from the dataset. For every input it will calculate a score for each of the function blocks. This is done by doing the dot product between the weights of the person in question and the input values from the dataset, which is how the main recommendation algorithm calculates a score to build a workflow (section 3.1.4).

Having the scores for a person for each function block, an intern function is called to choose the highest-scored block. So basically this function takes the matrix with the scores and turns into 1 the highest one, and into 0 the rest. This way the resulting score matrix will match the target matrix structure which will be filled with zeros and a one corresponding to the function block to be picked as the best in each input value (section 3.2.1).

So this way the fitness calculation goes through a check subtracting the new binary score matrix and the target matrix. The module of the subtraction between. There will be only a value 1 in each matrix and the remaining values are 0, thus the error will either be 0 if the right block was chosen, or 2 if the wrong one had the highest score. Looping over every case on the dataset, an average error is calculated, and that value is the average number of errors a person had through the dataset samples. The fitness value is not only composed by the average value of the errors. The other part missing is the absolute error of a solution for a given sample input. So, grabbing once again the original score matrix, it is then subtracted to the target matrix and calculated the sum of the modules of each number of the result. This gives the information of how close the scores are from the expected on the target. This is done to help differentiate several optimal solutions. After running through all the samples an average of this error is also done.

These two values are weight-summed. The final fitness value is composed by 99% of the average number of errors and 1% of the average error value. Repeating this intensive process for every single one individual will fulfill the fitness matrix.

The third function to be used from the *GA* module is the *select_mating_pool*. As explained previously, this function is called to select the best individuals in the current generation as parents to produce the offspring of the next generation. The function takes 3 parameters: the population, the fitness matrix of the population, and the number of parents for mating. This allows the parents matrix to be shaped exactly like the standard population one but with less solutions, being this number just the number of parents in the function argument.

So looping over the taken population it chooses the best one by checking the lowest fitness value, gets its index in the population array, and appending it to the parents array. Everytime a solution is chosen to be a parent, its fitness value is changed to a high value, such as 99999, so it won't be picked again on the next loop. This is done until it has chosen the desired number of solution to be the parents.

For example, with a population of 10 individuals, and it wants 4 individuals as parents, the function will get the 4 lowest fitness values and take the corresponding individuals as the parents. This guarantees that no matter what comes out of the offspring operations, the next generation won't have a worse best solution than the previous one as the best, since the best solutions are kept held into the next one.

The last two functions of this module are the ones that create and modify the offspring. Firstly there is the *crossover* function. It takes two arguments: the parents (calculated in the last one) and the population. The offspring has the exact same shape as the parents. Using from *NumPy* module the method *random.choice*, two parents are randomly chosen to have a crossover between them. The crossover is done by getting the first column of the chromosome, this being the time efficiency values of the first individual and mix it with the second column of the chromosome, the performance values, of the second individual. This is done in loop until the desired number of new mated individuals is achieved.

Usually after the crossover mutation happens. The *mutation* function mutates every person of the offspring generated in the crossover. A default 40% of the weights will randomly be chosen to suffer a mutation, evenly split between each column. The mutation consists on the sum to the selected weights of a value randomly generated between -1 and 1, by the *random.choice* function from *NumPy*.

For instance if the offspring has 4 people, and each people has 5 function blocks each with two weights, making a person have a shape of 5 by 2, then for each person, two random weights from each column will suffer a mutation.

Mutating random values by some number between -1 and 1 may seem a lot in absolute terms, since the final weights are supposed to be always between 0 and 1. This was done so when a population fitness is converging to a local minimum, adding a bit of variety to its solutions may help find better solutions. Also, a process is done in order to saturate overflow values. So if with mutation a value goes over 1 or under 0, the new values will be saturated so they stick within the limits. With this we can affirm that our algorithm will always discard non-admissible solutions.

3.2.5 Architecture

To help visualize the whole weight optimization system running the genetic algorithm, figure 3.10 shows its architecture. It starts from the user entering its dataset as an input into the main code block, initial population is then generated, used by the genetic algorithm block that loops over a user-defined number of generations. Have the termination condition ended, the best solution of the last population is then selected and goes under feature scaling normalization. The new normalized data goes under a checkup test afterwards, in order to verify it still is an optimal solution. Verified that condition, the new set of weights is then returned to the user.

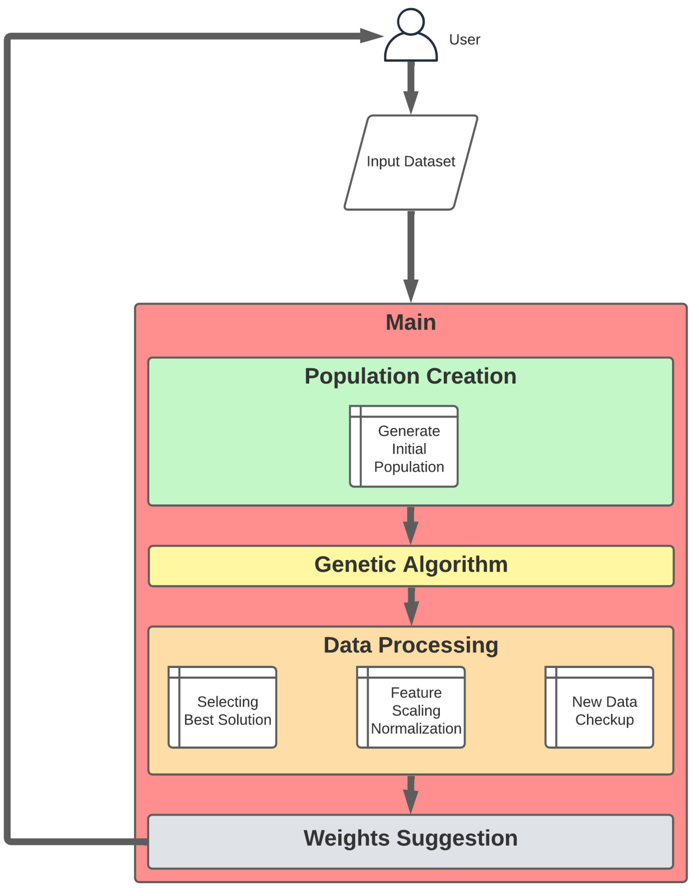


Figure 3.10: Weight Optimization System Architecture.

3.3 Overall Implementation

Now that both algorithms are thoroughly explained in the previous sections, it is important to emphasize how they may complement each other.

Firstly, the core correlation factor between the two algorithms are the weight values for performance and time efficiency for each function block. The recommendation program uses this values to calculate the score for the workflow options and selects the best one based on the maximum score obtained. The closer to the absolute correct values we have, the better are the results we get from the main program. That's why using the genetic algorithm is very important.

Before getting into the main recommendation program, all the function blocks must have valid weight values. The weight optimization system is recommended to be used prior to this in order to get the best weight values possible. If a user wants to add new FBs, he may want to use genetic algorithm (GA) in order to estimate those values.

The genetic algorithm returns to the user the optimized values for every function block within a category, but does not update the values automatically into the database csv file. This process must be done by the user manually. Whenever the user wants to add a function block of a certain category, he has to recalculate all the weights of all the blocks within that category, using for example, the genetic algorithm. Another crucial point is that function blocks from different categories can't be trained at the same time.

Once all the values for performance and time efficiency for every block in the database are updated, the main program is ready to be run.

To give a tangible example of every procedure the user has to follow let's suppose the user wants to add a new category to the database and some function blocks of that same category. Since both algorithms are fully expandable, this is very achievable. The first thing the user has to do is to get the performance and time efficiency weight values for the new function blocks. Since the function blocks are all from the same new category, these values can be retrieved all at once. Then the user has to make the datasets for input and target values for the genetic algorithm, following the structure presented in section 3.2.1. Once the datasets are done, the user runs the genetic algorithm and gets the values for all the weights for the blocks. The next stage consists on adding the blocks to the database csv file, once again following carefully the structure explained in section 3.1.1, with the weight values returned by the genetic algorithm. Note that if a new category is being added, then the respective function blocks will have a new value for the *requirement_id* block attribute.

Before running the main workflow recommendation system, the user has to also make some changes to the requirements csv file as well, adding the new specific requirement regarding the new category following the order that was told before. Having all these steps been completed, the last algorithm is ready to be run and should be returning the optimal top 3 workflows, having in account the new function blocks added by the user.

One last quick example is the case where the user wants to use the categories already presented in the database, but wants to add one or more function blocks to the database within those categories. Once again for every category updated, the weights must all be updated and the genetic algorithm should be run for this purpose. Also, changes to the input and target datasets must be done. With the new values, the weights of all the function blocks belonging to the category in question have to be updated in the database file. Since all the function blocks added were from an already existing category, there is no need to add new requirement lines in the requirements csv file. Once all this is done, the workflow recommending program is ready to be run.

3.3.1 Overall architecture

One of the most interesting features about the proposed work is the interopeability of the two implemented systems. The genetic algorithm uses history log data about workflows based on two performance metrics/requirements, and generates sets of weights that profile function blocks (FBs). The recommending system, uses those FBs weights to recommend workflows based on those same two performance requirements. After the generating the top 3 workflows for a given scenario, the recommendation system automatically generates datasets based on this results. These datasets can be used right away by the genetic algorithm based weight optimizer system. Figure 3.11 shows the architecture of the two systems working alongside.

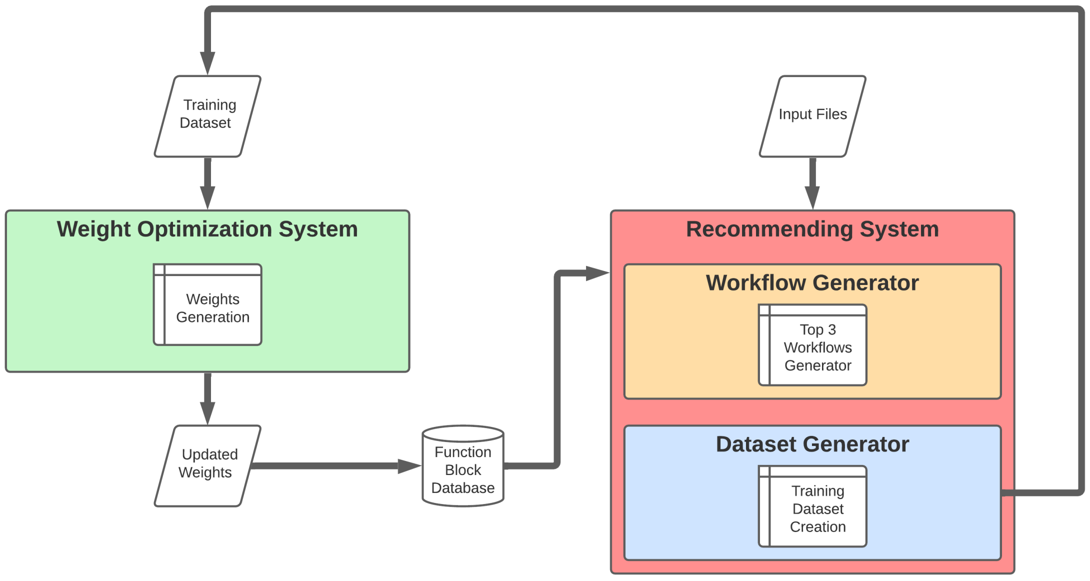


Figure 3.11: Overall Architecture.

The next chapter has different testing methods on both systems to evaluate the quality of the proposed implementation.

Chapter 4

Testing and Validation

In this chapter several tests were made using a real dataset from a study case [68] where workflows are also recommended and optimized based on two performance metrics defined by the user. Having done a lot of research it's quite hard to find good and clear testing examples for our system. Usually the procedure of orchestrating workflows is evaluated in several different metrics such as performance, accuracy, execution times, monetary cost, resources used, and more depending always on the case scenario where it's being applied. This way the approaches on most papers rely on optimization that suits one case or one type of case only, lacking generality and expandability.

Our work has that property of being able to be applied in as many case scenarios as possible. This of course is not easy to do and still demands a smart way of interpreting different cases into our system conditions. This means some ability to "normalize" and adjust datasets into our system structures. The paper used for testing purposes uses a dataset that suits that structure, with still some adjustments to be made in order to make it fully working with the proposed system.

As mentioned before our recommendation system recommends the 3 best workflows, in order, based on two user-defined performance requirements/metrics. The reason behind this number is that when dealing with this type of scenarios, there is some chance of not returning the best possible workflow, This is due to the fact that the program does not take in account every variable existing, taking only two in this case. Therefore, a lot of information may be unknown for the program, for instance, energy efficiency, resources consumption, execution costs, etc. This way, returning three workflows makes the system more helpful for real-life scenarios usage.

There are plenty of different ways to evaluate the performance of recommender systems to know how good the results returned by such are. In this work we proposed the use of MAP@K as evaluation metric for the whole system, due to the fact that it is suitable for ranking evaluation, rewarding the system if good solutions are suggested in the first positions.

Precision is one of the most used metrics for recommending systems [69]. In recommending systems it's given by the following expression:

$$Precision = \frac{\#RelevantRecommendations}{\#Recomendations} \tag{4.1}$$

Precision gives a good notation of how good a system is in terms of number of correct picked options, but doesn't seem to care much about order. Thus, precision at cutoff K, or P@K, is the precision of the system at the recommendation number K [52]. P@K always takes in account the precision of the recommendation at every point before K, making it an order evaluation metric.

Knowing our system is recommending three workflows, the ideal K point of analysis is K = 3, although K = 1, and K = 2 may also be used for some reference. Average precision at cutoff K, or AP@K, is the sum of the precision at every point until K, if relevant, divided by the number of relevant suggestions which will in our case be always K:

$$AP@K = \frac{1}{K}\sum_{n=1}^{K} P(n), ifk^{th} = relevant \tag{4.2}$$

For the same amount of relevant recommendations within a number of total recommendations, AP@K rewards a recommendation that has the relevant items on the first suggestions [54].

Finally, when a lot of samples are used, which may be the case, the mean of the average precision at cutoff k, is the mean value of AP@K of every test done by the system. We may conclude then that MAP@K treats the system a ranking task. So MAP@K, for K = 3 is our system's evaluation metric.

4.1 Case Study

The paper [68] introduces extract-transform-load (ETL) processes as workflows that process data in several ways and moves it between data sources (DSs) and a data warehouse (DW). They talk about the importance of performance otimization for ETL processes. It also addresses the fact that most of the cases, optimization always depends on the workflow designers, since every case becomes very specific and very personal, as referred above. They contribute with a cost model for user-defined functions (UDFs) that are seen as black-box operators, in order to optimize those UDFs choosing a degree of parallelism for an UDF, to meet user-defined performance metrics. The performance metrics used for the present work are execution time and monetary cost.

The used case studies the workflow of the Set-similarity joins using MapReduce (SSJ-MR) [70].

The workflow is composed by three stages of data processing:

- Token ordering
- Pair generation
- Record join

The workflow for SSJ-MR is shown in figure 4.1. Each stage can be completed by two of the optional UDFs: TO1 or TO2, PG1 or PG2, RJ1 or RJ2.

They previously presented an extendible theoretical ETL framework [71]. The recommender modules generates of the framework optimizes ETL workflows using a cost model library. It also works with statistics and machine learning algorithms, and stores previous ETL executions.

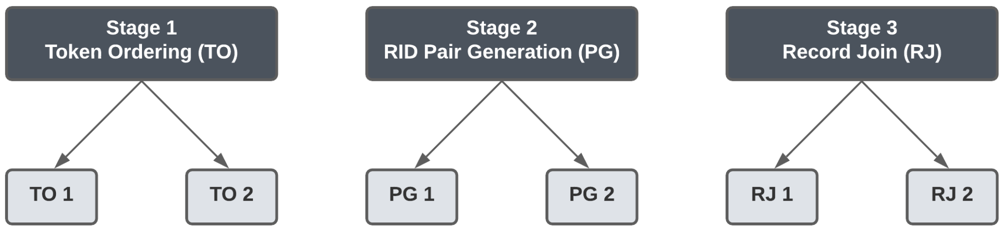


Figure 4.1: Set-similarity join workflow. From [68]

Each of the stages can be completed by two data process approaches, as perceivable in the workflow in the workflow figure. These approaches are:

- Token ordering: Basic Token Ordering (BTO) and One Phase Token Ordering (OPTO).
- RID Pair generation: Basic Kernel (BK) and Indexed Kernel (PK).
- Record join: Basic Record Join (BRJ) and One Phase Record Join (OPRJ).

They used a simulator to run the several options and retrieve information about execution time, estimated monetary cost, data size, etc. They tested the 6 different data processes approaches, two for each stage, and for each one calculates execution time and execution cost in $/h associated to a n-node cluster configuration to be executed in the Amazon Web Service[1], producing a total of 24 options. Table 4.1 has all these options for each algorithm and the number of nodes running in the integration environment.

Table 4.1: Execution time in seconds of each stage on different cluster sizes.

Stage	Algorithm	# Nodes [exec cost/h]			
		2 [0.4$/h]	4 [0.8$/h]	8 [1.6$/h]	10 [2$/h]
1	BTO	191.98	125.51	91.85	84.02
	OPTO	175.39	115.36	94.82	92.8
2	BK	753.39	371.08	198.7	164.57
	PK	682.51	330.47	178.88	145.01
3	BRJ	255.35	162.53	107.28	101.54
	OPRJ	97.11	74.32	58.35	58.11

For each data process, the execution time was calculated for the use of 2 nodes (costing 0.4$/h), 4 nodes (costing 0.8$/h), 8 nodes (costing 1.6$/h) and 10 nodes (costing 2$/h). The model used to generate the best workflow possible is called *Linear Integer Programming Model*[2]. This model is a mixed integer linear programming (MILP) solver using Simplex method [72]. So a user allocates a budget and the algorithm returns the fastest possible workflow for that price based on the values of Table 4.1.

[1] https://calculator.s3.amazonaws.com/index.html
[2] http://lpsolve.sourceforge.net/

So this case of study is very interesting given the points of similarity with the work proposed. The paper also recommends the best possible workflow based on two user-defined performance metrics, which is the perfect real case to test our system since it's also based on a two performance metrics. Another good value on the given dataset is that calculations were made available for the different UDFs. In order to keep our text homogeneous, we will refer to the UDFs of the given example as function blocks (FBs) or just blocks. Having the information on the execution time and cost for each possibility of executing each block will help on the weights calculation in order to run our recommendation system.

In chapter 3 the two performance requirements, used as an example of implementation only, were performance and time efficiency. From now on, in order to meet the used case context, the two performance requirements used to run our algorithms are time efficiency and cost efficiency. The way we estimate this values is explained next.

4.2 Recommendation system

In this section we evaluated the recommendation system with two different approaches for weight estimation. After that we used a lot samples to run the algorithm with the same weights of the two approaches and compared the performance in order to conclude if the system is consistent on the performance, no matter how many samples are used.

In order to test our recommendation system, we need the function block database filled with all the FBs and respective weights details. For any case scenario is very important to understand everything around it in order to make it more accessible to transform the necessary data into our system's structures.

The first step is the weights calculations for the FBs. In Table 4.1, we have reference values of execution time for every algorithm available with different nodes. The best way to extract this information to our database is to create a block for every possibility of running one of the algorithms. Since there are a total of six algorithms that can be executed in four different ways each. This way for each algorithm, four blocks were created, resulting in a total of 24 FBs. For instance, BTO algorithm is divided in four FBs, *BTO_2_Nodes*, *BTO_4_Nodes*, *BTO_8_Nodes*, *BTO_10_Nodes*.

Since we have three different stages, and each stage has to be done by executing one of its FBs, then this translates into three specific requirement categories. *BTO* and *OPTO* FBs belong to requirement 0, *BK* and *PK* FBs to requirement 1, and *BRJ* and *OPRJ* to requirement 2.

Therefore, the graph that represents the structure used is shown in figure 4.2.

After having the FBs, it is necessary to estimate their weight values. From the table we can grab values for cost and execution time of each FB. Our system is ready to take weight values between 0 and 1, and since less cost and less time means more efficiency, then we can't use directly the values of the table. Instead, and since our algorithm suggests workflows based on optimizing FBs scores, for each block was calculated a **time efficiency** and **cost efficiency** weight value. For this purpose, inverted feature scaling normalization was done to both costs and execution times.

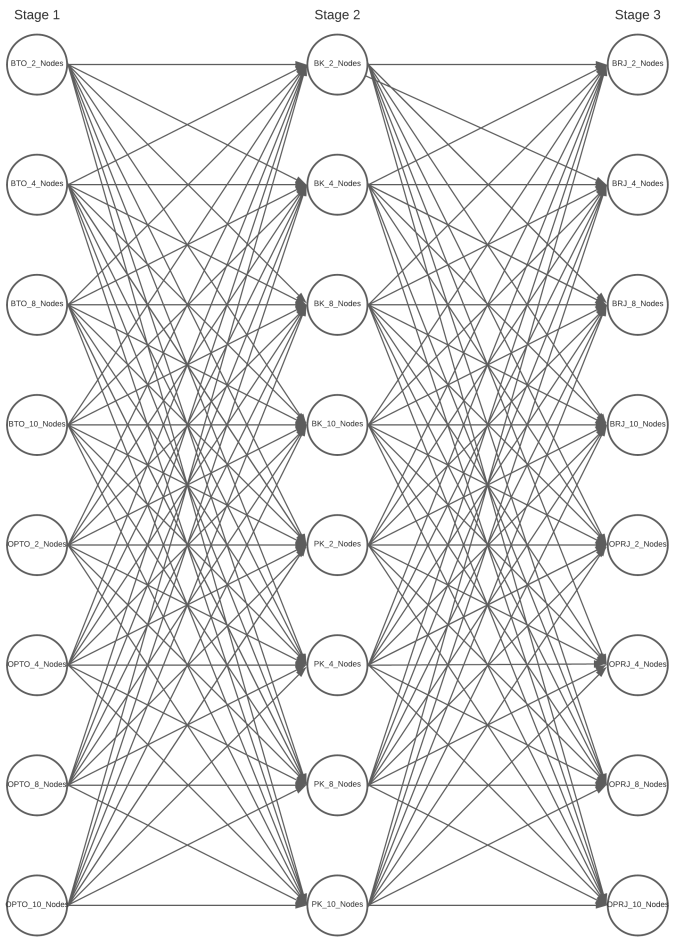


Figure 4.2: Graph used for this case study.

Inverted feature scaling normalization is an adaption of the original feature scaling normalization but instead of subtracting to the original value the minimum on the top of the fraction, the original value is subtracted to the maximum value. This is done since we want efficiency-related values, and since both requirements are as more efficient as smaller they are (cost and execution time) this will normalize the larger numbers into small normalized values and the opposite to the smaller values of time and cost. The formula of the inverted feature scaling normalization is:

$$X' = \frac{X_{max} - X}{X_{max} - X_{min}} \tag{4.3}$$

For example, a value 1 for every block that runs on 2 nodes (costs the minimum price, 0.4$/h), and a value 0 for blocks running 10 nodes (maximum cost of 2$/h) as cost efficiency values. Since the cost values are well-defined and there are only four options, this is pretty straight forward.

On the other hand, time efficiency is not that simple to calculate, and we can take two approaches in order to calculate the time efficiency value for a FB, a *Local Normalization* or *Global Normalization*.

Local Normalization (LN) normalizes the value of a function block relatively to every block on its own stage not minding the executing time values of blocks in different stages. This makes sense since we want to choose the best block in every category given the performance requirements ratio value defined by the user. And so the fastest-to-execute block in a stage should have a weight of time efficiency equal to 1, and the slowest a weight equal to 0.

Global Normalization (GN) normalizes each FB time efficiency value in accordance to the executing times of all the blocks available and not only in the same stage/requirement. This may not make sense yet, but since all the blocks, in this case, have the executing time in the same unit, seconds, then it may make sense to let the algorithm know how fast a block is compared to every other FB. For instance, with *LN* the time efficiency weight value for the block *BK_2_Nodes* would be the same of *BTO_2_Nodes* once they are the two slowest options of their stages. Even though the real difference between them on executing time, as referred in 4.1, is actually 561.41 seconds, which is a huge value.

So having in mind this two approaches for normalization purposes, we decided to run some tests on both of them, therefore we can make an overall evaluation of the system behavior on different interpretation of the same problem.

In order to evaluate quantitatively the system results, MAP@K, for K=1,2,3 is used. But to be able to calculate the MAP@K, a qualitative evaluation has to be made beforehand, since we need to define the concept of a relevant recommendation in order to calculate precision.

Thereby, the definition of a good/relevant recommendation is the best possible pipeline for a certain cost, making sure there are no faster pipelines at a smaller cost. In first place we need to know every possible combination for the cost of a workflow. We know that the cheapest workflow possible is one composed by three FBs running at 2 nodes, which gives us a cost of 1.2$/h, and the maximum cost is three FBs running at 10 nodes costing 6$/h. Knowing this, we used a tool based on linear programming simplex[3] and on evolutionary algorithms[4] to get the 3 smallest workflow execution times of every possible combination for cost. This values are presented in Table 4.2.

Table 4.2: The three shortest possible execution times for each combination of workflow cost.

	Execution time (s)		
Cost ($/h)	1st pipeline	2nd pipeline	3rd pipeline
6	🟩 287.14	295.92	306.7
5.6	🟩 287.38	🟨 294.97	296.16
5.2	🟩 295.21	🟨 298.18	314.77
4.8	🟩 303.35	🟥 312.13	318.48
4.4	🟩 311.18	🟨 314.15	🟥 318.72
4	🟩 333.97	🟥 336.94	345.05
3.6	🟩 334.69	🟨 344.84	🟥 354.25
3.2	🟩 357.48	🟨 367.63	🟥 368.56
2.8	🟩 391.35	🟨 401.5	🟥 411.17
2.4	🟩 451.38	🟨 467.97	🟥 471.2
2	🟩 542.94	🟨 553.09	🟥 580.18
1.6	🟩 602.97	🟨 619.56	🟥 643.58
1.2	🟩 955.01	🟨 971.6	🟥 1025.89

From this table we can take some conclusions. Every pipeline in the first column should be chosen before any of the others. So for the top recommendation to be relevant it must be one of the options from this column. The second column pipelines shall never be recommended before the first pipeline of that cost. The same rule applies to the third column pipelines that must never be chosen before the first and the second were chosen for a certain cost. For example, for a cost of 5.6$/h if the workflow with 294.97s of execution time is returned by the recommending system in a better rank than the workflow with 287.38s, that would be two wrong choices.

Apart from that rule and in order to test the recommending system performance at all levels, we also take a workflow choice as irrelevant if that workflow could never be picked no matter in what order since there are cheaper options that are faster. The color code in the Table 4.2 helps understanding this:

- Green pipelines can be recommended at any rank as long as they are the best ranked pipeline of its cost.
- Yellow pipelines should never be recommended with a better rank than the corresponding green pipeline of its cost, that is can only be recommended as the second or third best pipeline.
- Red pipelines should never be recommended in a better rank than third. This is the consequence of existing always at least two options that are faster at the same or even less costs.

[3] https://www.solver.com/linear-quadratic-technology
[4] https://www.solver.com/excel-solver-evolutionary-solving-method-stopping-conditions

- White pipelines SHALL never be recommended. This happens because there's always at least three options that are faster at the same or less costs.

For instance, the second pipeline of the cost 6$/h shall never be recommended since there are three pipelines that are faster. One at the same price and two others with a SMALLER price.

The recommending system will return workflows with costs based on the user-defined preference ratio value between cost and time efficiency and so it may return pipelines with different costs. And so performing a deep analysis of which recommendations are relevant may get a bit tricky.

Now that we decided the two different approaches to estimate the FBs weights and defined clearly a relevant recommendation we can proceed for the practical testing.

So after applying the Global Normalization (GN) technique explained previously in the beginning of this section the following weights were obtained for each FB are presented in Table 4.3.

Table 4.3: Weight values for the 24 function blocks after using the Global Normalization method. CE and TE stand for Cost Efficiency and Time Efficiency, respectively.

		# Nodes							
Stage	Algorithm	2		4		8		10	
		CE	TE	CE	TE	CE	TE	CE	TE
1	BTO	1.00	0.81	0.75	0.90	0.25	0.95	0.00	0.96
	OPTO	1.00	0.83	0.75	0.92	0.25	0.95	0.00	0.95
2	BK	1.00	0.00	0.75	0.55	0.25	0.80	0.00	0.85
	PK	1.00	0.10	0.75	0.61	0.25	0.83	0.00	0.88
3	BRJ	1.00	0.72	0.75	0.85	0.25	0.93	0.00	0.94
	OPRJ	1.00	0.94	0.75	0.98	0.25	1.00	0.00	1.00

Having everything set up to run the recommending algorithm, the first test included 13 samples varying on the ratio value. The results of this first test are available in Table 4.4.

Firstly we need to interpret this table. The first two columns represent the priority given by the ratio value to each of the performance requirements, cost efficiency and time efficiency. For instance, in the first sample a value of 0 was given to cost efficiency and 1 to time efficiency, this means the user wanted the fastest possible workflow no matter the cost.

The *Recommendation* columns calculate the execution time and cost for the chosen workflow. For instance on the first sample, the best recommended workflow was *BTO_10_Nodes, PK_10_Nodes, OPRJ_10_Nodes*. It has the cost of 6$/h since it runs on 10 nodes in each FB and is the best possible option for a 6$ cost workflow having an execution time of 287.14s. This was then evaluated for every workflow recommended on every sample, creating this way the *Best Option Chosen* columns. If a block is green then the best option was picked for that rank and that cost. If not then it should be red.

We can conclude from this results that the best pipeline for each cost and rank was recommended for every single sample but the last one. The explanation behind this is that when choosing a ratio value that gives all priority to the cost efficiency requirement, then all the information

Table 4.4: Results for the test 1, using GN for weight estimation.

Ratio value		Recommendation						Best option chosen?		
		1st pipeline		2nd pipeline		3rd pipeline				
cost	time	cost ($/h)	time (s)	cost ($/h)	time (s)	cost ($/h)	time (s)	Green - YES | Red - NO		
0.00	1.00	6.00	287.14	5.60	287.38	5.60	294.97	🟩 287.14	🟩 287.38	🟩 294.97
0.01	0.99	5.60	287.38	6.00	287.14	5.20	295.21	🟩 287.38	🟩 287.14	🟩 295.21
0.10	0.90	3.60	334.69	3.20	357.48	3.60	344.84	🟩 334.69	🟩 357.48	🟩 344.84
0.20	0.80	2.80	391.35	3.20	357.48	2.80	401.50	🟩 391.35	🟩 357.48	🟩 401.50
0.30	0.70	2.40	451.38	1.60	602.97	2.80	391.35	🟩 451.38	🟩 602.97	🟩 391.35
0.40	0.60	1.60	602.97	1.60	619.56	1.60	643.58	🟩 602.97	🟩 619.56	🟩 643.58
0.50	0.50	1.60	602.97	1.60	619.56	1.60	643.58	🟩 602.97	🟩 619.56	🟩 643.58
0.60	0.40	1.60	602.97	1.60	619.56	1.60	643.58	🟩 602.97	🟩 619.56	🟩 643.58
0.70	0.30	1.20	955.01	1.20	971.60	1.60	602.97	🟩 955.01	🟩 971.60	🟩 602.97
0.80	0.20	1.20	955.01	1.20	971.60	1.20	1025.89	🟩 955.01	🟩 971.60	🟩 1025.89
0.90	0.10	1.20	955.01	1.20	971.60	1.20	1025.89	🟩 955.01	🟩 971.60	🟩 1025.89
0.99	0.01	1.20	955.01	1.20	971.60	1.20	1025.89	🟩 955.01	🟩 971.60	🟩 1025.89
1.00	0.00	1.20	1042.48	1.20	1025.89	1.20	955.01	🟥 955.01	🟥 971.60	🟥 1025.89

about the time efficiency weights will be totally ignored. This happens because the recommendation algorithm calculates the score for each function block by doing the dot product of its weight values and the pair cost/time ratio value as explained in section 3.1.4 from the previous chapter. Therefore, if time ratio value is equal to 0, then the score will only be calculated with the product of the cost ratio value, which is 1 and the block cost efficiency weight value. Since every FB that runs on 2 nodes has the same cost efficiency weight, 1, and there are 2 in each stage, then the algorithm can't differentiate from the fastest one, this way recommending randomly the FB that respects the cost efficiency. This actually makes some sense, since when choosing a 1/0 ratio value for cost preference, the user is telling the program that he doesn't care on how much time the block runs, he just wants to get a cheap workflow, and so this is an ambiguous situation. To check this special situation we ran the program also on a 0.99/0.01 ratio value and it returned from the three cheapest workflows, the three fastest ones. This is an optimal result.

Another interesting result is when once again a 0/1 ratio value is introduced by the user, in this case when maximum priority was given to time efficiency. In this case the best workflow recommendation was a 287.14s, running on 10 nodes on every FB thus costing a total of 6$/h. However when we run the algorithm with a 0.01/0.99 input ratio value, the first two pipelines swap places in the ranking, and now the 5.60$/h workflow that takes 283.38s, is the best one, taking just 0.24s more to execute than the 6$/h option. This also makes lots of sense when we think about it in real life terms, paying less 0.40$/h while taking just 0.24s more to execute seems pretty reasonable.

To sum up, in order to get the best practical results we would not recommend running the algorithm for ratio values of 0/1 or 1/0, at least for this case scenario. Although algorithm still works and returns exactly what you can expect.

In order to evaluate the algorithm for this run, the average precision for different values of K were calculated Table 4.5.

The mean of all those values are in Table 4.6. MAP@K, with K taking values from 1 to 3,

Table 4.5: Calculation process of AP@K for test 1 results.

Ratio value		P			AP@K		
cost	time	P@1	P@2	P@3	AP@1	AP@2	AP@3
0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
0.01	0.99	1.00	1.00	1.00	1.00	1.00	1.00
0.10	0.90	1.00	1.00	1.00	1.00	1.00	1.00
0.20	0.80	1.00	1.00	1.00	1.00	1.00	1.00
0.30	0.70	1.00	1.00	1.00	1.00	1.00	1.00
0.40	0.60	1.00	1.00	1.00	1.00	1.00	1.00
0.50	0.50	1.00	1.00	1.00	1.00	1.00	1.00
0.60	0.40	1.00	1.00	1.00	1.00	1.00	1.00
0.70	0.30	1.00	1.00	1.00	1.00	1.00	1.00
0.80	0.20	1.00	1.00	1.00	1.00	1.00	1.00
0.90	0.10	1.00	1.00	1.00	1.00	1.00	1.00
0.99	0.01	1.00	1.00	1.00	1.00	1.00	1.00
1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

test 1 had a good value of 0.923. However, considering that ambiguous situation, as it doesn't happen because of a problem with a system, then if the last case is ignored then we can consider a MAP@K equal to 1, for every value of K.

Table 4.6: Test 1 MAP@K for different K values.

MAP@1	MAP@2	MAP@3
0.923	0.923	0.923

The second test was executed with the same samples of test 1 but using FBs weights calculated through Local Normalization (LN). The resulting weights are shown in Table 4.7.

Comparing the weight values resulting from the LN method and comparing them with the GN method, cost efficiency (CE) weights are actually the same. This happens because the maximum and minimum values for each stage are the same for every other stage. The big difference is on time efficiency (TE) values. With LN, blocks such *BTO_10_Nodes, PK_10_Nodes, OPRJ_10_Nodes* all have the same TE weight value. This happens because all of them are the fastest option within its stages, although they all have quite some difference on execution time. Another huge difference between the results from GN, is the values of TE for the FBs with 2 nodes. While with GN only the 2 slowest ones within the whole 24 pool were having values under 0.20. 5 out of the 6 FBs with 2 nodes got under 0.20 of TE in the LN method.

Running the same 13 samples with these new weight values, the following results were obtained. See Table 4.8.

So from this test 2 we can conclude immediately that the weight estimation using global normalization worked way better than local normalization. This was already expected. Despite the

Table 4.7: Weight values for the 24 function blocks after using the Local Normalization method.

		# Nodes							
Stage	Algorithm	2		4		8		10	
		CE	TE	CE	TE	CE	TE	CE	TE
1	BTO	1.00	0.00	0.75	0.62	0.25	0.93	0.00	1.00
	OPTO	1.00	0.15	0.75	0.71	0.25	0.90	0.00	0.92
2	BK	1.00	0.00	0.75	0.63	0.25	0.91	0.00	0.97
	PK	1.00	0.12	0.75	0.70	0.25	0.94	0.00	1.00
3	BRJ	1.00	0.00	0.75	0.47	0.25	0.75	0.00	0.78
	OPRJ	1.00	0.80	0.75	0.92	0.25	1.00	0.00	1.00

fact that LN may work in cases where there is more independence between blocks of different categories, LN is not the best approach for this case. The most reasonable explanation for this results is the fact that all FBs execution time have the same degree of magnitude (couple of hundreds of seconds) then performance should be better when they are normalized globally between them. In a case where different categories blocks would have significant difference on the degree of magnitude on the execution times, for instance seconds vs hours, then LN normalization would make more sense than in this particular case.

The same situation happened for the last sample, having returned the same exact three workflows in the same order. A particular result from the this table is the workflow suggested in the third place for the first sample that had a *0* returned instead of a workflow execution time on the colored column, but first let's understand better what the values of the *Best option chosen* column actually mean. The values on the columns of green and red are the values of execution time for the workflow that should have been returned with the same cost as the one recommended in that place. For instance, in sample 2, the third option was a 5.6$/h workflow with an executing time of 306.94s. This was returned as a non-relevant result since the second best workflow for that cost is the one with an execution time of 294.97s as it is displayed in our top-3 pipelines Table 4.2. As the best 5.6$/h pipeline had already been correctly chosen as the first pipeline, then the next with the same cost should be the one written in that column, with 294.97s of execution time, instead of the one returned, with 306.94s execution time. So now that the values from *Best option chosen* column are explained, let's have a look at the curious third pipeline recommended for the first sample. The workflow is a 6$/h workflow that takes 306.7s to execute. Checking Table 4.2 once again we can conclude that this result is non-relevant for two reasons. First and easiest one is the fact that the recommended pipeline is the third best option for the cost of 6$/h, and so since only one pipeline of that same cost had already been correctly recommended as the best option, then the second 6$/h pipeline to be recommended should be the 295.92s one. But why does it have a zero instead of this value? This leads us to the second reason why it is a non-relevant case. As also seen in the table the 2nd and 3rd fastest pipelines for the maximum cost shall never be recommended, since there are at least 3 faster workflows with the same or less cost, as already explained previously when explaining the meaning of the white colored values of that table. The *0* value on the results table mean that no more pipelines with that cost should be returned given

Table 4.8: Results for the test 2, using LN for weight estimation.

Ratio value		Recommendation						Best option chosen?		
		1st pipeline		2nd pipeline		3rd pipeline				
cost	time	cost ($/h)	time (s)	cost ($/h)	time (s)	cost ($/h)	time (s)	Green - YES I Red - NO		
0.00	1.00	6.00	287.14	5.60	287.38	6.00	306.70	🟩 287.14	🟩 287.38	🟥 0.00
0.01	0.99	5.60	287.38	6.00	287.14	5.60	306.94	🟩 287.38	🟩 287.14	🟥 294.97
0.10	0.90	5.60	287.38	4.80	303.35	6.00	287.14	🟩 287.38	🟩 303.35	🟩 287.14
0.20	0.80	4.40	337.22	4.80	303.35	4.00	345.05	🟥 311.18	🟩 303.35	🟥 333.97
0.30	0.70	4.00	345.05	3.20	368.56	3.60	367.84	🟥 333.97	🟥 357.48	🟥 334.69
0.40	0.60	2.00	542.94	2.40	520.15	2.00	583.55	🟩 542.94	🟥 451.38	🟥 553.09
0.50	0.50	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.60	0.40	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.70	0.30	1.20	955.01	1.60	602.97	1.60	894.98	🟩 955.01	🟩 602.97	🟥 619.56
0.80	0.20	1.20	955.01	1.20	1025.89	1.20	971.60	🟩 955.01	🟥 971.60	🟥 1025.89
0.90	0.10	1.20	955.01	1.20	1025.89	1.20	971.60	🟩 955.01	🟥 971.60	🟥 1025.89
0.99	0.01	1.20	955.01	1.20	1025.89	1.20	971.60	🟩 955.01	🟥 971.60	🟥 1025.89
1.00	0.00	1.20	1042.48	1.20	1025.89	1.20	955.01	🟥 955.01	🟥 971.60	🟥 1025.89

the already recommended pipelines, which keeps coherent with the with our reasoning.

Now that the results have been qualitatively analyzed, Table 4.9 shows the calculations made for the average precision at different K values for every sample.

Table 4.9: Calculation process of AP@K for test 2 results

Ratio value		P			AP@K		
cost	time	P@1	P@2	P@3	AP@1	AP@2	AP@3
0.00	1.00	1.00	1.00	0.67	1.00	1.00	0.67
0.01	0.99	1.00	1.00	0.67	1.00	1.00	0.67
0.10	0.90	1.00	1.00	1.00	1.00	1.00	1.00
0.20	0.80	0.00	0.50	0.33	0.00	0.25	0.17
0.30	0.70	0.00	0.00	0.00	0.00	0.00	0.00
0.40	0.60	1.00	0.50	0.33	1.00	0.50	0.33
0.50	0.50	1.00	0.50	0.33	1.00	0.50	0.33
0.60	0.40	1.00	0.50	0.33	1.00	0.50	0.33
0.70	0.30	1.00	1.00	0.67	1.00	1.00	0.67
0.80	0.20	1.00	0.50	0.33	1.00	0.50	0.33
0.90	0.10	1.00	0.50	0.33	1.00	0.50	0.33
0.99	0.01	1.00	0.50	0.33	1.00	0.50	0.33
1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

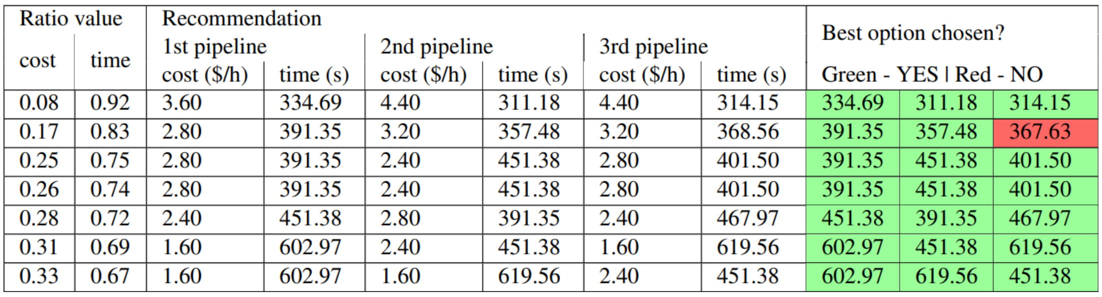
From Table 4.10 we can see the values for MAP@K with different K values. It's quite noticeable that results from test 2 really aren't really the best, with MAP@K for K=3 dropping to 0.397. Even if we don't take in consideration the ambiguous sample of 1/0 ratio value for cost/time efficiency input, this value goes up to only 0.431. So we can conclude that the choice of weight values can really make a huge difference on the performance of our recommending system.

Table 4.10: Test 2 MAP@K for different K values.

MAP@K		
MAP@1	MAP@2	MAP@3
0.769	0.558	0.397

With two tests running with 13 samples some conclusions were already made about our system performance. Weight optimization is an important key factor since tweaking those values really make a difference on the results, and so this topic will be better addressed in the next section 4.3. Running our system for 13 samples gave us some information on its performance but how does the system behaves in terms of performance for more than 13 cases. To test the scalability of the system, two more tests were run. We used a normal distribution, with mean equal to 0.5 and standard deviation of 0.145, to generate 99 random values between 0 and 1 in order to test our system for several different cases, using both weights from the GN and LN approaches that were used for tests 1 and 2. Note that some values for the input ratio are repeated and so those were removed from the results. Also, different sets of values for inputs were generated for tests 3 and 4. So the resultant number of unique testing samples was 47 for both test 3 and test 4. In the appendix we can find the results of both test 3 and test 4 in the Appendix A.1 and A.2, respectively.

Once again our recommending system performed very nicely for the GN weights optimization. In a total of 141 workflows recommended, it returned only one non-optimal solution, the third pipeline for the second sample. Part of the results table is shown in figure 4.3, it order to analyze the one wrong case. So two 3.2$/h workflow were returned in the last two recommendations. The first one is a correct solution but the second one was a 368.56s workflow instead of the optimal 367.63s option. Looking once again at Table 4.2, the 368.56s is the third fastest workflow for a cost of 3.2$/h and shouldn't be returned before the 367.63s pipeline which is the second best for the current cost. Although the difference of time is almost non-significant, 0.93s this can actually be explained. In order to keep coherence between the recommending system and the weight optimization based on genetic algorithm program, all the weights, and performance requirements ratio values were all used with two decimal digits. This avoids getting different score results for the exact same cases between the two frameworks. This ends up limiting the capacity of our recommending system. In this example the 368.56s workflow is composed by the FBs *OPTO_4_Nodes, PK_8_Nodes, OPRJ_4_Nodes*. The optimal solution for this case, would be the 367.63s pipeline with the blocks *BTO_4_Nodes, PK_10_Nodes, OPRJ_2_Nodes*. The score calculated by the recommending algorithm are done by doing the sum of the dot product between the two weights of each block and the two ratio values. These two workflows had the exact same score when rounding to two decimal digits, *2.56*. So if weights and scores were all calculated using more than two decimal digits could be the difference between having the best workflow at all times and sometimes be recommended a non-optimal solution. This is also why when generating 100 different samples, around 50 of them were repeated. If we used more than two decimals digits then much more samples could be run without repeating.



Ratio value		Recommendation						Best option chosen?		
		1st pipeline		2nd pipeline		3rd pipeline		Green - YES | Red - NO		
cost	time	cost ($/h)	time (s)	cost ($/h)	time (s)	cost ($/h)	time (s)			
0.08	0.92	3.60	334.69	4.40	311.18	4.40	314.15	🟩 334.69	🟩 311.18	🟩 314.15
0.17	0.83	2.80	391.35	3.20	357.48	3.20	368.56	🟩 391.35	🟩 357.48	🟥 367.63
0.25	0.75	2.80	391.35	2.40	451.38	2.80	401.50	🟩 391.35	🟩 451.38	🟩 401.50
0.26	0.74	2.80	391.35	2.40	451.38	2.80	401.50	🟩 391.35	🟩 451.38	🟩 401.50
0.28	0.72	2.40	451.38	2.80	391.35	2.40	467.97	🟩 451.38	🟩 391.35	🟩 467.97
0.31	0.69	1.60	602.97	2.40	451.38	1.60	619.56	🟩 602.97	🟩 451.38	🟩 619.56
0.33	0.67	1.60	602.97	1.60	619.56	2.40	451.38	🟩 602.97	🟩 619.56	🟩 451.38

Figure 4.3: Small portion of results of test 3.

As expected for test 4, it recommended several non-relevant workflows, concluding once again that weight estimation is very important in order to get good results. To evaluate quantitatively the system performance on both tests, it's presented in Table 4.11 the MAP@K for test 3 and in Table 4.12 the same metric for test 4.

Table 4.11: MAP@K evaluation for test 3.

MAP@K		
MAP@1	MAP@2	MAP@3
1.000	1.000	0.993

Table 4.12: MAP@K evaluation for test 4.

MAP@K		
MAP@1	MAP@2	MAP@3
0.830	0.484	0.346

Comparing the MAP@K values of tests that used the same weights we can verify that they are pretty close. So the test 1 MAP@3 is 0.923 and with the same weights on test 3 inputs is equal to 0.993. Between test 2 and 4, that also used the same weights, the MAP@3 also has no significant difference being 0.397 and 0.346, respectively. The values for K equal to 1 and 2 also didn't differ too much.

Given that, we can conclude that our system is expandable to every pair of values a user decides to use. Therefore, it doesn't matter which performance requirement the user gives more importance the recommendation system will always perform equally. Also having more samples we conclude that for optimal weights estimation, the recommendation system gives almost perfect results for every recommendation with the MAP@3 of 0.993.

In order to help visualizing the different approaches and how they affect the system's performance, the graphic in figure 4.4 shows the MAP@K comparison for the 4 different tests.

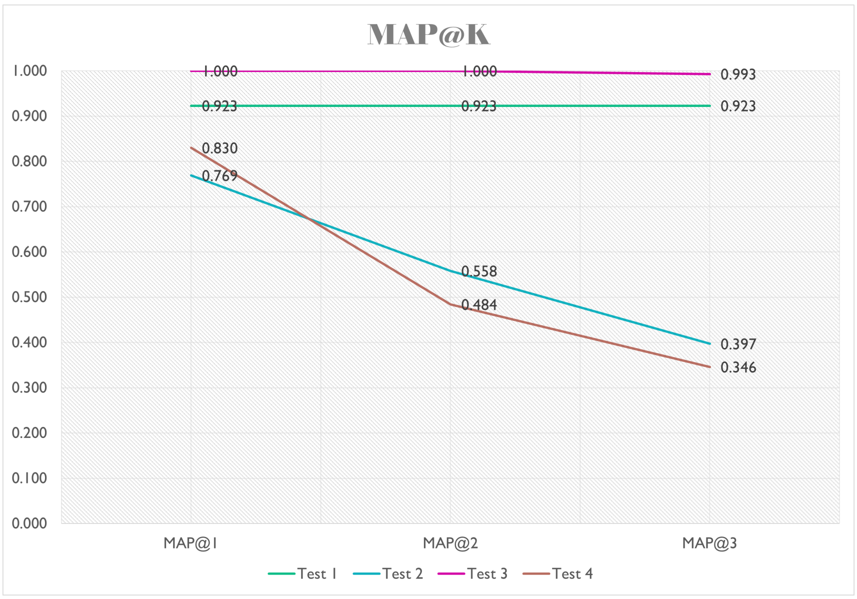


Figure 4.4: MAP@K comparison between tests 1, 2, 3, and 4.

It's very noticeable the coherency on the values between the tests with the same weights and different samples, showing the expandability feature. Apart from that, it is also very clear the two tests with global normalization with way better precision than the two tests with local normalization.

4.3 Genetic algorithm as weights estimator

In this section we introduce test results on the integration of both the genetic algorithm and recommendation system, doing a completely different strategy on the problem interpretation. We take some conclusions on how the genetic algorithm allows the user to estimate the weights automatically.

So as concluded several times already, weights estimation is a crucial procedure to guarantee the best results of the recommendation system. In the previous section we made this estimation using two different approaches. Both approaches were done manually and required time. For this case scenario it wasn't a big deal since we are dealing with just 24 FBs. Also, the testing case included the Table 4.1 allowed the process to be much easier and the calculations to be linear. Several times in real-life scenarios, the user may not have such well structured and detailed information about how well each function block performs in the two user-defined metrics. Therefore, the weight estimation procedure may get very tricky, hard, and time-consuming to do, specially if we speak about a way larger number of FBs.

Thereby, we propose to use the implemented genetic algorithm (GA) in order to perform this task based on history logs of workflows. So if the user has available a set of workflows recommendations historic based on the two defined metrics, but not the information about any function block, then he can use the performance requirements as input to the GA and expect a set of weights to be returned, based on the workflows used for each case. This weights will describe the performance of the function blocks, and will be ready to be used in the recommendation system.

To test this strategy, we used some of the tests that were already done, from the previous section, and used the results as inputs on the GA. The idea is to use the weights returned by the GA, use them as the FBs weights on the function blocks database, and run the same scenarios once again on the recommendation and compare if the results become similar to the ones done before.

So in first place we will use the results from test 1, shown in Table 4.4, as inputs to the GA. First things first, we need to remember that the genetic algorithm takes as input a set of values that represent the ratio value of cost/time efficiency, and as the target the recommended workflow. The first property about the GA is that it estimates weight values for FBs of a specific stage at a time. Thus, the target for each scenario is the function block chosen in that scenario, for the stage of the blocks that are being optimized. Since we have blocks belonging to 3 different stages, the GA will have to be run 3 times. The second note on the implemented GA is that it only takes one workflow as the target. Since on the test 1 results for each scenario, three workflows were recommended, we will chose the first ones as the input of our GA. This is due to the fact that GA always optimizes the weight values according to the best possible workflow for each case, as explained previously in chapter 3.

The first thing to do is to get the top 1 workflow for each cost/time ratio value from test 1 results. These workflows are shown in Table 4.13. It shows a set of workflows based on performance metrics, composed by different FBs for different stages. This is the information the user needs to run the GA in order to get the best weight optimization based on this information.

Table 4.13: Top 1 workflow composition, recommended in test 1.

Ratio value		Benchmark pipeline		
cost	time	Stage 1 FB	Stage 2 FB	Stage 3 FB
0.00	1.00	*BTO_10_Nodes*	*PK_10_Nodes*	*OPRJ_10_Nodes*
0.01	0.99	*BTO_10_Nodes*	*PK_10_Nodes*	*OPRJ_8_Nodes*
0.10	0.90	*OPTO_4_Nodes*	*PK_10_Nodes*	*OPRJ_4_Nodes*
0.20	0.80	*OPTO_4_Nodes*	*PK_8_Nodes*	*OPRJ_2_Nodes*
0.30	0.70	*OPTO_2_Nodes*	*PK_8_Nodes*	*OPRJ_2_Nodes*
0.40	0.60	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.50	0.50	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.60	0.40	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.70	0.30	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.80	0.20	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.90	0.10	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.99	0.01	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
1.00	0.00	*BTO_2_Nodes*	*BK_2_Nodes*	*OPRJ_2_Nodes*

So now that we have the history log of workflows for each scenario, we use this information to run the genetic algorithm. For each run, the number of iterations, the execution time, the average fitness of the population, and the number of errors were analyzed. For the first run, we estimated the weights for the FBs of stage 1. For each population a total 500 solutions were used in each one. Each iteration will pool the 100 best solutions as parents. A mutation rate is 40%. A maximum of 2000 generations was defined, but the algorithm breaks if the best solution fitness value doesn't improve for 100 generations in a row and the best solution has no more than 0 errors compared to the target block choice. Figure 4.5 represents the evolution of the average fitness of the population as well as the fitness value of the current best solution in the population along the different iterations. The results on the graphic show a negative exponential tendency on the average fitness throughout the generations. A reminder on the fact that this fitness value actually represents an error associated to the solutions, and so it is minimized, working more as a cost function. In this case we can tell that the algorithm found an optimal solution right after the 50th generation, by the sharp fall on the blue line.

In figure 4.6 is represented a histogram of the distribution of the weights values along the whole last population, separated in intervals of 0.1. We can tell that most of weights rely on values close to 1 and 0, with a bit more density in low values close to 0.

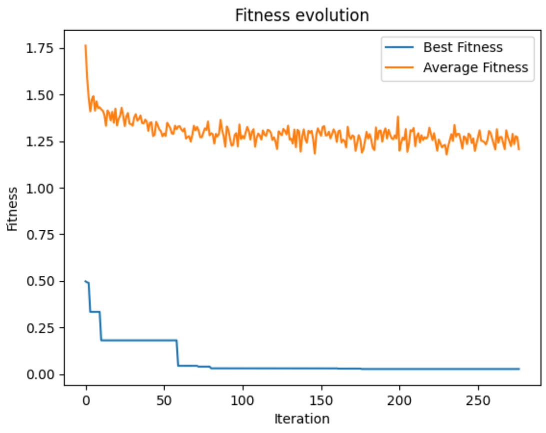


Figure 4.5: Graphic of the average population and best solution fitness evolution through generation iterations for stage 1.

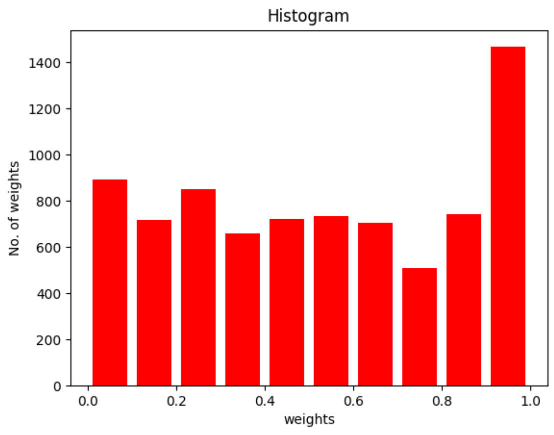


Figure 4.6: Histogram with the weights distribution on the last population for stage 1.

Figure 4.7 and figure 4.8 show the results for the stage 2 blocks weight training. They are pretty similar to the stage 1 estimation., though a bit slower.

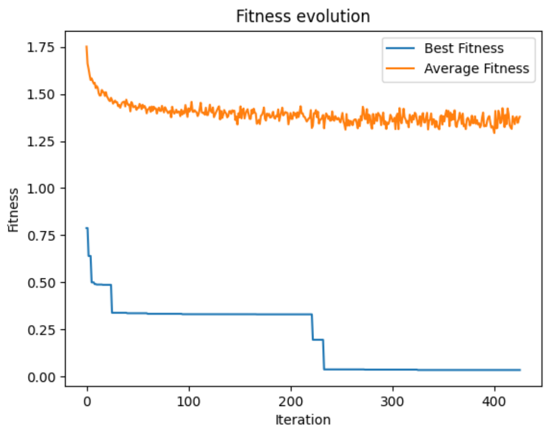


Figure 4.7: Graphic of the average population and best solution fitness evolution through generation iterations for stage 2.

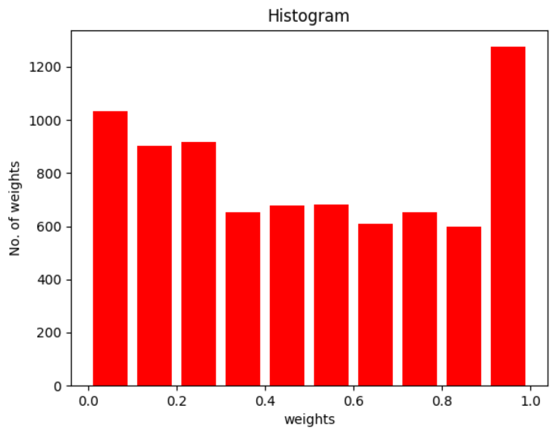


Figure 4.8: Histogram with the weights distribution on the last population for stage 2.

Last but not least, for the third and last run on the GA, blocks from stage 3 were estimated by the GA. Once again the results shown in figures 4.9 and 4.10 show an identical look of the previous two cases. Although it's noticeable on the histogram there more weights between 0 and 0.1 than the usual. The average execution time for the 3 runs of the genetic algorithm was 48.24 seconds.

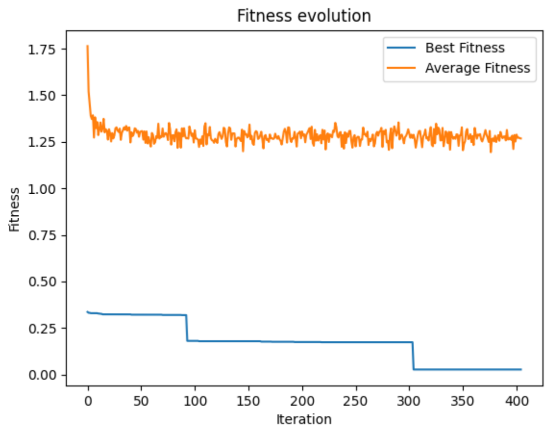


Figure 4.9: Graphic of the average population and best solution fitness evolution through generation iterations for stage 3.

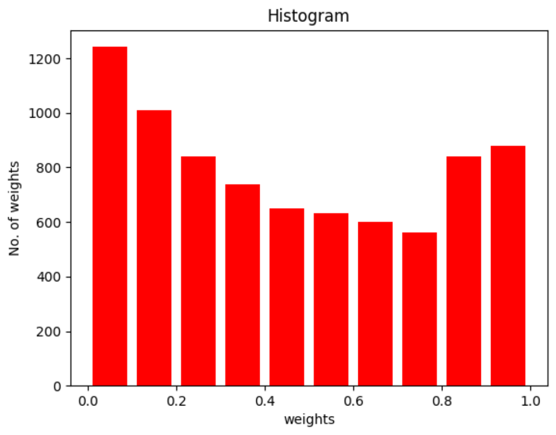


Figure 4.10: Histogram with the weights distribution on the last population for stage 2.

The estimated weights for each one of the 24 function block are available at Table 4.14. Some interesting details can be concluded from this values. If we compare this table with 4.3, which is the set of weights used in test 1 whose results were the reference for the estimation of these new weights, there's a correlation of similar values on FBs that appear the most on the results from Table 4.13. This is what we could expect, since the genetic algorithm relied on these FBs as the reference to assign weights and if some FBs were not chosen, then the tendency might be to have some off values since there was no reference for them in first place. So a solution for this problem would be using more samples to estimate values in the GA.

Table 4.14: Weights generated by the GA for each block based on test 1.

		# Nodes							
Stage	Algorithm	2		4		8		10	
		CE	TE	CE	TE	CE	TE	CE	TE
1	BTO	1.00	0.00	0.27	0.27	0.28	0.01	0.06	1.00
	OPTO	1.00	0.87	0.62	0.99	0.03	0.31	0.27	0.33
2	BK	1.00	0.13	0.98	0.07	0.00	0.41	0.00	0.22
	PK	1.00	0.24	0.73	0.76	0.38	0.97	0.15	1.00
3	BRJ	0.27	0.42	0.42	0.27	0.00	0.15	0.23	0.24
	OPRJ	1.00	0.81	0.49	0.87	0.09	0.88	0.03	0.88

Having these weights, to test the performance of the GA let's compare the results of running this new estimated weights on the recommending system with the results of test 1, using the exact same input values. The expected would be to have the same exact workflows for the top-1 options, since these were the ones used as a reference, but the 2nd and 3rd options may have interesting results as well. The results of the recommendation system are listed in Table 4.15. MAP@K is shown in Table 4.16

From this results we can take some conclusions about the performance of the GA. As expected the first pipelines recommended match perfectly the first pipelines from test 1 in 4.4. The second pipelines were the exact same for the first 3 cases and the last one. In the third pipeline column we

Table 4.15: Results for the test 5, using weights estimated by the GA.

Ratio value		Recommendation						Best option chosen?		
		1st pipeline		2nd pipeline		3rd pipeline				
cost	time	cost ($/h)	time (s)	cost ($/h)	time (s)	cost ($/h)	time (s)	Green - YES | Red - NO		
0.00	1.00	6.00	287.14	5.60	287.38	4.80	318.48	🟩 287.14	🟩 287.38	🟥 303.35
0.01	0.99	5.60	287.38	6.00	287.14	4.40	318.72	🟩 287.38	🟩 287.14	🟥 311.18
0.10	0.90	3.60	334.69	3.20	357.48	3.20	368.56	🟩 334.69	🟩 357.48	🟥 367.63
0.20	0.80	2.80	391.35	2.40	451.38	3.20	357.48	🟩 391.35	🟩 451.38	🟩 357.48
0.30	0.70	2.40	451.38	2.80	391.35	1.60	602.97	🟩 451.38	🟩 391.35	🟩 602.97
0.40	0.60	1.60	602.97	2.40	451.38	2.00	542.94	🟩 602.97	🟩 451.38	🟩 542.94
0.50	0.50	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.60	0.40	1.60	602.97	1.20	955.01	1.20	1025.89	🟩 602.97	🟩 955.01	🟥 971.60
0.70	0.30	1.20	955.01	1.20	1025.89	1.60	602.97	🟩 955.01	🟥 971.60	🟩 602.97
0.80	0.20	1.20	955.01	1.20	1025.89	1.60	643.58	🟩 955.01	🟥 971.60	🟥 602.97
0.90	0.10	1.20	955.01	1.20	1025.89	1.60	643.58	🟩 955.01	🟥 971.60	🟥 602.97
0.99	0.01	1.20	955.01	1.20	1025.89	1.20	971.60	🟩 955.01	🟥 971.60	🟥 1025.89
1.00	0.00	1.20	1042.48	1.20	1025.89	1.20	955.01	🟥 955.01	🟥 971.60	🟥 1025.89

see that only two of them matched the test 1 results. So for the first-ranked workflow it matched in 100% of the cases. The overall result, 19 out of the total 39 workflows recommendations matched between the two tests which means a percentage of 48.72%. In terms of performance of these weights, the evaluation metric shows us a MAP@3 of 0.632. We can then conclude that if the user has no information on the FBs performance for defined metrics, that he would be able to set up decent weight values in order to use on the recommendation system. The results from the GA are quite impressive. Noting that a MAP@1 of 0.923, and it's not 1 because of the ambiguous last scenario, actually represents a good performance. Knowing that the GA estimates and normalizes weights within categories, it actually does something somewhat like the LN method used in section 4.2. Looking at the performance metrics of the LN approach, Table 4.10, we got a MAP@3 of 0.397. So, even though the GA uses a bit of the theory behind the Linear Normalization method used in the first tests to the recommending system, it performs way better. Moreover, it's all done automatically taking just a few minutes to estimate all the weight values.

The next step is to test the expandability of our genetic algorithm and understand how well it behaves when it receives a larger dataset of history log data. We have concluded that due to lack of samples, some blocks weren't assigned with the best weight estimations by the GA. So, to test this, we are using the top 1 recommendations of test 3 results from Table A.1 as the input references for the GA. The workflows of each sample are shown in Table A.3.

So using these workflows as input on our GA we estimate new weights for the FBs. Comparing the graphics in figures 4.11, 4.12, 4.13 with the ones from the previous runs of the GA, we notice a slight difference on the number of generations that it takes for the GA to converge to an optimal solution, unless in the case for stage 2. We also see a sharper exponential drop right in the first generations. This means that the genetic algorithm works better when he receives larger datasets as targets, since it converges to optimal solutions in less generations although taking more time because of the larger number of samples.

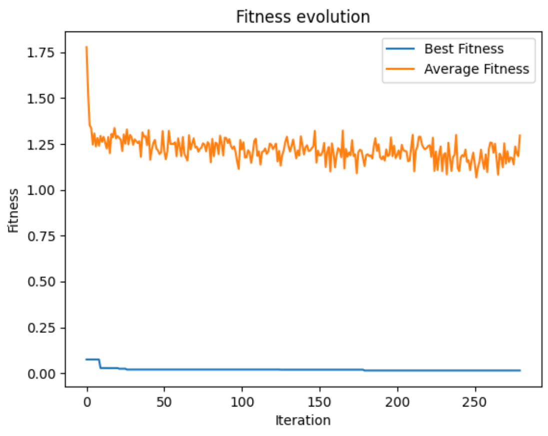


Figure 4.11: Graphic of the average population and best solution fitness evolution through generation iterations for stage 1.

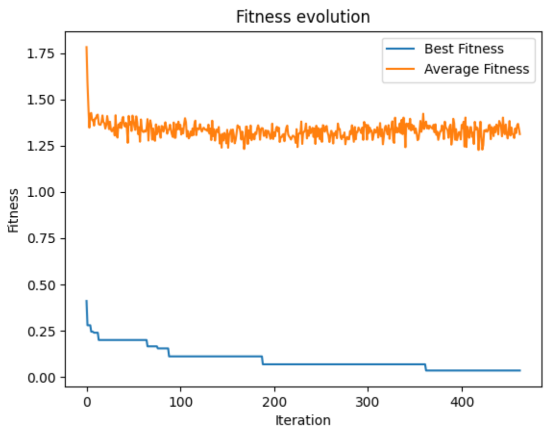


Figure 4.12: Graphic of the average population and best solution fitness evolution through generation iterations for stage 2.

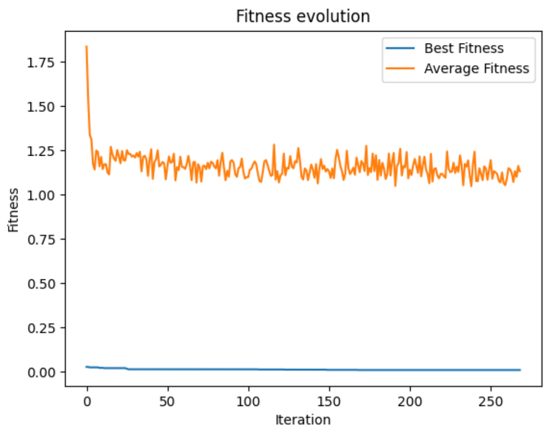


Figure 4.13: Graphic of the average population and best solution fitness evolution through generation iterations for stage 3.

Table 4.16: MAP@K evaluation for test 5.

MAP@K		
MAP@1	MAP@2	MAP@3
0.923	0.769	<u>0.632</u>

From the histograms 4.14, 4.15, 4.16 we don't notice much differences from the last examples apart from the fact that in stages 1 and 2 optimization, values between 0 and 0.1 seem to have been increased, which may mean that the GA now has better information on less used FBs. The average time of execution increased to 127.24 seconds.

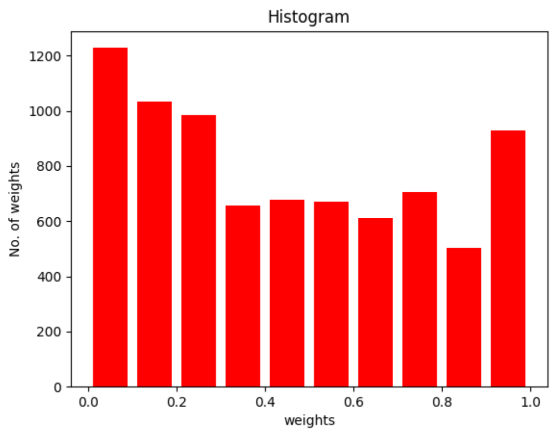


Figure 4.14: Histogram with the weights distribution on the last population for stage 1.

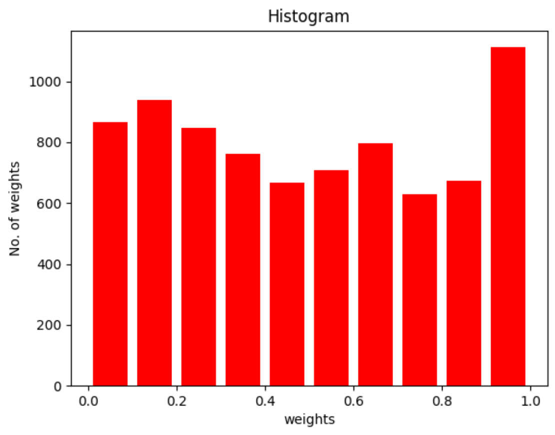


Figure 4.15: Histogram with the weights distribution on the last population for stage 2.

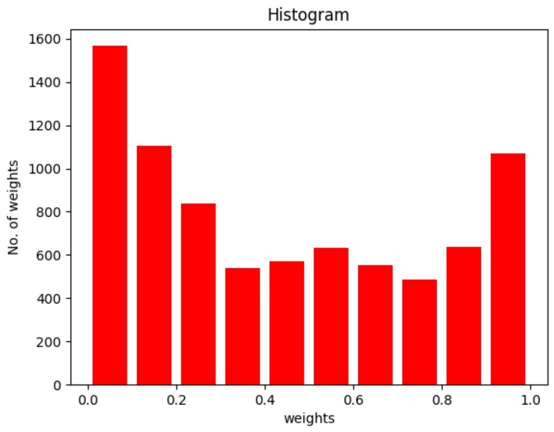


Figure 4.16: Histogram with the weights distribution on the last population for stage 3.

So new weight values were generated and are found in Table 4.17. Once again the most chosen FBs in test 3 results for top 1 workflows are the ones whose weight values are closer to the reference in Table 4.3. Running the recommending system with these weights for the same cases of test 3 returns us the results on Table A.4.

Table 4.17: Weights generated by the GA for each block based on test 3.

		# Nodes							
Stage	Algorithm	2		4		8		10	
		CE	TE	CE	TE	CE	TE	CE	TE
1	BTO	0.19	0.07	0.02	0.05	0.43	0.11	0.01	0.03
	OPTO	1.00	0.71	0.17	1.00	0.06	0.00	0.20	0.18
2	BK	0.09	0.04	0.37	0.08	0.66	0.20	0.32	0.58
	PK	1.00	0.04	0.61	0.84	0.43	0.92	0.00	1.00
3	BRJ	0.05	0.15	0.01	0.06	0.11	0.02	0.06	0.17
	OPRJ	0.91	0.92	0.10	1.00	0.08	0.00	0.02	0.00

The results are actually quite good. Every recommended workflow in the first and second pipeline is a relevant solution. For the third pipeline, only a third of the workflows are non-relevant solutions. With this results we can confirm that the larger the history log sample the user has access to use as input reference on the GA, the better the weights estimated by the GA are. This makes sense since as long the GA has its hyperparameters well tuned to converge, the more information it gathers, the better the quality of the weights assigned to each FB. Even though the solutions recommended on the second option are not all the same as from the reference in test 3, they still are optimal solutions. Quantitatively speaking, MAP@K for this last test is shown in Table 4.18. Maximum MAP at cutoff points 1 and 2 were obtained and an impressive 0.887 value for MAP@3.

To sum up this section, the problem was approached from a different perspective. Picturing what could be a real-life situation, we found an automatized and precised way of estimating weight

Table 4.18: MAP@K evaluation for test 6.

MAP@K		
MAP@1	MAP@2	MAP@3
1.000	1.000	0.887

values for our FBs. Doing it manually is subjected to some factors such as the problem interpretation from the user. In section 4.2 we showed two interpretations for the case study, following distinctive ways of weights estimation for the same FBs using the exact same information, although creating significantly different results performance wise. In order to solve that, we propose a GA that is ready to estimate a set of values for FBs within a category based on historical logs on workflows composed by those FBs, evaluated by the user-defined performance metrics. Despite the fact that the genetic algorithm itself uses a normalization process that only involves the FBs from the same category at a time, somewhat similar to the LN method shown in the previous section, its the results are vastly improved. And so, the genetic algorithm is a powerful tool that should be used alongside the main recommending algorithm.

In order to help visualizing the different approaches and how they affect the system's performance, the graphic in figure 4.17 shows the MAP@K comparison for the 4 different tests.

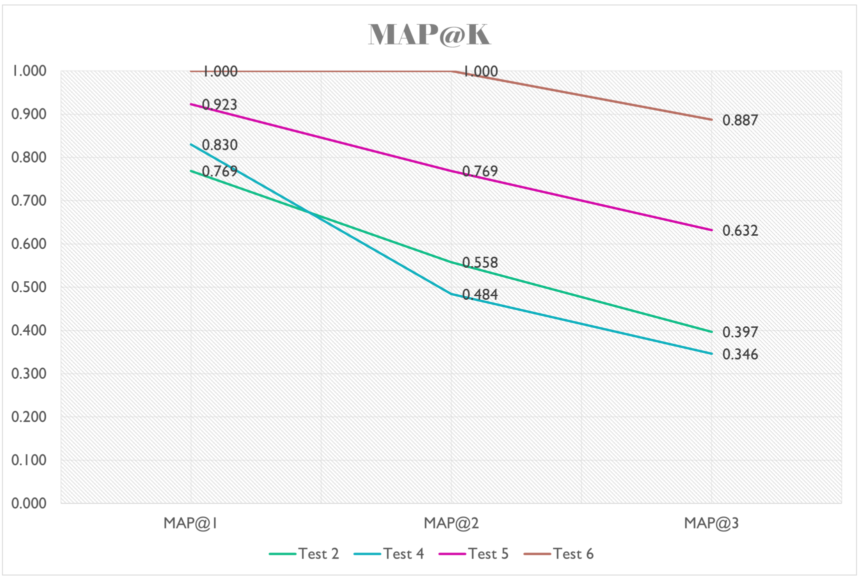


Figure 4.17: MAP@K comparison between tests 2, 4, 5, and 6.

The graphic shows clearly that enriched input datasets for the genetic algorithm creates better and more informed weights, with the increased precision values on test 6 over test 5. Comparing with tests 2 and 4, the automatized feature of the optimizer system shows better results than when doing it manually.

4.4 Genetic algorithm as weights optimizer

The final look into the two programs developed during this work has to do with weight optimization in specific situations. Firstly, we decided to use the recommendation system by calculating manually the FBs weights using information given about their performance on two defined metrics. This showed that the main recommending algorithm works pretty well, but its performance is dependent on the quality of the weights and how well they represent the performance of the different FBs. Knowing that we then focused on a way to automatize those weights estimation in order to try to get higher quality values while facilitating the process, avoiding human related errors upon that process. We proposed using the GA to generate those values, showing to be a powerful tool to use alongside the recommending system.

In this section we will propose to use the genetic algorithm as a tool to improve recommendation performance in specific situations. For instance, taking a look at the test 4 results, in Table A.2, we know that they are poor with a MAP@K of around 0.346. And we know as well that the explanation for those results is the fact that the approach of calculating the FBs values, LN was poor and resulted in bad recommendations. If we face this problem as if it was a real-life example, a user could have some troubles estimating weights, or not having enough data about the FBs performance. So the idea is to go through the recommendations on test 4, know which recommendations we want to have improving, and use the GA to generate new weight values in order to do that procedure.

First of all we know that the GA optimizes weights based on the best workflows for a given situation, therefore we will use the first pipeline recommendations from test 4 as a starting point. First we need to know how each of the top 1 recommended workflows are composed for each input ratio value. This information is described in Table A.5. From this list we know that from the third to the tenth recommended workflow were not optimal solutions, thereby we want to work on those.

The idea is to use these results, modify them on the non-optimal recommendations for the workflows we want, a use it in the GA to have better weight values. To calculate best workflow for each of the costs of those non-optimal recommendations we will resort once again to table 4.2. So for a 4.4$/h cost the best workflow has an execution time of 311.18s. For 4$/h the best option has a running time of 333.97s. Finally the 3.2$/h cost has the fastest workflow at 357.48$/h. Using the Simplex method explained previously and calculating which blocks will return us this workflows, Table A.5 has been updated with the new desired solutions. The updated version is in Table A.6. Note on the changes in referred cases.

So using the information on this table as inputs for the GA will return us new weight values. The GA allows us to add to the initial population an initial solution. Since we know the weights being used for the results in test 4, which are bad quality weights, we may use them as a way of perhaps help the GA to converge faster to an optimal solution. We will run on both conditions, with and without an initial user solution in order to compare the GA performance.

For the first stage optimization the GA struggled to find an optimal solution, thereby we ran the code once again with an initial good solution concluding that as a matter of fact, the GA turned into better solutions in shorter time. Figures 4.18 and 4.19 show the graphical differences on these two optimizing runs. As we can see from the blue line, the best fitness is way lower right from the beginning. In terms of fitness average, no significant differences were noted.

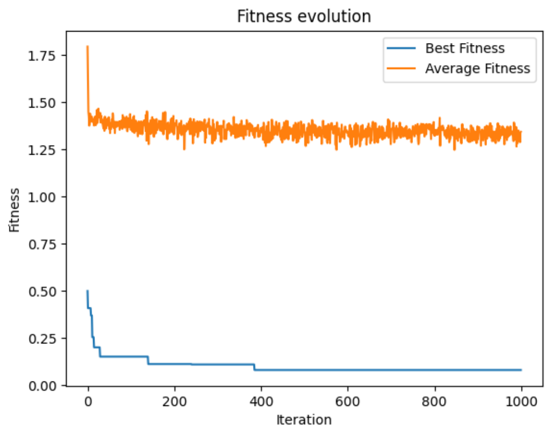


Figure 4.18: Graphic of the average population and best solution fitness evolution through generation iterations for stage 1, without initial solution.

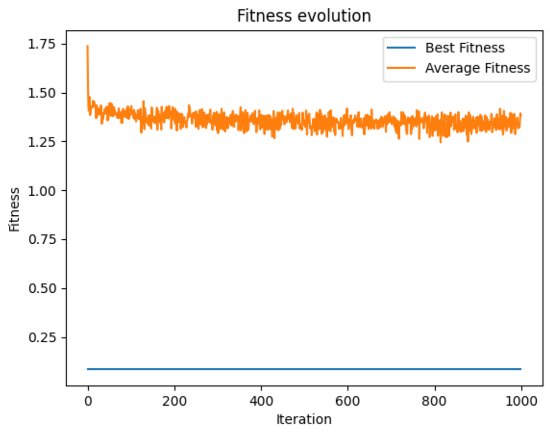


Figure 4.19: Graphic of the average population and best solution fitness evolution through generation iterations for stage 1, with initial solution.

In terms of weight distribution for the last population, we can compare the figures 4.20 and 4.21. The second case shows a significant denser area on the weights between 0 and 0.1, while the first case has bit more homogeneous distribution for the weights.

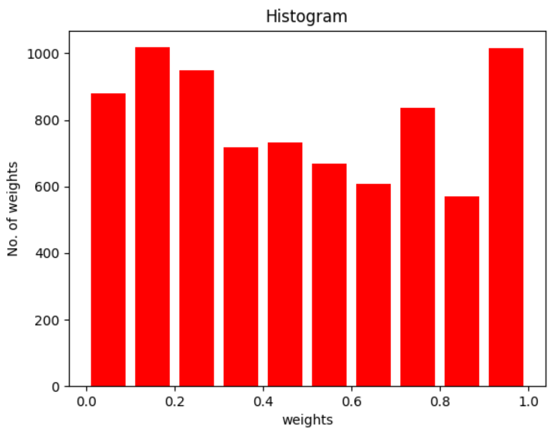


Figure 4.20: Histogram for the weights distribution on the last population for stage 1, with no initial solution.

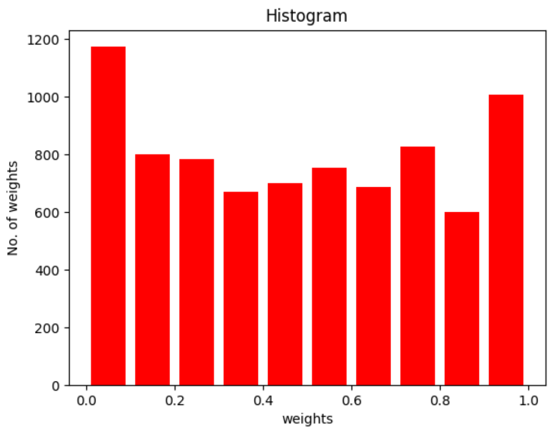


Figure 4.21: Histogram for the weights distribution on the last population for stage 1, with initial solution.

In the second stage of training, both approaches found it easier to calculate a good optimal solution, although an interesting note on the fact that the algorithm with no initial solution actually converged faster, around 78 seconds. This seems to be an outlier, with the first case reaching good solutions faster than expected. Also a sharper drop in the first case average population fitness is notable in the first couple of generations. These graphical differences are shown in figures 4.22 and 4.23.

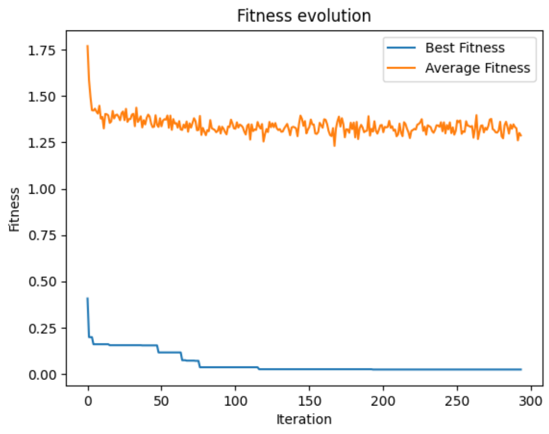


Figure 4.22: Graphic of the average population and best solution fitness evolution through generation iterations for stage 2, without initial solution.

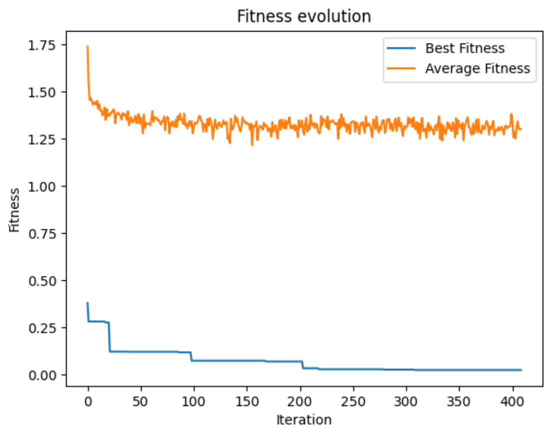


Figure 4.23: Graphic of the average population and best solution fitness evolution through generation iterations for stage 2, with initial solution.

The look on the weight distribution for the last population on both cases, we take a look at figures 4.24 and 4.25. No significant differences are seen. Plus, in both runs a lot of weights were between the 0.1 and 0.3 mark.

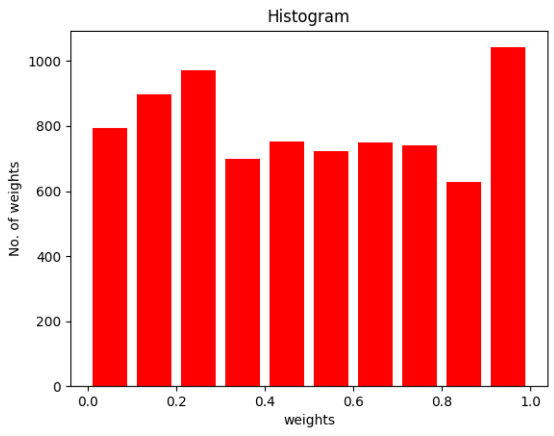


Figure 4.24: Histogram for the weights distribution on the last population for stage 2, with no initial solution.

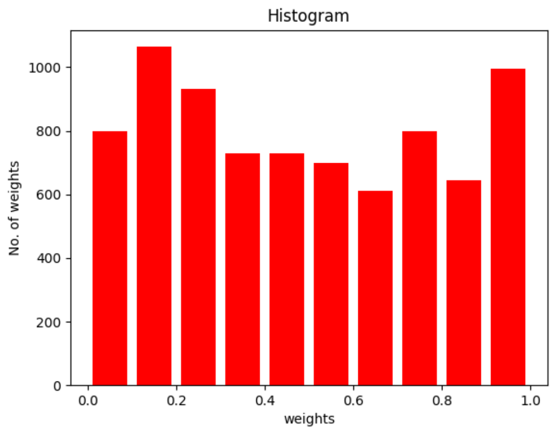


Figure 4.25: Histogram for the weights distribution on the last population for stage 2, with initial solution.

For the third and last stage of optimization, once again both approaches performed well to find an optimal solution. Giving it an initial solution made it almost 3 minutes faster, which is an incredible achievement. Figures 4.26 and 4.27 have the average fitness evolution of the two approaches. As seen, optimal solutions were found almots in a third of generations on the second case.

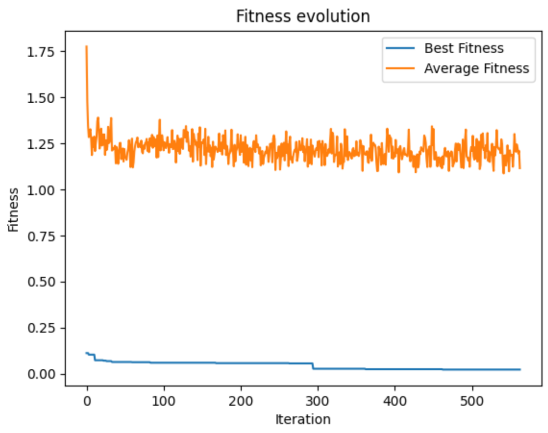


Figure 4.26: Graphic of the average population and best solution fitness evolution through generation iterations for stage 3, without initial solution.

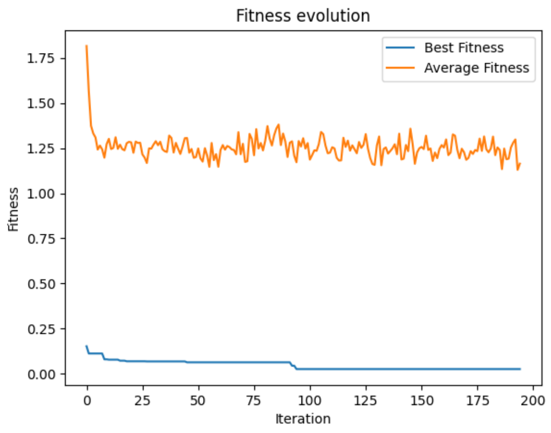


Figure 4.27: Graphic of the average population and best solution fitness evolution through generation iterations for stage 3, with initial solution.

As a matter of weight distribution in the last population, figures 4.28 and 4.29, shows some differences on the behaviour of the last population for the different cases. The second case has a better distribution between all the weights, opposing to the much more concentrated on low-value weights on the first case.

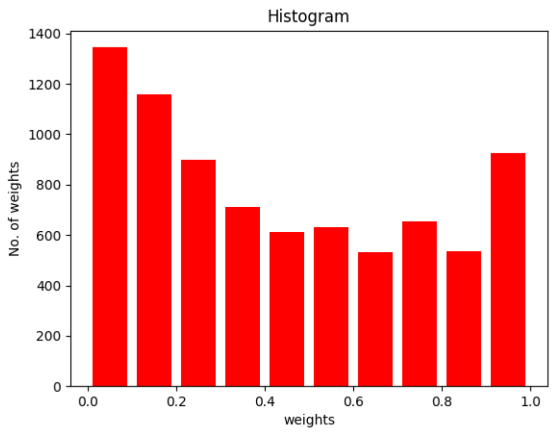


Figure 4.28: Histogram for the weights distribution on the last population for stage 3, with no initial solution.

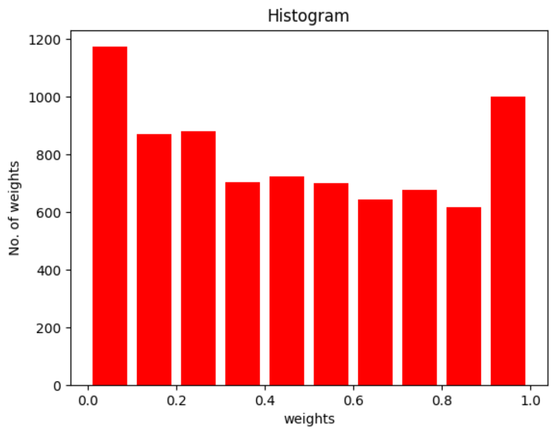


Figure 4.29: Histogram for the weights distribution on the last population for stage 3, with initial solution.

Summing up all the analysis on the two approaches on the GA, we concluded that most of the times introducing an already good but non-optimal solution, helps the algorithm to converge faster. The average time for the GA to find good weights was 284.45 seconds when no solution is given to the system and 257.16 seconds when it is given. This average should tend to increase with more testing, since for this case, we had an outlier result on stage two training. For this reason, we used the weights generated on runs where solutions were introduced into the initial population since they were, on average, faster.

These final weights are shown in Table 4.19.

Table 4.19: Weights generated by the GA when good solutions are introduced in the initial population. Reference values used are from test 4.

		# Nodes							
Stage	Algorithm	2		4		8		10	
		CE	TE	CE	TE	CE	TE	CE	TE
1	BTO	1.00	0.00	0.75	0.62	0.25	0.93	0.00	1.00
	OPTO	1.00	0.15	0.75	0.71	0.25	0.90	0.00	0.92
2	BK	0.32	0.07	0.13	0.04	0.00	0.63	0.46	0.13
	PK	1.00	0.23	0.80	0.69	0.04	0.01	0.19	1.00
3	BRJ	0.00	0.43	0.14	0.45	0.08	0.13	0.52	0.07
	OPRJ	1.00	0.83	0.39	0.99	0.09	1.00	0.27	0.03

To check if this strategy worked, these weights were used on the recommending system using the same input values as in test 4. The results are shown in Table A.7. These results show that there was a major improvement on the weights.

So by comparing the test 7 and test 4 results, in Table A.2, we may conclude that using the GA as a weight optimizer for certain case scenarios retrieves incredibly improved results. MAP@K values for test 7 are in Table 4.20, and we can immediately notice that the performance increased amazingly, with MAP@K having values of 0.979, 0.941, and 0.746 for cutoff points of K=1,2 and 3. So not only it increased the precision on the first recommended workflows for almost 100% of the cases, but the second and third pipelines recommended also had pretty amazing results. MAP@2 almost doubled compared to test 4, Table 4.12, while MAP@3 got an astonishing improvement for more than double.

Table 4.20: MAP@K evaluation for test 7.

MAP@K		
MAP@1	MAP@2	MAP@3
0.979	0.941	0.746

Therefore, even though we used optimal solutions just from the top 1 recommended workflows as reference to the GA, the overall precision of top 2 and top 3 recommendations improved as well.

In short, using the GA as a weight optimizer works very well. We used the logs of recommendations that used bad quality weight values, thus returning poor performance, in order to improve those weights for better ones, by using just the top 1 workflow recommendations and changing them to optimal workflows, as the input reference of our genetic algorithm. Also, explained two ways of using the GA, one using the test 4 weights as good initial solutions, since they had around 87% of the recommendations on top 1 workflows correctly, in order to help the GA to converge better and faster, and the other one just using the input references and let the algorithm itself try to find a solution. Once again the genetic algorithm proved to be a very strong tool to be used alongside our recommending system so as to not only estimate weight values automatically, but also as a weight optimizer tool.

Down below in figure 4.30 we can finally visualize better the increased performance when using the genetic algorithm to optimize a set of weights.

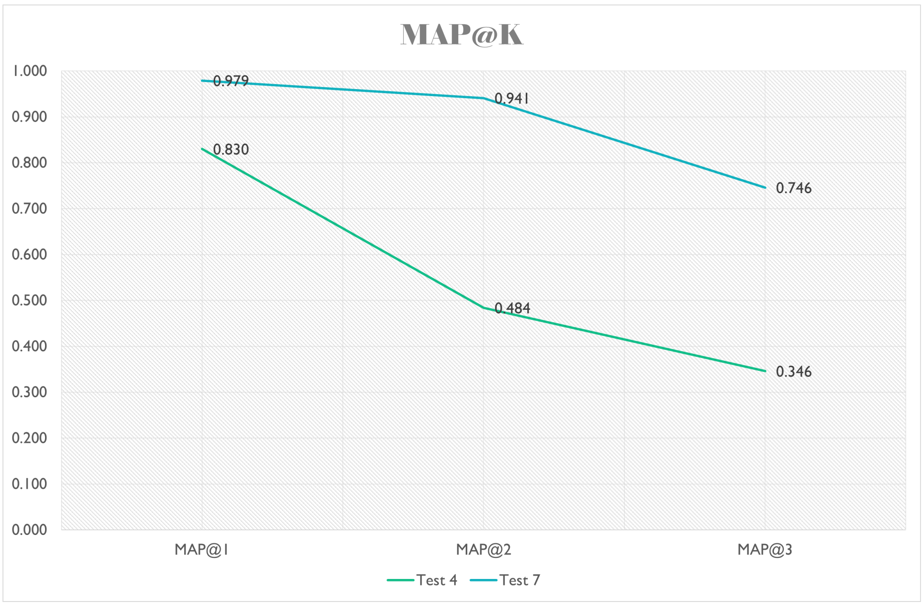


Figure 4.30: MAP@K comparison between tests 4 and 7.

Chapter 5

Conclusions and Future Work

Taking a look back on the main objective for this dissertation, in section 1.4, three main points were addressed: the importance of understanding the different requirements that define a workflow as a pipeline, the implementation of an adaptive recommending system that is able to calculate the best workflow and recommend it to the user, and knowing the best way to assess the system performance.

The tests performed in the previous chapter evaluated the proposed work in several manners. Both implemented systems were evaluated on their performance and efficiency, scalability, and capacity of working together. In order to calculate the system performance, the metric MAP@K was used, meeting then the third objective of the work.

A few tests were done on the main recommending system, in order to check how well it performs for different weights values. Then the performance with an expanded range of scenarios to check whether or not that performance would be untouchable. The following tests were made on the capacity of merging the two systems. Firstly we tested the weights estimation using the genetic algorithm so the user wouldn't need to rely solely on interpretation of a study case, and also avoiding calculating values manually. This estimation was done based on a set of history logs of workflows for different case scenarios of cost and time efficiency. The last couple of tests were about weight optimization using the genetic algorithm and some case scenarios where the recommendation system didn't return such precise results. All the results showed that the proposed work performs really well in different occasions and gives user the chance of using it in any situation involving workflow specification.

All the tests made in the previous chapter were based on the testing case using a dataset from a paper. The tests made could still be run on different datasets and case scenarios in order to have a better overview on the expandability of the proposed work.

Concerning the weight estimation/optimization, the genetic algorithm was not the first option to be thought. Since the time it may take to optimize weights in cases where there are a larger number of function blocks and/or specific requirements, execution time might be the bottleneck of the system. Deep learning techniques may be more efficient solutions. Using neural networks that use methods such as backpropagation to train weights could be used as an optimization method in the place of the genetic algorithm.

It was tried to implement a neural network in order to use the known backpropagation method to train the weights into giving better solutions. It had several problems of convergence to optimal solutions. A simpler genetic algorithm was implemented. Since it was retrieving good results, easier implementation and also expandability, it was chosen to be perfected as most as possible.

So the results were overall very positive, with a complete system used for workflow definition purposes, with the possibility of being applied in the most varied situations. Most of the cases studied during literature review shown in state-of-art worked their ways towards specific situations, specific workflows, and specific requirements. So creating the most general possible solution in order to be used in any situation was the priority of this work.

This way we may conclude the system does meet the first two objectives pointed out for this dissertation as the system showed to be adaptive for different scenarios, performed really well when recommending workflows, always with very good values for the best workflow recommendation, which was the main goal for the work, while meeting all the requirements given by the user, which was the first objective addressed.

As for future work, using the already talked strategy of using deep learning methods in order to estimate weights for function blocks based on history logs. Deep learning could improve efficiency of the weight optimization. On the other hand some tweaks could be made as well on the genetic algorithm. For instance, the current system is a bit limited in terms of working just for blocks of a specific requirement at a time. Another limiting property of the system is the fact that for each case of cost/time it uses only the best workflow recommendation as a reference for optimizing purposes. Making it more flexible in this measure would very likely produce much better results.

In terms of the recommending system, the first thing to be noticed would be the fact that it works only for two performance metrics. Although the implemented priority feature between two requirements of preference is very useful, the system could be more expandable on this direction. Some exploitation could be done as well on the scoring techniques. Even though this method showed very positive and hard to find results with the dot product calculation, some other methods could be explored in order to try to get higher performance.

Last but not least, some automatization could be done on the process of data exchange between the two systems. The proposed implementation has a separated structure of the two presented systems, although with some auto features such as the datasets generated by the recommendation system ready to be used by the weight optimization system.

Appendix A

Appendix

Table A.1: Results for the test 3, using GN for weight estimation.

Ratio value		Recommendation						Best option chosen?		
		1st pipeline		2nd pipeline		3rd pipeline				
cost	time	cost ($/h)	time (s)	cost ($/h)	time (s)	cost ($/h)	time (s)	Green - YES | Red - NO		
0.08	0.92	3.60	334.69	4.40	311.18	4.40	314.15	334.69	311.18	314.15
0.17	0.83	2.80	391.35	3.20	357.48	3.20	368.56	391.35	357.48	367.63
0.25	0.75	2.80	391.35	2.40	451.38	2.80	401.50	391.35	451.38	401.50
0.26	0.74	2.80	391.35	2.40	451.38	2.80	401.50	391.35	451.38	401.50
0.28	0.72	2.40	451.38	2.80	391.35	2.40	467.97	451.38	391.35	467.97
0.31	0.69	1.60	602.97	2.40	451.38	1.60	619.56	602.97	451.38	619.56
0.33	0.67	1.60	602.97	1.60	619.56	2.40	451.38	602.97	619.56	451.38
0.34	0.66	1.60	602.97	1.60	619.56	2.40	451.38	602.97	619.56	451.38
0.35	0.65	1.60	602.97	1.60	619.56	2.00	542.94	602.97	619.56	542.94
0.36	0.64	1.60	602.97	1.60	619.56	2.00	542.94	602.97	619.56	542.94
0.38	0.62	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.39	0.61	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.40	0.60	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.41	0.59	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.42	0.58	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.43	0.57	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.44	0.56	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.46	0.54	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.47	0.53	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.48	0.52	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.49	0.51	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.50	0.50	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.51	0.49	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.52	0.48	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.53	0.47	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.54	0.46	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.55	0.45	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.56	0.44	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.57	0.43	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.58	0.42	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.59	0.41	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.60	0.40	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.61	0.39	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.62	0.38	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.63	0.37	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.64	0.36	1.60	602.97	1.60	619.56	1.60	643.58	602.97	619.56	643.58
0.65	0.35	1.60	602.97	1.60	619.56	1.20	955.01	602.97	619.56	955.01
0.66	0.34	1.60	602.97	1.60	619.56	1.20	955.01	602.97	619.56	955.01
0.67	0.33	1.60	602.97	1.20	955.01	1.60	619.56	602.97	955.01	619.56
0.68	0.32	1.20	955.01	1.20	971.60	1.60	602.97	955.01	971.60	602.97
0.70	0.30	1.20	955.01	1.20	971.60	1.60	602.97	955.01	971.60	602.97
0.75	0.25	1.20	955.01	1.20	971.60	1.20	1025.89	955.01	971.60	1025.89
0.76	0.24	1.20	955.01	1.20	971.60	1.20	1025.89	955.01	971.60	1025.89
0.78	0.22	1.20	955.01	1.20	971.60	1.20	1025.89	955.01	971.60	1025.89
0.79	0.21	1.20	955.01	1.20	971.60	1.20	1025.89	955.01	971.60	1025.89
0.80	0.20	1.20	955.01	1.20	971.60	1.20	1025.89	955.01	971.60	1025.89
0.86	0.14	1.20	955.01	1.20	971.60	1.20	1025.89	955.01	971.60	1025.89

Table A.2: Results for the test 4, using LN for weight estimation.

Ratio value		Recommendation						Best option chosen?		
cost	time	1st pipeline		2nd pipeline		3rd pipeline				
		cost ($/h)	time (s)	cost ($/h)	time (s)	cost ($/h)	time (s)	Green - YES | Red - NO		
0.03	0.97	5.60	287.38	6.00	287.14	5.60	306.94	🟩 287.38	🟩 287.14	🟥 294.97
0.16	0.84	4.80	303.35	4.40	337.22	5.60	287.38	🟩 303.35	🟥 311.18	🟩 287.38
0.21	0.79	4.40	337.22	4.00	345.05	4.80	303.35	🟥 311.18	🟥 333.97	🟩 303.35
0.23	0.77	4.00	345.05	4.40	337.22	4.40	311.18	🟥 333.97	🟥 311.18	🟥 314.15
0.26	0.74	4.00	345.05	4.40	337.22	4.40	311.18	🟥 333.97	🟥 311.18	🟥 314.15
0.27	0.73	4.00	345.05	4.40	337.22	3.60	367.84	🟥 333.97	🟥 311.18	🟥 334.69
0.29	0.71	4.00	345.05	3.20	368.56	3.60	367.84	🟥 333.97	🟥 357.48	🟥 334.69
0.30	0.70	4.00	345.05	3.20	368.56	3.60	367.84	🟥 333.97	🟥 357.48	🟥 334.69
0.31	0.69	3.20	368.56	4.00	345.05	2.80	391.35	🟥 357.48	🟥 333.97	🟩 391.35
0.32	0.68	3.20	368.56	2.80	391.35	2.40	520.15	🟥 357.48	🟩 391.35	🟥 451.38
0.35	0.65	2.00	542.94	2.40	520.15	2.80	391.35	🟩 542.94	🟥 451.38	🟩 391.35
0.36	0.64	2.00	542.94	2.40	520.15	2.80	391.35	🟩 542.94	🟥 451.38	🟩 391.35
0.38	0.62	2.00	542.94	2.40	520.15	2.80	391.35	🟩 542.94	🟥 451.38	🟩 391.35
0.39	0.61	2.00	542.94	2.40	520.15	2.00	583.55	🟩 542.94	🟥 451.38	🟥 553.09
0.40	0.60	2.00	542.94	2.40	520.15	2.00	583.55	🟩 542.94	🟥 451.38	🟥 553.09
0.41	0.59	2.00	542.94	2.40	520.15	2.00	583.55	🟩 542.94	🟥 451.38	🟥 553.09
0.42	0.58	2.00	542.94	2.40	520.15	2.00	583.55	🟩 542.94	🟥 451.38	🟥 553.09
0.43	0.57	2.00	542.94	2.40	520.15	2.00	583.55	🟩 542.94	🟥 451.38	🟥 553.09
0.44	0.56	2.00	542.94	2.00	583.55	2.40	520.15	🟩 542.94	🟥 553.09	🟥 451.38
0.45	0.55	2.00	542.94	2.00	583.55	2.40	520.15	🟩 542.94	🟥 553.09	🟥 451.38
0.47	0.53	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.48	0.52	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.49	0.51	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.50	0.50	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.51	0.49	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.52	0.48	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.53	0.47	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.54	0.46	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.55	0.45	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.56	0.44	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.57	0.43	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.58	0.42	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.59	0.41	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.60	0.40	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.61	0.39	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.62	0.38	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.63	0.37	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.64	0.36	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.65	0.35	2.00	542.94	2.00	583.55	2.00	553.09	🟩 542.94	🟥 553.09	🟥 580.18
0.68	0.32	2.00	542.94	1.60	602.97	1.60	894.98	🟩 542.94	🟩 602.97	🟥 619.56
0.69	0.31	2.00	542.94	1.60	602.97	1.60	894.98	🟩 542.94	🟩 602.97	🟥 619.56
0.70	0.30	1.20	955.01	1.60	602.97	1.60	894.98	🟩 955.01	🟩 602.97	🟥 619.56
0.72	0.28	1.20	955.01	1.60	602.97	1.60	894.98	🟩 955.01	🟩 602.97	🟥 619.56
0.73	0.27	1.20	955.01	1.60	602.97	1.60	894.98	🟩 955.01	🟩 602.97	🟥 619.56
0.76	0.24	1.20	955.01	1.20	1025.89	1.20	971.60	🟩 955.01	🟥 971.60	🟥 1025.89
0.77	0.23	1.20	955.01	1.20	1025.89	1.20	971.60	🟩 955.01	🟥 971.60	🟥 1025.89
0.80	0.20	1.20	955.01	1.20	1025.89	1.20	971.60	🟩 955.01	🟥 971.60	🟥 1025.89

Table A.3: Top 1 workflow composition, recommended in test 3.

Ratio value		benchmark pipeline		
cost	time	stage 1	stage 2	stage 3
0.08	0.92	*OPTO_4_Nodes*	*PK_10_Nodes*	*OPRJ_4_Nodes*
0.17	0.83	*OPTO_4_Nodes*	*PK_8_Nodes*	*OPRJ_2_Nodes*
0.25	0.75	*OPTO_4_Nodes*	*PK_8_Nodes*	*OPRJ_2_Nodes*
0.26	0.74	*OPTO_4_Nodes*	*PK_8_Nodes*	*OPRJ_2_Nodes*
0.28	0.72	*OPTO_2_Nodes*	*PK_8_Nodes*	*OPRJ_2_Nodes*
0.31	0.69	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.33	0.67	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.34	0.66	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.35	0.65	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.36	0.64	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.38	0.62	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.39	0.61	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.40	0.60	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.41	0.59	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.42	0.58	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.43	0.57	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.44	0.56	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.46	0.54	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.47	0.53	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.48	0.52	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.49	0.51	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.50	0.50	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.51	0.49	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.52	0.48	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.53	0.47	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.54	0.46	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.55	0.45	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.56	0.44	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.57	0.43	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.58	0.42	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.59	0.41	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.60	0.40	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.61	0.39	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.62	0.38	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.63	0.37	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.64	0.36	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.65	0.35	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.66	0.34	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.67	0.33	*OPTO_2_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.68	0.32	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.70	0.30	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.75	0.25	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.76	0.24	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.78	0.22	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.79	0.21	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.80	0.20	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.86	0.14	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*

Table A.4: Results for the test 6, using weights estimated by the GA.

Ratio value		Recommendation						Best option chosen?		
		1st pipeline		2nd pipeline		3rd pipeline				
cost	time	cost ($/h)	time (s)	cost ($/h)	time (s)	cost ($/h)	time (s)	Green - YES | Red - NO		
0.08	0.92	3.60	334.69	3.20	357.48	3.20	368.56	🟩 334.69	🟩 357.48	🟥 367.63
0.17	0.83	2.80	391.35	3.20	357.48	2.00	542.94	🟩 391.35	🟩 357.48	🟩 542.94
0.25	0.75	2.80	391.35	2.40	451.38	2.00	542.94	🟩 391.35	🟩 451.38	🟩 542.94
0.26	0.74	2.40	451.38	2.80	391.35	1.60	602.97	🟩 451.38	🟩 391.35	🟩 602.97
0.28	0.72	2.40	451.38	1.60	602.97	2.80	391.35	🟩 451.38	🟩 602.97	🟩 391.35
0.31	0.69	1.60	602.97	2.40	451.38	2.00	542.94	🟩 602.97	🟩 451.38	🟩 542.94
0.33	0.67	1.60	602.97	2.40	451.38	2.00	542.94	🟩 602.97	🟩 451.38	🟩 542.94
0.34	0.66	1.60	602.97	2.40	451.38	2.00	542.94	🟩 602.97	🟩 451.38	🟩 542.94
0.35	0.65	1.60	602.97	2.40	451.38	2.00	542.94	🟩 602.97	🟩 451.38	🟩 542.94
0.36	0.64	1.60	602.97	2.40	451.38	2.00	542.94	🟩 602.97	🟩 451.38	🟩 542.94
0.38	0.62	1.60	602.97	2.40	451.38	2.80	417.51	🟩 602.97	🟩 451.38	🟥 391.35
0.39	0.61	1.60	602.97	2.40	451.38	2.80	417.51	🟩 602.97	🟩 451.38	🟥 391.35
0.40	0.60	1.60	602.97	2.40	451.38	2.80	417.51	🟩 602.97	🟩 451.38	🟥 391.35
0.41	0.59	1.60	602.97	2.40	451.38	2.80	417.51	🟩 602.97	🟩 451.38	🟥 391.35
0.42	0.58	1.60	602.97	2.40	451.38	2.80	417.51	🟩 602.97	🟩 451.38	🟥 391.35
0.43	0.57	1.60	602.97	2.40	451.38	2.80	417.51	🟩 602.97	🟩 451.38	🟥 391.35
0.44	0.56	1.60	602.97	2.40	451.38	2.80	417.51	🟩 602.97	🟩 451.38	🟥 391.35
0.46	0.54	1.60	602.97	2.40	451.38	2.80	417.51	🟩 602.97	🟩 451.38	🟥 391.35
0.47	0.53	1.60	602.97	2.40	451.38	2.80	417.51	🟩 602.97	🟩 451.38	🟥 391.35
0.48	0.52	1.60	602.97	2.40	451.38	2.80	417.51	🟩 602.97	🟩 451.38	🟥 391.35
0.49	0.51	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.50	0.50	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.51	0.49	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.52	0.48	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.53	0.47	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.54	0.46	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.55	0.45	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.56	0.44	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.57	0.43	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.58	0.42	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.59	0.41	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.60	0.40	1.60	602.97	2.40	451.38	1.20	955.01	🟩 602.97	🟩 451.38	🟩 955.01
0.61	0.39	1.60	602.97	1.20	955.01	2.40	451.38	🟩 602.97	🟩 955.01	🟩 451.38
0.62	0.38	1.60	602.97	1.20	955.01	2.40	451.38	🟩 602.97	🟩 955.01	🟩 451.38
0.63	0.37	1.60	602.97	1.20	955.01	2.40	451.38	🟩 602.97	🟩 955.01	🟩 451.38
0.64	0.36	1.60	602.97	1.20	955.01	2.40	451.38	🟩 602.97	🟩 955.01	🟩 451.38
0.65	0.35	1.60	602.97	1.20	955.01	2.40	451.38	🟩 602.97	🟩 955.01	🟩 451.38
0.66	0.34	1.60	602.97	1.20	955.01	2.40	451.38	🟩 602.97	🟩 955.01	🟩 451.38
0.67	0.33	1.60	602.97	1.20	955.01	2.40	451.38	🟩 602.97	🟩 955.01	🟩 451.38
0.68	0.32	1.20	955.01	1.60	602.97	2.40	451.38	🟩 955.01	🟩 602.97	🟩 451.38
0.70	0.30	1.20	955.01	1.60	602.97	2.40	451.38	🟩 955.01	🟩 602.97	🟩 451.38
0.75	0.25	1.20	955.01	1.60	602.97	2.40	451.38	🟩 955.01	🟩 602.97	🟩 451.38
0.76	0.24	1.20	955.01	1.60	602.97	2.40	471.20	🟩 955.01	🟩 602.97	🟥 451.38
0.78	0.22	1.20	955.01	1.60	602.97	2.40	471.20	🟩 955.01	🟩 602.97	🟥 451.38
0.79	0.21	1.20	955.01	1.60	602.97	2.40	471.20	🟩 955.01	🟩 602.97	🟥 451.38
0.80	0.20	1.20	955.01	1.60	602.97	2.40	471.20	🟩 955.01	🟩 602.97	🟥 451.38
0.86	0.14	1.20	955.01	1.60	602.97	2.40	471.20	🟩 955.01	🟩 602.97	🟥 451.38

Table A.5: Top 1 workflow composition, recommended in test 4.

Ratio value		benchmark pipeline		
cost	time	stage 1	stage 2	stage 3
0.03	0.97	*BTO_10_Nodes*	*PK_10_Nodes*	*OPRJ_8_Nodes*
0.16	0.84	*BTO_10_Nodes*	*PK_10_Nodes*	*OPRJ_4_Nodes*
0.21	0.79	*BTO_10_Nodes*	*PK_8_Nodes*	*OPRJ_4_Nodes*
0.23	0.77	*BTO_8_Nodes*	*PK_8_Nodes*	*OPRJ_4_Nodes*
0.26	0.74	*BTO_8_Nodes*	*PK_8_Nodes*	*OPRJ_4_Nodes*
0.27	0.73	*BTO_8_Nodes*	*PK_8_Nodes*	*OPRJ_4_Nodes*
0.29	0.71	*BTO_8_Nodes*	*PK_8_Nodes*	*OPRJ_4_Nodes*
0.30	0.70	*BTO_8_Nodes*	*PK_8_Nodes*	*OPRJ_4_Nodes*
0.31	0.69	*OPTO_4_Nodes*	*PK_8_Nodes*	*OPRJ_4_Nodes*
0.32	0.68	*OPTO_4_Nodes*	*PK_8_Nodes*	*OPRJ_4_Nodes*
0.35	0.65	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.36	0.64	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.38	0.62	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.39	0.61	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.40	0.60	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.41	0.59	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.42	0.58	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.43	0.57	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.44	0.56	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.45	0.55	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.47	0.53	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.48	0.52	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.49	0.51	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.50	0.50	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.51	0.49	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.52	0.48	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.53	0.47	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.54	0.46	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.55	0.45	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.56	0.44	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.57	0.43	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.58	0.42	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.59	0.41	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.60	0.40	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.61	0.39	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.62	0.38	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.63	0.37	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.64	0.36	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.65	0.35	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.68	0.32	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.69	0.31	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.70	0.30	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.72	0.28	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.73	0.27	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.76	0.24	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.77	0.23	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.80	0.20	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*

Table A.6: Modified list from top 1 workflows of test 4, with optimal solutions.

Ratio value		benchmark pipeline		
cost	time	stage 1	stage 2	stage 3
0.03	0.97	*BTO_10_Nodes*	*PK_10_Nodes*	*OPRJ_8_Nodes*
0.16	0.84	*BTO_10_Nodes*	*PK_10_Nodes*	*OPRJ_4_Nodes*
0.21	0.79	*BTO_8_Nodes*	*PK_10_Nodes*	*OPRJ_4_Nodes*
0.23	0.77	*BTO_8_Nodes*	*PK_10_Nodes*	*OPRJ_2_Nodes*
0.26	0.74	*BTO_8_Nodes*	*PK_10_Nodes*	*OPRJ_2_Nodes*
0.27	0.73	*BTO_8_Nodes*	*PK_10_Nodes*	*OPRJ_2_Nodes*
0.29	0.71	*BTO_8_Nodes*	*PK_10_Nodes*	*OPRJ_2_Nodes*
0.30	0.70	*BTO_8_Nodes*	*PK_10_Nodes*	*OPRJ_2_Nodes*
0.31	0.69	*BTO_8_Nodes*	*PK_10_Nodes*	*OPRJ_2_Nodes*
0.32	0.68	*BTO_8_Nodes*	*PK_10_Nodes*	*OPRJ_2_Nodes*
0.35	0.65	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.36	0.64	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.38	0.62	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.39	0.61	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.40	0.60	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.41	0.59	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.42	0.58	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.43	0.57	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.44	0.56	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.45	0.55	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.47	0.53	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.48	0.52	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.49	0.51	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.50	0.50	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.51	0.49	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.52	0.48	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.53	0.47	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.54	0.46	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.55	0.45	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.56	0.44	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.57	0.43	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.58	0.42	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.59	0.41	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.60	0.40	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.61	0.39	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.62	0.38	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.63	0.37	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.64	0.36	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.65	0.35	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.68	0.32	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.69	0.31	*OPTO_4_Nodes*	*PK_4_Nodes*	*OPRJ_2_Nodes*
0.70	0.30	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.72	0.28	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.73	0.27	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.76	0.24	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.77	0.23	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*
0.80	0.20	*OPTO_2_Nodes*	*PK_2_Nodes*	*OPRJ_2_Nodes*

Table A.7: Results for the test 7, using GA for weight optimization.

Ratio value		Recommendation						Best option chosen?		
		1st pipeline		2nd pipeline		3rd pipeline				
cost	time	cost ($/h)	time (s)	cost ($/h)	time (s)	cost ($/h)	time (s)	Green - YES | Red - NO		
0.03	0.97	5.60	287.38	4.80	303.35	5.20	295.21	🟩 287.38	🟩 303.35	🟩 295.21
0.16	0.84	4.80	303.35	4.40	311.18	4.40	326.14	🟩 303.35	🟩 311.18	🟥 314.15
0.21	0.79	4.40	326.14	4.80	303.35	4.00	333.97	🟥 311.18	🟩 303.35	🟩 333.97
0.23	0.77	4.00	333.97	4.40	326.14	4.40	311.18	🟩 333.97	🟥 311.18	🟥 314.15
0.26	0.74	4.00	333.97	4.40	326.14	4.00	336.94	🟩 333.97	🟥 311.18	🟩 336.94
0.27	0.73	4.00	333.97	4.40	326.14	4.00	336.94	🟩 333.97	🟥 311.18	🟩 336.94
0.29	0.71	4.00	333.97	3.20	357.48	4.00	336.94	🟩 333.97	🟩 357.48	🟩 336.94
0.30	0.70	4.00	333.97	3.20	357.48	4.00	336.94	🟩 333.97	🟩 357.48	🟩 336.94
0.31	0.69	3.20	357.48	4.00	333.97	4.00	336.94	🟩 357.48	🟩 333.97	🟩 336.94
0.32	0.68	3.20	357.48	4.00	333.97	2.00	542.94	🟩 357.48	🟩 333.97	🟩 542.94
0.35	0.65	2.00	542.94	3.20	357.48	2.80	519.43	🟩 542.94	🟩 357.48	🟥 391.35
0.36	0.64	2.00	542.94	3.20	357.48	2.80	519.43	🟩 542.94	🟩 357.48	🟥 391.35
0.38	0.62	2.00	542.94	3.20	357.48	2.80	519.43	🟩 542.94	🟩 357.48	🟥 391.35
0.39	0.61	2.00	542.94	3.20	357.48	2.00	553.09	🟩 542.94	🟩 357.48	🟩 553.09
0.40	0.60	2.00	542.94	2.00	553.09	3.20	357.48	🟩 542.94	🟩 553.09	🟩 357.48
0.41	0.59	2.00	542.94	2.00	553.09	3.20	357.48	🟩 542.94	🟩 553.09	🟩 357.48
0.42	0.58	2.00	542.94	2.00	553.09	3.20	357.48	🟩 542.94	🟩 553.09	🟩 357.48
0.43	0.57	2.00	542.94	2.00	553.09	3.20	357.48	🟩 542.94	🟩 553.09	🟩 357.48
0.44	0.56	2.00	542.94	2.00	553.09	3.20	357.48	🟩 542.94	🟩 553.09	🟩 357.48
0.45	0.55	2.00	542.94	2.00	553.09	2.80	519.43	🟩 542.94	🟩 553.09	🟥 391.35
0.47	0.53	2.00	542.94	2.00	553.09	2.80	519.43	🟩 542.94	🟩 553.09	🟥 391.35
0.48	0.52	2.00	542.94	2.00	553.09	2.80	519.43	🟩 542.94	🟩 553.09	🟥 391.35
0.49	0.51	2.00	542.94	2.00	553.09	2.80	519.43	🟩 542.94	🟩 553.09	🟥 391.35
0.50	0.50	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.51	0.49	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.52	0.48	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.53	0.47	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.54	0.46	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.55	0.45	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.56	0.44	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.57	0.43	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.58	0.42	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.59	0.41	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.60	0.40	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.61	0.39	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.62	0.38	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.63	0.37	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.64	0.36	2.00	542.94	2.00	553.09	1.60	894.98	🟩 542.94	🟩 553.09	🟥 602.97
0.65	0.35	2.00	542.94	1.60	894.98	2.00	553.09	🟩 542.94	🟥 602.97	🟩 553.09
0.68	0.32	2.00	542.94	1.60	602.97	1.60	894.98	🟩 542.94	🟩 602.97	🟥 619.56
0.69	0.31	2.00	542.94	1.60	602.97	1.60	894.98	🟩 542.94	🟩 602.97	🟥 619.56
0.70	0.30	1.20	955.01	1.60	602.97	1.60	894.98	🟩 955.01	🟩 602.97	🟥 619.56
0.72	0.28	1.20	955.01	1.60	602.97	1.60	894.98	🟩 955.01	🟩 602.97	🟥 619.56
0.73	0.27	1.20	955.01	1.60	602.97	1.60	894.98	🟩 955.01	🟩 602.97	🟥 619.56
0.76	0.24	1.20	955.01	1.20	971.60	1.60	602.97	🟩 955.01	🟩 971.60	🟩 602.97
0.77	0.23	1.20	955.01	1.20	971.60	1.60	602.97	🟩 955.01	🟩 971.60	🟩 602.97
0.80	0.20	1.20	955.01	1.20	971.60	1.60	602.97	🟩 955.01	🟩 971.60	🟩 602.97

References

[1] Eliseu Pereira, João Reis, and Gil Gonçalves. DINASORE: A dynamic intelligent reconfiguration tool for cyber-physical production systems. *Eclipse SAM IoT*, 2739:63–71, 2020.

[2] Rajiv Ranjan, Lydia Y Chen, Prem Prakash Jayaraman, and Albert Y Zomaya. A note on advances in scheduling algorithms for cyber-physical-social workflows, 2020.

[3] Ruben Sipos, Dmitriy Fradkin, Fabian Moerchen, and Zhuang Wang. Log-based predictive maintenance. In *Proceedings of the 20th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1867–1876, 2014.

[4] Dmitri Panfilenko, Peter Poller, Daniel Sonntag, Sonja Zillner, and Martin Schneider. BPMN for knowledge acquisition and anomaly handling in CPS for smart factories. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4. IEEE, 2016.

[5] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, 2015.

[6] Fei Tao, Qinglin Qi, Lihui Wang, and AYC Nee. Digital twins and cyber–physical systems toward smart manufacturing and industry 4.0: Correlation and comparison. *Engineering*, 5(4):653–661, 2019.

[7] Shiyong Wang, Jiafu Wan, Daqiang Zhang, Di Li, and Chunhua Zhang. Towards smart factory for industry 4.0: a self-organized multi-agent system with big data based feedback and coordination. *Computer networks*, 101:158–168, 2016.

[8] Ioan Dumitrache, Simona Iuliana Caramihai, Ioan Stefan Sacala, Mihnea Alexandru Moisescu, and Dragos Constantin Popescu. Future enterprise as an intelligent cyber-physical system. *IFAC-PapersOnLine*, 53(2):10873–10878, 2020.

[9] Zhijiao Xiao and Zhong Ming. A method of workflow scheduling based on colored petri nets. *Data & Knowledge Engineering*, 70(2):230–247, 2011.

[10] Jianwu Wang, Prakashan Korambath, Ilkay Altintas, Jim Davis, and Daniel Crawl. Workflow as a service in the cloud: architecture and scheduling algorithms. *Procedia computer science*, 29:546–556, 2014.

[11] Marek Wieczorek, Radu Prodan, and Thomas Fahringer. Scheduling of scientific workflows in the askalon grid environment. *Acm Sigmod Record*, 34(3):56–62, 2005.

[12] Ahmed Ismail, Hong-Linh Truong, and Wolfgang Kastner. Manufacturing process data analysis pipelines: a requirements analysis and survey. *Journal of Big Data*, 6(1):1–26, 2019.

[13] Gonzalo De La Torre, Paul Rad, and Kim-Kwang Raymond Choo. Driverless vehicle security: Challenges and future research opportunities. *Future Generation Computer Systems*, 108:1092–1111, 2020.

[14] Jian Shen, Chen Wang, Anxi Wang, Qi Liu, and Yang Xiang. Moving centroid based routing protocol for incompletely predictable cyber devices in cyber-physical-social distributed systems. *Future Generation Computer Systems*, 108:1129–1139, 2020.

[15] Yiping Wen, Jianxun Liu, Wanchun Dou, Xiaolong Xu, Buqing Cao, and Jinjun Chen. Scheduling workflows with privacy protection constraints for big data applications on cloud. *Future Generation Computer Systems*, 108:1084–1091, 2020.

[16] Liwei Huang, Yutao Ma, Yanbo Liu, and Arun Kumar Sangaiah. Multi-modal bayesian embedding for point-of-interest recommendation on location-based cyber-physical–social networks. *Future Generation Computer Systems*, 108:1119–1128, 2020.

[17] Hong Yao, Muzhou Xiong, Hui Li, Lin Gu, and Deze Zeng. Joint optimization of function mapping and preemptive scheduling for service chains in network function virtualization. *Future Generation Computer Systems*, 108:1112–1118, 2020.

[18] Shijian Li, Minhao Shi, Runhe Huang, Xinwei Chen, and Gang Pan. Perception-enhancement based task learning and action scheduling for robotic limb in cps environment. *Future Generation Computer Systems*, 108:1069–1083, 2020.

[19] Anish Jindal, Neeraj Kumar, and Mukesh Singh. Internet of energy-based demand response management scheme for smart homes and phevs using svm. *Future Generation Computer Systems*, 108:1058–1068, 2020.

[20] Reem Y Ali, Shashi Shekhar, Shounak Athavale, and Eric Marsman. Ulama: A utilization-aware matching approach for robust on-demand spatial service brokers. *Future Generation Computer Systems*, 108:1030–1048, 2020.

[21] Xiaodao Chen, Junqing Fan, Qing He, Yuewei Wang, Dongbo Liu, and Shiyan Hu. Economical and balanced production in smart petroleum cyber–physical system. *Future Generation Computer Systems*, 95:364–371, 2019.

[22] Shangguang Wang, Yan Guo, Yan Li, and Ching-Hsien Hsu. Cultural distance for service composition in cyber–physical–social systems. *Future Generation Computer Systems*, 108:1049–1057, 2020.

[23] Henan Zhao and Rizos Sakellariou. An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In *European Conference on Parallel Processing*, pages 189–194. Springer, 2003.

[24] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. Pegasus: Mapping scientific workflows onto the grid. In *European Across Grids Conference*, pages 11–20. Springer, 2004.

[25] R Keith Mobley. *An introduction to predictive maintenance*. Elsevier, 2002.

[26] Hedda Lüttenberg, Christian Bartelheimer, and Daniel Beverungen. Designing predictive maintenance for agricultural machines. 2018.

[27] David J Edwards, Gary D Holt, and FC Harris. Predictive maintenance techniques and their relevance to construction plant. *Journal of Quality in Maintenance Engineering*, 1998.

[28] Ying Peng, Ming Dong, and Ming Jian Zuo. Current status of machine prognostics in condition-based maintenance: a review. *The International Journal of Advanced Manufacturing Technology*, 50(1-4):297–313, 2010.

[29] Peter O'Donovan, Kevin Leahy, Ken Bruton, and Dominic TJ O'Sullivan. An industrial big data pipeline for data-driven analytics maintenance applications in large-scale smart manufacturing facilities. *Journal of Big Data*, 2(1):1–26, 2015.

[30] W Freudling, M Romaniello, DM Bramich, P Ballester, V Forchi, CE García-Dabló, S Moehler, and MJ Neeser. Automated data reduction workflows for astronomy-the eso reflex environment. *Astronomy & Astrophysics*, 559:A96, 2013.

[31] K Banse, P Crane, P Grosbol, F Middleburg, C Ounnas, D Ponz, and Banse Waldthausen, H. Midas-eso's new image processing system. *The Messenger*, 31:26–28, 1983.

[32] Tody Tody, Doug. Iraf in the nineties. In *Astronomical Data Analysis Software and Systems II*, volume 52, page 173, 1993.

[33] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Altintas Mock, Steve. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.*, pages 423–424. IEEE, 2004.

[34] William F Godoy, Peter F Peterson, Steven E Hahn, and Jay J Billings. Efficient data management in neutron scattering data reduction workflows at ornl. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 2674–2680. IEEE, 2020.

[35] Mark Könnecke, Frederick A Akeroyd, Herbert J Bernstein, Aaron S Brewster, Stuart I Campbell, Björn Clausen, Stephen Cottrell, Jens Uwe Hoffmann, Pete R Jemian, David Männicke, et al. The nexus data format. *Journal of applied crystallography*, 48(1):301–305, 2015.

[36] Owen Arnold, Jean-Christophe Bilheux, JM Borreguero, Alex Buts, Stuart I Campbell, L Chapon, Mathieu Doucet, N Draper, R Ferraz Leal, MA Gigg, et al. Mantid—data analysis and visualization package for neutron scattering and μ sr experiments. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 764:156–166, 2014.

[37] A Traficante, L Calzoletti, M Veneziani, B Ali, G De Gasperis, AM Di Giorgio, F Faustini, D Ikhenaode, S Molinari, Paolo Natoli, et al. Data reduction pipeline for the hi-gal survey. *Monthly Notices of the Royal Astronomical Society*, 416(4):2932–2943, 2011.

[38] Brad Cavanagh, T Jenness, F Economou, and MJ Currie. The orac-dr data reduction pipeline. *Astronomische Nachrichten: Astronomical Notes*, 329(3):295–297, 2008.

[39] Lupton Lupton, Robert. *Statistics in theory and practice*. Princeton University Press, 2020.

[40] Matthew Overlin, Christopher Smith, Marija Ilic, and James L Kirtley. A workflow for non-linear load parameter estimation using a power-hardware-in-the-loop experimental testbed. In *2020 IEEE Applied Power Electronics Conference and Exposition (APEC)*, pages 581–588. IEEE, 2020.

[41] Sophia Ulonska, Paul Kroll, Jens Fricke, Christoph Clemens, Raphael Voges, Markus M Müller, and Christoph Herwig. Workflow for target-oriented parametrization of an enhanced mechanistic cell culture model. *Biotechnology journal*, 13(4):1700395, 2018.

[42] Bence Bécsy, Peter Raffai, Neil J Cornish, Reed Essick, Jonah Kanner, Erik Katsavounidis, Tyson B Littenberg, Margaret Millhouse, and Salvatore Vitale. Parameter estimation for gravitational-wave bursts with the bayeswave pipeline. *The Astrophysical Journal*, 839(1):15, 2017.

[43] Neil J Cornish and Tyson B Littenberg. Bayeswave: Bayesian inference for gravitational wave bursts and instrument glitches. *Classical and Quantum Gravity*, 32(13):135012, 2015.

[44] Peter J Green. Reversible jump markov chain monte carlo computation and bayesian model determination. *Biometrika*, 82(4):711–732, 1995.

[45] Xiaoyu Chen and Ran Jin. Adapipe: A recommender system for adaptive computation pipelines in cyber-manufacturing computation services. *IEEE Transactions on Industrial Informatics*, 2020.

[46] Xiaoyu Chen and Ran Jin. Data fusion pipelines for autonomous smart manufacturing. In *2018 IEEE 14th international conference on automation science and engineering (CASE)*, pages 1203–1208. IEEE, 2018.

[47] Chengrun Yang, Yuji Akimoto, Dae Won Kim, and Madeleine Udell. Oboe: Collaborative filtering for automl model selection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1173–1183, 2019.

[48] Dylan Bates. Recommending more efficient workflows to software developers. *arXiv preprint arXiv:2102.03670*, 2021.

[49] Frank Linton, Deborah Joy, Hans-Peter Schaefer, and linton Charron, Andrew. Owl: A recommender system for organization-wide learning. *Educational Technology & Society*, 3(1):62–76, 2000.

[50] Justin Matejka, Wei Li, Tovi Grossman, and matejka Fitzmaurice, George. Community-commands: Command recommendations for software applications. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, page 193–202, New York, NY, USA, 2009. Association for Computing Machinery. URL: https://doi.org/10.1145/1622176.1622214, doi:10.1145/1622176.1622214.

[51] Emerson Murphy-Hill, Rahul Jiresal, and murphy Murphy, Gail C. Improving software developers' fluency by recommending development environment commands. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery. URL: https://doi.org/10.1145/2393596.2393645, doi:10.1145/2393596.2393645.

[52] Buyun Sheng, Ruiping Luo, Gaocai Fu, Hui Wang, and Xincheng Lu. Workflow model recommendation approach based on a design information model. *IEEE Access*, 7:159622–159634, 2019.

[53] Kamran Soomro, Kamran Munir, and Richard McClatchey. Incorporating semantics in pattern-based scientific workflow recommender systems: Improving the accuracy of recommendations. In *2015 Science and Information Conference (SAI)*, pages 565–571. IEEE, 2015.

[54] Lu Zhen, George Q Huang, and Zuhua Jiang. Recommender system based on workflow. *Decision Support Systems*, 48(1):237–245, 2009.

[55] Samet Ayhan, Pablo Costas, and Hanan Samet. Prescriptive analytics system for long-range aircraft conflict detection and resolution. In *Proceedings of the 26th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 239–248, 2018.

[56] Ahmed Ismail, Hong-Linh Truong, and Wolfgang Kastner. Manufacturing process data analysis pipelines: a requirements analysis and survey. *Journal of Big Data*, 6(1):1–26, 2019.

[57] Taesung Park, Sung-Gon Yi, Sung-Hyun Kang, SeungYeoun Lee, Yong-Sung Lee, and Richard Simon. Evaluation of normalization methods for microarray data. *BMC bioinformatics*, 4(1):1–13, 2003.

[58] Stephen Strother, Stephen La Conte, Lars Kai Hansen, Jon Anderson, Jin Zhang, Sujit Pulapura, and David Rottenberg. Optimizing the fmri data-processing pipeline using prediction and reproducibility performance metrics: I. a preliminary group analysis. *Neuroimage*, 23:S196–S207, 2004.

[59] Jia Yu, Rajkumar Buyya, and Kotagiri Ramamohanarao. Workflow scheduling algorithms for grid computing. In *Metaheuristics for scheduling in distributed computing environments*, pages 173–214. Springer, 2008.

[60] Artur Andrzejak, Ulf Hermann, and Akhil Sahai. Feedbackflow-an adaptive workflow generator for systems management. In *Second International Conference on Autonomic Computing (ICAC'05)*, pages 335–336. IEEE, 2005.

[61] Johannes Kunze Von Bischhoffshausen, Markus Paatsch, Melanie Reuter, Gerhard Satzger, and Hansjoerg Fromm. An information system for sales team assignments utilizing predictive and prescriptive analytics. In *2015 IEEE 17th Conference on Business Informatics*, volume 1, pages 68–76. IEEE, 2015.

[62] Golshan Madraki and Robert P Judd. Recalculating the length of the longest path in perturbed directed acyclic graph. *IFAC-PapersOnLine*, 52(13):1560–1565, 2019.

[63] Mirko Stojiljkovic. Linear regression in python, 2021.

[64] Jing Li, Ji-hang Cheng, Jing-yuan Shi, and Fei Huang. Brief introduction of back propagation (bp) neural network algorithm and its improvement. In *Advances in computer science and information engineering*, pages 553–558. Springer, 2012.

[65] JVN Lakshmi. Stochastic gradient descent using linear regression with python. *International Journal on Advanced Engineering Research and Applications*, 2(7):519–524, 2016.

[66] Ahmed Fawzy Gad. Python genetic algorithm!, 2020. URL: `https://pygad.readthedocs.io/en/latest/`.

[67] Ahmed Fawzy Gad. Pygad: An intuitive genetic algorithm python library. *arXiv preprint arXiv:2106.06158*, 2021.

[68] Syed Muhammad Fawad Ali and Robert Wrembel. Towards a cost model to optimize user-defined functions in an etl workflow based on user-defined performance metrics. In *European Conference on Advances in Databases and Information Systems*, pages 441–456. Springer, 2019.

[69] Sharan Srinivas and A Ravi Ravindran. Optimizing outpatient appointment system using machine learning algorithms and scheduling rules: a prescriptive analytics framework. *Expert Systems with Applications*, 102:245–261, 2018.

[70] Rares Vernica, Michael J Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506, 2010.

[71] Syed Muhammad Fawad Ali. Next-generation etl framework to address the challenges posed by big data. In *DOLAP*, 2018.

[72] James P Evans and Ralph E Steuer. A revised simplex method for linear multiple objective programs. *Mathematical Programming*, 5(1):54–72, 1973.