

Individual Diploma Thesis

**MACHINE LEARNING INFERENCE PERFORMANCE  
EVALUATION ON EDGE TO CLOUD INFRASTRUCTURES**

**Panikos Christou**

**University of Cyprus**



**Department of Computer Science**

**May 2022**

**UNIVERSITY OF CYPRUS**  
**DEPARTMENT OF COMPUTER SCIENCE**

**MACHINE LEARNING INFERENCE PERFORMANCE EVALUATION ON EDGE TO  
CLOUD INFRASTRUCTURES**

**Panikos Christou**

Supervisor

DR. George Pallis

The Individual Diploma Thesis was submitted for partial fulfilment of the requirements for obtaining the degree of Computer Science of the Department of Computer Science of the University of Cyprus

May 2022

## **Acknowledgements**

I would like to thank from the bottom of my heart Dr. George Pallis for choosing me and giving me this great opportunity to work on this subject. From the beginning of the project, he knew that the subject suited me greatly and fully utilized my knowledge on machine learning, software development and operations deployment. This was a huge adventure for me since the first day of my assignment here. I am grateful because not only was he there for me whenever I needed him but also for assisting and guiding me in completing the given milestones in the project and in the end completing it successfully.

Moreover, I need to thank the doctoral student supervisor Moysis Simonides for being by my side whenever I needed him as well when I needed guidance on how to continue and where to focus. He was always happy to have a meeting and always wanted to assist me in moving forward with the project and assisting me when questions and confusion arrived. I can't imagine how much worse this thesis would be without him.

Additionally, I would like to thank the Department of computer science for the knowledge and skills that it has provided to me. With it, I feel ready to take on the outside world and the job market.

Lastly, I would like to thank my family for always being there for me and assisting me financially and psychologically in studying in this great University and even better Course.

# Abstract

Many new technologies and approaches have been developed in order to tackle modern problems when it comes to application development. Cloud-to-edge architectures are the go-to way for building IoT applications and Machine Learning has been introduced as a wonderful solver for a variety of problems the current world faces. On the one hand, the use of cloud computing and specifically containerizing applications to work on the cloud is certainly where the future is headed so any research done to optimize the whole procedure is vital. On the other hand, there are limited to almost no tools that help a business choose the best ML that will help save them costs and optimize their performance in the long run. Thus, there is a gap when it comes to optimizing these applications all ranging from application architecture, which Machine Learning model is most suited for the current task, and which is the best data reduction approach that keeps the network traffic low while also having the least drawbacks when it comes to performance.

This thesis introduces an easy to configure cloud to edge architecture targeted towards ML, Dev-ops and Architecture engineers which utilizes ML that performs Object Detection. This tool is a perfect foundation for any of the above users to improve upon and test out their applications. ML engineers can easily replace the ML's and the data sent to the edges to suit their custom problems; Dev-ops engineers can use the source code to create their own easy to launch multiservice application and lastly Architecture engineers can easily take the source code and extend it to any application of their liking which will already be configured to gather a multitude of important data to them. Finally, the tool provides post-experimentation analytic functions which help with data related tasks and the data the project generates, are important and lead to conclusive results.

# Contents

<b>Abstract</b>	<b>3</b>
<b>List of figures:</b>	<b>6</b>
<b>Notable Abbreviations</b>	<b>7</b>
<b>Chapter 1: Introduction</b>	<b>8</b>
<b>Chapter 2: Background and related work</b>	<b>10</b>
2.1 Cloud and Edge computing:	10
2.2 Docker Ecosystem:	11
2.3 Restful APIs and Server programming	13
2.4 ML and Data Analysis tools	13
2.5 Related work to get familiar with the concept	16
2.6 Helpful documentation	16
<b>Chapter 3: Systems overview</b>	<b>17</b>
3.1 High level description:	17
3.1.1 Description:	17
3.1.2 Pipeline of operations:	18
3.2 Targeted users and what to expect from the system:	21
3.2.1 ML engineers:	21
3.2.2 Dev-ops engineers:	22
3.2.3 Architecture optimizers:	22
<b>Chapter 4: Implementation</b>	<b>23</b>
4.1 Implementation details:	24
4.1.1 Dockerfiles and Docker-compose information:	24
4.1.2 Dockerfile information:	25
4.1.3 Flask Server information:	26
4.1.4 Fifty-one information:	26
4.2 How to run and gather results:	27
4.2.1 Requirements:	27
4.2.2 Building the Base_image:	27
4.2.3 How to run the default project configurations:	27
4.2.4 How to edit the configurations:	29
4.2.5 Where to view the results:	30
4.2.6 Interpreting the csv's:	30
4.3 How to use the colab file:	32
<b>Chapter 5: Experimentation</b>	<b>37</b>
5.1 Experimentation goals:	37
5.2 What to experiment with to achieve said goals:	37

5.3 Experimentation details: -----	38
5.3.1 Using different models: -----	38
5.3.2 Using reduced dimensions: -----	38
5.3.3 Using reduced quality:-----	39
<b>Chapter 6: Evaluation-----</b>	<b>40</b>
6.1 Results overview: -----	40
6.1.1 Machine Learning models:-----	41
6.1.2 Pre-processing : -----	45
6.1.3 Object recognition :-----	52
6.2 Results comments and limitations: -----	54
6.2.1 Comments -----	54
6.2.2 limitations -----	55
<b>Chapter 7: Conclusions -----</b>	<b>56</b>
<b>Bibliography -----</b>	<b>56</b>

## List of figures:

Figure 1 Cloud and Edge computing " <a href="https://www.alibabacloud.com/knowledge/what-is-edge-computing">https://www.alibabacloud.com/knowledge/what-is-edge-computing</a> " .....	11
Figure 2 Docker Architecture " <a href="https://thingsolver.com/hello-docker/">https://thingsolver.com/hello-docker/</a> " .....	11
Figure 3 Docker Compose Demo " <a href="https://hosting.analythium.io/shiny-apps-with-docker-compose-part-1-development/">https://hosting.analythium.io/shiny-apps-with-docker-compose-part-1-development/</a> " .....	12
Figure 4 REST API Demonstration " <a href="https://www.astera.com/type/blog/rest-api-definition/">https://www.astera.com/type/blog/rest-api-definition/</a> " .....	13
Figure 5 Machine learning illustration " <a href="https://www.researchgate.net/figure/Illustration-of-the-differences-in-complexity-and-implementation-between-traditional_fig3_336660735">https://www.researchgate.net/figure/Illustration-of-the-differences-in-complexity-and-implementation-between-traditional_fig3_336660735</a> " .....	14
Figure 6 Architecture overview .....	18
Figure 7 Visualization of how relaunching the project works .....	19
Figure 8 Docker-compose containers communication visualized .....	20
Figure 9 Visualization of which data each node collects .....	21
Figure 10 Docker-compose YAML file.....	24
Figure 11 Edge container's Dockerfile .....	25
Figure 12 runconf.py Default runtime configurations.....	29
Figure 13 An edge request csv .....	30
Figure 14 A workload's Request csv .....	31
Figure 15 A monitor csv of a node .....	32
Figure 16 Code that organises the csv's to each runtime configuration .....	33
Figure 17 Visualization of the runtime configurations dictionary .....	34
Figure 18 How we create a DataFrame from a list of csv's .....	35
Figure 19 Sample of how to query the DataFrame and get the average time each ML model took to process.....	36
Figure 20 Chart of average f1_score.....	41
Figure 21 Chart of average milliseconds taken per request .....	43
Figure 22 Chart of average f1_score/milli_taken.....	44
Figure 23 Chart of how quality change effects average f1_score.....	47
Figure 24 Chart of quality vs image size reduction .....	48
Figure 25 Chart of how resizing effects f1_score.....	49
Figure 26 Chart of resizing vs image size reduction .....	50
Figure 27 Illustration of the JPEG compression algorithm <a href="https://www.eetimes.com/wp-content/uploads/media-1101219-fig1.jpg">https://www.eetimes.com/wp-content/uploads/media-1101219-fig1.jpg</a> .....	50
Figure 28 Chart comparing f1_score per pre-processing option .....	51
Figure 29 Chart of items f1_score and support sorted by f1_score.....	53
Figure 30 Chart of items f1_score and support sorted by support .....	54

# Notable Abbreviations

	Abbreviation		Full Term
→	ML	....	Machine learning
→	VM	....	Virtual machine
→	CNN	....	Convolutional Neural network
→	ANN	....	Artificial Neural network
→	APIs	....	Application Programming Interfaces
→	URL	....	Uniform Resource Locator
→	CV	....	Computer vision
→	IoT	....	Internet of Things
→	OS	....	Operating System
→	AI	....	Artificial intelligence
→	Vram	....	Virtual RAM memory
→	HW	....	Hardware

# Chapter 1: Introduction

Cloud computing has taken the world by storm and now it is the most reliable and cost-effective way of acquiring computing resources as well as anything you might need without having to maintain them yourself. Meaning that all sizes of enterprises use cloud providers to serve their applications or in general their solutions. IoT devices keep popping up in our lifestyle and serve to make our life easier. From our homes to our work, we are flooded with smart devices that collect and send data to their respective companies to be processed. These companies most often or not, have ML models running behind them because they provide solutions to a wide array of problems, from classification to regression to even Natural Language Processing.

Despite how amazing Machine Learning is, there is a huge drawback to it. Firstly, it is heavily reliant on the amount of data its fed, and it requires immense computing resources. The modern world has solved both problems, the first is solved by the aforementioned IoT devices and the second by the cost-effective cloud services.

Edge computing has also been a huge contributing factor to these companies since it helps optimize the IoT architecture by reducing the network bandwidth, reduces the workload of the cloud and can scale better. But this is not as easy implement as it is to say since each subject bring their own set challenges to a company.

Each solution carries over some challenges and merging them together can only cause even more trouble. Currently, there are many challenges an organization must solve to successfully launch such a deployment. Firstly, there are a plethora of ML models to choose from which all have their own benefits and drawbacks. So, deciding which ML to choose from a collection of implementations available online is hard has its own profession associated with it. ML engineers are tasked to find the best ML for a problem and even sometimes build it, train, and test themselves. This is a big challenge which can cause a company a lot of time and resources to get right. Secondly, since the topics are relatively new, deployments architecture has not been stabilized and each can vary depending on the problem, the devices, the data and even the MLs themselves. So, a lot of testing needs to be done on how an architectures solution must be. Lastly, coordinating containerized applications to be working together is no easy task. There are a lot of problems that can occur and a lot of bugs to be fixed. Networking can also be an issue since almost all of the times, the network available is not as powerful as one might hope so there must be compression challenges that have to be solved for the deployment to even be running.

As briefly discussed in the abstract, We have created a system that launches multiple configurations of a multiservice application. This multiservice application has edge services receiving images and their annotations by a workloader service. These services can pre-process the images and then run an ML Object detection algorithm. While running they store their HW metrics and ML detection statistics for someone to apply data science on them. We can for example deduce which ML works best for compressed images or which one is the most efficient etc.

The application itself has provided with some satisfactory results and it its working flawlessly. The services themselves are built in such a way that someone can replace any part of the project and create his own problem solver and test out stuff.

The System's responsibility is to serve as fundamental project for anyone who wants to experiment with cloud and edge computing, as well Machine learning. Hopefully this project will help developers not to get stuck on the little bugs and configuration problems we had to resolve, and instead focus on their project.

The main selling point is that an individual or a company could take the source code and then themselves built application architectures, run them, and collect any data they might want. It's all expandable with most of the important code already appearing in the fundamental version. Moreover, the system is preconfigured to collect HW data and can be customized to collect application specific data as well. In our case, it specifically collects ML object detection and the image's size for each different configuration.

Lastly, we have also gathered some Object detection data in which we have extracted which ML is the most accurate, the most efficient and which pre-processing technique works best with how much of a size reduction.

## Chapter 2: Background and related work

---

2.1 Cloud and Edge computing:

2.2 Docker Ecosystem

2.3 RESTful APIs and Server Programming

2.4 ML and Data Analysis tools

2.5 Related work to get familiar with the concept

2.6 Helpful Documentation

---

### 2.1 Cloud and Edge computing:

Cloud computing is the on-demand availability of computer system resources, especially data storage (cloud storage) and computing power, without direct active management by the user. Large clouds often have functions distributed over multiple locations, each location being a datacentre. Cloud computing relies on sharing of resources to achieve coherence and typically using a "pay-as-you-go" model which can help in reducing capital expenses but may also lead to unexpected operating expenses for unaware users. (Hertzfeld)

If an operator has an IoT application, it may not want to carry the costs needed to build and run a server capable of handling the task. It's best to rent the server resources through a cloud provider. So, our sensors transmit their data to these servers to run our algorithms. This the standard practice in the IoT space since there are many reasons to prefer resorting to a cloud provider.

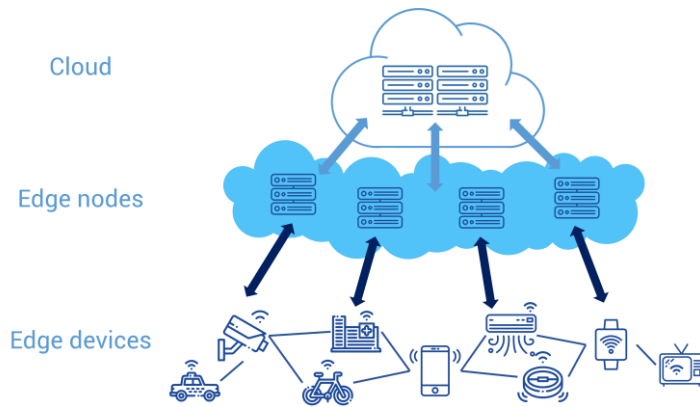


Figure 1 Cloud and Edge computing "<https://www.alibabacloud.com/knowledge/what-is-edge-computing>"

Contrary to the centralised orientation of Cloud Computing, Edge computing is a technique used to bring computation and data storage closer to the sources of the data. For example, if we had some sensors that capture and send data, we could only apply edge computing if they had compute and store capabilities as well. (Edge computing) We could use these resources for example to lower the size and frequency of network traffic sent or even run some computation so that a cloud server does not have to take it upon itself to do it for thousands of edge machines.

## 2.2 Docker Ecosystem:

Docker is an open-source containerization platform. It enables developers to package applications into containers—standardised executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment. Every single node in a microservice architecture runs in its own containerized OS. (What is Docker? - India, 2021)

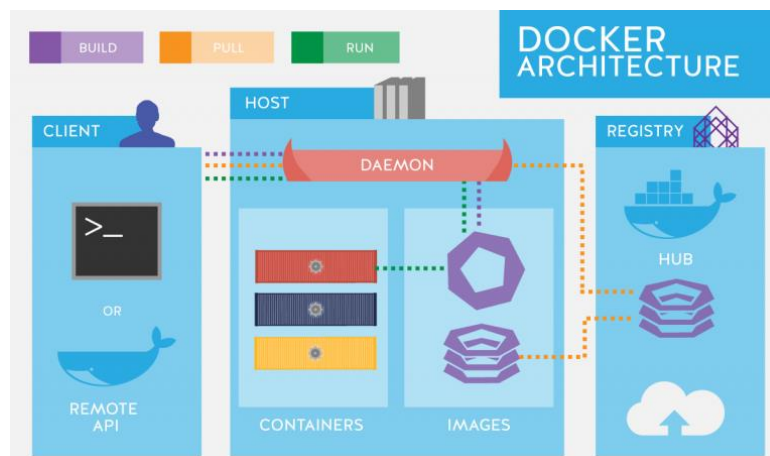


Figure 2 Docker Architecture "<https://thingsolver.com/hello-docker/>"

The services use the environment variables of their container to set up configurations or pass parameters in a running containerized program. Docker allows the easy build and launch of a service while also allowing it to act as an independent environment. Information about the docker build and launch exists in the “dockerfile” of each node. Lastly, prebuilt docker images with all possible technologies and libraries needed usually are utilised by developers to minimise the development and build time. So that the services don’t have to redownload and build everything and can build faster.

Moreover, Docker-compose is a tool that was developed to help define and share multi-container applications. With Compose, one can create a YAML file to define the services and with a single command, can spin everything up or tear it all down. The big advantage of using Compose is that a user can define his/her application stack in a file, and then can easily deploy the multi-serviced application by calling Docker-compose. (Use Docker Compose)

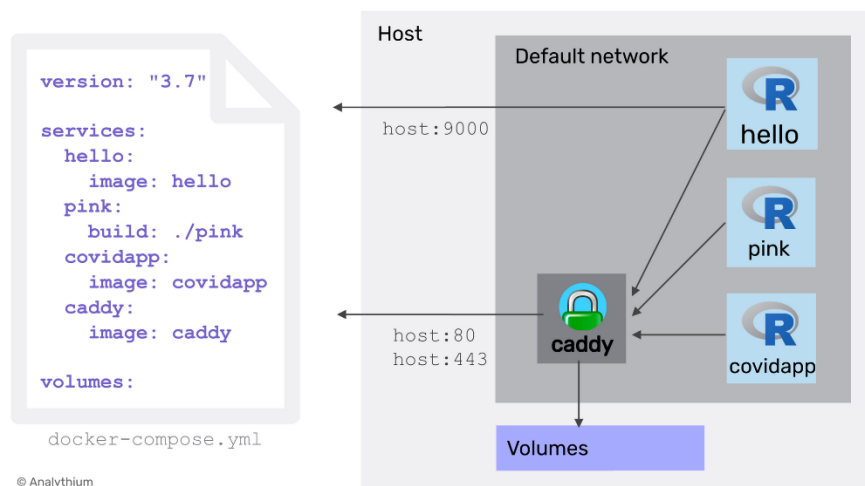


Figure 3 Docker Compose Demo “<https://hosting.analythium.io/shiny-apps-with-docker-compose-part-1-development/>”

Essentially Docker-compose was the glue that allows independent docker containers to launch together as a single architecture. Through it, one could dynamically set up each container's environment variables and easily stop a launch after a specified time and relaunch the same but with different application properties (e.g., ML’s configurations). Lastly, it creates its own network so that each container could easily talk to the other using its name as URL and did not need further setup.

## 2.3 Restful APIs and Server programming

Restful APIs became the default way for service-to-service communication in a microservice-based application. Flask is a web framework, it's a Python module that lets you develop web applications easily. (What is Flask Python) For instance, a server could receive a post request from the workloader that contains the encoded images, and has to process, running ML and/or pre-processing, forward the compressed image to the cloud. A server can be supposed to store or retrieve data from a Database. Furthermore, communication between backends and underlying operating systems can be conducted via automated bash script execution. For instance, a program may run by creating a dynamic bash script that runs Docker-compose, waits a specific time and then stops the architecture. A bash script is a file in which Linux commands are written and can be run in a sequence.

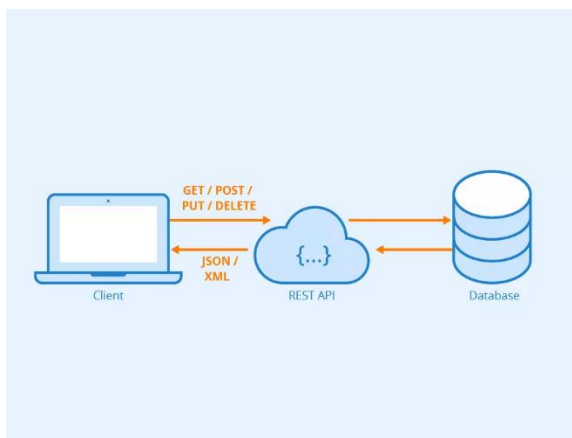


Figure 4 REST API Demonstration "<https://www.astera.com/type/blog/rest-api-definition/>"

## 2.4 ML and Data Analysis tools

Machine learning (ML) is the study of computer algorithms that can improve automatically through experience and by the use of data. It is seen as a part of artificial intelligence. Machine learning algorithms build a model based on sample data, known as training data, in order to make predictions or decisions without being explicitly programmed to do so. Machine learning algorithms are used in a wide variety of applications, such as in medicine, email filtering, speech recognition, and computer vision, where it is difficult or unfeasible to develop conventional algorithms to perform the needed tasks. (Machine learning) ML models are often very computationally expensive to train and run so they are one of the most common uses of using the cloud, as cloud providers like Microsoft have developed products such as Azure Machine Learning to have better efficiency and pricing on such specific use cases. Moreover, in

most cases the data fed into them needs to be pre-processed so edge computing can help their application even more.

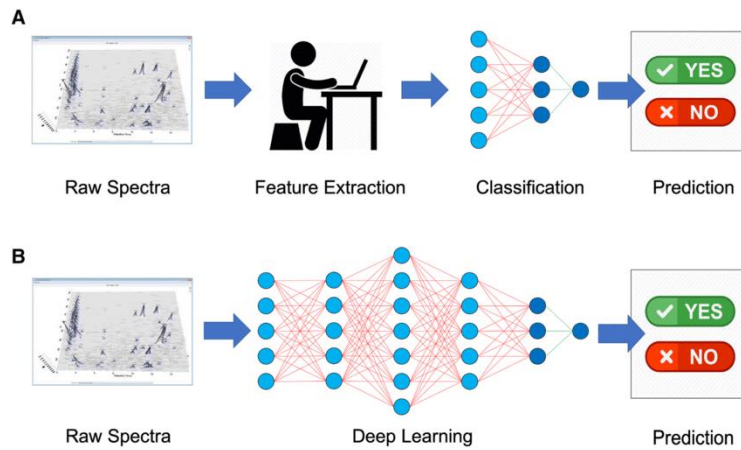


Figure 5 Machine learning illustration "[https://www.researchgate.net/figure/Illustration-of-the-differences-in-complexity-and-implementation-between-traditional\\_fig3\\_336660735](https://www.researchgate.net/figure/Illustration-of-the-differences-in-complexity-and-implementation-between-traditional_fig3_336660735)"

PyTorch is an open-source machine learning (ML) framework based on the Python programming language and the Torch library. It is one of the preferred platforms for deep learning research. The framework is built to speed up the process between research prototyping and deployment. The popularity of PyTorch continues to rise as it simplifies the creation of artificial neural network (ANN) models. PyTorch is mainly used for applications of research, data science and artificial intelligence (AI) (Lewis). Fifty-one library provides 5 CNN ML models that can perform image classification and with a few lines users could download a pre-trained ML model and be able to get its predictions on the sample images.

To manipulate ML datasets and generally data in python, Pandas is a well-known open-source Python package that is most widely used. Specifically, scientists utilise pandas for data science/data analysis and machine learning tasks. It is built on top of another package named Numpy, which provides support for multi-dimensional arrays and a high-performance in-memory data structure that amplifies the performance of data processing. As one of the most popular data wrangling packages, Pandas works well with many other data science modules inside the Python ecosystem and is typically included in every Python distribution. (What Is Pandas in Python? Everything You Need to Know, 2021)

Except for the purely ML libraries, there are also utility tools that help developers in ML pipelines and implementation. For instance, FiftyOne is the first open-source tool that empowers CV/ML engineers and scientists to rapidly evaluate their datasets and models. FiftyOne helps developers to visualise and explore datasets, identify key scenarios, and debug your models, enabling you to build better ML models more quickly. (FiftyOne — FiftyOne 0.15.0)

documentation) It provides an easy way for downloading datasets, querying a number of images and their annotations and encoding them to be sent over the network. Moreover, it allows for ML models to run predictions on the images and provides performance evaluations. Each dataset is represented by “Samples” so the user can select the size of a sample for his/her project. Then, samples are stored in specific dictionaries and reverse, so users can reuse them at any future time. Additionally, each sample can have multiple tags attached to it which can help in predictions and evaluation of each ML algorithm.

Last part of an analytic process is the data plotting and graph presentation. Matplotlib is a comprehensive library for creating static, animated, and interactive visualisations in Python. (Matplotlib — Visualization with Python) This library can be used to create plots of the data that users want to analyse and display to visualise their analysis results. Some examples of charts are bar charts, scatter plots, box plots, timelines, and, generally, every commonly used plotting technique. So, users can compare diverse metrics on the same plot or graph in order to extract intuitive information and insights about their metrics.

## 2.5 Related work to get familiar with the concept

When we started this journey, we first had to understand what microservice applications were and why there are used as a default architecture in modern applications. There is a plethora of positives of this approach, and we needed to see some implementation examples as well as some papers on the architecture's performance and ease of use. So, we read the below sources to get better acquainted with it.

- <https://www.csl.cornell.edu/~delimitrou/papers/2019.asplos.microservices.pdf>
- <https://www.slideshare.net/DemetrisTrihinas/designing-scalable-and-secure-microservices-by-embracing-devopsasaservice-offerings>
- [http://linc.ucy.ac.cy/assets/files/publications/pdfs/cloudcom\\_v1.pdf](http://linc.ucy.ac.cy/assets/files/publications/pdfs/cloudcom_v1.pdf)
- <https://github.com/microservices-demo/microservices-demo>
- <https://github.com/GoogleCloudPlatform/microservices-demo>
- <https://github.com/UCY-LINC-LAB/CloudCom2018-Tutorial>

## 2.6 Helpful documentation

When it was time to get into implementing our system there were a lot of technologies and subjects that we had to get familiar with in order to make everything run together seamlessly. Firstly, we needed to read the documentations on Docker and Docker-compose to make an application run on multiple containers and communicate with each other. Then we needed to read the Flask Server RESTful documentation to make it listen on requests and Fifty-One documentation to understand how to work with datasets and samples. Lastly, we went over which object detection models Pytorch has and how to download them pretrained and use them. Some of the sources we needed for the above are below:

- <https://docs.docker.com/engine/>
- <https://docs.docker.com/compose/>
- <https://flask.palletsprojects.com/en/2.1.x/>
- [https://voxel51.com/docs/fiftyone/user\\_guide/evaluation.html](https://voxel51.com/docs/fiftyone/user_guide/evaluation.html)
- [https://pytorch.org/tutorials/intermediate/torchvision\\_tutorial.html](https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html)

# Chapter 3: Systems overview

---

## 3.1 High level overview

### 3.1.1 Description:

### 3.1.2 Pipeline of operations:

## 3.2 Targeted users and what to expect from the system:

### 3.2.1 ML engineers:

### 3.2.2 Dev-ops engineers:

### 3.2.3 Architecture optimizers:

---

## 3.1 High level description:

### 3.1.1 Description:

In short, we have made a simple edge to cloud architecture in dockerized containers. Each edge container receives several images from a workload container, from there the edges can pre-process the images and/or run an ML algorithm on them (simulating edge computing) and then sending their results and the images to a cloud service container. The most powerful part of our system is that it's easy to configure each launch parameters as we will see below.

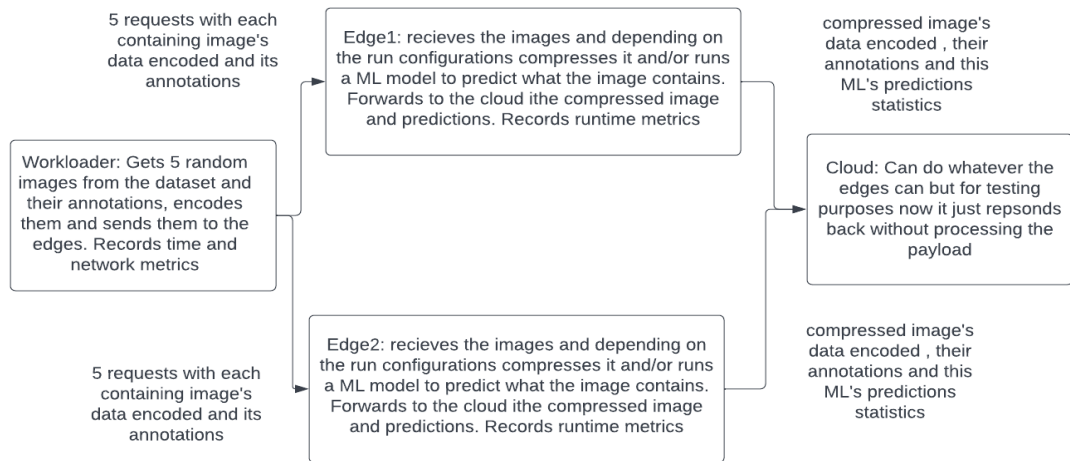


Figure 6 Architecture overview

I have made it so that this whole procedure can be launched automatically and dynamically using docker-compose to launch different configurations and variations of the architecture. We can tweak stuff like how much compression should be done in an image, what ML model each edge runs, how many images are sent and more.

I have also implemented some metric data collection while the architecture runs like how much time each image takes, how much network traffic is sent/received and more.

Lastly, after collecting the data, we have made some analysis on our public google colab file: <https://colab.research.google.com/drive/1MkReYIf3Qu63cK5pZX87qUCGNbxZtw6?usp=sharing>

In which we extract some results on the effects of the different ML models and pre-processing.

### 3.1.2 Pipeline of operations:

Firstly, we launch the project by running a python file that dynamically changes the environment of the workspace thus dynamically changing the runtime configuration of the architecture's nodes. Then it calls upon Docker-compose to build and run the project for a predetermined number of minutes. When the time passes, the project stops, and a new environment is written for the new runtime and the whole process repeats until all the configurations are done.

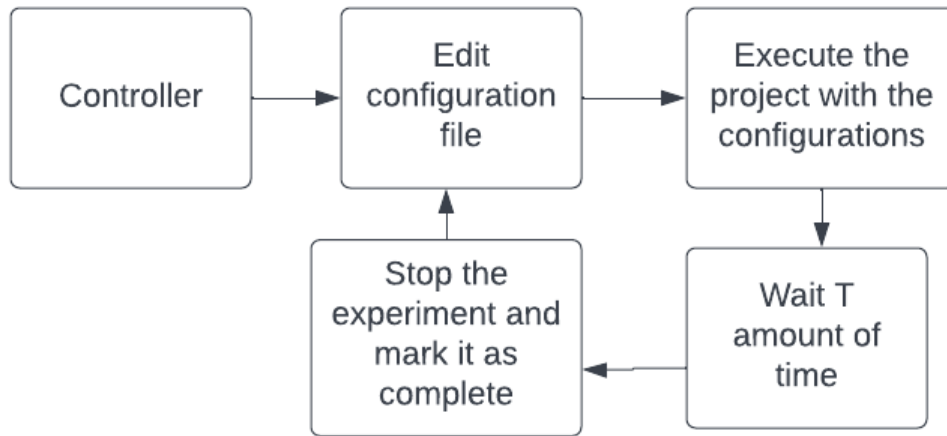


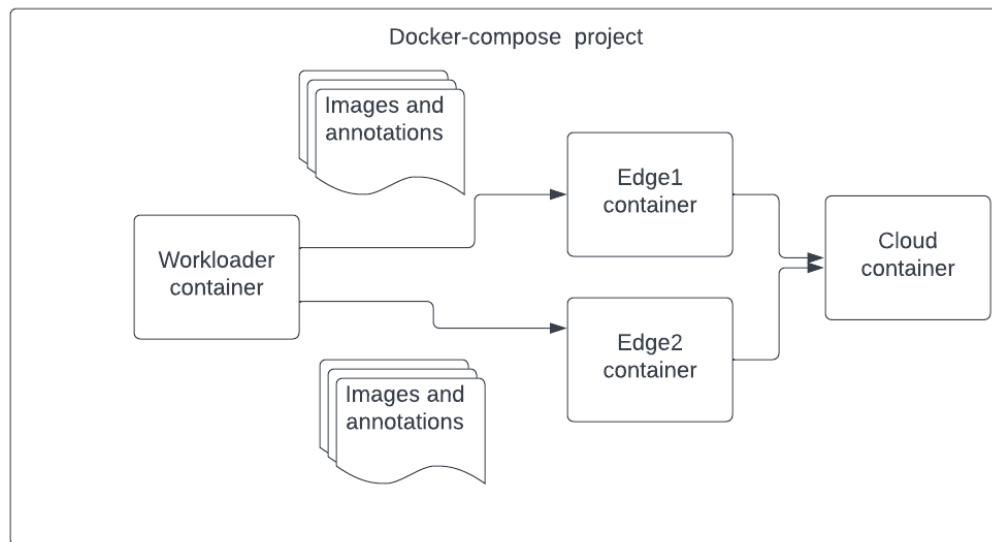
Figure 7 Visualization of how relaunching the project works

The configurations that our current project experiments with are how many edges there are, the ML model and which pre-processing options each edge runs and how many seconds the HW for that container sleeps for to get the new data. What we mean by pre-processing options, we currently support making an image black and white, reducing its width and height dimensions and reducing the quality of the stored image.

So, when we need to create 2 edge configurations (we have 2 processors to work with), we let the python file edit the configuration file to add the appropriate environmental variables for each edge (check section [4.2.4](#)), and creates a docker-compose.yml file with all the services required and the environment variables names each was assigned above. Then the python file creates a bash file that consists of 3 commands. Start the docker-compose project in the background, sleep for T time and then call docker-compose stop.

As mentioned above, the workload node sends an N number of random images and their annotations X times to an edge. When this process is done X times, the workload goes to the next edge and does the same. So, for all the time T the project is up, an  $N \cdot X$  number of images are sent to each edge circulating each one so its fair and have chance to test out their predictions.

When an edge receives the images, it can run an ML algorithm to predict what the images contain and/or run some pre-processing on it which can reduce the quality and/or the image dimensions. It then sends the new image and its results to the cloud container.



*Figure 8 Docker-compose containers communication visualized*

When it comes to the metrics we collect, from all the nodes we collect some HW metrics like CPU cycles, RAM usage and network traffic (sent and received) with an internal timer that we specify in the environments, currently it is set at 4 seconds.

The edges since they are running an ML model and pre-process the image, also collect the predictions and prediction metrics, much pre-processing affected the image size and how much time each of these took.

Lastly with the metrics we collected we extract some information like which ML is the most accurate, which was the fastest, how pre-processing effects the predictions and more. You can view our result for the fundamental project at [section 6](#).

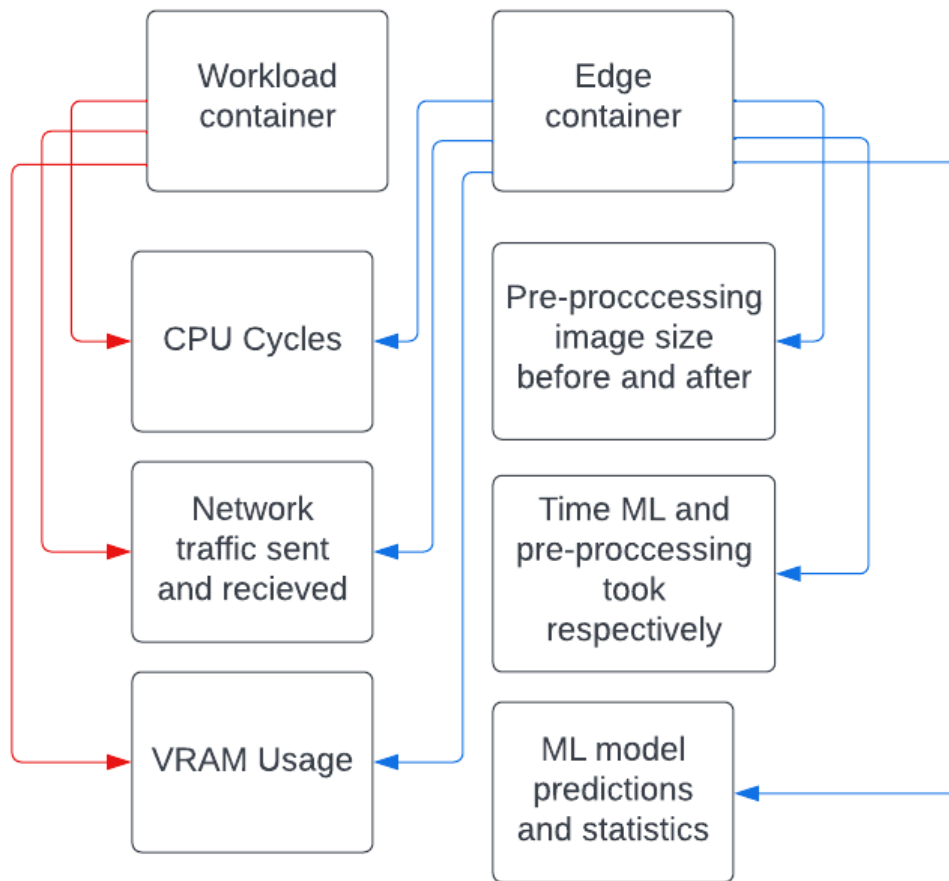


Figure 9 Visualization of which data each node collects

## 3.2 Targeted users and what to expect from the system:

As we have already mentioned this system is targeted towards ML engineers, Dev-ops engineers, Architecture optimizers or in general anyone who wants to build a multiservice application which utilises ML. A user should expect a system that on runs out of the box a Machine Learning multiservice application that solves an object recognition problem. The system should run smoothly and collect data as we mentioned above. From there on the user can come and alter any component of the system with their implementations and the rest will stay intact so they can still for example say communicate between services or collect data.

### 3.2.1 ML engineers:

A machine learning engineer can use the project to test out multiple models and their performance in a real runtime scenario. For example, if the engineer has created 10 different ML models all specialising in object detection, then it will be extremely easy in adapting the current source code to exchange current ML models to her/his. She/He will simply have to replace the model names in the `runconf.py` as well as implement her/his own function of

predicting and gathering metrics and replacing the current implementation. The project will run the same and she/he will be able to extract the results by running the colab file with his own zip of the source.

### 3.2.2 Dev-ops engineers:

The project uses a lot of technologies which are vital for developing application back-ends and in general a multiservice app. Meaning that the engineer can easily take any part of the architecture and use it in developing its own implementation of a similar project.

She/He could simply use the N number of already preconfigured http servers to run distributed computing algorithms and respond back with their results. In general, she/he could use it a fundamental and basic multiservice application and from there, extend it to serve an even bigger purpose while also collecting various important metrics.

### 3.2.3 Architecture optimizers:

The Docker-compose that the project utilizes to launch is an amazing tool in the modern world, since it helps launching a project with a single command anywhere.

So, an architecture engineer could easily take our source code and replace and add other nodes to do other stuff. It is quite easy to change what the workloader sends to the edges. Moreover, they can also increase the number of edges and make them handle the new data to serve a different purpose. The core architecture will still be the same and capture the HW metrics so the engineer will be able to measure the performance and then optimize his project.

# Chapter 4: Implementation

---

## 4.1 Implementation details:

### 4.1.1: Dockerfiles and Docker-compose information:

#### 4.1.1.1: Common elements seen in the compose YAML file:

#### 4.1.1.2: Cloud container:

#### 4.1.1.3: Edges container:

#### 4.1.1.4: Workload container:

### 4.1.2: Dockerfile information:

### 4.1.3: Flask Server information:

### 4.1.4: Fifty-one information:

## 4.2 How to run and gather results:

### 4.2.1 Requirements:

#### 4.2.1.1 Software Requirements

#### 4.2.1.2 Hardware Requirements

### 4.2.2: Building the Base\_image:

### 4.2.3: How to run the default project configurations:

### 4.2.4: How to edit the configurations:

### 4.2.5: Where to view the results:

### 4.2.6: Interpreting the csv's:

#### 4.2.6.1: Edge Request csv's:

#### 4.2.6.2: Workload Request csv's:

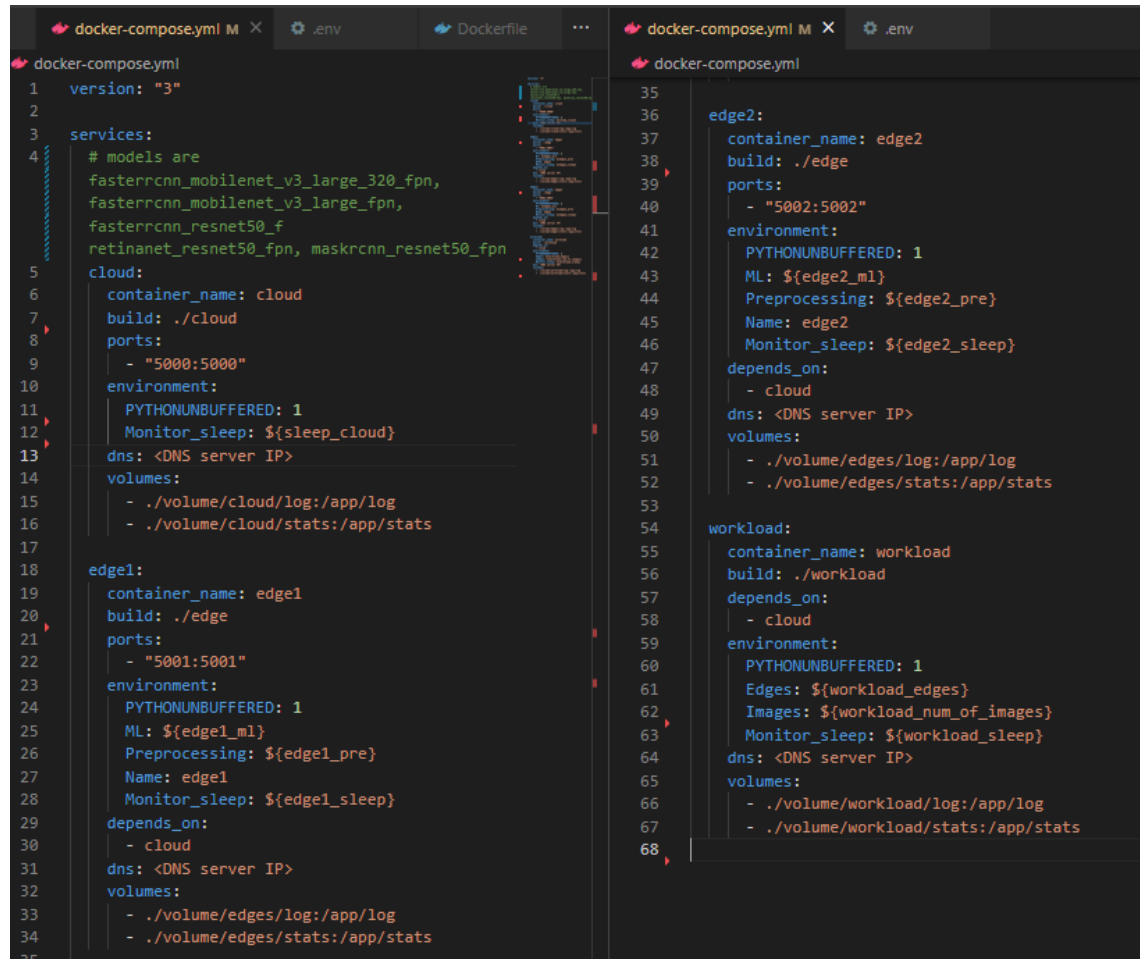
#### 4.2.6.3: Edge and Workload Monitor csv's:

---

## 4.1 Implementation details:

### 4.1.1 Dockerfiles and Docker-compose information:

Here is the Docker-compose file:



```
docker-compose.yml
1 version: "3"
2
3 services:
4   # models are
5   fasterrcnn_mobilenet_v3_large_320_fpn,
6   fasterrcnn_mobilenet_v3_large_fpn,
7   fasterrcnn_resnet50_f
8   retinanet_resnet50_fpn, maskrcnn_resnet50_fpn
9   cloud:
10    container_name: cloud
11    build: ./cloud
12    ports:
13      - "5000:5000"
14    environment:
15      PYTHONUNBUFFERED: 1
16      Monitor_sleep: ${sleep_cloud}
17    dns: <DNS server IP>
18    volumes:
19      - ./volume/cloud/log:/app/log
20      - ./volume/cloud/stats:/app/stats
21
22   edge1:
23    container_name: edge1
24    build: ./edge
25    ports:
26      - "5001:5001"
27    environment:
28      PYTHONUNBUFFERED: 1
29      ML: ${edge1_ml}
30      Preprocessing: ${edge1_pre}
31      Name: edge1
32      Monitor_sleep: ${edge1_sleep}
33    depends_on:
34      - cloud
35    dns: <DNS server IP>
36    volumes:
37      - ./volume/edges/log:/app/log
38      - ./volume/edges/stats:/app/stats
39
40   workload:
41    container_name: workload
42    build: ./workload
43    depends_on:
44      - cloud
45    environment:
46      PYTHONUNBUFFERED: 1
47      Edges: ${workload_edges}
48      Images: ${workload_num_of_images}
49      Monitor_sleep: ${workload_sleep}
50    dns: <DNS server IP>
51    volumes:
52      - ./volume/workload/log:/app/log
53      - ./volume/workload/stats:/app/stats
54
55   edge2:
56    container_name: edge2
57    build: ./edge
58    ports:
59      - "5002:5002"
60    environment:
61      PYTHONUNBUFFERED: 1
62      ML: ${edge2_ml}
63      Preprocessing: ${edge2_pre}
64      Name: edge2
65      Monitor_sleep: ${edge2_sleep}
66    depends_on:
67      - cloud
68    dns: <DNS server IP>
69    volumes:
70      - ./volume/edges/log:/app/log
71      - ./volume/edges/stats:/app/stats
```

Figure 10 Docker-compose YAML file

As you can here, we declare that we want a total of 4 containers being deployed.

#### 4.1.1.1 Common elements seen in the compose YAML file:

Before starting, there are some common elements you will see in the docker-compose YAML file and the dockerfiles. Whenever you see `${var}`, it means that that value is located in the `.env` file. For example, one line of the `.env` file at one runtime instance is “`edge2_ml = 'fasterrcnn_resnet50_fpn'\n`”. Meaning that when the project launches the docker container `edge2` will have in its environment a key-value pair (`'ML' , ' fasterrcnn_resnet50_fpn'` ). In each configuration of a launch, these values change depending on the `.env` file.

Additionally, the “`PYTHONUNBUFFERED:1`” means that we want the stdout to be on the console when we run the containers, “`dns: <DNS server IP>`” helps the docker containers use the established dns of the host to send data to other containers by a URL like

<http://edge1:5000/endpoint> and "monitor\_sleep" is how many seconds the containers monitors sleeps until it records the results.

Lastly the most vital part each container has it the volume arguments. They are used as way to bind directories between the host and the container so that we have persistent storage. The data the container contains are also saved on our local directory as well as the original container, so when we stop the container, we still have the data generated.

#### 4.1.1.2 Cloud container:

Starting with the cloud, we do not have much environmental variables since in the current case as soon as it gets forwarded the image and an edge results it immediately responds back without doing anything. It is capable though of running its own ML model or doing many stuffs with the data it collects, but it was not the main point of this experiment so for now it's not utilised.

#### 4.1.1.3 Edges containers:

As you can see, we have 2 services called "edge1" and "edge2" each with their own set of environmental variables. As you can probably guess, we can easily add an N number of edges in the project as long as we update the runconf.py to set their environmental variables as well to run. The ML environmental variable can be one of the 4 PyTorch ML models we work with and the pre-processing can be a string of "BW", "quality" or "resize" with their values like, for example we can have "edge1\_pre = 'BW,1,quality,50%' and edge2\_pre = 'BW,1,resize,50%,quality,25%'" and try to avoid spaces.

#### 4.1.1.4 Workload containers:

As you can see the last service is the "workload" container which is responsible for sending the images to the edges. Here you specify the edges in the "Edges" variable like "workload\_edges = 'edge1,edge2,edgeN'" and images is how many images to send to each edge, and the final one is how many iterations. For example, "workload\_num\_of\_images = '1,1,5'" means that we will send 1 image to edge1 and 2 5 times in a row and then switch.

#### 4.1.2 Dockerfile information:

As mentioned above all the above are Docker containers which at their core are a single file which contains instructions on how to build the container.

```
1 FROM base_image:latest
2
3 COPY . /app
4
5 WORKDIR /app
6
7 CMD ["python3", "app.py"]
8
```

Figure 11 Edge container's Dockerfile

As you can see, the dockerfile for an edge simply says from the docker image "base\_image:latest" which we built above, copy everything to an "app" folder in the container,

use that a working directory and then always start by executing the command “python3 app.py” which starts the flask server and its ready to go.

#### 4.1.3 Flask Server information:

All edges and the cloud containers run a Flask application. They all start running on host “0.0.0.0” so that they can be accessible by anyone and listen on port 5000 on endpoint “endpoint”. Meaning that when the workloader wants to send a post request containing a json of information we use

```
requests.post(" http://edge1:5000/endpoint", json=json.dumps(
    DictionaryOfStuffToSend), proxies={"http": None, "https": None, })
```

Then we have the app.py which at its core we have

```
@app.route('/endpoint', methods=['POST'])
async def hello():
    content = json.loads(request.get_json())
```

Which we then use the content dictionary to do what we want.

We also use in these some functions which are worth mentioning like

```
image = Image.open(BytesIO(b64decode(content[0][data])))
```

Which we use to create a PIL image by the encoded bytes of each of the request’s image.

#### 4.1.4 Fifty-one information:

As we mentioned in the beginning, Fifty-one is a library which makes it easy to work with datasets.

We first need to have access to a dataset by downloading it locally. Thankfully all our nodes are built on the “base\_image” which already has downloaded the whole dataset so all of them have access to it.

Each fifty-one dataset consists of samples. So, when we choose N number of images we call

```
dataset.take(images[i])
```

and then for each sample of the above view of the dataset. We convert the sample to a dictionary (sample.to\_dict()) , we encode the original image and add it to the data to be sent by

```
sample['data'] = b64encode(open(sample.filepath, "rb").read()).decode('utf-8')
```

and then post to the edge the json.dumps(sample).

When an edge or cloud flask receive a request, as we saw they decode the string to a dictionary of samples.

## 4.2 How to run and gather results:

### 4.2.1 Requirements:

In order to run the `runconf.py` file which basically automatically runs the project you need to fulfil specific hardware and software requirements.

#### 4.2.1.2 Software requirements:

You are required to run in a Linux distribution OS which has Docker and Docker-compose set up. Moreover, you need to have a python3 interpreter and be available on the `$HOME` variable so that you can call it from a terminal.

#### 4.2.1.3 Hardware requirements:

It is suggested to have at least a 2-core processor and 8 Gb of RAM in order to be able to handle the Docker-compose launch. Lastly at least 10 GB of free memory are needed to be able to store the images and the created data.

### 4.2.2 Building the Base\_image:

Firstly, it is needed to build the “Base image” which is an Ubuntu Docker image with all necessary libraries needed. It takes approximately 7GB of memory and consists of all python libraries needed to run the project, the COCO 2017 Validation dataset (2GB) and about 1GB of 5 pretrained Machine learning models from pytorch. Building the base image takes dozens of minutes but it is worth it since creating it will make it so when building all other containers build instantly since all their data and dependency is already downloaded.

To build it, simply navigate to the root project directory (`cd Diplomatic_project`) and run  
`‘ docker build --rm -f "base_image/Dockerfile" -t base_image:latest "base_image" ‘`

### 4.2.3 How to run the default project configurations:

The project runs on the background by navigating to the root directory and running “`nohup python3 run_conf.py &`” which runs the python file in the background. From there on you can check out the `progress.txt` for updates on what has configuration has been ran until the project stops, and all configurations are done. While running, files will be created to the volume directory in the project containing runtime results.

If at any case you want to stop the process entirely before all configurations are done. You can kill the python command and then calling “`docker-compose stop`” to stop the containers as well.

Last thing you should do is “./clear\_logs.sh” which cleans up the volume directory so that no leftover data are left from the previous executions.

One example of the progress.txt is:

1. BW,0,quality,50%,resize,75%,BW,0,quality,50%,resize,75%,0,fasterrcnn\_mobilenet\_v3\_large\_320\_fpn,maskrcnn\_resnet50\_fpn,13\_59\_14\_04
2. Pog
3. DONE,14\_06\_14\_04
4. BW,0,quality,50%,resize,75%,BW,0,quality,50%,resize,75%,0,fasterrcnn\_mobilenet\_v3\_large\_fpn,fasterrcnn\_mobilenet\_v3\_large\_320\_fpn,14\_06\_14\_04

Which indicates that at 13.59 14/4 it started a project launch with without making the images black and white (BW,0), the edges make the image 75% of the original on both width and height (9/16 of the original pixels), and the quality saved is 50% which means the image is not as sharp. Lastly, we can see that the first edge is using as a ML a “fasterrcnn\_mobilenet\_v3\_large\_fpn” whereas the edge2 is running “maskrcnn\_resnet50\_fp”. The “Pog” keyword means that the environment was written and the docker containers started. The Done as implied means that the time limit of the executions successfully is reached. In the default environment each project is launched for 6 mins and then stopped. Which you can see from the next run starting 7 minutes later but now, the model on the second edge is different.

#### 4.2.4 How to edit the configurations:

As mentioned above the `runconf.py` file is the bread and butter of the whole operation.

```
251 if __name__ == '__main__':
252
253     secs = 60 * 6 # 6 mins
254     step_one(secs)

99 def step_one(secs: int):
100
101     for bw in ["0"]:
102         for q in ["", "25%", "50%", "75%"]:
103             for r in ["", "25%", "50%", "75%"]:
104                 for model1 in models:
105                     other = [x for x in models if x != model1]
106                     for model2 in other:
107                         write_new_env(default_env)
108                         str1 = f'BW,{bw}'
109                         str2 = f'BW,{bw}'
110
111                         if q != "":
112                             str1 = f'{str1},quality,{q}'
113                             str2 = f'{str2},quality,{q}'
114                         if r != "":
115                             str1 = f'{str1},resize,{r}'
116                             str2 = f'{str2},resize,{r}'
117
118                         update_env(
119                             'edge1_pre', str1)
120                         update_env(
121                             'edge2_pre', str2)
122
123                         update_env('edge1_m1', model1)
124                         update_env('edge2_m1', model2)
125
126                         curr_time = datetime.now().strftime('%H_%M_%d_%m')
127                         string = f'{str1},{str2},{bw},{model1},{model2},{curr_time}'
128                         write_to_done_file(string)
129                         run_compose(secs)
130
131                         curr_time = datetime.now().strftime('%H_%M_%d_%m')
132                         string = f'DONE,{curr_time}'
133                         write_to_done_file(string)
```

Figure 12 `runconf.py` Default runtime configurations

This project makes it easy to dynamically change the configuration of the edges and the workloader. Here as we can see on the step 1 function. We pass how many seconds we want each launch to take. Then we can use python's list operations to use custom aspects.

We first indicate that we do not want to experiment with black and white images since we found that the default .jpeg image format did not benefit in size when making the image black and white.

Secondly, we indicate that the quality and resize variables will be chosen separately, then the models are chosen so that they are not the same and we write to the current environment our current runtime variables.

This setup will launch  $1 * 4 * 4 * 5 * 4 = 320$  projects, with each taking about 7 mins, so we can safely say that it will run  $320 * 7 \text{ mins} = 37.3 \text{ hours}$  or about 1,5 days.

You can easily modify or add configurations to your liking like running more pre-processing qualities and using a static ML instead.

This all works because in the end we modify the .env file that Docker-compose will look to assign the environment variables to our architecture's containers. So essentially, we dynamically edit the .env file for the project to behave different.

Lastly, the project in each runtime is launched by creating a bash file which we then run. It launches a project in the background, sleeps for the number of second we had as arguments and then calls docker compose stop. When everything stops it leaves and then a new .env is written and the whole process is repeated.

## 4.2.5 Where to view the results:

When a launch is initiated, some new files are created in the /volume directory. There we have 5 different folders. In the current version we only care about 3. The cloud, edges and workload folder. In each of these folders, there 2 subfolders, logs and stats which capture data for each node of the architecture. As the name implies the logs folder contains the logs of the of each of the containers launch and its only there for debugging run time issues.

What we care about is the stats folder. There you will find 2 kinds of files, a monitor and a requests csv we will see more about them below. Each file is unique and identifiable by 4 variables in its name. Each file first starts by a time stamp, then the containers name, which of the 2 kinds of files is and then the runtime parameters.

For example, if we look in the {root}/volume/edges/stats you will see a file a called "04\_15\_02\_11\_edge2\_requests\_retinanet\_resnet50\_fpn\_BW\_0\_quality\_75%\_resize\_75%.csv" This means that this file was created at 15/4 at 2.11 am on the edge 2 container in which it was running the "retinanet\_resnet50\_fpn" ML, No grayscale, 75% original quality, and 75% of the original image dimensions. This way we can sort the results and then easily create a dictionary of each configuration to extract data from. Please see google colab for this implementation.

## 4.2.6 Interpreting the csv's:

### 4.2.6.1 Edge Request csv's:

04\_13\_16\_18\_edge1\_requests\_fasterrcnn\_mobilenet\_v3\_large\_320\_fpn\_BW\_0.csv

item_names	precision	recall	f1_score	support	cpu_cycle	ml_cycles	pre_cycles	image_size	milli_take	total_before_size	total_after_size	total_ml_time_milli	total_pre_time_milli
['couch', 'cat', 'mouse', 'micro avg', 'm']	[0.0, 1.0, 1]	[0.0, 1.0, 1]	[0.0, 1.0, 1]	[1, 1, 1, 3, 2.5E+12]	1.5E+09	3.8E+07	100	188.943	0.138318062	0.13832	721.438	18.156	
['car', 'truck', 'bus', 'micro avg', 'macro']	[1.0, 1.0, 1]	[0.33, 0.5, 0.5]	[0.5, 0.67, 0.67]	[6, 2, 1, 9, 2.5E+12]	2.3E+09	5.3E+07	100	404.831	0.244493484	0.24449	82.017	25.326	
['person', 'skis', 'backpack', 'micro avg']	[1.0, 0.5, 1]	[0.33, 1.0, 1]	[0.5, 0.67, 0.67]	[3, 1, 1, 5, 2.6E+12]	1.5E+09	3E+07	100	987.233	0.138894081	0.13889	732.174	14.341	
['person', 'horse', 'chair', 'accuracy', 'r']	[1.0, 1.0, 1]	[1.0, 1.0, 1]	[1.0, 1.0, 1]	[3.0, 1.0, 1]	2.6E+12	1.7E+09	3.9E+07	100	108.171	0.185777664	0.18578	833.19	18.68
['skis', 'person', 'micro avg', 'macro avg']	[0.0, 1.0, 1]	[0.0, 1.0, 1]	[0.0, 1.0, 1]	[1, 1, 2, 2, 2.6E+12]	2E+09	2.7E+07	100	189.271	0.12120533	0.12121	944.366	12.997	
['motorcycle', 'person', 'bicycle', 'micro']	[1.0, 1.0, 1]	[1.0, 1.0, 1]	[1.0, 1.0, 1]	[1, 1, 1, 3, 2.6E+12]	1.6E+09	4.3E+07	100	71.203	0.215143204	0.21514	749.621	20.37	
['vase', 'micro avg', 'macro avg', 'weight']	[0.25, 0.25, 1]	[1.0, 1.0, 1]	[0.4, 0.4, 0.4]	[1, 1, 1, 1, 2.6E+12]	1.7E+09	4.4E+07	100	66.074	0.21421051	0.21421	806.817	21.098	
['person', 'surfboard', 'accuracy', 'macro']	[1.0, 1.0, 1]	[1.0, 1.0, 1]	[1.0, 1.0, 1]	[1.0, 1.0, 1]	2.6E+12	1.4E+09	4.8E+07	100	905.197	0.211296082	0.2113	657.295	22.676
['stop sign', 'car', 'truck', 'micro avg', 'r']	[1.0, 0.0, 1]	[0.5, 0.0, 1]	[0.67, 0.0, 0.67]	[2, 1, 1, 4, 2.6E+12]	1.7E+09	3.4E+07	100	48.774	0.150271416	0.15027	802.264	16.297	
['carrot', 'broccoli', 'oven', 'spoon', 'mi']	[1.0, 0.22, 0.5]	[0.5, 0.5, 1]	[0.67, 0.3, 0.3]	[8, 4, 1, 1, 2.6E+12]	2E+09	4.7E+07	100	379.33	0.224750519	0.22475	948.179	22.227	
['person', 'keyboard', 'laptop', 'cup', 'r']	[1.0, 1.0, 1]	[0.67, 0.2, 0.8]	[0.33, 0.9, 5, 5, 4, 2.6E+12]	1.8E+09	3.9E+07	100	449.892	0.171022415	0.17102	854.893	18.634		
['car', 'truck', 'bus', 'micro avg', 'macro']	[0.5, 0.0, 1]	[0.33, 0.0, 0.4]	[0.0, 0.3, 2, 1, 6, 2.6E+12]	2.2E+09	3.9E+07	100	333.503	0.191295624	0.1913	44.202	18.427		
['chair', 'cup', 'vase', 'book', 'tv', 'couch']	[1.0, 0.0, 1]	[1.0, 0.0, 1]	[1.0, 0.0, 1]	[2, 2, 2, 1, 2.6E+12]	1.9E+09	3.7E+07	100	265.651	0.17027092	0.17027	926.778	17.697	
['car', 'traffic light', 'person', 'micro avg']	[1.0, 0.0, 1]	[0.54, 0.0, 0.7]	[0.0, 0.13, 3, 1, 2.6E+12]	2.1E+09	3.8E+07	100	406.306	0.184670448	0.18467	993.659	18.146		
['person', 'snowboard', 'micro avg', 'm']	[1.0, 0.0, 1]	[1.0, 0.0, 1]	[1.0, 0.0, 1]	[2, 1, 3, 3, 2.6E+12]	2E+09	4.3E+07	100	216.899	0.218671799	0.21867	940.81	20.537	

Figure 13 An edge request csv

In the current configurations we select 5 random images and send them one at a time in each edge. Then after sending 5 images, we switch and do the same for the other edge. We do this for the whole duration of the project. The project can also be configured to send batches of N images, but we decided to send them one by one for clarity.

For each request an edge receives from the workloader, we add a row to the csv with some stats we have captured.

We first add some stats the for the top 10 items the ML predicted. We add the item names, the precisions, recall, support, f1\_score and then we add some hardware metrics like the time and cycles the ML and pre-processing part took as well as the request as whole.

#### 4.2.6.2 Workload Request csv's:

04\_13\_16\_18\_workload\_requests\_edge1\_1\_edge2\_1\_sleepTime\_4.csv

	A	B	C	D
1		edge_name	total_time_milli	req_times_milli
2	0	http://edge1:5000/endpoint	665.89	[733.57, 559.644, 540.576, 351.068, 480.538]
3	1	http://edge2:5000/endpoint	18.072	[224.129, 432.562, 15.49, 130.367, 215.381]
4	2	http://edge1:5000/endpoint	347.231	[446.513, 485.908, 467.966, 457.227, 489.262]
5	3	http://edge2:5000/endpoint	605.876	[93.874, 93.658, 931.697, 78.258, 408.235]
6	4	http://edge1:5000/endpoint	158.471	[474.034, 418.786, 380.312, 425.469, 459.592]
7	5	http://edge2:5000/endpoint	821.924	[485.231, 363.624, 292.148, 437.775, 243.0]

Figure 14 A workload's Request csv

The name of the csv indicates that it was made at 16:18 on the 13/4 at a workloader container where it was sending one image at each of 2 total edges and the monitor who we will see below sleeps and wakes up after every 4 seconds.

The workloader requests csv's just contain the times each batch of images it took. For example, above we have that for the first request each images took about between about 35m milliseconds to 733. Whereas the second edge was in general faster. There appear to be some edge cases in which sometimes are too little but that's because probably the request was dropped by chance. In the grand scale of things this does not affect us. Moreover, we do an exhaustive data collection with each edge running the same variables as the other at another time so the errors will be distributed in all ML's and Pre-processing configurations. Lastly, we are working with dozens of thousands requests so if errors appear, they will not affect our results at all.

#### 4.2.6.3 Edge and Workload Monitor csv's:

04\_13\_16\_18\_edge1\_monitor\_fasterrcnn\_mobilenet\_v3\_large\_320\_fpn\_BW\_0\_sleepTime\_4

	A	B	C	D	E	F
1		cpu_cycles	KBytes_sent	KBytes_recieved	vram_used_MBytes	ram_active_MBytes
2	0	8.39E+09	5.898.757	6.902.132	327.850.390.625	455.457.421.875
3	1	8.39E+09	541.385	793.23	339.642.578.125	467.001.171.875
4	2	8.39E+09	5784.35	6.206.952	349.568.359.375	476.998.828.125
5	3	8.39E+09	619.719	772.237	350.359.765.625	477.810.546.875
6	4	8.38E+09	5.285.626	6.063.088	366.498.828.125	494.108.203.125
7	5	8.39E+09	0	0	362.816.796.875	490.286.328.125

Figure 15 A monitor csv of a node

As mentioned above, the second type of files are monitor CSVs. Which have a thread that monitors HW metrics every couple of seconds which can be specified at the launch. Currently it was set at 4 seconds.

So, the rows we will be seeing here are the network metrics of each container after resetting the counter and the current Vram our container takes up of the host device as well as the actual ram it takes.

### 4.3 How to use the colab file:

As mentioned above there will be thousands of csv's in the volume directory, so we need an easy way of finding the metrics of the stuff we want. So, we came up with the below code:

```

def get_dict_of_names_per_var():

    dict_of_names = {'pre':{}}

    edges_files_dir = (vol_dir+'edges/stats/')
    workload_files_dir = (vol_dir+'workload/stats/')
    edges_files = os.listdir(edges_files_dir)
    workload_files = os.listdir(workload_files_dir)

    edges_files.sort()
    workload_files.sort()
    iter = -4

    for q in ["", "25%", "50%", "75%"]:
        dict_of_names[q] = {}
        for r in ["", "25%", "50%", "75%"]:
            dict_of_names[q][r] = {}
            for model in models:
                dict_of_names[q][r][model] = {"monitor":[], "requests":[], "workload":{"requests":[], "monitor":[] } }
            for model1 in models_1:
                for model2 in models_2:
                    iter += 4
                    dict_of_names[q][r][model1]["monitor"].append(
                        edges_files_dir + edges_files[iter])
                    dict_of_names[q][r][model1]["requests"].append(
                        edges_files_dir + edges_files[iter+1])
                    dict_of_names[q][r][model2]["monitor"].append(
                        edges_files_dir + edges_files[iter+2])
                    dict_of_names[q][r][model2]["requests"].append(
                        edges_files_dir + edges_files[iter+3])
                    dict_of_names[q][r][model2]["workload"]["monitor"].append(
                        (workload_files_dir + workload_files[int(iter/2)], 0,))
                    dict_of_names[q][r][model1]["workload"]["monitor"].append(
                        (workload_files_dir + workload_files[int(iter/2)], 1,))
                    dict_of_names[q][r][model1]["workload"]["requests"].append(
                        (workload_files_dir + workload_files[int(iter/2) + 1], 0,))
                    dict_of_names[q][r][model2]["workload"]["requests"].append(
                        (workload_files_dir + workload_files[int(iter/2) + 1], 1, ))

```

Figure 16 Code that organises the csv's to each runtime configuration

Which creates a dictionary of many iterations of dictionaries that help determine each csv. Here is a very vital part if you wish to change the configurations. If you create your own configurations, then they must be added here with the same order as we see above so that each csv gets assigned to its own runtime.

\*\*\*\*Do not run this program when the Year changes because then the naming scheme of the csv's gets a reset. If its vital you can sort them by date modified and not name and you should be good.

```

{
  "pre": {},
  "": {
    "": {
      "fasterrcnn_mobilenet_v3_large_320_fpn": {
        "monitor": [
          "/content/Diplomatic_project/volume/edges/stats/04_13_16_18_edge1_monitor_fasterrcnn_mobilenet_v3_large_320_fpn_BW_0_sleepTime_4.csv",
          "/content/Diplomatic_project/volume/edges/stats/04_13_16_25_edge1_monitor_fasterrcnn_mobilenet_v3_large_320_fpn_BW_0_sleepTime_4.csv",
          "/content/Diplomatic_project/volume/edges/stats/04_13_16_31_edge1_monitor_fasterrcnn_mobilenet_v3_large_320_fpn_BW_0_sleepTime_4.csv",
          "/content/Diplomatic_project/volume/edges/stats/04_13_16_38_edge1_monitor_fasterrcnn_mobilenet_v3_large_320_fpn_BW_0_sleepTime_4.csv",
          "/content/Diplomatic_project/volume/edges/stats/04_13_16_44_edge1_monitor_fasterrcnn_mobilenet_v3_large_fpn_BW_0_sleepTime_4.csv"
        ],
        "requests": [
          "/content/Diplomatic_project/volume/edges/stats/04_13_16_18_edge1_requests_fasterrcnn_mobilenet_v3_large_320_fpn_BW_0.csv",
          "/content/Diplomatic_project/volume/edges/stats/04_13_16_25_edge1_requests_fasterrcnn_mobilenet_v3_large_320_fpn_BW_0.csv",
          "/content/Diplomatic_project/volume/edges/stats/04_13_16_31_edge1_requests_fasterrcnn_mobilenet_v3_large_320_fpn_BW_0.csv",
          "/content/Diplomatic_project/volume/edges/stats/04_13_16_38_edge1_requests_fasterrcnn_mobilenet_v3_large_320_fpn_BW_0.csv",
          "/content/Diplomatic_project/volume/edges/stats/04_13_16_44_edge1_requests_fasterrcnn_mobilenet_v3_large_fpn_BW_0.csv"
        ],
        "workload": {
          "requests": [
            [
              "/content/Diplomatic_project/volume/workload/stats/04_13_16_18_workload_requests_edge1_1_edge2_1_sleepTime_4.csv",
              0
            ],
            [
              "/content/Diplomatic_project/volume/workload/stats/04_13_16_25_workload_requests_edge1_1_edge2_1_sleepTime_4.csv",
              0
            ],
            [
              "/content/Diplomatic_project/volume/workload/stats/04_13_16_31_workload_requests_edge1_1_edge2_1_sleepTime_4.csv",
              0
            ],
            [
              "/content/Diplomatic_project/volume/workload/stats/04_13_16_38_workload_requests_edge1_1_edge2_1_sleepTime_4.csv",
              0
            ],
            [
              "/content/Diplomatic_project/volume/workload/stats/04_13_16_44_workload_requests_edge1_1_edge2_1_sleepTime_4.csv",
              0
            ]
          ],
          "monitor": [
            [
              "/content/Diplomatic_project/volume/workload/stats/04_13_16_18_workload_monitor_edge1_1_edge2_1_sleepTime_4.csv",
              1
            ]
          ]
        ]
      }
    }
  }
}

```

Figure 17 Visualization of the runtime configurations dictionary

As you can see above, if we want to see the csv's relating to 25% quality and 25% dimension reduction with the model as X we can call `dict_of_names = get_dict_of_names_per_var()["25%"]["25%"][X]["requests"]` and we can see all the 5 or 4 times the model has been called. This makes it great because the list of csv's that this returns can be dynamically created, and we can get for example all instances of this specific pre-processing by making X be ALL the models.

I have created a function that takes as input the list of csv's a user passes and returns a DataFrame object of all of them combined:

```

def create_df_from_list(q,r,model,type_metric ,type2=None):

    df = pandas.DataFrame()

    ar = []
    if type_metric == "workload":
        ar = dict_of_names[q][r][model]["workload"][type2]

    for tupl in ar:
        df2 = pandas.read_csv(tupl[0],index_col=0)
        df2 = df2.iloc[tupl[1]::2]
        df = df.append( df2, ignore_index=True)

    else:
        ar = dict_of_names[q][r][model][type_metric]
        for file_name in ar:
            try:
                df = df.append(pandas.read_csv(file_name,index_col=0) , ignore_index=True)
            except Exception as e:
                continue
        return df

df = create_df_from_list("25%", "25%",models[0],"monitor")

print(df)

```

	cpu_cycles	KBytes_sent	KBytes_recieved	vram_used_MBytes	\
0	8.390966e+09	2329.032	2853.993	3067.339844	
1	8.390108e+09	381.204	562.686	3218.863281	
2	8.389410e+09	2646.464	3491.885	3389.238281	
3	8.389529e+09	9.689	184.407	3362.851562	
4	8.390627e+09	3225.168	4158.105	3498.117188	
..	...	...	...	...	
424	8.387122e+09	1902.839	2533.057	4136.910156	
425	8.391401e+09	2561.905	3540.820	4194.082031	
426	8.384751e+09	1186.171	1501.089	4208.812500	
427	8.389293e+09	2364.077	3148.395	4263.316406	
428	8.385795e+09	1726.070	2016.460	4219.980469	

Figure 18 How we create a DataFrame from a list of csv's

From there on its just a matter of creating a list of csv's using the method above and using the method below it to have a dataframe to work on.

For example, with simple dynamic model in the appropriate dictionary position, we can read the Dataframe and find how much the average time of a request was.

If you are familiar with python, then it will be trivial to be able to do the same stuff for your records as well.

```

def get_avg_time_all():
    ret = []
    for model in models:
        df = create_df_from_list("", "", model, "workload", t
        if model == models[0]:
            print(df['req_times_milli'])
            times = df['req_times_milli'].to_list()

            i = 0
            sum = 0
            for time_str in times:
                ar = list(ast.literal_eval(time_str))
                if ar:
                    for time in ar:
                        i+=1
                        sum+=float(time)
                    # print(sum/i)
                    ret.append(sum/i)
            return ret
    get_avg_time_all()

```

```

0      [733.57, 559.644, 540.576, 351.068, 480.538]
1      [446.513, 485.908, 467.966, 457.227, 489.262]
2      [474.034, 418.786, 380.312, 425.469, 459.592]
3      [537.54, 490.933, 391.149, 406.866, 433.181]
4      [392.713, 473.807, 366.017, 516.416, 459.73]
...
124     [36.913, 311.332, 898.197, 886.43, 941.101]
125     [319.751, 957.406, 681.793, 116.688, 12.663]
126     [683.483, 474.927, 925.426, 31.621, 28.413]
127     [897.318, 855.789, 338.731, 472.298, 8.675]
128     [66.606, 349.027, 344.329, 283.563, 909.818]
Name: req_times_milli, Length: 129, dtype: object
[476.48762170542625,
 494.3135809523809,
 511.3006297872343,
 544.6161066666667,
 500.9842314285713,
 470.4146409090909,
 495.86163829787216,
 528.3484755555555,
 454.0533830985918]

```

Figure 19 Sample of how to query the DataFrame and get the average time each ML model took to process

Then we can plot the average times. But we will see it below in the results section.

# Chapter 5: Experimentation

---

5.1 Experimentation goals:

6.2 What to experiment to achieve said goals:

6.3 Experimentation details:

6.3.1 Using different models:

6.3.2 Using reduced dimensions:

6.3.3 Using reduced quality:

---

## 5.1 Experimentation goals:

When we were creating the project, we wanted to experiment on how a different architecture effects a multiservice application, how ML models can influence the architectures performance and how image preprocessing can affect image size and prediction accuracy.

So the goals are to experiment a variety of different options and configurations of the SAME architecture so that we can see how each tweak on a subject effect each part of the architecture experimentation goal.

## 5.2 What to experiment with to achieve said goals:

Firstly we have experimented running the application with different ML models to see each model's accuracy compared to the other models and crown a winner. Moreover we wanted to see the performance impact of the models and then crown a new winner which is the fastest but also relatively the most accurate in object recognition.

Secondly, we have experimented with different preprocessing settings on the image. We have firstly reduced the image dimensions to 25%,50%,75% of their original to see how much smaller the file size becomes and how it affects the accuracy. Moreover we have also reduced the image quality to again 25%,50%,75% to see if it has a more positive on negative impact than reducing dimensions.

## 5.3 Experimentation details:

### 5.3.1 Using different models:

Pytorch offers 6 pretrained model architectures on object recognition which you can read about here : <https://pytorch.org/vision/stable/models.html#object-detection-instance-segmentation-and-person-keypoint-detection>

Which have used all of them!

```
(models = [
"fasterrcnn_mobilenet_v3_large_320_fpn",
"fasterrcnn_mobilenet_v3_large_fpn",
"fasterrcnn_resnet50_fpn",
"retinanet_resnet50_fpn",
"maskrcnn_resnet50_fpn" ,
"fcos_resnet50_fpn"
,"keypointrcnn_resnet50_fpn"
,"ssdlite320_mobilenet_v3_large"
,"ssd300_vgg16" ])
```

The models took several GBs for storage but after they were downloaded once they could be used by one image without redownloading. Also there are already some metrics on the models in the above link like their RAM usage which can vary wildly.

The models are chosen by their name which is set as the environment variable ‘edge1\_ml’ = ‘fasterrcnn\_resnet50\_fpn’ for example and so in the edge container we dynamically assign it in the edge:

```
73 | model_name = environ['ML']
74 | method = getattr(models.detection, model_name)
75 | model = method(pretrained=True)
76 | print('Using ', model_name)
77 | model.to(device)
78 | model.eval()
79 |
```

So now in the edge1 we have model = torchvision.models.detection.  
fasterrcnn\_resnet50\_fpn(pretrained = True)

### 5.3.2 Using reduced dimensions:

As you can see above, we can pass a reduction in the image’s dimensions. If we have in an edge’s preprocessing variable ‘resize,X%’ then when the edge preprocesses the image like below:

```
if 'resize' in pref_dict:
    resize = pref_dict['resize']
    if resize[-1] == '%':
        resize = resize[:-1]
    resize = float(resize)/100

    s = image.size
    new0 = float(s[0])*resize
    new1 = float(s[1])*resize
    image = image.resize((int(new0), int(new1)))
```

So if resize was 50% then it becomes  $50/100 = 0.5$  and then we create the new X,Y by multiplying the images dimensions and changing them in the end.

### 5.3.3 Using reduced quality:

Just like above when there is a string in the preprocessing variable like “quality,X%” then we save the image with the new quality:

```
if 'quality' in pref_dict:
    quality = pref_dict['quality']
    if quality[-1] == '%':
        quality = quality[:-1]
        quality = int(quality)

image.save(new_path, quality=quality, optimize=True)
```

# Chapter 6: Evaluation

---

## 6.1 Results overview:

### 6.1.1 Machine Learning models results:

### 6.1.2 Preprocessing results:

### 6.1.3 Object recognition results:

## 6.2 Results explanations:

### 6.1 Machine Learning explanations:

### 6.1 Preprocessing explanations:

### 6.1 Object recognition explanations:

## 6.3 Results comments:

## 6.4 Limitations:

---

## 6.1 Results overview:

Here we will have an overview of the results we have gathered. In total we have gathered data when it comes to the object detection ML models, how the image's size in bytes was affected by the preprocessing/compress it and lastly about the objects the ML's we recognized.

The experiment we have run is the default experiment we above at [section 4.2.3](#). Where we have done a full permutation of all possible configurations. We ran config permutation of [Q,R,ML1,ML2] where Q and R are the options for preprocessing X in [no pre,25%,50%,75%] and ML1 is 1 out of the 9 models and ML2 is all the other except ML1. Meaning we had a total of  $4*4*9*8 = 1152$  runs which took 9 days to complete running.

The experiment was run on an azure virtual machine. It is running "Linux (ubuntu 18.04)" and has as specs "Standard E2s v3 (2 vcpus, 16 GiB memory)". We have been running this VM for free since azure provided €100 worth of runtime to UCY students. In total we have spent the €64 since the day we started working with it. The cost for running the default configuration was about 18€. The other capital was spent in setting up the project, debugging it and running smaller incremental configurations.

## 6.1.1 Machine Learning models:

### 6.1.1.1 Machine Learning models results:

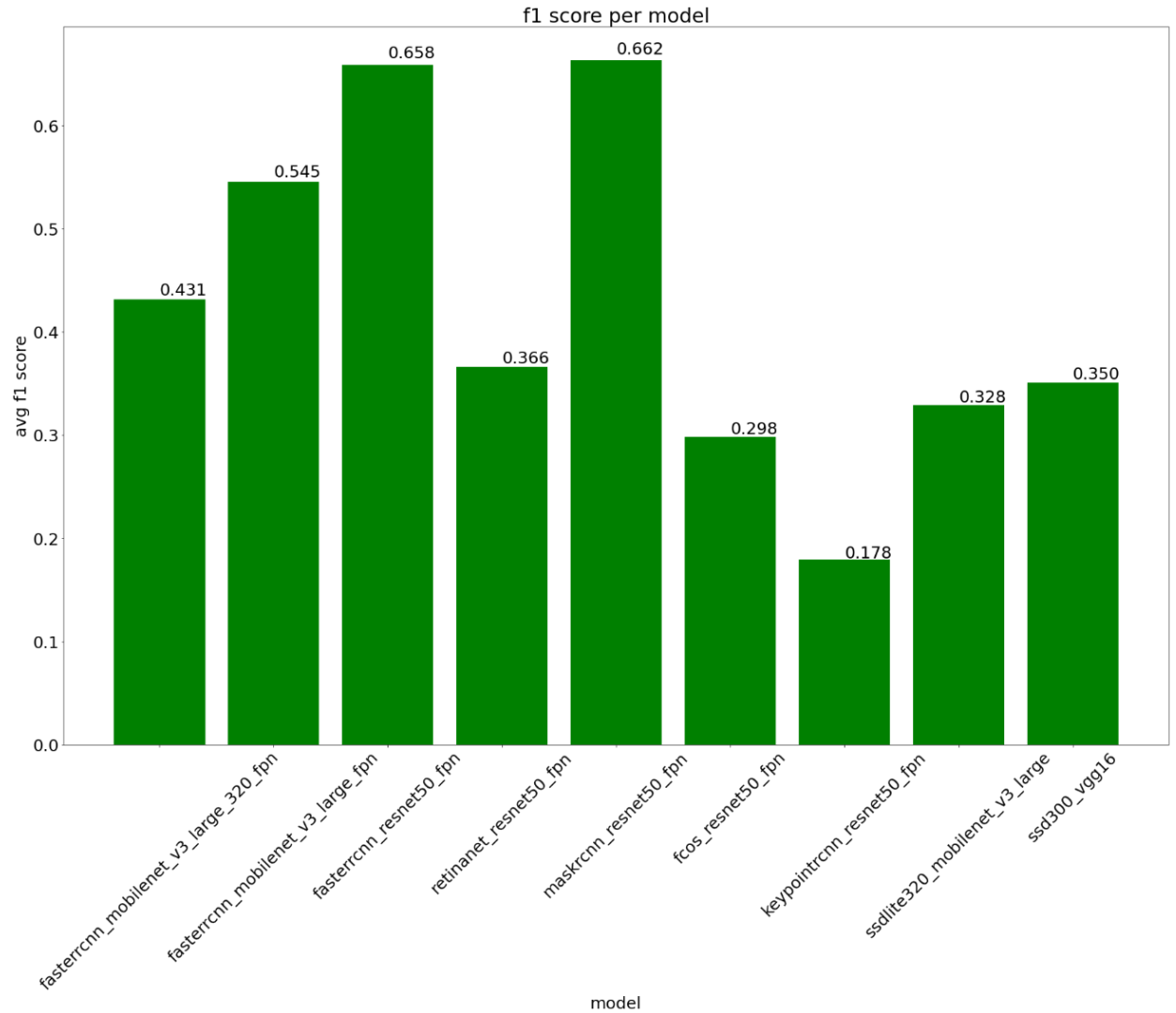


Figure 20 Chart of average f1\_score

Firstly, we have calculated the average f1\_score. We chose this metric because as you can see below:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1-score = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

The F1-score combines the precision and recall of a classifier into a single metric by taking their harmonic mean. It is primarily used to compare the performance of two classifiers. Suppose that classifier A has a higher recall, and classifier B has higher precision. In this case, the F1-scores for both the classifiers can be used to determine which one produces better results.

(<https://www.educative.io/edpresso/what-is-the-f1-score> )

We took into account all the top 10 items in all images that the model was more than 75% confident on them (can be less than 10 as well since we sent only 1 image). Then we summed up the f1\_score for all these items for all the predictions each model had where the image was NOT PREPROCESSED. Then we divided it by their number and got an average f1\_score for each model.

As you can see maskrcnn\_resnet50 and fasterrcnn\_resnet50 were the most accurate with 0.662 and 0.658 avg f1\_score. As you can probably guess by their names let's see if the "faster" is actually faster and be considered a better candidate overall.

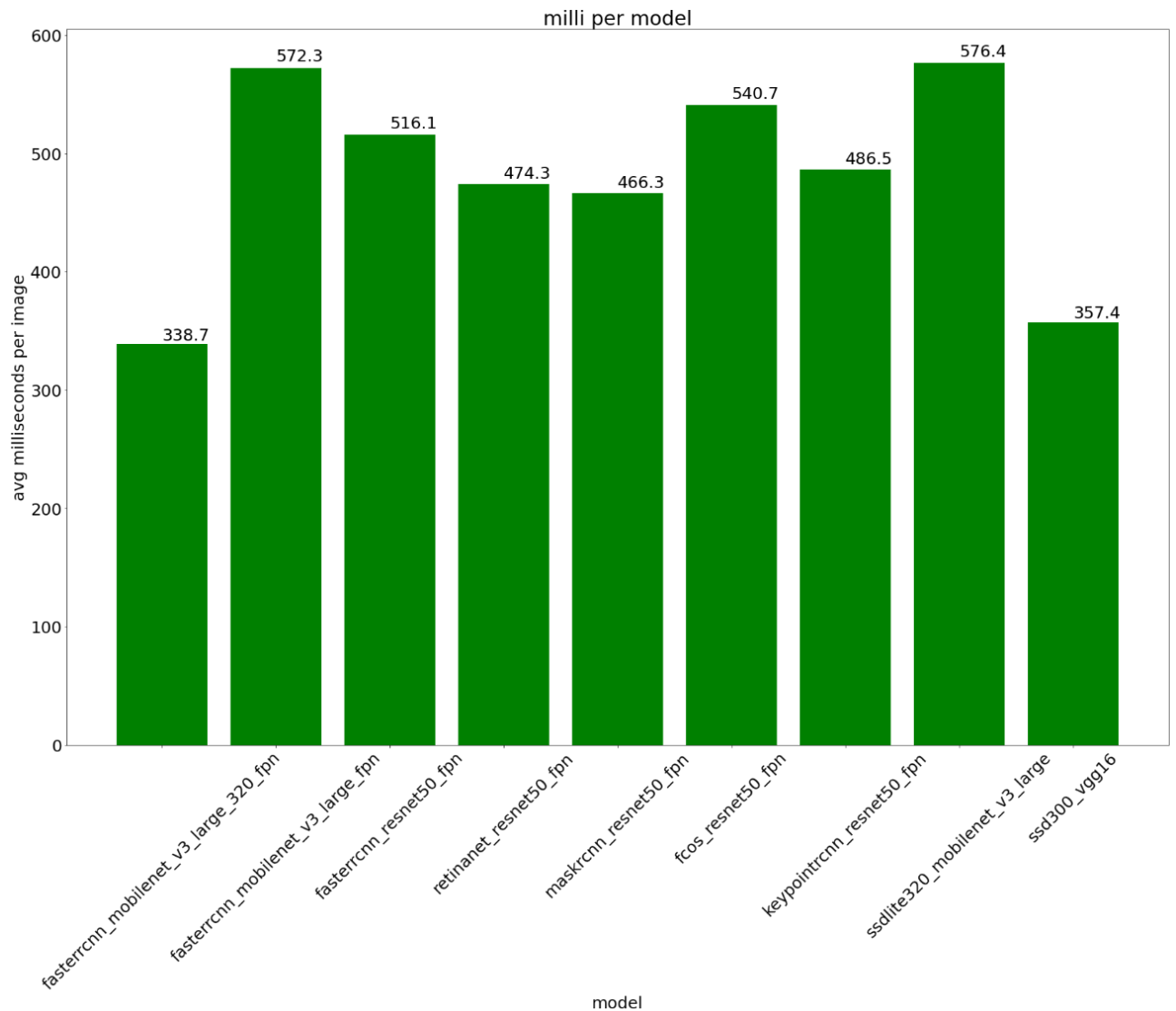


Figure 21 Chart of average milliseconds taken per request

Here we can see the average milliseconds each ML took to respond back WITHOUT PREPROCESSING. We come to the conclusion that ssd300 and faster\_mobilenet were the fastest to respond in general, meaning the detected stuff in an image the fastest. Moreover, maskrcnn was actually faster than fasterrcnn so we can probably guess that below, maskrcnn is the overall winner since it beat both categories.

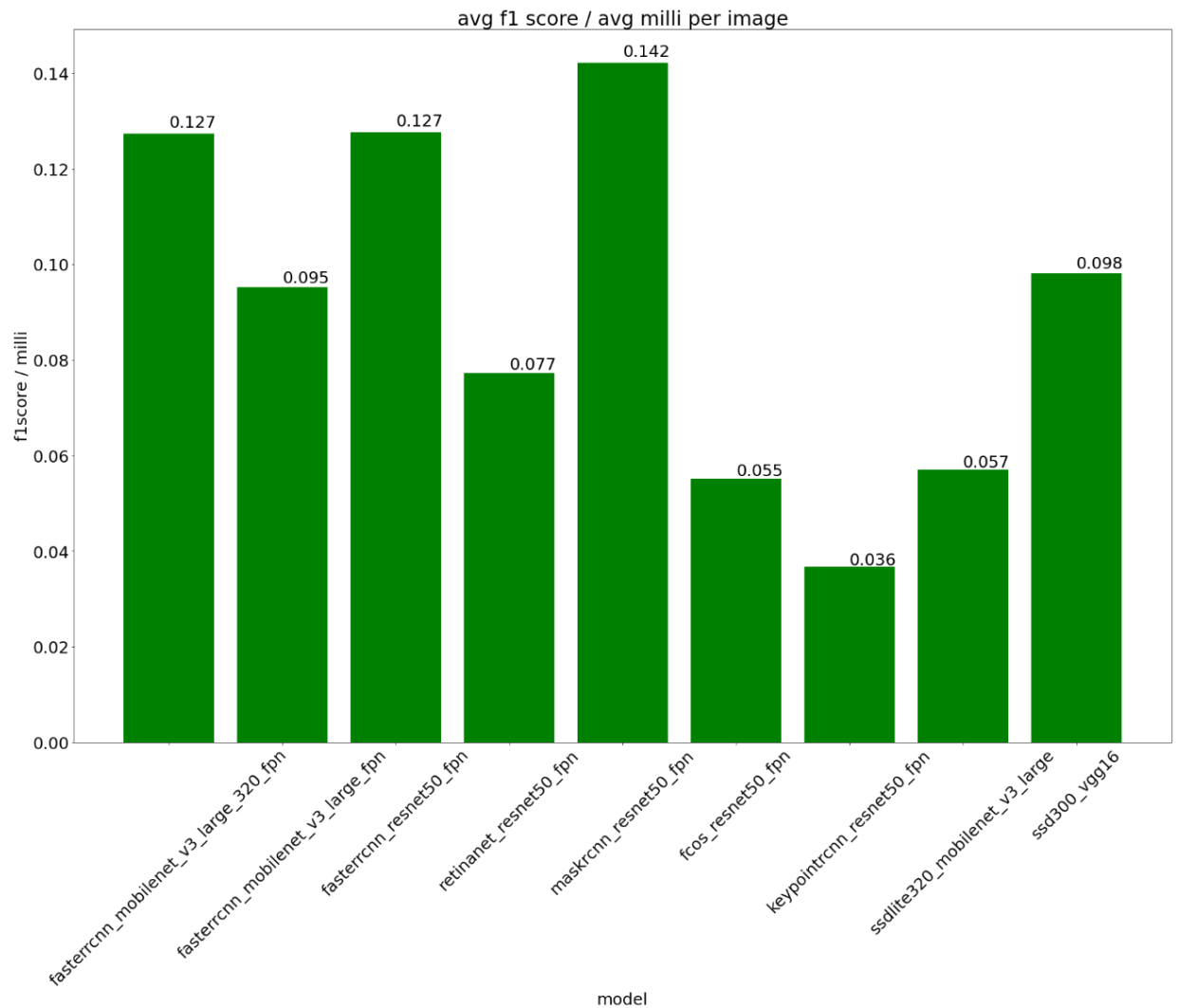


Figure 22 Chart of average f1\_score/milli\_taken

As a metric which would accurately represent an overall ML's performance, we decided to use the average f1\_score divided by the average time. Since we prefer higher accuracy and less time taken the higher this quotient is, the better the ML is overall.

So, as you can see above maskrcnn has the highest f1\_score per millisecond so we can declare it the "best" ML to use for object recognition with the current dataset.

#### 6.1.1.1 Machine Learning explanations:

As for ML model's performances, it's hard to determine a winner because each ML is depended in its implementation in the pytorch library. So technically our results are not 100% accurate since maybe some hyperparameter tuning in the ML could make them better for our purposes or maybe someone could come up with a faster python implementation in the future of the library.

The point is in this fundamental block project, in the current object recognition problem we have come up with. The models ran well and had nice and clear results. A person interested in the project can easily put his own ML's inside and declare a winner as well with some custom statistics to go along with.

#### 6.1.2 Pre-processing :

##### 6.1.2.1 Preprocessing results:

Quality of an image in as in our terms, is how compressed in JPEG format the image becomes. The amount of JPEG compression is typically measured as a percentage of the quality level. An image at 100% quality has (almost) no loss, and 1% quality is a very low quality image. In general, quality levels of 90% or higher are considered "high quality", 80%-90% is "medium quality", and 70%-80% is low quality. Anything below 70% is typically a very low quality image.

( <https://fotoforensics.com/tutorial.php?tt=estq#:~:text=The%20amount%20of%20JPEG%20compression,%25%2D80%25%20is%20low%20quality.> )

For example, on the top half we have on the left the image with 90% quality and on the right the "baseline" of that image which shows that the edges are not very "crisp" but we can easily tell what's in the image.





The below half above is the same image but with 70% quality. Edges are no longer crisp and compression artifacts are visible. E.g., the base of the pitcher's curved handle (behind the nearly empty glass) is very blurry and distorted.



Lastly, above we see the same image but with now at 20% quality. Significant JPEG artifacts are visible around all edges. Most appear as ripples and echo lines. But we can still distinguish the objects somewhat.

In our project we have used 4 quality options, no quality change and 75%,50%,25% quality.

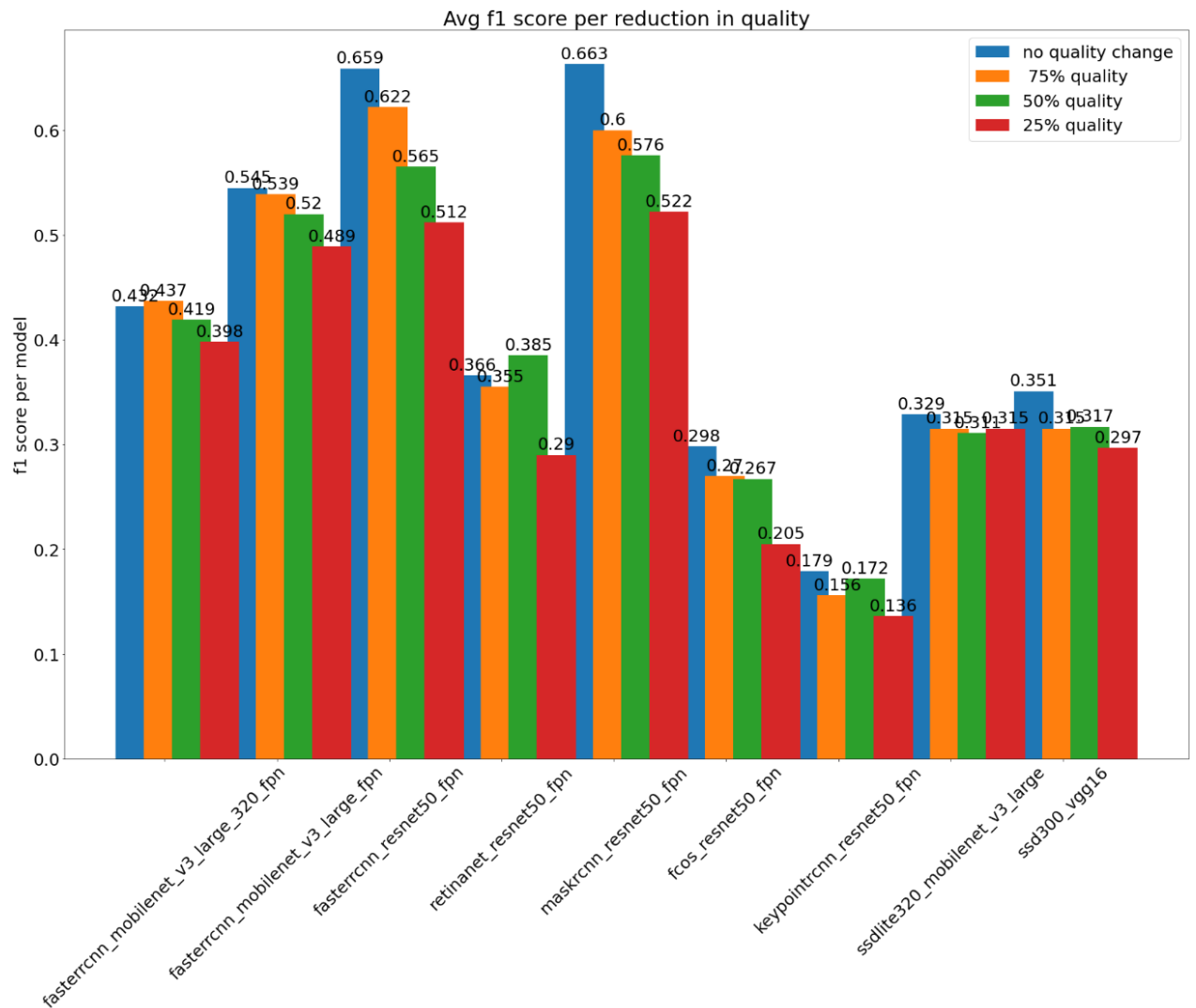


Figure 23 Chart of how quality change effects average f1\_score

As we see above, reducing the quality has small impact to the overall accuracies, but it is much more negatively effective to our best models. Let's see below if the worse accuracies are worth the data we save.

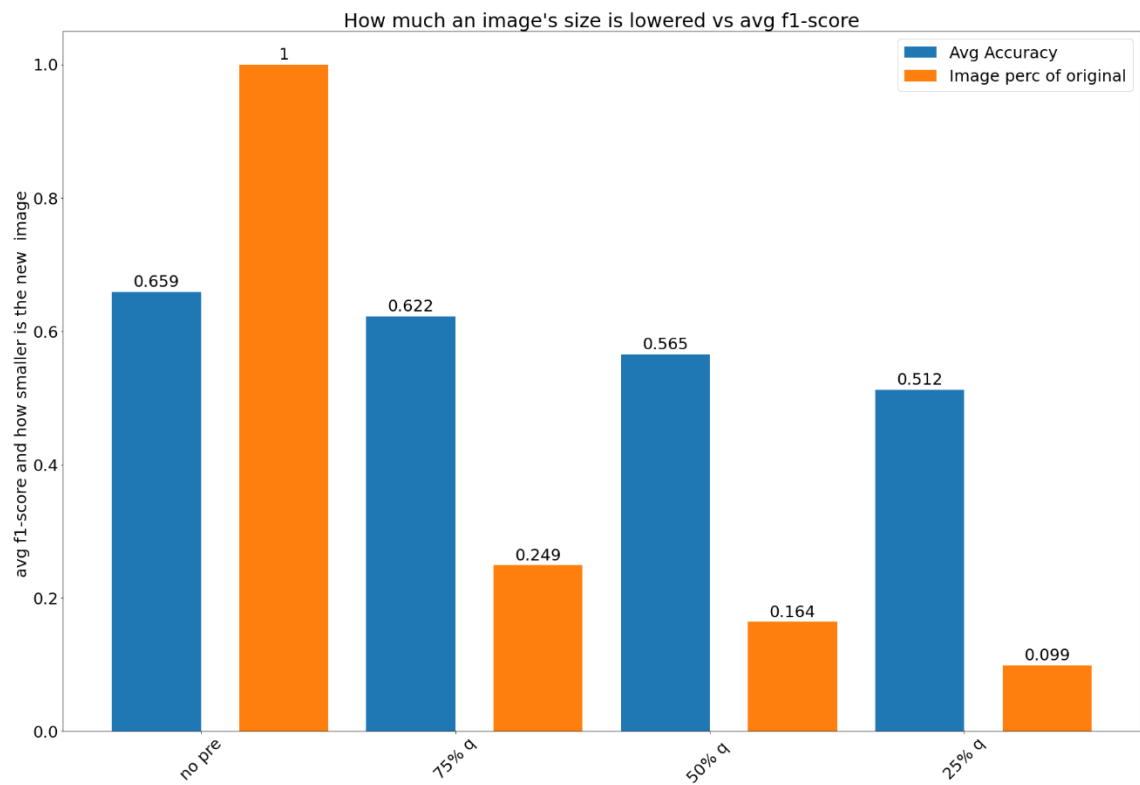


Figure 24 Chart of quality vs image size reduction

With blue we have the accuracies of the most accurate model (mask\_rcnn) and on the right, we have the how much smaller the new file is in relation to the original one.

For example if we have an f1 score for an image of 0.60 for a non-compressed image and it is 1000 bytes, then if we were to lower the quality down to 75% we would achieve a 75% reduction on image size while only dropping the f1\_score by 0.037 which we believe is worth the it.

Next aspect of the image we have compressed is the images dimensions.

For example, if an image had as original dimensions a width of 1000 pixels and a height of 800 pixels then if we were to resize it to 75%, we would have an image of 750 x 600 thus reducing the pixel by  $1000 \times 800 - 750 \times 600 = 350\,000$ . So, we actually reduced the total pixel count by 43.75%!

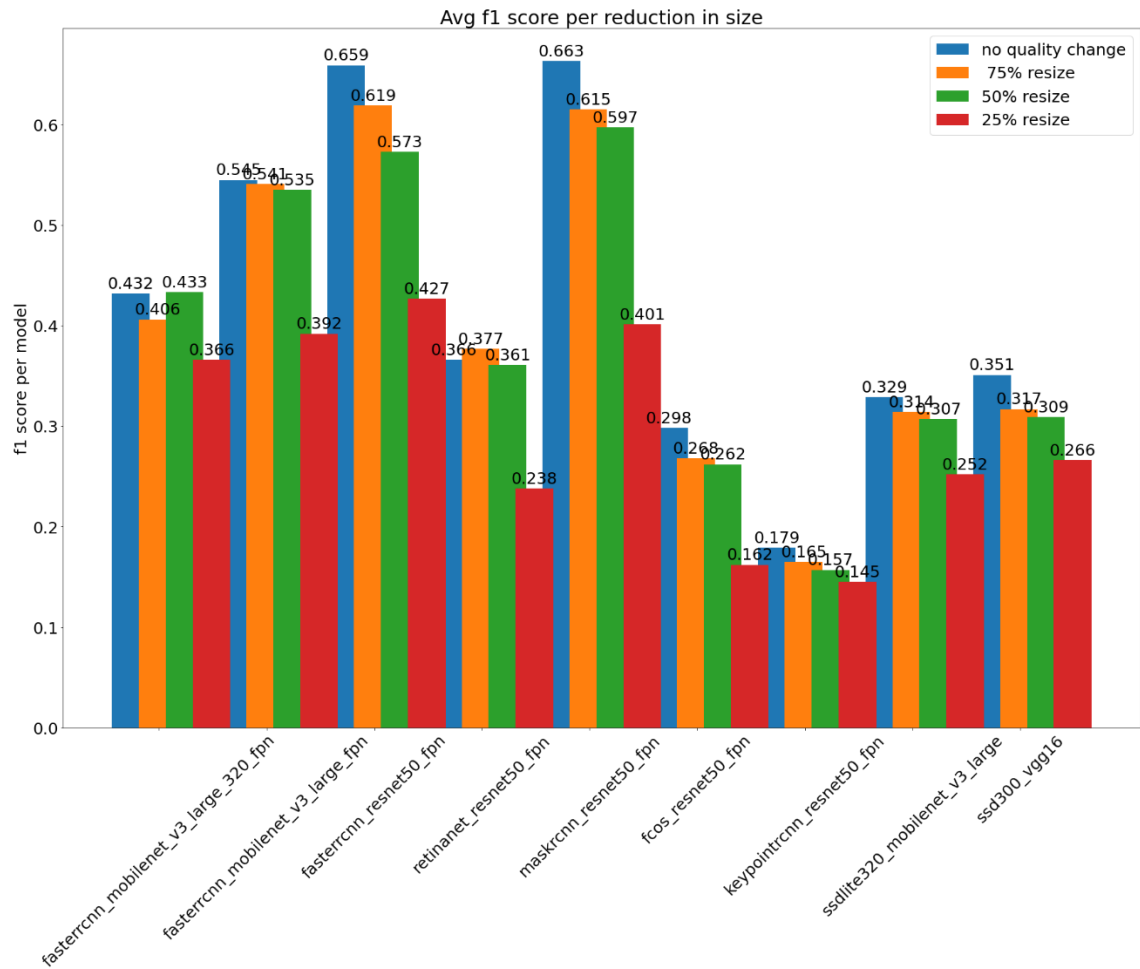


Figure 25 Chart of how resizing effects f1\_score

Above we see again that as expected the accuracies drop when we resize the image, but the results are as we saw them in the quality graph. What is different though is that at 25% resize the accuracies drops much more significantly than the quality had. We will see a direct comparison a few graphs below.

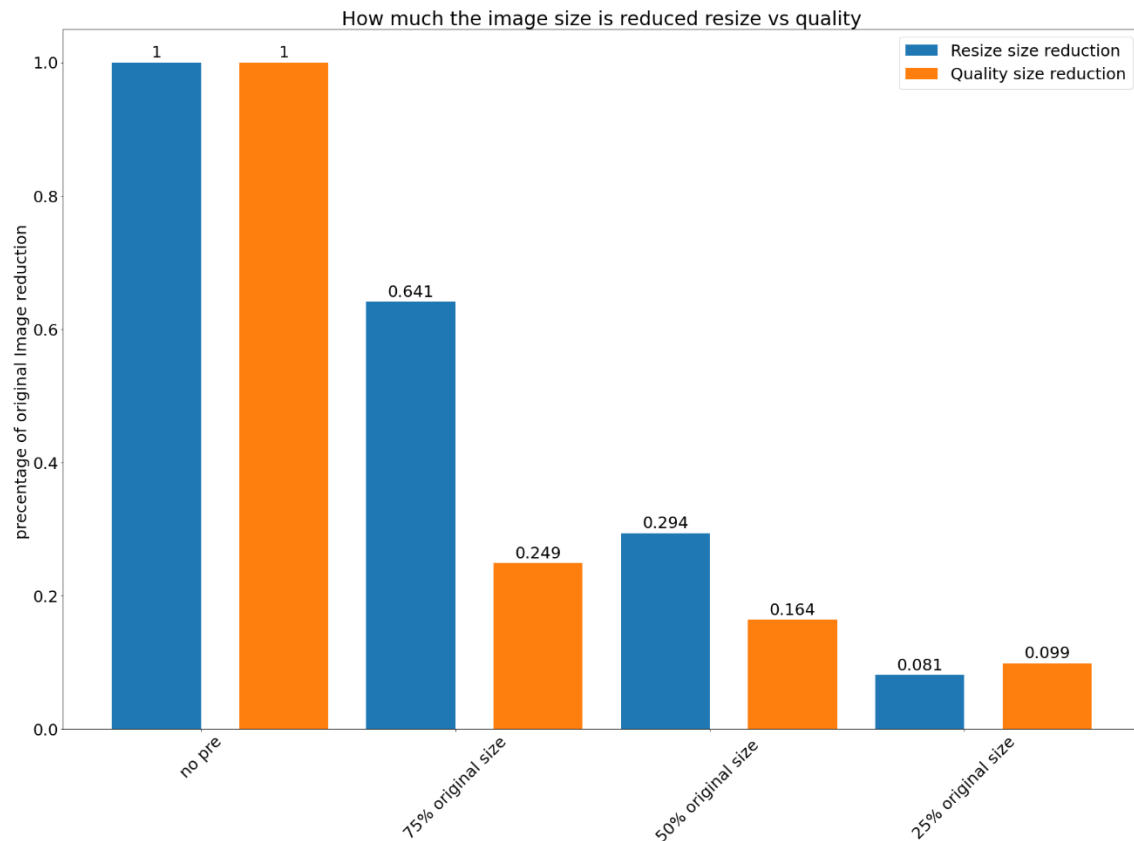


Figure 26 Chart of resizing vs image size reduction

Here we compare how much the image size was reduced to. As you can see changing the quality has more effect than changing the resize. But at 25% smaller image the picture is so small in dimensions that it prevails.

As we saw above, resizing intuitively should reduce the image size more efficiently than quality. This however is not true because the JPEG format we are using is a really amazing compression algorithm which can is really complicated as seen below:

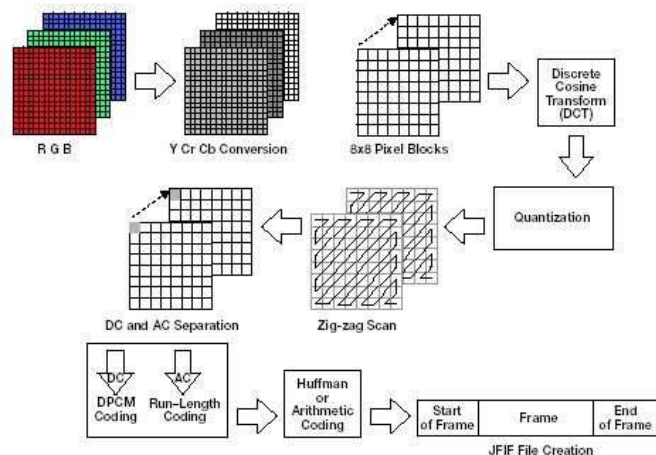


Figure 27 Illustration of the JPEG compression algorithm <https://www.eetimes.com/wp-content/uploads/media-1101219-fig1.jpg>

Point is that by reducing the quality of an image can make it more efficient than reducing the dimensional size. We cannot explain this but as the data shows, the image has less size on disk. But we can expect that at 25% the quality must yield better results to be this bigger.

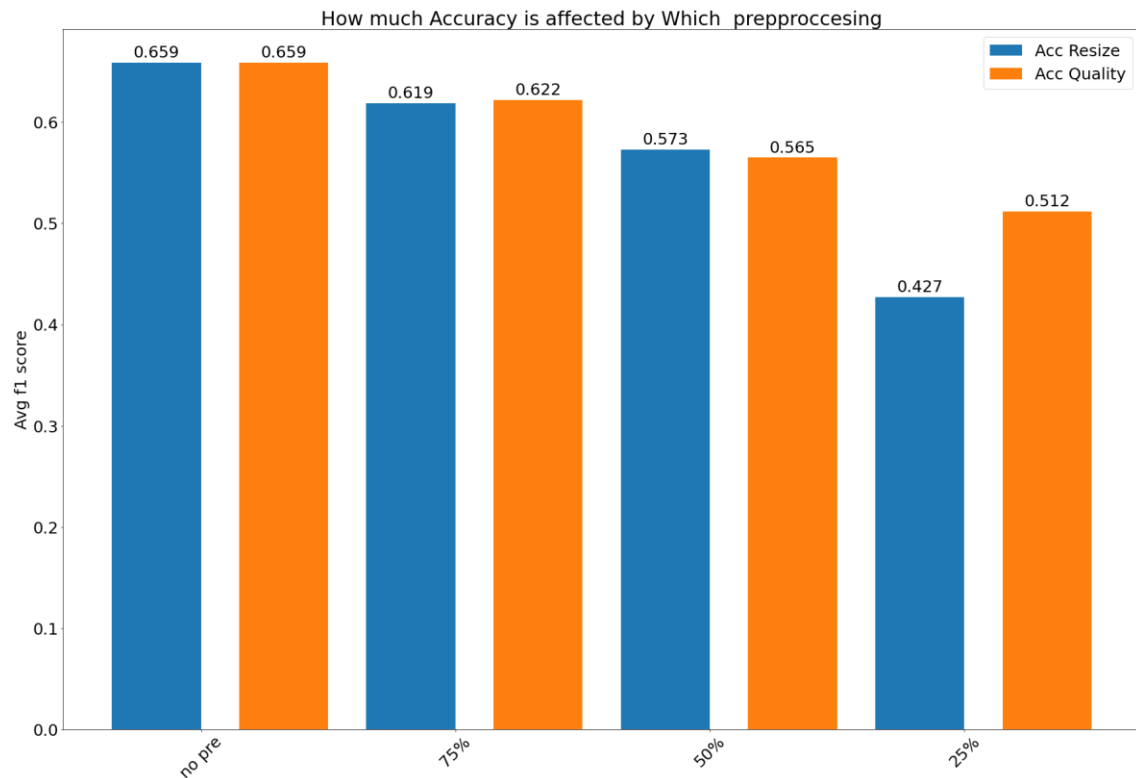


Figure 28 Chart comparing f1\_score per pre-processing option

Above we see how accuracies are affected by each processing. We see that both methods all have about the same effect but at 25% it is better to reduce quality instead of size (as we saw above its also bigger in size) so we can say we expected it.

### 6.1.2.2 Preprocessing explanations:

One clear problem someone can see is that 0.75% quality or resize reduction results in less than 75% of the original image size. This is normal since the PIL library uses optimization in saving jpeg images meaning that it can further compress than what the original images were.

```
image.save(new_path, quality=quality, optimize=True)
new_size = path.getsize(new_path) / (1024*1024)
# string = "New image size = " + str(new_size) + 'MBytes'
# logger.info(string)

percent_smaller = (new_size/prev_size) * 100
```

As for the results we have gotten, we see that nothing can beat not preprocessing, but in actuality we believe 75% dimensions reductions and a 75% quality instead of 100% does not affect the accuracy that much and is preferable when someone will want to test a out a deployment with limited bandwidth.

When it comes to object recognition, our system has helped us form the above opinion and helped justifying it. So if anyone wants to grab our project and run his own processing on any kinds of data it wants to send, for example compressing text files. Then our system will come and assist them in making decisions about it without needing to change a lot of stuff.

### 6.1.3 Object recognition:

#### 6.1.3.1 Object recognition statistics:

Here we will see all the items that appeared in all models handling them with NO PREPROCESSING. In total we saw 80 distinct types of objects in the images. We will display the top and bottom 3 and 1 for every 7 of the rest. All sorted by either the support or f1\_score. So we are only displaying the [0, 1, 2, 3, 10, 17, 24, 31, 38, 45, 52, 59, 66, 73, 77, 78, 79] numbered items as seen below in the red.

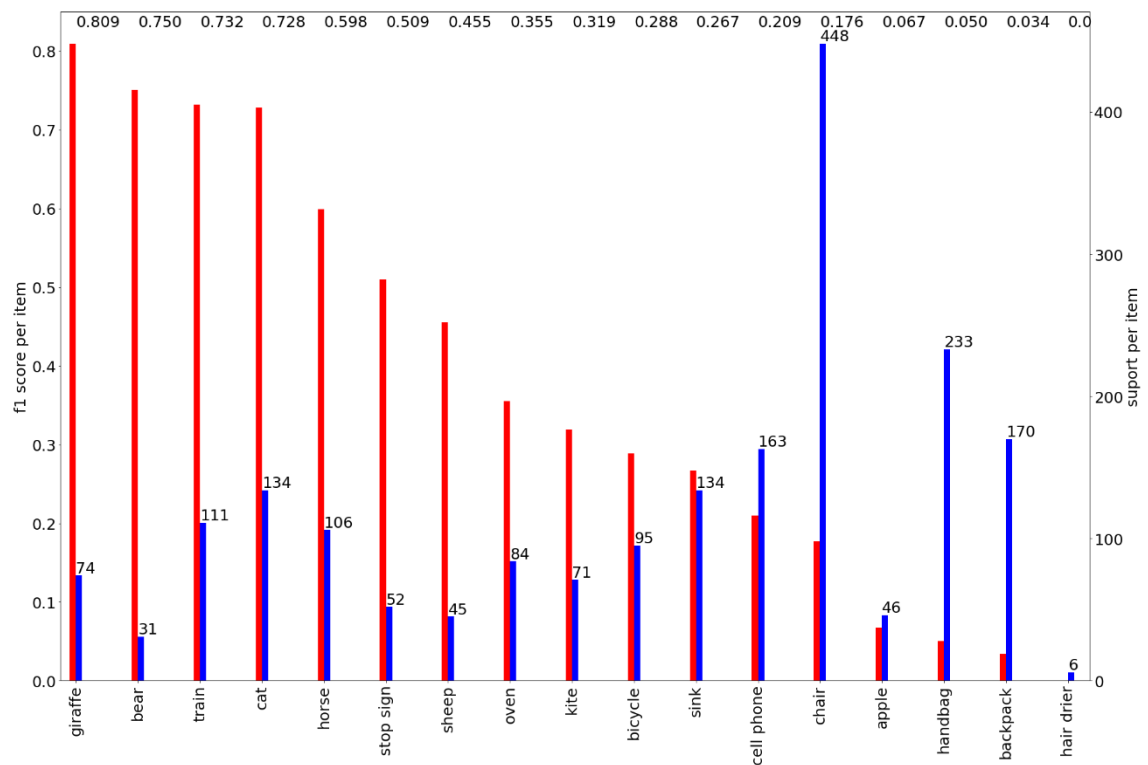


Figure 29 Chart of items f1\_score and support sorted by f1\_score

Here we see the items sorted by their f1\_score on our best ML, the “mask\_rcnn”. We had giraffes having the top average f1\_score of 0.8 but a total support of 74. So, they did not appear that many times. But chairs for example appeared 448 times but we failed miserably at detecting them. Lastly, even though handbags, and backpacks appeared 233 and 170 times we still had a score less than 0.05.

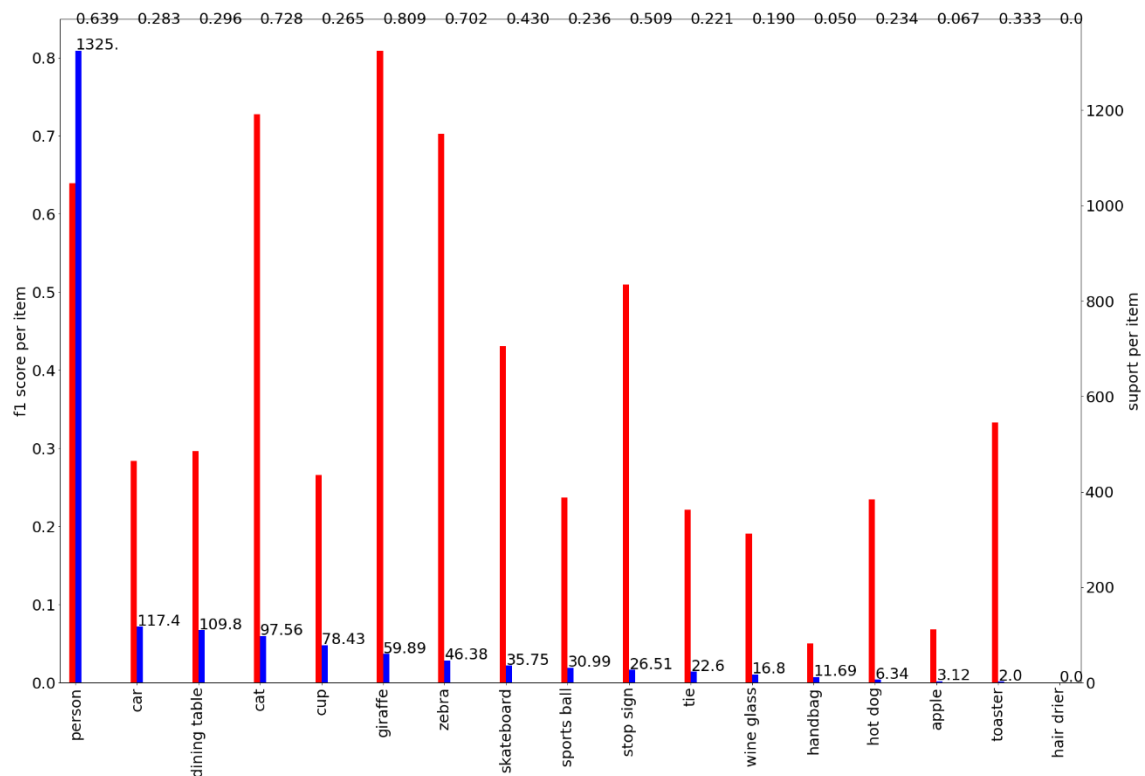


Figure 30 Chart of items f1\_score and support sorted by support

Here we see the items with the same form as we saw before, but now they are sorted by their support, i.e., how many times they appeared in total in the predictions. We see that there were many people in the images, specifically 1325 but their average f1\_score was an ok 0.63. So, we can say that the Dataset contained many pictures with people on them and the ML's were ok at spotting them, but the dataset may contain a lot of people so in general application we could have an equal distribution of objects.

### 6.1.3.2 Object recognition explanations:

We see above there is a clear confusion when it comes to the objects we detected. Well, again this just comes down to the ML models themselves, their hyperparameters and with what dataset and how much they have been trained.

We do know that Models we used in this problem were trained on the same dataset so we can safely say they were operating normally on data they could have been trained on.

Point is, in this object recognition problem we have established that we have the ability to recognize objects well enough and have found the approach to do so. Someone could come up and change the project to recognition of other stuff like fraud detections. Then after changing the relative code it could use the same structure seen in the colab to extract results easily saving a lot of hassle of rewriting stuff.

## 6.2 Results comments and limitations:

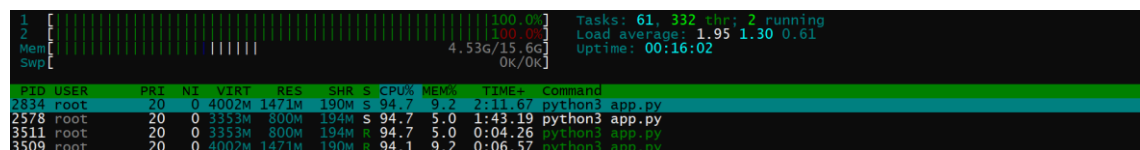
### 6.2.1 Comments:

For the sake of this project, the results were sufficient for our analysis and we got the project ran without any unexpected problems. Our focus was the ability of the project to gather data for a wide range of purposes to fulfill the users' needs. It can easily be expanded to gather more or different data depending on what is the ML inference application a user wants to put it through.

## 6.2.2 limitations:

There are a lot of limitations when it comes to the results, the most important one being time. We have made the project to run a total of 3 days in order to gather results. In a better scenario, we want to run it for 20 to gather much more data with better variety. But since we rushed a bit in order to prepare for this thesis and colab notebook we are satisfied with the current ones.

Another limitation was the hardware. We needed to rent an Azure VM with 2 cores and 16 GB of RAM in order to run this project. If it were for example 4 cores, we could have 4 edges running sparing execution time. Since Docker is multithreaded, we can see that both edges utilized all the cores of the system:



PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
2834	root	20	0	4002M	1471M	190M	S	94.7	9.2	2:11.67	python3 app.py
2578	root	20	0	3353M	800M	194M	S	94.7	5.0	1:43.19	python3 app.py
3511	root	20	0	3353M	800M	194M	R	94.7	5.0	0:04.26	python3 app.py
3509	root	20	0	4002M	1471M	190M	R	94.1	9.2	0:06.57	python3 app.py

Lastly there are limitations to this approach since there are many points of failure. A lot of systems need to be orchestrated to work with each other so it took a lot of time in building and debugging it. Which could have been spent in building a better problem to get better results about a different subject.

## Chapter 7: Conclusions

---

To sum up, we are pretty happy for the system we have developed, the results we got and in general we believe it serves its purpose well. This system was really hard to develop and test but, in the end, we pulled through and delivered a complete system despite the limitations.

When it comes to the results, we are really happy with the results we got. They are accurate and most of, informative for the specific problem we are addressing. When it comes to object recognition, maskrcnn from pytorch is the best of its kind and a reduction of quality and dimensions of 75% of the original delivers a great accuracy while also reducing its size considerably.

Our system is complete and has a lot of future ahead. As we mentioned above it can serve as a great building block for future systems. They can simply form our publicly available source code and build their own implementations of cloud to edge implementations with their own problems and solutions in mind.

# Bibliography

(n.d.). Retrieved April 11, 2022, from Matplotlib — Visualization with Python:

<https://matplotlib.org/>

*Edge computing*. (n.d.). Retrieved April 11, 2022, from Wikipedia:

[https://en.wikipedia.org/wiki/Edge\\_computing](https://en.wikipedia.org/wiki/Edge_computing)

*FiftyOne — FiftyOne 0.15.0 documentation*. (n.d.). Retrieved April 11, 2022, from Voxel51:

<https://voxel51.com/docs/fiftyone/>

Hertzfeld, A. (n.d.). *Cloud computing*. Retrieved April 11, 2022, from Wikipedia:

[https://en.wikipedia.org/wiki/Cloud\\_computing](https://en.wikipedia.org/wiki/Cloud_computing)

Lewis, S. (n.d.). *Key Features of PyTorch*. Retrieved April 11, 2022, from TechTarget:

<https://www.techtarget.com/searchenterpriseai/definition/PyTorch>

*Machine learning*. (n.d.). Retrieved April 11, 2022, from Wikipedia:

[https://en.wikipedia.org/wiki/Machine\\_learning](https://en.wikipedia.org/wiki/Machine_learning)

*Use Docker Compose*. (n.d.). Retrieved April 11, 2022, from Docker Documentation:

[https://docs.docker.com/get-started/08\\_using\\_compose/](https://docs.docker.com/get-started/08_using_compose/)

*What is Docker? - India*. (2021, June 23). Retrieved April 11, 2022, from IBM:

<https://www.ibm.com/in-en/cloud/learn/docker>

*What is Flask Python*. (n.d.). Retrieved April 11, 2022, from Python Tutorial:

<https://pythonbasics.org/what-is-flask-python/>

*What Is Pandas in Python? Everything You Need to Know*. (2021, November 25). Retrieved

April 11, 2022, from ActiveState: <https://www.activestate.com/resources/quick-reads/what-is-pandas-in-python-everything-you-need-to-know/>