Master Thesis

September 14, 2022

enseMbLer

Designing a Scalable Architecture for Ensemble Machine Learning & Collaboration

Shubhankar Joshi

of New Delhi, India (19-734-342)

supervised by

Prof. Dr. Harald C. Gall

Dr. Pasquale Salza & Marco Palma

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master Thesis

enseMbLer

Designing a Scalable Architecture for Ensemble Machine Learning & Collaboration

Shubhankar Joshi

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master Thesis

Author: Shubhankar Joshi, shubhankar.joshi@uzh.ch
URL: https://www.linkedin.com/in/jos/
Project period: 14.03.2022 - 14.09.2022

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

First and foremost, I would like to thank Prof. Dr. Harald C. Gall and Dr. Pasquale Salza at Software Evolution & Architecture Lab of the University of Zurich, for taking me under their supervision and allowing me to write my thesis. Furthermore, I would like to express my sincere gratitude towards my supervisors, Dr. Pasquale Salza and Marco Palma, for their invaluable insights, patience and constant guidance throughout the process. This work would not have been possible without their support.

I would also like to thank my colleagues at intelliCard Solutions AG, for giving me time off from work to focus on my thesis.

Finally, I thank my family and friends for their constant motivation and wishes throughout the past six months.

Abstract

The past few decades have seen a huge rise in the amount of data being generated online and a significant boom in the adoption of big data analytics and machine learning techniques with the aim of solving complex problems and driving innovation further. However, this is easier said than done as big data presents significant challenges. This thesis aims to explore this intersection of fields of big data and machine learning, examining the challenges and reviewing current state-of-the-art techniques. We further design and develop our own cloud ready scalable architecture, *enseMbLer*, building on the principles of data parallelism and massively parallel ensemble learning to tackle the big data problem.

We demonstrate the capability of our solution by preforming an empirical analysis. To this end, we augment a popular public dataset using WGAN-GP, a generative adversarial network. We then develop and train standard models, and run a bunch of different experiments over Google Cloud Platform, comparing our results to those of others. We successfully demonstrate the effectiveness of REST based HTTP infrastructure to handle distributed machine learning without significant overheads and further provide evidence of the increased performance gains of ensemble based techniques. Finally, we contribute a solution to tackle real-time stream processing and machine learning by suggesting a lambda architecture using our solution.

Zusammenfassung

In den letzten Jahrzehnten hat die Menge der online generierten Daten enorm zugenommen und die Einführung von Big-Data-Analysen und maschinellen Lerntechniken mit dem Ziel, komplexe Probleme zu lösen und die Innovation voranzutreiben, hat einen erheblichen Aufschwung erlebt. Dies ist jedoch leichter gesagt als getan, da Big Data erhebliche Herausforderungen mit sich bringt. Ziel dieser Arbeit ist es, diesen Schnittpunkt von Big Data und maschinellem Lernen zu erforschen, die Herausforderungen zu untersuchen und den aktuellen Stand der Technik zu überprüfen. Darüber hinaus entwerfen und entwickeln wir unsere eigene Cloud-fähige, skalierbare Architektur, *enseMbLer*, die auf den Prinzipien der Datenparallelität und des massiv-parallelen Ensemble-Lernens aufbaut, um das Big-Data-Problem zu lösen.

Wir demonstrieren die Leistungsfähigkeit unserer Lösung, indem wir eine empirische Analyse durchführen. Zu diesem Zweck erweitern wir einen beliebten öffentlichen Datensatz mit WGAN-GP, einem generativen kontradiktorischen Netzwerk. Anschließend entwickeln und trainieren wir Standardmodelle und führen eine Reihe verschiedener Experimente über die Google Cloud Platform durch, wobei wir unsere Ergebnisse mit denen anderer vergleichen. Wir demonstrieren erfolgreich die Effektivität einer REST-basierten HTTP-Infrastruktur, um verteiltes maschinelles Lernen ohne signifikanten Overhead zu handhaben, und liefern weitere Beweise für die Leistungssteigerung von Ensemble-basierten Techniken. Abschließend stellen wir eine Lösung für die Echtzeit-Stream-Verarbeitung und das maschinelle Lernen vor, indem wir eine Lambda-Architektur vorschlagen, die unsere Lösung nutzt.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Recent years have seen an alarming rise in the amount of data being generated over the internet due to the growing popularity of web & mobile technologies, social media and advancements in the fields of IoT and cloud computing. It is quite evident that we are in the *age of data*, as more industries across numerous domains have started investing in big data analytics. Big Data, a term coined since the early 1990s, refers to data that is too large or complex to be processed through traditional computing techniques. It provides huge potential in terms of business value by providing the ability to make better decisions through knowledge discovery. It is estimated that by the year 2027, the big data market will grow to 103 billion US dollars, more than doubling its market size in the year 2018 [1].

The past decade has also seen a rapid adoption of machine learning techniques across multiple data rich industries such as pharmaceutical, astronomy, aviation & finance. Machine learning techniques aim to analyse data and provide meaningful insights, and further learn from data to provide optimizations and predictions. And now, the intersection of the fields of machine learning and big data seems to be very promising and has been drawing attention of the research community and industry because of the potential gains it can bring to solving complex problems, improve quality of life and drive innovation further. However, learning from big data using traditional machine learning techniques is not so straightforward.

Managing and learning from big data is a growing challenge that many researchers and companies face. These challenges can be best summarized by the 3 V's models, referring to the volume, variety and velocity of big data [7]. The tremendous scale and volume of large datasets make traditional algorithms very inefficient, as they were deisgned for smaller datasets and do not scale well. For example, SVMs or Support Vector Machines, have a training time complexity of O($dataSize^3$) [31]. Furthermore, large datasets may be distributed across multiple nodes, and the entire dataset may not fit into memory, breaking some traditional algorithms. Similarly, velocity of data in terms of real-time stream processing also proves to be challenging [11]. This has caused research to focus on newer innovative techniques to handling big data for machine learning.

This thesis aims to contribute towards research in this intersection of fields of machine learning and big data. We first review the current state-of-the-art techniques to leverage machine learning for large datasets, from popular frameworks like Apache Hadoop's MapReduce & Apache Spark, to approaches utilizing massive data parallelism through ensemble learning on the cloud. We then propose a cloud ready, scalable architecture design to address the limitations of these approaches and later perform empirical analysis to demonstrate its effectiveness. To this end, we use a popular public dataset and augment it using state-of-the-art generative adversarial network and train multiple machine learning models developed by us to explore the following research questions:

1. *RQ1. How do traditional machine learning models scale with big data?*
2. *RQ2. How do ensembles created through factorization techniques compare against their standard models?*
3. *RQ3. How much overhead does a HTTP based RESTful infrastructure add to distributed machine learning?*
4. *RQ4. How does degree of parallelization affect the performance of massively data parallel ensembles?*

The main contributions of this thesis are:

1. We designed and developed a scalable architecture, *enseMbLer*, which leverages data parallelism & cloud technologies to handle big datasets through ensemble learning and supports machine learning collaboration.
2. We demonstrated the effectiveness of our solution and performed empirical analysis using augmented datasets and various machine learning algorithms to derive more insights into the proposed research questions.
3. We demonstrated that HTTP based REST infrastructures do not add significant communication overhead & further proposed how our solution could be used in a lambda architecture to handle real-time stream data processing and machine learning.
4. We verified and contributed further evidence to support that ensembles can achieve higher accuracy with more efficient performances.

The entire source code of *enseMbLer* is available on GitHub [1] and all docker images developed are available on DockerHub [2].

The remainder of this thesis is organized as follows: Chapter 2 provides a brief introduction to the theoretical background knowledge required for better a understanding of this thesis. Chapter 3 provides a review of the start-of-the-art techniques currently in use to leverage machine learning for big data analytics. Chapter 4 presents the problem statement and describes in detail the main motivation, goals and research potential of this thesis. Chapters 5 and 6 form the main body of the thesis and discuss the approach that was taken to selecting and preparing the models & dataset, as well as the implementation details and architecture of *enseMbLer*. Chapter 7 provides a detailed summary of all the results and our evaluations. And finally, Chapter 8 provides the summary, conclusion, and an outlook on future work.

[1] `https://github.com/shobuxtreme/ensembler`
[2] `https://hub.docker.com/u/shobuxtreme`

Chapter 2

Theoretical Background

This thesis assumes that the reader is familiar with baisc principles and concepts of machine learning. To provide a better understanding of the work described in the later Chapters of this thesis, the theoretical framework on which this thesis builds, i.e. the concepts of parallel computing and ensemble learning, is explained in this Chapter.

2.1 Data Parallelism vs Task Parallelism

Parallel computing involves breaking down large complex problems into smaller parts and processing these smaller independent parts simultaneously. In the context of machine learning, algorithms can attain significant gains if operations can be performed concurrently. This concurrency can be realized through two major ways, *data parallelism* & *task parallelism* [6].

Data Parallelism. Data parallelism refers to performing the same computation on multiple *different* inputs simultaneously. Many machine learning algorithms process data in batches, and this approach is a natural fit for such algorithms. Such algorithms are termed *embarrassingly parallel* as there is no intercommunication in between the batch operations. Master-Slave models work on this principle, where a master process distributes work across slave processes, which perform the *same* operations. Later in this thesis in Chapter 3.3, we review state-of-the-art techniques which build upon this principle to process huge datasets through a divide-and-conquer approach.

Task Parallelism. Task parallelism on the other hand refers to breaking up an overall algorithm into different parts which can be executed simultaneously. This requires more in-depth knowledge of the algorithms and frameworks, but significant efficiency gains can be achieved [6]. An example is the use of GPUs. We review an approach in Chapter 3.2 that uses DAG, directed acyclic graph, based computation framework to achieve task parallelism.

2.2 Ensemble Learning

Ensemble systems are multi-classifier systems which were originally developed to improve decision making accuracy by reducing variance [24]. They are used in a wide variety of setting to help solve problems related to feature selection, class-imbalanced data and confidence estimation among others. The work by weighing and combining the output of multiple learners and there exist many ways in which these ensemble members can be combined, including averaging the

outputs, selecting the majority through weighted voting and many more. This process of combination is termed *classifier fusion*. Bagging, Random Forests, Boosting, AdaBoost are all state-of-the-art classifier fusion techniques, and this thesis later describes implementation of many machine learning and ensemble learning models using these techniques.

We make use of XGBoost stacking algorithm to create our ensembles by fusing multiple kinds of learners. Stacked generalization algorithms are trainable i.e. they *learn* to combine and fuse the models in the best way possible to achieve high performance. The architecture of such models usually has multiple levels. Level 0 or l0 models are called base models which fit on training data. Level 1 or l1 models are called meta models, which learn to fuse the base models in the best way possible. We support this configurablity of ensemble architectures by designing level into our framework's job configurations, further described in Chapter 6.1.3.

Chapter 3

Review of the State of the Art

Prior work had been done in an attempt to use various frameworks and architecture patterns to handle large datasets. In this chapter we provide an overview of the current research and major ideas on ways of using machine learning with big data.

Section 3.1 discusses approaches using popular Apache frameworks which use distributed computing across a cluster of nodes. Section 3.2 mentions an approach utilizing a framework which relies on using graph-based data models to achieve task parallelism in a single node setting. Finally, Section 3.3 highlights approaches using data parallel factorization techniques and cloud technologies to scale machine learning across a cluster of nodes.

3.1 MapReduce and Spark

Apache MapReduce [1] is the processing engine of the Apache Hadoop framework, and allows for scalability and computation of vast volumes of data. It consists of *Map* and *Reduce* operations which aggregate and combine output in a distributed setting, and works well when algorithms are embarrassingly parallel. The *Map* operation distributes sections of the dataset to mappers which work in parallel and perform computational tasks. The *Reduce* operation then aggregates and combines the output from all the mappers and provides the overall result.

Apache Spark [2] is a similar big data processing framework which extends the capabilities of MapReduce, and is emerging to be the next generation replacement of Hadoop. It provides real time stream processing features in addition to the batch processing feature supported by Hadoop. Spark makes use of RDDs i.e. Resilient Distributed Datasets, which can be stored in memory in between queries.

Numerous studies highlight the ability of both these frameworks for handling distributed machine learning workloads [18] [12] [4] [20]. Gillick *et al.* concluded that MapReduce proved to be a great choice for parallelizing simple machine learning applications [12]. They benchmarked the performance of searching and sorting algorithms across clusters of 9 and 80 machines and offered improvements to make MapReduce work with more machine learning problems. Similarly, Liu *et al.* experimented with Back Propagation Neural Networks (BPNNs) developed using MapReduce framework to solve classification problems [18]. They were able to demonstrate the effectiveness of MapReduce in parallelizing classification tasks. Nair *et al.* developed a real-time health status prediction system using Decision Trees algorithms on streaming Big Data with the Apache Spark framework [20]. Similarly, Assefi *et al.* demonstrated the ability of Apache Spark by analysing big

[1] `https://hadoop.apache.org`
[2] `https://spark.apache.org`

datasets through SVM (Support Vector Machines), Decision Trees, Random Forests and K-Means clustering algorithms [4].

Gopalni & Arora compared the two frameworks, highlighting their differences and providing reasons to choosing one over the other [14]. They performed comparative analysis using the K-Means clustering algorithm on a sensor dataset 1240 MB in size and concluded that MapReduce is not efficient for multi-pass applications and that Spark will become the new de facto framework for big data processing.

3.2 GraphLab

Low *et al.* at Carnegie Mellon University developed the GraphLab framework to achieve excellent parallel machine learning performance with large-scale real world problems [19]. In their study, they highlight the shift in computer architecture industry from frequency scaling to parallel scaling and mention the drawbacks of approaches using existing frameworks such as MapReduce abstractions, DAG abstractions and Systolic abstractions. They mention that such frameworks fail when there are computational dependencies in the dataset, which makes it difficult to operate them on structured data.

They then propose a framework in which the data model consists of a directed *data graph*. The graph denotes all the data and computational dependencies and an *update* function performs all the computations on the graph vertices and their neighbours. A *sync* mechanism then aggregates data across all the graph vertices. The GraphLab framework makes use of both the synchronous and round-robin scheduling algorithms, implemented using FIFO and Prioritized queues for task scheduling. They were able to support structured data dependencies and iterative computations and demonstrated the effectiveness of GraphLab and its parallel performance gains on real-world problems using MRF Parameter Learning, Gibbs Sampling, CO-EM Named Entity Recognition and Shooting Algorithms, in a single node multi-core setting.

3.3 FlexGP, FCUBE, AMQPGA and cCube

FlexGP. Proposed by Veeramachaneni *et al.* at Massachusetts Institute of Technology, FlexGP was one of the first systems which performed symbolic regression on a large-scale dataset on the cloud, using massively parallel ensemble learning with genetic programming algorithms [32]. It uses a parallelization framework which decomposes a dataset into multiple smaller subsets and trains a large quantity of independent learning models. Each model independently makes a prediction and the results are fused together in the form of an ensemble. Thus the framework works on the principal of data parallelism. They demonstrated the effectiveness of their solution with the Million Song Dataset year prediction challenge. The goal of this challenge is to predict the release year of 515,000 songs and FlexGP outperformed a variety of other state-of-the-art regression learners and obtained more accurate solutions in a shorter time frame.

FCUBE. Arnaldo *et al.* at Massachusetts Institute of Technology further extended FlexGP's concepts and proposed a cloud-based framework to let researchers contribute their own learners to efficiently tackle large data problems [3]. With their "Bring Your Own Learner" model, researchers could contribute their custom algorithms in a plug-n-play style. Similar to FlexGP, FCUBE also exploited the principle of data parallelism to factor large datasets into random sub samples and used them to train multiple models simultaneously, fusing their results in the end. They further demonstrated the capabilities of FCUBE by integrating five different learners on the Higgs dataset with 11 million exemplars and obtained competitive performance and fast learning times.

AMQPGA & cCube. Building upon the concepts of work mentioned above, Salza *et al.* proposed two similar cloud based solutions using software containers, AMQPGA and cCube [25] [26]. AMQPGA uses a master-slave model to speed up genetic algorithms [25]. The master node manages the communication with slave worker nodes which compute the fitness evaluations. AMQP communication protocol is used and messages are dispatched in a round-robin fashion. Once the workers evaluate the fitness functions, they publish the results in a response queue. They used open source cloud orchestration techniques to allocate resources and deploy the workload and succeeded in accelerating the execution times of their algorithm, thereby demonstrating the effectiveness of cloud based solutions in scaling genetic algorithms.

Furthermore, cCube builds upon FCUBE's "Bring Your Own Learner" model to let users collaborate on the same machine learning problems [26]. It uses a microservices based architecture to scale software containers on open cloud and manages communication using AMQP messaging protocol to divide tasks among multiple learners. cCube uses DockerSwarm to provide a flexible infrastructure thereby avoiding any "lock-in" with specific cloud service providers. The authors demonstrated the workings of cCube by running GPFunction EML (Evolutionary Machine Learning) algorithm on a split of the Higgs dataset, with nodes deployed on a hybrid cloud comprising OpenStack private cloud and AWS.

Chapter 4

Problem Statement

In the past few decades alone, enormous amounts of data has been generated and this trend is only expected to grow exponentially. Big data presents huge potentials in terms of knowledge discovery and better decision making ability across multiple domains and industries, and as a result more and more machine learning techniques are being adopted. Although machine learning has been growing over the years to become a more mature field, there still exists uncertainty in the relationship between training datasets and model performance.

This means that there may be situations when model performance improves as the size of the dataset increases, but at the same time, there may also be other situations where a model does not necessarily scale well enough to handle datasets that are large, resulting in a point of diminishing returns. Model performance may even degrade beyond a certain dataset size as traditional machine learning algorithms were not designed to work with huge volumes of data. In Chapter 3 we looked at the current state-of-the-art techniques which attempt to provide solutions and more insights into this problem. And although they seem promising and are able to demonstrate their effectiveness, each has certain drawbacks and limitations to their use.

4.1 Limitations

1. The use of traditional machine learning algorithms with big data brings tremendous challenges as they work by loading the data completely into memory [10]. This approach fails in the context of large datasets. Much research is still required to address the challenges posted by big data, best expressed by the 3 V's model. Volume, Variety and Velocity, respectively refer to the large scale of data, different variety of data and the speed of streaming data [7].

2. While MapReduce and Hadoop framework can be used in machine learning workloads with large datasets, they often lead to high overheads and communication bottlenecks. Liu *et al.* discovered high computation overhead when using BPNNs over a cluster of machines, which happened as a result of the mappers and reducers continually starting and stopping [18].

3. Another drawback of Apache frameworks is that they are limited to certain programming languages. Salza *et al.* highlighted the fact that Hadoop development is only limited to the Java programming ecosystem and requires the developers to have specific skills and knowledge of the framework [25]. Similarly, the Spark framework is currently limited to Python, Scala, Java and R programming languages and limits the users who wish to use other languages and frameworks, such as JavaScript.

4. Massively data parallel ensemble learning techniques prove to be very promising, however more research needs to be done for evaluating ways to aggregate the results and fuse the contributions from individual learners. Parallel execution of different models requires communication between multiple components. Even though cCube serves as a proof-of-concept, it is limited to fusing results from different models if and only if they are available in the same containers [26].

5. Velocity or the speed of data in the context of big data proves to be another challenge for machine learning. Real world time-sensitive scenarios such as stock market prediction or natural calamities prediction, require the models to work with stream data and have quicker prediction times. Additionally, as the data distribution changes with time, models need to learn data as stream and be constantly updated [10].

4.2 Goals

The goal of this thesis is to build upon the state-of-the-art cloud based techniques and propose a cloud-ready solution to address the limitations mentioned above. We define the following as goals that need to be met by the proposed architecture design:

- **G1 Scalability.** To allow distributed computing and enable parallelization of machine learning.
- **G2 Portability.** To support cross platform execution from anywhere without requiring access to any specialized hardware.
- **G3 Plug-n-Play.** To support on-the-go execution of models without requiring any recompilation of source code and support "Bring Your Own Learner" model.
- **G4 Polyglot Development.** To support deployment of machine learning models developed with any programming language and framework.
- **G5 Robustness.** To support high fault tolerance.
- **G6 Black Box Execution.** To support functionality without requiring special knowledge about the workings of the model algorithms.
- **G7 Open Source.** To use open source technologies and avoid strict dependence on specific software or cloud vendors.
- **G8 Graphical User Interface.** To aid human supervision and support all operations via a modern graphical user interface, providing feedback and control to the users.

4.3 Research Questions

Through this thesis, we also aim to gain more insights into using massively parallel ensemble learning to handle big data, and compare it to techniques using traditional machine learning. We use this as an opportunity to demonstrate the validity of our proposed solution and contribute to research towards collaborative machine learning techniques. We explore the following research questions:

- **RQ1. How do traditional machine learning models scale with big data?** As highlighted in Section 4.1, traditional machine learning models should perform poorly and even fail when scaled to big data. Through this research question, we want to verify this limitation by computing the performance of such models against varying dataset sizes and further aim to discover if there exists a point of diminishing returns for the respective models. We first augment and expand a standard dataset using GANs (Generative Adversarial Networks) and use our proposed solution to train models serially without using distributed computing. This also serves as a benchmark against which we compare further results.
- **RQ2. How do ensembles created through data parallel factorization techniques compare against their standard models?** For the purposes of this thesis, we use the PIMA Indians Diabetes Dataset and provide more details and reasoning for our choice further on in Chapter 5.1.1. This is a heavily studied dataset and there exist many studies which use different kinds of machine learning algorithms, including ensemble algorithms. Through this research question we want to compare how ensembles created through factorization techniques as mentioned in Section 3 compare against their standard models and other ensemble algorithms. To this end, we first train our models using our proposed solution without any parallelization and compare our results to those of others. We then use or solution to form ensembles with various configurations and compare the results to derive more insights.
- **RQ3. How much overhead does a HTTP based RESTful infrastructure add to distributed machine learning?** Some form of communication is required in a distributed cloud setup involving multiple nodes & microservices to transfer and share data among components. In a REST architecture style with communication via HTTP, the payload size of the HTTP messages will increase as the overall dataset size increases. Through this research question, we aim to investigate and study the relationship between communication overhead and dataset size to determine if this leads to performance bottlenecking. To this end, we schedule multiple machine learning training jobs on our proposed solution and vary the dataset size and sampling size parameter. As we use our own developed algorithms, we can precisely calculate the execution runtimes of the machine learning algorithm and the worker microservice, and calculate the difference to pinpoint the overhead time due to communication over the network.
- **RQ4. How does degree of parallelization affect the performance of massively data parallel ensembles?** State-of-the-art techniques and their respective studies mentioned in Section 3 provide evidence that ensembles created through data-parallel distributed computing using cloud technologies can handle big data. Advancements in cloud technologies makes it easy to horizontally scale resources and increase compute performance. Through this research question we aim to find how such horizontal scaling and increase in the degree of parallelization affect the performance of such techniques. We want to investigate if there is a linear relationship between the two or if there exists a point of diminishing returns. We address this research question by varying both the dataset sizes and number of model replicas, and compare our results against results of **RQ1**.

Chapter 5

Approach

Before proposing an architecture design, it was important verify the limitations of standard machine learning algorithms with big data. Most of the prior studies and similar work done either derived insights from smaller datasets, or we could not find multiple use cases with the same large dataset for drawing meaningful comparisons. Moreover, access to the data and infrastructure used for these studies was limited. As we wanted to compare and benchmark the performance of some of the standard & most widely used machine learning algorithms with varying dataset sizes, we determined to implement our own models from scratch with a standard dataset. By implementing similar machine learning models in different programming languages, we got the opportunity to demonstrate the polyglot nature of our solution.

The following chapter describes in detail the approach that was taken to select, prepare and augment a dataset, implement and train the machine learning models, and benchmark their performance.

5.1 Dataset

5.1.1 Selection of dataset

The UCI Machine Learning Repository [1] is one of the largest and most widely referenced collection of datasets used by researches for empirical analysis of machine learning algorithms. Tanwani *et al.* performed experiments on more than 30 biomedical datasets available by using and evaluating multiple machine learning algorithms [30]. The PIMA Indians Diabetes Dataset seemed to be the most promising choice for selection [21], as several similar extensive studies had been performed on the dataset by using Random Forest and Decision Trees algorithms [9], Neural Networks & Support Vector Machines [27], and Genetic Algorithms [16]. Furthermore Akyol *et al.* evaluated the performances of AdaBoost, Gradient Boosted Trees and Random Forest ensemble learning algorithms, achieving a classification accuracy of 73.88%, 73.16% and 73.45% respectively, thus setting a benchmark to compare our results against [2]. Hence, this dataset was selected.

5.1.2 Description of Dataset

The dataset was originally developed in a study by Smith *et al.* at the National Institute of Diabetes and Digestive and Kidney Diseases [29]. It comprises 8 attributes as input variables, and

[1] `https://archive.ics.uci.edu/ml/index.php`

one output variable with 2 class values depicting diabetic and non-diabetic instances among 768 patients. Figure 5.1 below shows the correlation matrix plot describing the dataset.

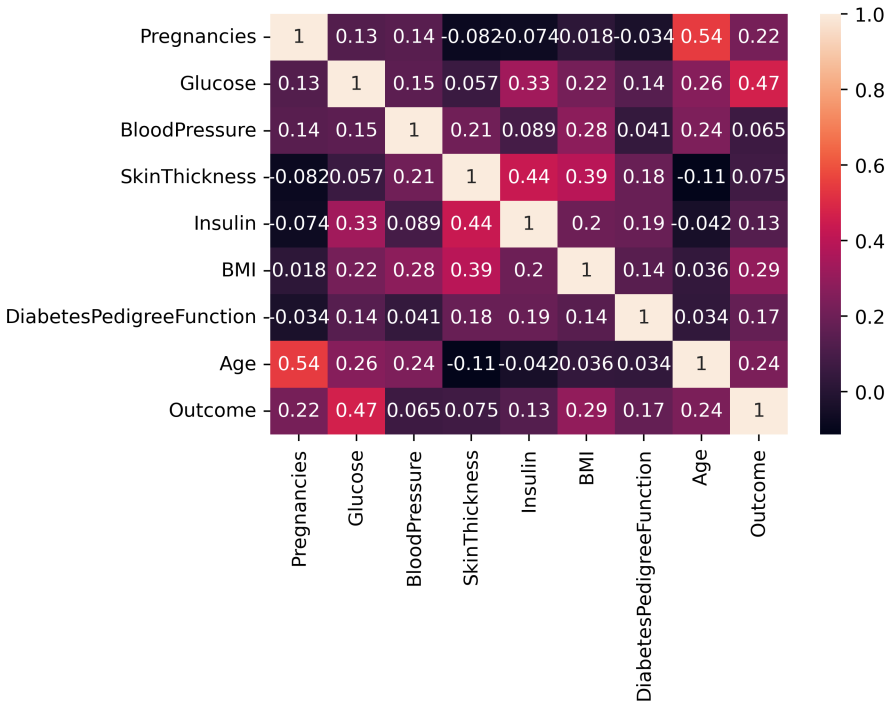

Figure 5.1: Correlation Matrix of PIMA Indians Diabetes Dataset

The dataset in its original form was used as a reference to compute model metrics and compare different machine learning algorithms. These results are presented later in Chapter 7. However, with only 768 rows and a size of 23 KB, the dataset wasn't fit to be used as a "massive" or "big" dataset for conducting our empirical analysis with big data. Hence, the dataset needed to be augmented and expanded.

5.1.3 Augmentation and Expansion of Dataset

Data augmentation is a suite of techniques to increase the size or quality of a dataset by modifying the original data or synthetically generating new data from existing data. Typically used in the fields of image classification and speech recognition, data augmentation helps in reducing overfitting during model training [28]. Data augmentation is similar in nature to regularization and oversampling techniques used in data analysis and deep learning, such as Batch Normalization, Transfer Learning, Pretraining & Zero-shot Learning [17].

Since the proposal in 2014 by Goodfellow *et al.*, Generative Adversarial Networks or GANs have become popular in generating training data through unsupervised learning [13]. There have been many extensions to GANs, such as CycleGANs & DCGANs, and numerous studies show their effectiveness in data augmentation [23].

This thesis makes use of a more recent GAN variant called Wasserstein GAN with Gradient Penatly or WGAN-GP. Proposed and demonstrated by Gulrajani *et al.*, WGAN-GP was shown

to have a stronger modelling performance in terms of training speed and stability in scaling datasets [15]. We implemented our adversarial network using **ydata-synthetic**, an open source python library which makes use of TensorFlow 2.0. The following code snippet depicts the training parameters and GAN model definition. The entire model training and data augmentation process took 3 hours and 50 minutes to completion, and was performed on the Google Cloud Platform.

```
from ydata_synthetic.synthesizers.regular import WGAN_GP
from ydata_synthetic.synthesizers import ModelParameters, TrainParameters

# Import Dataset
data = pd.read_csv('./data.csv')
num_cols = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
            'BMI', 'DiabetesPedigreeFunction', 'Age']
cat_cols = ['Outcome']

# Define the GAN and training parameters
noise_dim = 128
dim = 128
batch_size = 10
log_step = 100
epochs = 50
learning_rate = [5e-4, 3e-3]
beta_1 = 0.5
beta_2 = 0.9
n_critic = 3

gan_args = ModelParameters(batch_size=batch_size,
                           lr=learning_rate,
                           betas=(beta_1, beta_2),
                           noise_dim=noise_dim,
                           layers_dim=dim)

train_args = TrainParameters(epochs=epochs,
                             sample_interval=log_step)

# Define augmentation sample size, 50 million rows
sample_size = 50000000

# Random noise for sampling both generators
noise = uniform([sample_size, noise_dim], dtype=float32)

# Training the GAN model
model = WGAN_GP
synthesizer = model(gan_args, n_critic)
synthesizer.train(data, train_args, num_cols, cat_cols)

# Generate samples
gs_samples = synthesizer.sample(sample_size)[:sample_size]
```

Listing 5.1: WGAN-GP Python Implementation

The outcome of this augmentation process was a synthetic dataset based on the original PIMA Indians Diabetes Dataset, with 50 million exemplars & 2.84 GB in size.

5.2 Machine Learning Models

5.2.1 Model Selection & Implementation

Once the dataset had been selected and augmented, the next step was to prepare multiple machine learning models for our analysis study. As discussed in section 5.1.1, the PIMA Indians Diabetes Dataset had already been heavily studied with numerous machine learning & ensemble learning algorithms. Table 5.1 below depicts the machine learning algorithms we implemented for this thesis.

Algorithm	Type	Implementation Language(s)
Random Split	Sampling	Python
Stratified Split	Sampling	Python, Java, JavaScript
Kennard Stone	Sampling	Python
AdaBoost	Classification, Ensemble	Python
Deep Learning	Classification	Python, Java, JavaScript
Extra Trees	Classification, Ensemble	Python
Gradient Boosted Trees	Classification, Ensemble	Python
Random Forest	Classification, Ensemble	Python
Stacking XGBoost	Ensemble	Python
Max Vote	Ensemble	Python, JavaScript

Table 5.1: Machine Learning Algorithms used in this Thesis.

Depending on the libraries and frameworks used by various developers and machine learning researchers, machine learning models may perform several operations, but almost always certainly perform *training* & *prediction*. To enforce standardisation and ensure that all our models behave entirely in the same manner, and can be executed externally via command line, we defined the following specifications as depicted in Table 5.2 below. Thus, in future, if any researcher wants to implement a model or modify an existing model to make it work with *enseMbLer*, they can refer to these specifications.

Specification	Value	Description
Input Data Format	csv	Input data must be in .csv format.
Output Data Format	csv	Output data must be in .csv format.
Input Features	cli	Model input variables must be configurable via command line arguments.
Model Hyperparameters	cli	Model hyperparameters must be configurable via command line arguments.
Model	binary	Trained model must be saved in binary format.
Commands	cli	Model must support **TRAIN**, **PREDICT** & **VALIDATE** commands, execuatble via command line. Optionally, other commands such as FILTER may be defined.

Table 5.2: *enseMbLer* Machine Learning model specifications.

5.2.2 Model Metrics & Benchmark

After implementing the models, we trained them on a 30-70 test-train sample of the original PIMA Indians Diabetes Dataset. Three different sampling algorithms were used, Kennard Stone, Stratified & Random Sampling. The best accuracy scores were obtained using the Kennard Stone sampling algorithm. Table 5.3 below summarizes the binary accuracy metrics of all the models with Kennard Stone sampling algorithm. The results are presented later in Chapter 7.

Model	Kennard Stone
AdaBoost	85.71%
Deep Learning (Python)	77.92%
Deep Learning (Java)	51.94%
Deep Learning (JavaScript)	71.42%
Extra Trees	81.16%
Gradient Boosted Trees	**94.15%**
Random Forest	83.11%

Table 5.3: **Benchmark.** Model Binary Accuracy Summary.

5.2.3 Model Performance with Varying Dataset Size

There have been some studies which investigated the effect of dataset size in certain machine learning problems. In their study using NASA datasets from the PROMISE repositories, Catal & Diri discovered that Random Forest ensembles & parallel AIRS2 algorithms provided best performance for large datasets in terms of accuracy, however they did not mention compute or time efficiency details [8]. It has also been concluded that some algorithms, such as Deep Learning & Convolutional Neural Networks only work well when enough samples from the dataset are available [5]. Furthermore, it has even been investigated that for smaller datasets, a size of 3179 samples serves as a threshold for supervised training performance estimation [22].

To our knowledge, there haven't been extensive research done comparing model performances with "big data" datasets i.e. datasets with millions of samples. Therefore, drawing inspiration from some of the studies mentioned above, we used this thesis as an opportunity to also investigate how some of the most widely used machine learning algorithms perform against large datasets, to address **RQ1** in Chapter 4.3. We used our synthetically generated dataset with 50 million samples, and trained our models serially i.e without using any parallel or distributed computing, on the cloud using our proposed solution. These results are discussed in Chapter 7.

Chapter 6

Architecture Design & Implementation

Drawing inspiration from **FlexGP**, **FCUBE** & **cCube** as described in Chapter 3, we focused on ways to improve the performance of machine learning algorithms with big data by using principles of ensemble learning, data parallelism and cloud computing. We propose a microservices based architecture design utilising containerization & orchestration technologies to achieve our goals of scalablity, parallelization and supporting polyglot machine learning development.

The following chapter provides an overview of the architecture of our solution ***enseMbLer***, and the various technologies and components involved. First, Section 6.1 explains the microservices based architecture, describing how each service interacts with the other. A brief overview of the graphical user interface is also presented. Section 6.2 explains the various stages of operations, from factorization and sampling of data to prediction using trained models. Finally, Section 6.3 describes the various use cases and scenarios in which the proposed solution can be used.

6.1 Architecture

Over the recent years in software engineering, microservices have become *the* standard architectural style when it comes to distributed computing. Microservices are lightweight processes built around separate business capabilities, each independently deployable. This allows breaking applications into smaller services which can be scaled and maintained separately. In traditional monolith applications, which only run a single process, function calls and methods are used by the different components to invoke each other. However, microservice-based applications run many different services. These independent services require a communication standard and an orchestration machinery to accomplish a common task. They typically interact using communication protocols such as HTTP, AMQP and standards such as REST or SOAP. Therefore, designing a microservices based architecture also involves determining which protocols and orchestration technologies to use.

6.1.1 Design Goals

Table 6.1 below provides a summary of the technologies used to meet the design goals defined in Chapter 4.2.

Design Goal	Technologies
G1 Scalability	Docker, Kubernetes
G2 Portability	Docker
G3 Plug-n-Play	Docker, Kubernetes
G4 Polyglot Development	Docker
G5 Robustness	Kubernetes, RabbitMQ
G6 Black Box Execution	Docker, RabbitMQ
G7 Open Source.	Docker, Kubernetes, MongoDB, RabbitMQ
G8 Graphical User Interface.	React, Axios, ChakraUI

Table 6.1: Meeting Design Goals.

6.1.2 Architecture Overview

An overview of *enseMbLer* is depicted in Figure 6.1. The microservices based architecture follows a client-server model based upon REST communication with HTTP protocol. The core services of *enseMbLer* include *data-service*, *executor*, *factorizer*, *worker* and *gui*. The *data-service*, *executor*, *factorizer* & *worker* services are developed using **Java Spring** framework and form the back-end, providing key functionalities of job scheduling, data sampling & factorization and data persistence. Whereas the *gui* service, a **React** SPA, forms the front-end, providing access and control to all the functionalities through a graphical user interface. The *worker* service is a special wrapper service which encapsulates the user's machine learning model, thereby enabling the model's communication with the rest of the *enseMbLer* infrastructure. Each service is fully autonomous and provides a well defined REST API. In addition to the core services, **MongoDB** is used as the database to provide persistence and **RabbitMQ** is used as the message broker for AMQP protocol. AMQP enables asynchronous communication between the components and provides a means for robust message transfer, which is used in *enseMbLer* for scheduling of jobs and synchronisation.

These services and components are made cloud-ready following principles of containerization and industry proven Dev-Ops practices. Containerization is the process of packaging software with all the necessary libraries and dependencies required for execution, into lightweight containers. **Docker** is used as the container management system to package these components into docker containers which can run on any system using the docker engine. Finally, at the heart of *enseMbLer* lies **Kubernetes**, which is used as the container orchestration system for automating deployment and scaling of the services. Together, these technologies enable us to meet the design goals.

6.1.3 Components

Job

A job is the primary unit of work in the *enseMbLer* infrastructure. It represents a set of machine learning models, may or may not forming an ensemble, and the corresponding commands that need to be executed. It also comprises additional configuration parameters such as the degree of parallelization required, *factorizer* service URL and command, model hyperparameters, level, number of replicas and commands, and auto generated ids and temporal information for management. The *executor* microservice accepts the job in the JSON format. An example job configuration in the JSON format is depicted in Figure 6.2 below, with its corresponding ensemble configuration depicted in Figure 6.3.

In an ensemble of many machine learning models, there may be multiple levels i.e. output

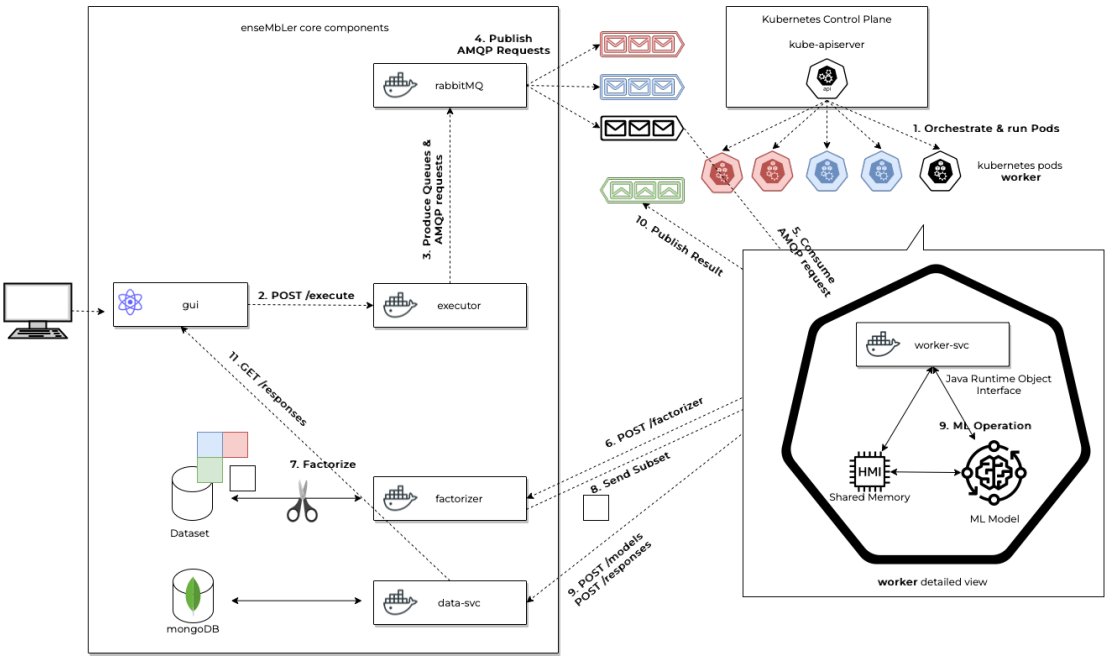

Figure 6.1: *enseMbLer* Architecture Overview.

from models may be fed into another model such as a stacking model or a max vote model, and this may be repeated a number of times, forming hierarchical levels. In the Figure 6.3, we show an ensemble of 7 learners, fused together using a stacking model, having two levels *l0* and *l1*. To allow such a configuration and flexibility for more in future, we assign each model with a *level* configuration.

Executor

The *executor* microservice handles the orchestration and scheduling of various machine learning jobs. It provides a REST API interface which is used by the *gui* to send job requests in the JSON format, and can additionally be used by external tools and scripts. It implements an AMQP message producer and communicates with RabbitMQ, the AMQP message broker.

Once a job request is received, the *executor* creates new AMQP topic exchange queues for each set of machine learning models. The respective models are bound to their corresponding queues via the binding key, defined by the *worker.model.name* meta property. Thus, if a job request comprises 3 distinct sets of models, 3 unique topic queues will be created, as depicted in Figure 6.1. Based on the degree of parallelisation and number of replicas, new AMQP messages containing details about which commands to execute are created , and published to the respective queues through RabbitMQ. The *executor* also implements an AMQP message receiver bound to a results exchange queue, common to all the models. Once the models execute their commands, the result messages are published to this queue. This is used for synchronisation purposes, to ensure that models at lower hierarchical levels in the configuration are scheduled first.

JSON Auth Query Header 1 Docs

```
{
  "jobName": "14sep0001",
  "trainDataSource": "http://factorizer-svc:9001/factorizer",
  "validateDataSource": "http://factorizer-svc:9001/factorizer",
  "predictDataSource": "http://factorizer-svc:9001/factorizer",
  "factorizeCmd" : "python3 Stratified.py factorize --test_size 0.3 --dataset data-51200k.csv",
  "jobKind": "TRAIN",
  "modelSet": {
    "set1": {
      "level": 0,
      "model": {
        "modelName" : "AdaBoost",
        "modelImage" : "adaB",
        "trainCmd" : "python3 AdaBoost.py train -x Pregnancies -x Glucose -x BloodPressure -x SkinThickness -x Insulin -x BMI -x
DiabetesPedigreeFunction -x Age -y Outcome --n_estimators 400 --learning_rate 0.65",
        "validateCmd" : "python3 AdaBoost.py validate -x Pregnancies -x Glucose -x BloodPressure -x SkinThickness -x Insulin -x BMI -x
DiabetesPedigreeFunction -x Age -y Outcome",
        "predictCmd" : "python3 AdaBoost.py predict -x Pregnancies -x Glucose -x BloodPressure -x SkinThickness -x Insulin -x BMI -x
DiabetesPedigreeFunction -x Age -y Outcome",
        "level" : 4
      },
      "replicas": 1,
      "tuneParameters": false
    },
    "set2" : {
        "level" : 0,
        "model" : {
          "modelName" : "GBoost",
          "modelImage" : "gBoost",
          "trainCmd" : "python3 GBoost.py train -x Pregnancies -x Glucose -x BloodPressure -x SkinThickness -x Insulin -x BMI -x
DiabetesPedigreeFunction -x Age -y Outcome --n_estimators 400 --max_depth 6",
          "validateCmd" : "python3 GBoost.py validate -x Pregnancies -x Glucose -x BloodPressure -x SkinThickness -x Insulin -x BMI -x
DiabetesPedigreeFunction -x Age -y Outcome",
          "predictCmd" : "python3 GBoost.py predict -x Pregnancies -x Glucose -x BloodPressure -x SkinThickness -x Insulin -x BMI -x
DiabetesPedigreeFunction -x Age -y Outcome"
        },
        "replicas" : 3,
        "tuneParameters" : false
    },
    "set3" : {
      "level" : 1,
      "model" : {
        "modelName" : "L1-XGBoost",
        "modelImage" : "xgbStack",
        "trainCmd" : "python3 L1-XGBoost.py train --n_inputs 7 --n_estimators 2000 --min_child_weight 2 --gamma 0.9 --subsample 0.8 --
colsample_bytree 0.8 --objective  binary:logistic --scale_pos_weight 1 --y Outcome",
        "validateCmd" : "python3 L1-XGBoost.py validate --n_inputs 7 --y Outcome",
        "predictCmd" : "python3 L1-XGBoost.py predict --n_inputs 7 --y Outcome"
      },
      "replicas" : 1,
      "tuneParameters" : false
    }
  }
}
```

Figure 6.2: *enseMbLer* Job in JSON format.

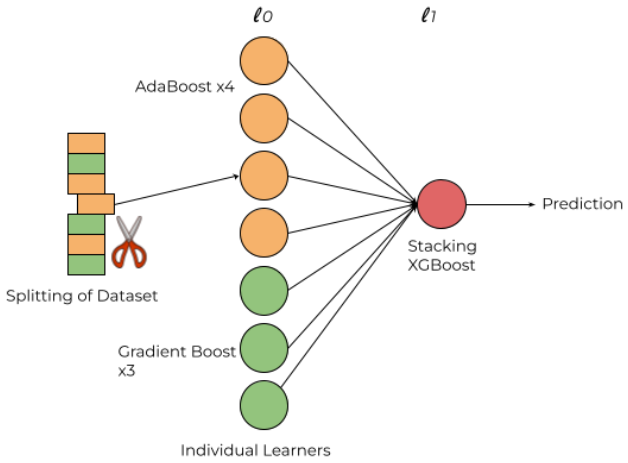

Figure 6.3: *enseMbLer* Ensemble configuration of 7 learners.

Worker

The *worker* microservice encapsulates the machine learning model and provides the necessary communication mechanism to fetch & store both data and models themselves. Essentially, it is a web server developed using the Java Spring framework. The server provides a REST API interface for management and implements an AMQP message receiver for machine learning job scheduling. It executes the machine learning model commands through the Java Runtime Object, by providing an interface with the environment in which the application is running, as depicted in Figure 6.1. This means that both the machine learning model and *worker* service must reside in the same runtime environment i.e. in the same docker container. This is achieved by using the *ensembler.worker.template* Dockerfile, which automatically injects the machine learning model into the *worker* service container. During configuration, an appropriate *worker.model.name* meta property needs to be set, which tells the infrastructure which machine learning model is encapsulated.

Once a job request is sent to the *executor* service, it schedules the individual worker request messages and sends them to topic exchange queues. Each *worker* is bound to an appropriate topic exchange queue by means of the *worker.model.name* key. Hence, when a corresponding message is published, the respective *worker* consumes it and executes the appropriate command for the machine learning operation. When the command is executed successfully, a success message containing all the relevant details is sent out to a common results queue.

The *worker* also interacts with the *factorizer* microservice to either fetch the training sample data or data for inference. This is done via REST request-response messages over HTTP. As there is no theoretical size limit to the HTTP payloads, even large datasets can be transferred this way, but may lead to overheads. We discuss our findings later in Chapter 7.1.4. Similarly, the *worker* also interacts with the *data-service* microservice using its REST API. This is done to either store trained models, store the results or fetch trained models & output data from previous jobs.

In an ensemble configuration, *worker* at a higher level must wait for *worker*(s) at lower levels to finish their executions before it can begin its task. It may also need to fetch the output of the models from the corresponding lower-levels. This is ensured in the *worker* control logic, and also taken care of partly by the *executor* service, which only proceeds to new higher levels once *worker*(s) at previous lower levels have finished their tasks.

Factorizer

The *factorizer* microservice provides a REST API and interacts with a *worker* to provide data for machine learning operations. It primarily fetches the training dataset and uses a sampling algorithm to provide a sample subset for machine learning training. This sampling process allows splitting of the large dataset into smaller chunks to be sent to the different workers, and we use the term "factorization" to denote this entire splitting process. The sampling algorithm, splitting size and the command is configurable in the job request. If the algorithm supports random seeds, such as in the case of scikit-learn's stratified or random sampling algorithms, same seed values can be used to ensure consistency among different requests and jobs, which can be used for hyperparameter optimization or performing k-fold cross validation of the different machine learning models. It also provides the datasets for validation and prediction in a similar manner, however without using any further sampling processes.

Similar to creating a *worker* which encapsulates a machine learning model, *factorizer* models can be created by injecting the desired sampling algorithm into the same runtime environment using the *ensembler.factorizer.template* Dockerfile. The sampled dataset is transferred over HTTP responses, with no theoretical payload size limits.

Data-Service

The *data-service* microservice provides persistence of input and output data of the respective *worker*(s), as well as the trained models. It implements database connectivity with MongoDB and exposes a REST API to perform all basic CRUD (Create, Read, Update & Delete) operations. It is used by both the *gui* and *worker* microservices.

MongoDB

MongoDB is a cross-platform, NoSQL, document oriented database, which works with collections and documents instead of relations and tables. This allows for a dynamic schema and higher level of flexibility. One of the reasons for choosing MongoDB over other traditional relational databases was the distributed architecture of MongoDB, allowing for horizontal scaling. This suits the microservices architecture paradigm, and is more suitable for cloud solutions.

RabbitMQ

RabbitMQ is an open source, lightweight message broker that supports AMQP protocol, among other messaging protocols. We use AMQP to asynchronously distribute machine learning workload among the many *workers*. Specifically, we use it to establish topic exchanges, which lets the *executor* send messages to queues based on the binding keys derived from the machine learning model names. Hence, the *workers* receive tasks only for the machine learning models they encapsulate.

GUI

React is an open source JavaScript library for building graphical user interfaces. The *gui* microservice is essentially a React SPA (Single Page Application), to aid human supervision and let users monitor & control the machine learning training processes. It was developed using Chakra UI component library, and internally uses the Axios promised-based HTTP client to exchange REST requests & responses with the *executor* and *data-service* microservices. The Figure 6.4 depicts the user interface. It comprises of a job configuration widget which lets the user enter all necessary configuration commands and parameters to start a machine learning task. This includes configuring the *factorizer* url and commands, as well as configuring all the machine learning models in an ensemble. Additionally, hyperparameters and training commands for the models can also be configured. Instead of manually typing in and configuring a job, entire JSON configurations can also be loaded. Finally, as the models finish their executions, the results are displayed in the table widget. Additional details are visible when the cursor hovers over the table columns.

6.2 Stages of Operation

In this section, we describe the different stages of operation when a user interacts with *enseMbLer*. By user, we mean an end user i.e. a person who wishes to use *enseMbLer* for one of the use cases we mention in the following Section 6.3. This could be:

- Researchers in academia who wish to compare algorithms and collaborate solutions to machine learning problems.
- Engineers who wish to optimize hyperparameters and select best features for a dataset.
- Students who wish to create and study ensembles.

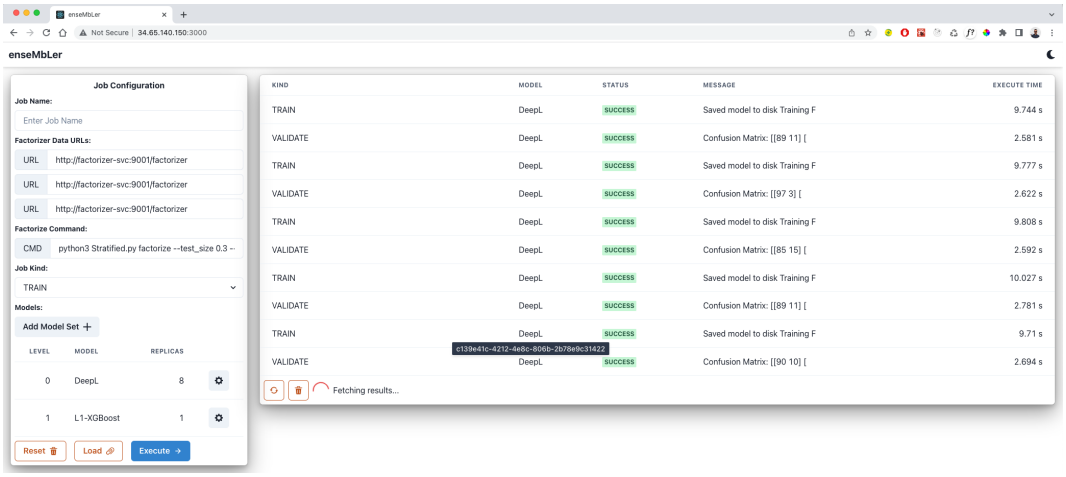

Figure 6.4: *enseMbLer* Graphical User Interface

- Any other end user in the machine learning community.

6.2.1 Infrastructure Setup

First it is necessary to set up the infrastructure. This involves configuring local machines or virtual machines in the cloud to handle the workload. Since *enseMbLer* uses open source technologies, there is no restriction in choosing a cloud provider. It is also possible to use a combination of multiple cloud providers and local machines, forming a hybrid cloud. For the purposes of this thesis and conducting our empirical analysis, we chose the Google Cloud Platform and created a Kubernetes cluster using the Google Kubernetes Engine (GKE). Once the Kuberenetes cluster is ready, we use kubectl to launch our deployments. All the scripts and deployments are attached in the Appendix. This setup process is only required once in the beginning, and doesn not need to be repeated for running different jobs.

6.2.2 Preparation

To use a custom machine learning model developed by the user with *enseMbLer*, the user needs to inject the model into the *worker* microservice. This can be done using the *ensembler.worker.template* Dockerfile. The user needs to add all the relevant steps required to containerize their model code and build a docker image. Furthermore, they need to upload the image to a public repository. We followed these steps to prepare all of our models mentioned in Table 5.1 and uploaded the respective images to Dockerhub. The Listing below depicts our Deep Learning (Java) model's Dockerfile. The image was tagged and uploaded as shobuxtreme/ensembler.worker:deeplj. The template Dockerfile is listed in the Appendix.

```
# Ensembler ML-worker Deep Learning Java
FROM ubuntu

# 2. Inject ensembler base worker msvc
ADD worker-install.sh .
RUN chmod +x /worker-install.sh
RUN /worker-install.sh
```

```
WORKDIR /usr/src/ensembler
COPY worker-0.0.1.jar .
ENTRYPOINT ["java","-jar","worker-0.0.1.jar"]

# 4. ----- BEGIN INSTRUCTIONS -----
# Your dockerfile instructions
ADD install.sh .
RUN chmod +x install.sh
RUN ./install.sh
COPY . .

# 4. ----- END OF INSTRUCTIONS -----
```

Listing 6.1: Deep Learning Java Worker Dockerfile

Similarly, the user can also use their own factorization and sampling algorithms for splitting the training dataset. Once all the images are uploaded to a public repository, the user can start configuring jobs using the *enseMbLer* user interface.

6.2.3 Factorization and Training

First, the user needs to train a model or multiple models in an entire ensemble. This is done by setting the job kind to TRAIN. When the job is executed, the *executor* will start producing and publishing the requests to model queues based on the level of the models in the ensemble. The *worker* bound to the queue will consume the request, and make a corresponding REST request to the *factorizer* to fetch the data. The *factorizer* will use the configured sampling algorithm to produce a subset for training, and send it back to the *worker*. When multiple such requests are received, the effect is similar to that of "splitting", as depicted in Figure 6.5 below. The big dataset is broken into smaller splits, and these splits are sent to separate *worker*s.

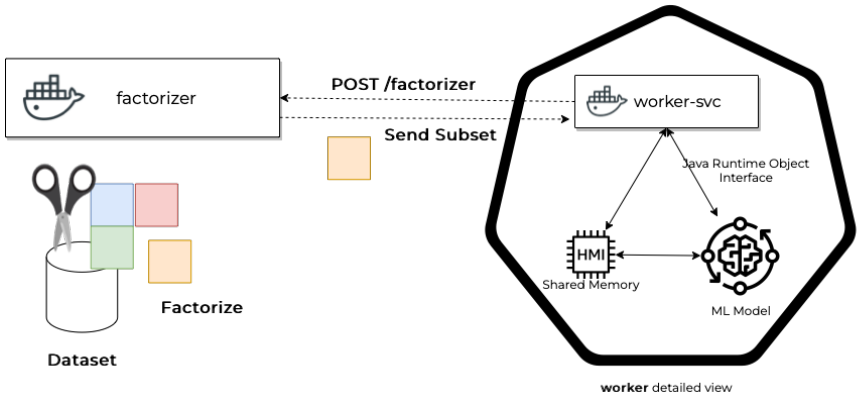

Figure 6.5: *enseMbLer* Factorization Process.

After receiving the training subset, the *worker* executes the training command which is contained in the request message. This results in the execution of the machine learning algorithm's training. However, if the model has a higher level in an ensemble job configuration, the *worker* will also fetch all the input datatsets from previous levels as required. This is needed ,for example, in the case of max vote or stacking algorithms. Once the training is finished, the corresponding

trained model, input training dataset and execution result are saved through a REST request to the *data-service*. A final result message is published to the results queue, which is consumed by the *executor*. The results are then updated on the *gui*.

6.2.4 Validation

Once models have been trained, their metrics can be computed using the VALIDATE job kind. When such a job is executed, like in the previous training stage, the *executor* publishes requests to model queues and the *worker*s make a request to *factorizer*. However this time, the *factorizer* sends a separate validation dataset, instead of using a sampling algorithm. A constant dataset for validation across all requests and job runs ensures that all models can be evaluated equally. This can be configured while creating the *factorizer* docker image. The *worker* also fetches the latest trained models by making the appropriate REST API call to the *data-service*. Once the model is loaded, the validate command is executed and the results are saved. A result message is published to the results queue. The choice of precise metrics to be calculated remains with the machine learning algorithm's developer. In our case, we calculate the model's binary classification accuracy, confusion matrix, precision score, recall score & F1 score.

6.2.5 Prediction

Finally, trained models can be used to make predictions on new data. This process is known as inference. Similar to the validation stage, when a job of kind PREDICT is executed, the *executor* publishes requests to model queues and the *worker*s make a request to *factorizer*. The predict dataset is sent back to the *worker*s. This dataset can be configured while creating the *factorizer* docker image. Once the *worker* receives the dataset, it makes a REST API call to the *data-service* to fetch the latest trained model. The *worker* then executes the prediction command and the result message & output dataset are saved. A subsequent result message is published to the results queue, and the results are updated on the *gui*.

If the machine learning model has a higher hierarchical level in the ensemble configuration, the *worker* will also fetch the output predictions from models at the lower levels.

6.3 Use Cases

We designed *enseMbLer* with the main aim of helping users create ensemble models & train machine learning models on large datasets, as well as facilitating machine learning collaboration. While there could be more use cases & potential users of such an infrastructure, we consider the following to be of most significance:

1. **Handle Big Data.** To make use of large datasets for model training which may span over multiple databases, resulting in hundreds of gigabytes in size. Using a suitable sampling and factorization algorithm, the dataset can be split into smaller chunks and models can be trained parallelly on these chunks. The models can then be fused to form an ensemble, combining the training results from all individual models. This approach doesn't require any expertise or knowledge of special frameworks such as Apache Hadoop or Apache Spark, and the entire training can be done through the graphical user interface.

2. **Machine Learning Collaboration.** To allow different machine learning algorithms, potentially developed in different programming languages by different people, to collaborate and solve the same problem. This requires the infrastructure to be open source, distributed,

polyglot and user friendly. Machine learning collaboration enables a *"bring your own model"* paradigm, where users have the flexibility to bring their own models and at the same time, use models developed by others in a plug-n-play manner. This can especially be useful in comparing models or optimising ensembles.

3. **Hyperparameter Optimization.** To tune the hyperparameters for a model. Hyperparameters are parameters external to the model, whose value must be set manually. These are used to control the learning process, and depending on the specific problem and dataset, need to be tuned to solve the problem optimally. Same random seed values can be used to control the sampling and factorization processes, and multiple replicas of the same machine learning model can be used with different hyperparameter values. These models can then be trained simultaneously on the same data, and hence models with best hyperparameters can be selected, resulting in an efficient tuning process.

4. **Feature optimization.** To simplify and optimize machine learning process by removing input features. Feature optimization is the process of reducing input variables to reduce the computational costs of training and increase performance of the model. Given that the model allows configuring features, i.e. input columns, via the command line and thus respecting the specifications, replicas can be trained in parallel and the models with the best results and metrics can be filtered.

5. **Lambda Architectures.** To deal with massive amounts of data efficiently. Lambda architectures are special data processing architectures which provide low latency, increased throughput and high fault tolerance by using a combination of batch processing and stream processing methods. These architectures typically comprise 3 layers, namely the batch layer, the serving layer and the speed layer. New data comes in continuously and is fed to both the batch & speed layers. The speed layers process the data in real-time, however it doesn't prioritize completeness or accuracy. Batch layer aims to provide perfect accuracy and handles all data in batches. In a machine learning scenario, *enseMbLer* can be used as a batch processing infrastructure to train models on big data, and the updated models can then be used in the speed layer, thus supporting lambda architectures for machine learning as depicted in Figure 6.6.

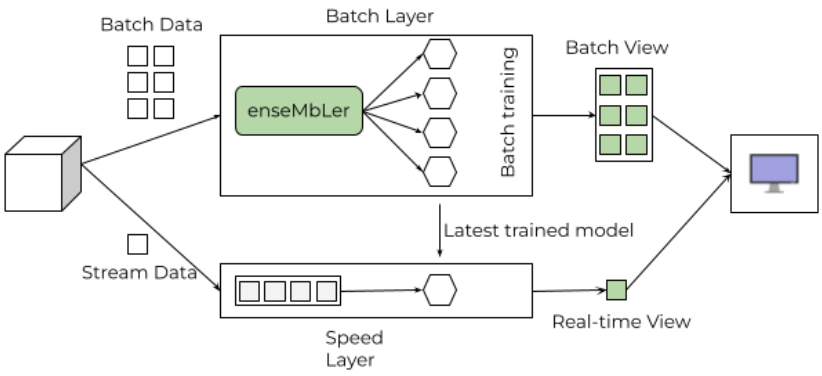

Figure 6.6: *enseMbLer* as batch processing infrastructure in lambda architecture

Chapter 7

Results & Evaluation

Once the *enseMbLer* architecture was implemented and set up, we were ready to test its capabilities & demonstrate its effectiveness in supporting the use cases mentioned in Chapter 6.3, addressing the limitations of the state-of-the-art techniques mentioned in Chapter 4.1.

We first used *enseMbLer* to run our standard machine learning models with the original PIMA Indians Diabetes Dataset to create a benchmark for further evaluations. The summary of these results was already briefly expressed in Chapter 5.2.2, and we present the full results in Section 7.1.1 below. We then proceeded to conduct an empirical analysis by running experiments to explore and gain insights into the research questions described in Chapter 4.3 and discuss our findings in Sections 7.1.2 through 7.1.5. We further discuss the possible threats to validity of our findings in Section 7.2

7.1 Results

The following Table 7.1 denotes the abbreviations we have used for our machine learning models in the subsequent sections. Furthermore, Table 7.2 provides a summary of the datasets we used for our empirical analysis.

Abbreviations	Model	Hyperparameters
AB	AdaBoost (Py)	nEstimators=400, learningRate=0.65
DL	Deep Learning (Py)	epochs=150, lossfn=binaryCrossentropy
DJ	Deep Learning (Java)	epochs=10, lossfn=binaryCrossentropy
DS	Deep Learning (JS)	epochs=150, lossfn=binaryCrossentropy
ET	Extra Trees (Py)	nEstimators=400, maxDepth=5, minSamplesLeaf=2 nJobs=1
GB	Gradient Boosted Trees (Py)	nEstimators=400, maxDepth=6
RF	Random Forest (Py)	nEstimators=400, maxDepth=5, nJobs=3, maxFeatures=sqrt

Table 7.1: Abbreviations used to denote Machine Learning Models.

No. of exemplars	Dataset size
100K	4.4 MB
200K	8.8 MB
400K	22.5 MB
800K	35.2 MB
1600K	90 MB
3200K	180 MB
6400K	360 MB
12800K	720.1 MB
25600K	1.44 GB
50000K	2.84 GB

Table 7.2: Augmented Datasets based on the PIMA Indian Diabetes Dataset

7.1.1 PIMA Indians Diabetes Dataset Benchmark Results

Figure 7.1 depicts the binary accuracy scores of our machine learning models with a 30-70 test-train split of the PIMA Indians Diabetes dataset. We notice that best results are achieved with the Kennard Stone sampling algorithm, with Gradient Boosting algorithm achieving a high accuracy of 94.15 %. We will use these results as a benchmark for further comparisons. Figure 7.2 depicts the training time in seconds taken by the respective models. We notice that the Deep Learning JavaScript (DS) algorithm takes significantly higher time. On further analysis we found that this is due to no GPU support on our GKE nodes, which forces the TensorFlow JS library to rely on a much slower CPU computations.

7.1.2 RQ1. How do traditional machine learning models scale with big data?

Figure 7.3 depicts the training times (in seconds) of the standard machine learning models as a function of the input dataset size. We used our synthetically generated dataset and trained the models in a serial manner using *enseMbLer*. We notice that the training time is directly proportional to the input dataset size, and grows with a rough factor of 1.8 - 2.2 times. We used Google Cloud Platform's C2 Series compute optimized nodes with 8 vCPUs and 32 GB memory. We were unable to perform these machine learning training tasks with E2 micro instances. Therefor, we can conclude that **traditional machine learning models do not scale well to handle large datasets**. Furthermore, Figure 7.4 highlights that the binary accuracy of these models significantly reduces as the dataset size increases.

7.1.3 RQ2. How do ensembles created through data parallel factorization techniques compare against their standard models?

To address the research question, we created ensembles of each model using *enseMbLer* with 8 as a degree of parallelization and compared our results against those of Akyol *et al*. and our previously derived benchmark from Section 7.1.1. The Figure 7.5 below shows the ensemble configuration, which is similar for all the 7 models. The results from the individual learners are combined using a Stacking XGBoost algorithm, and listed in Table 7.3. As we can see, **ensembles created through**

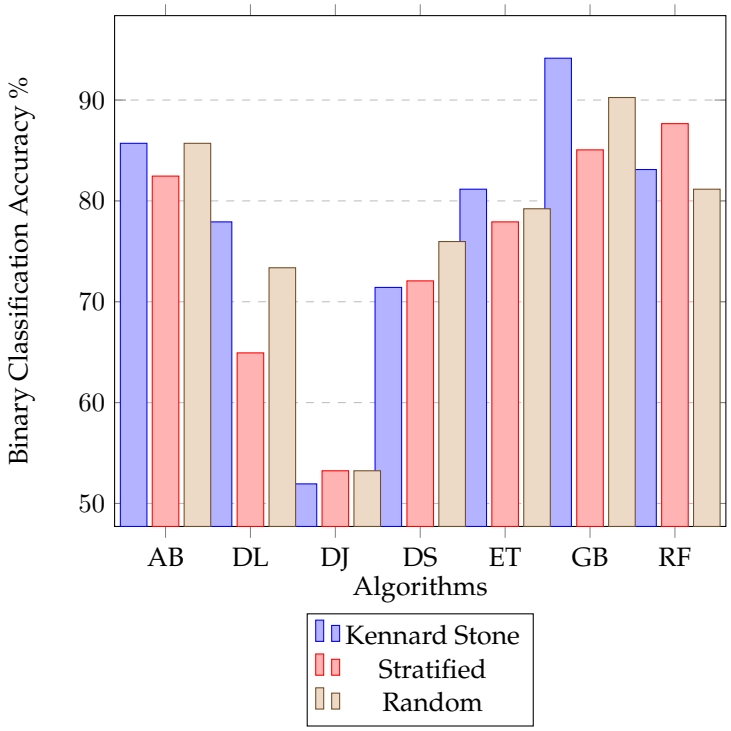

Figure 7.1: **Benchmark** Model Binary Accuracy Metrics for PIMA Indians Diabetes Dataset

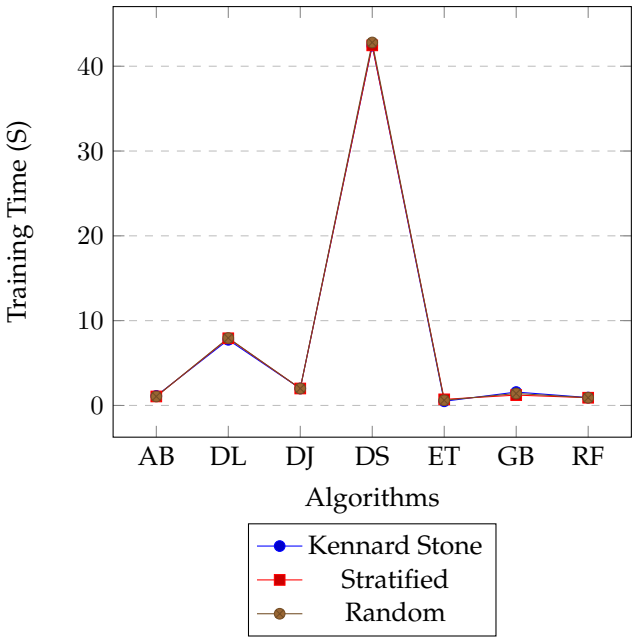

Figure 7.2: **Benchmark** Model Training Time Metrics for PIMA Indians Diabetes Dataset

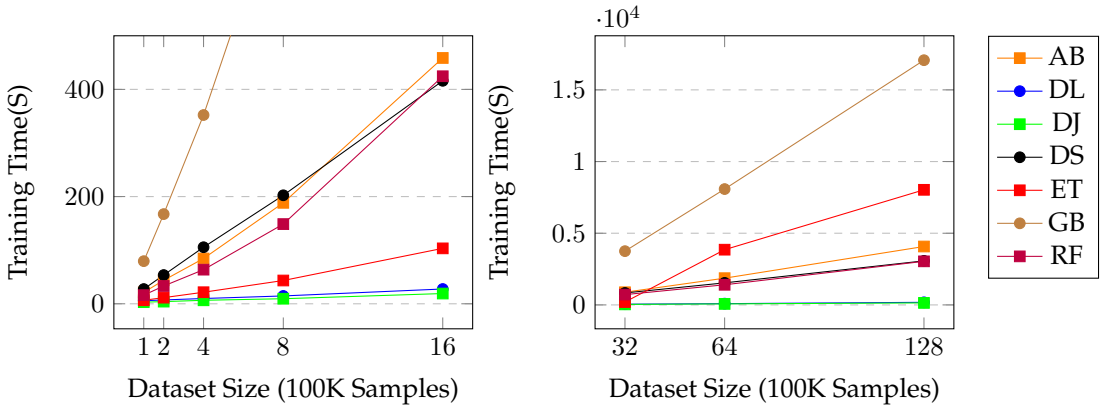

Figure 7.3: **RQ1.** Model Training Time against Varying Dataset Size

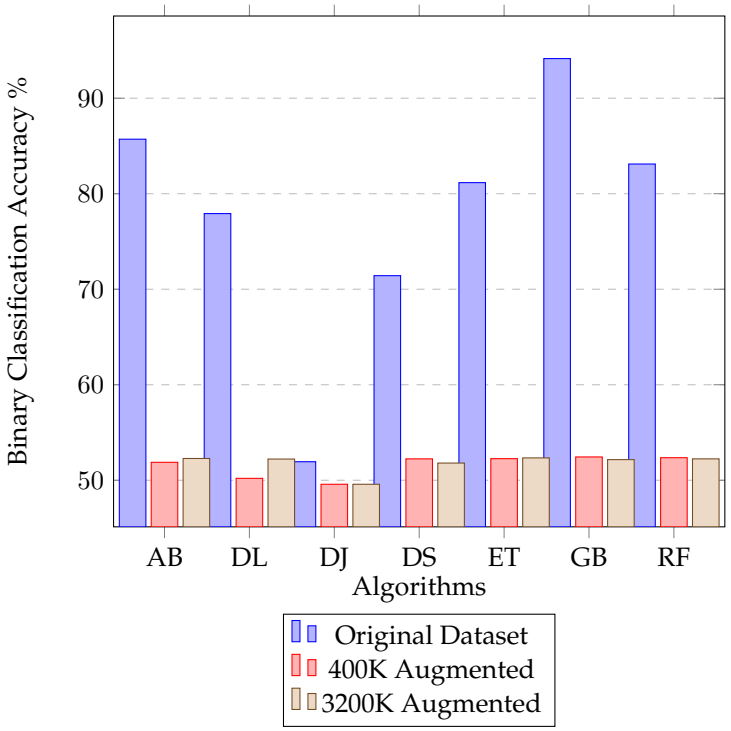

Figure 7.4: **RQ1.** Comparison of Model Binary Accuracy Scores against Varying Dataset Size

data parallel factorization techniques outperform all the other models, achieving highest accuracy.

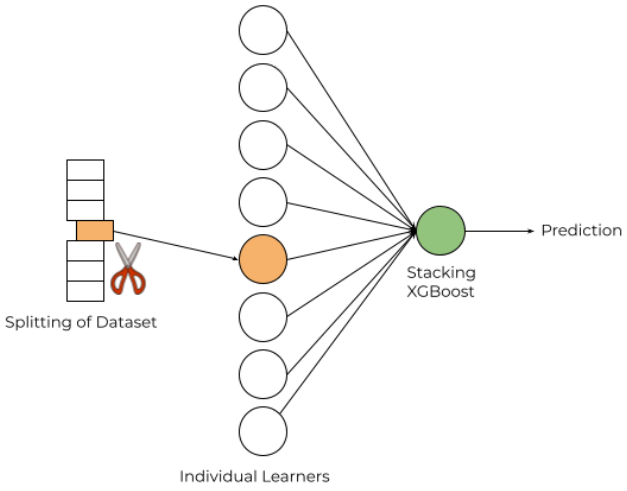

Figure 7.5: **RQ2.** Ensemble configuration

Model	Our Benchmark	RQ2 Ensemble	Akyol *et al*'s [2]
AdaBoost	85.71%	**95.45%**	73.88%
Gradient Boosted Trees	94.15%	**96.10%**	73.16%
Random Forest	83.11%	**86.36%**	73.45%
Extra Trees	81.16%	**86.11%**	n/a
Deep Learning	77.92%	**86.36%**	n/a

Table 7.3: **RQ2.** Binary Accuracy Comparison.

7.1.4 RQ3. How much overhead does a HTTP based RESTful infrastructure add to distributed machine learning?

Figure 7.6 depicts the overhead as time in seconds due to factorization and transfer of dataset over HTTP protocol. As expected, factorization time increases as the dataset size increases. However it is interesting to note that the time taken to transfer data between the microservices and components does not increase. We ran our *enseMbLer* infrastructure over Google Cloud Platform, with nodes in the same zone i.e europe-west-6a (Zurich). On further analysis, we found that our network bandwith on GCP was 16Gbps and with paid plans this could be increased upto 100 Gbps. Hence, we conclude that **there is no significant overhead due to data transfer between components in a HTTP based RESTful infrastructure over public cloud providers. There is however an overall overhead due to factorization of datasets**. This can further be minimised with pre-processing and batch pre-factorization techniques.

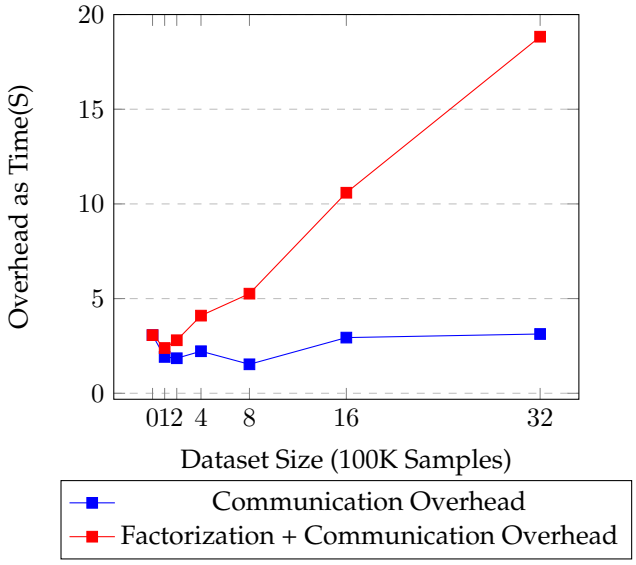

Figure 7.6: **RQ3.** Comparison of Overhead against Varying Dataset Size

7.1.5 RQ4. How does degree of parallelization affect the performance of massively parallel ensembles?

We used the DL Deep Learning Python machine learning model and trained multiple ensembles by varying the degree of parallelization i.e. by varying the number of replica learners. Figure 7.7 depicts the binary accuracy of these models with the PIMA Indians Diabetes Dataset. We notice a significant increase in the accuracy on using parallelization, however we conclude that **in this scenario there is no significant affect in the performance of ensembles on increasing the degree of parallelization beyond 2**.

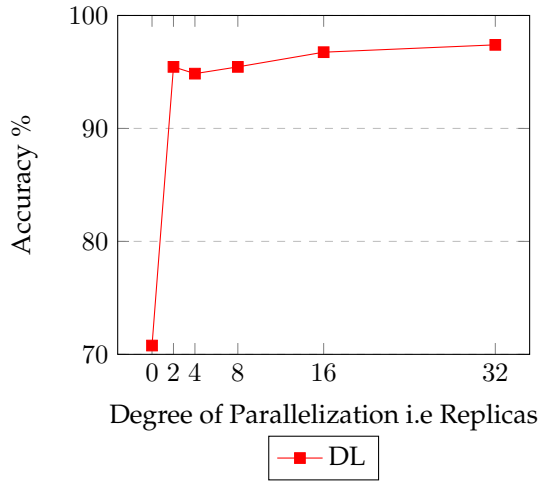

Figure 7.7: **RQ4.** Ensemble Model Accuracy against Degree of Parallelization

7.2 Threats to Validity

Internal Validity. Internal validity when testing cause-effect relationships represents the confidence and trustworthiness that the relationship being tested is not influenced by other factors. Due to the nature of our experiments in investigating compute times and overheads using cloud technologies, multiple machine learning algorithms and synthetically generated datasets, we faced the following the threats:

- Our synthetically generated dataset comprised of 50 million exemplars. Due to the randomness in factorization, different runs could yield very different metrics and computational times. We mitigated these risks by taking an average of 10 runs for each experiment. Moreover, using the same random seed values allowed us to maintain consistency across multiple runs.
- We used both the original PIMA Indians Diabetes Dataset and a synthetically generated large dataset based on the original to conduct our empirical analysis. However, we used hyperparameters which provided optimum results with the original dataset in both scenarios. This is clearly visible in the difference between accuracy metrics of models with the two datasets. By focusing on just overhead and speedup calculations, we avoided drawing conclusions for accuracy related model metrics for the larger dataset.
- We ran our infrastructure on the Google Cloud Platform's Google Kubernetes Engine (GKE). Although we had full control over configuring the hardware and software, Kubernetes automatically distributes and schedules pods across the cluster. This could result in some nodes having more intensive workloads and affect the computational performance of some algorithms. Unfortunately, there is not much that we could do to mitigate these risks, other than taking an average of multiple results.

External Validity. External validity represents the extent to which discovered results and conclusions from a study can be generalized to other events and work. Machine learning is a complex field of study comprising different types of techniques like supervised, unsupervised and reinforced learning. Due to the way in each each type of algorithm works, in general it is not possible to generalise and apply findings from one study with a certain dataset to other studies involving different datasets and algorithms. Therefor we only focus on comparing our model metric results and findings to that of similar studies with the PIMA Indians Diabetes Dataset. We can however generalise the findings and conclusions on overhead, speedup and computational times using massively parallel factorization and ensemble learning, and apply it to other scenarios involving distributed cloud computing.

Chapter 8

Conclusion

8.1 Conclusion

This thesis was devoted to research work in the intersection of the fields of machine learning and big data. We aimed to discover more insights into the challenges of using traditional machine learning techniques with large datasets, which were best represented by the 3V's model. We reviewed some of the current state-of-the art techniques for using machine learning with big data and further identified the limitations and drawbacks associated with each. We then proposed *enseMbLer* an open source, scalable, cloud-ready architecture, building upon the principle of massively data parallel factorization to build ensembles and support machine learning collaboration.

To demonstrate its functionality, we first augmented a heavily researched dataset using a state-of-the-art *Generative Adversarial Network*, WGAN-GP and later used this dataset to conduct multiple experiments with 7 different machine learning models. Through these experiments:

1. We verified that traditional machine learning models do not scale well with big data.
2. We demonstrated that ensembles created through massively data parallel factorization techniques outperform other standard models and ensemble models.
3. We determined that a RESTful infrastructure for distributed machine learning does not add significant overheads due to communication and data transfer over HTTP protocol.
4. We discovered that there is no significant affect in the performance of ensembles on increasing the degree of parallelization, in the case of PIMA Indians Diabetes Dataset.

8.2 Summary of Contributions

The main contributions of this thesis are:

1. We designed an developed a massively parallel cloud-ready architecture, **enseMbLer** [1] [2], which met our goals of being *open-source*, *scalable*, *portable*, *plug-n-play* friendly, *robust*, supported *black-box* execution and provided a *graphical user interface*.
2. We demonstrated the capabilities of **enseMbLer** and its effectiveness in addressing the limitations of the present state-of-the-art cloud based techniques through an **empirical analysis** by running multiple experiments against our synthetically generated big dataset.

[1] `https://github.com/shobuxtreme/ensembler`
[2] `https://hub.docker.com/u/shobuxtreme`

3. We further demonstrated that a massively parallel HTTP-based REST infrastructure does not lead to significant communication overhead. It can further be used in the batch processing layer of a *lambda* architecture to train models in batches which can support stream processing.

4. We also verified and contributed further evidence to support that ensembles can achieve higher accuracy with more efficient performance than traditional machine learning algorithms. Specifically, our ensemble based on data-parallel factorization technique achieved a very high accuracy with the PIMA Indians Diabetes Dataset, beating other models.

8.3 Future Work

This thesis proposed *enseMbLer*, an open source cloud architecture for scaling massively parallel ensemble learning and machine learning collaboration. While the proposed microservices based design lets us meet our goals and addresses the limitation of other state-of-the-art techniques, further enhancements can be made to improve the capabilities of the solution.

1. Kuberenetes is a very powerful, state-of-the-art, orchestration system for management of containerized applications. The current implementation of *enseMbLer* uses **kubectl**, the Kubernetes command-line tool, during infrastructure setup for deployment of all the containers and services. Scaling of all the components and workers, thus, requires interaction through kubectl scripts. User experience can further be optimized if these tasks can be done through the graphical user interface. This will involve extending the React front-end SPA to make REST requests to Kubernetes' **kube-apiserver**.

2. The current state-of-the-art massively parallel cloud based ensemble techniques use factorization i.e splitting of larger dataset into smaller chunks by means of various sampling algorithms. As presented in Chapter 7, factorization time increases as the overall dataset size grows. Even if factorization operations can scale horizontally across multiple nodes, the workers would still need to wait for the factorization process to be finished before they receive the training datasets. This waiting time can be minimized and the overall factorization process can be optimized through **batch factorization**. This involves factoring the dataset into subsets in batches, *before* the workers request for training data.

3. The current implementation of *enseMbLer* uses MongoDB's **BinData** BSON data type to store binary files such as trained models or model output. This limits the file size to 16MB i.e if the trained model & its weights need more than 16MB for storage in a binary format, *enseMbLer* will not be able to persist the model. This can be addressed by extending the *data-service* microservice to use MongoDB's **GridFS** specification which divides larger files into chunks.

4. The current implementation of *enseMbLer* requires the user to manually input all the hyperparameter values. Hyperparameter tuning can be performed by using scripts to compare and select the values which provide best results. This capability can be further improved by implementing auto tuning support into the infrastructure. This would require extending the *executor* microservice to tweak the hyperparameters using grid based and random search algorithms. The entire infrastructure can then behave as a genetic algorithm, and can be used to obtain optimum hyperparameter values for multiple models simultaneously.

Bibliography

[1] "Big data market size revenue forecast worldwide from 2011 to 2027," 2022. [Online]. Available: https://www.statista.com/statistics/254266/global-big-data-market-forecast/

[2] K. Akyol and B. Sen, "Diabetes mellitus data classification by cascading of feature selection methods and ensemble learning algorithms," *International Journal of Modern Education and Computer Science*, vol. 10, pp. 10–16, 06 2018.

[3] I. Arnaldo, K. Veeramachaneni, A. Song, and U.-M. O'Reilly, "Bring your own learner: A cloud-based, data-parallel commons for machine learning," *IEEE Computational Intelligence Magazine*, vol. 10, no. 1, pp. 20–32, 2015.

[4] M. Assefi, E. Behravesh, G. Liu, and A. P. Tafti, "Big data machine learning using apache spark mllib," in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 3492–3498.

[5] J. G. A. Barbedo, "Impact of dataset size and variety on the effectiveness of deep learning and transfer learning for plant disease classification," *Computers and Electronics in Agriculture*, vol. 153, pp. 46–53, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168169918304617

[6] R. Bekkerman, M. Bilenko, and J. Langford, "Scaling up machine learning: Parallel and distributed approaches," ser. KDD '11 Tutorials. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2107736.2107740

[7] M. Beyer and D. Laney, "The importance of'big data': A definition. stamford, ct: Gartner. retrieved june 22, 2014," 2012.

[8] C. Catal and B. Diri, "Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem," *Information Sciences*, vol. 179, no. 8, pp. 1040–1058, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025508005173

[9] V. Chang, J. Bailey, Q. A. Xu, and Z. Sun, "Pima indians diabetes mellitus classification based on machine learning (ml) algorithms," *Neural Computing and Applications*, Mar 2022. [Online]. Available: https://doi.org/10.1007/s00521-022-07049-z

[10] X.-W. Chen and X. Lin, "Big data deep learning: Challenges and perspectives," *IEEE Access*, vol. 2, pp. 514–525, 2014.

[11] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *International Journal of Information Management*, vol. 35, pp. 137–144, 04 2015.

[12] D. Gillick, A. Faria, and J. DeNero, "Mapreduce: Distributed computing for machine learning," *Berkley, Dec*, vol. 18, 2006.

[13] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

[14] S. Gopalani and R. Arora, "Comparing apache spark and map reduce with performance analysis using k-means," *International journal of computer applications*, vol. 113, no. 1, 2015.

[15] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved training of wasserstein gans," 2017. [Online]. Available: https://arxiv.org/abs/1704.00028

[16] A. Karegowda, A. Manjunath, and J. M.A, "Application of genetic algorithm optimized neural network connection weights for medical diagnosis of pima indians diabetes," *International Journal on Soft Computing (IJSC)*, vol. 2, 04 2011.

[17] J. Kukaka, V. Golkov, and D. Cremers, "Regularization for deep learning: A taxonomy," *ArXiv*, vol. abs/1710.10686, 2017.

[18] Y. Liu, J. Yang, Y. Huang, L. Xu, S. Li, and M. Qi, "Mapreduce based parallel neural networks in enabling large scale machine learning," *Intell. Neuroscience*, vol. 2015, jan 2016. [Online]. Available: https://doi.org/10.1155/2015/297672

[19] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," 2014. [Online]. Available: https://arxiv.org/abs/1408.2041

[20] L. R. Nair, S. D. Shetty, and S. D. Shetty, "Applying spark based machine learning model on streaming big data for health status prediction," *Computers Electrical Engineering*, vol. 65, pp. 393–399, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790617305359

[21] "Pima Indians Diabetes Database, NIDDK," National Institute of Diabetes and Digestive and Kidney Diseases. [Online]. Available: https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database

[22] Z. Omary and F. Mtenzi, "Machine learning approach to identifying the dataset threshold for the performance estimators in supervised learning," *International Journal for Infonomics*, vol. 3, 09 2010.

[23] L. Perez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning," 2017. [Online]. Available: https://arxiv.org/abs/1712.04621

[24] R. Polikar, *Ensemble Learning*. Boston, MA: Springer US, 2012, pp. 1–34. [Online]. Available: https://doi.org/10.1007/978-1-4419-9326-7_1

[25] P. Salza and F. Ferrucci, "Speed up genetic algorithms in the cloud using software containers," *Future Generation Computer Systems*, vol. 92, pp. 276–289, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17324147

[26] P. Salza, E. Hemberg, F. Ferrucci, and U.-M. O'Reilly, "Towards evolutionary machine learning comparison, competition, and collaboration with a multi-cloud platform," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1263–1270. [Online]. Available: https://doi.org/10.1145/3067695.3082474

[27] P. V. Sankar Ganesh and P. Sripriya, "A comparative review of prediction methods for pima indians diabetes dataset," in *Computational Vision and Bio-Inspired Computing*, S. Smys, J. M. R. S. Tavares, V. E. Balas, and A. M. Iliyasu, Eds. Cham: Springer International Publishing, 2020, pp. 735–750.

[28] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, no. 1, p. 60, Jul 2019. [Online]. Available: https://doi.org/10.1186/s40537-019-0197-0

[29] J. Smith, J. Everhart, W. Dickson, W. Knowler, and R. Johannes, "Using the adap learning algorithm to forcast the onset of diabetes mellitus," *Proceedings - Annual Symposium on Computer Applications in Medical Care*, vol. 10, 11 1988.

[30] A. K. Tanwani, J. Afridi, M. Z. Shafiq, and M. Farooq, "Guidelines to select machine learning scheme for classification of biomedical datasets," in *Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics*, C. Pizzuti, M. D. Ritchie, and M. Giacobini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 128–139.

[31] I. Tsang, J. Kwok, and P.-M. Cheung, "Core vector machines: Fast svm training on very large data sets." *Journal of Machine Learning Research*, vol. 6, pp. 363–392, 04 2005.

[32] K. Veeramachaneni, I. Arnaldo, O. Derby, and U.-M. O'Reilly, "Flexgp - cloud-based ensemble learning with genetic programming for large regression problems," *J. Grid Comput.*, vol. 13, pp. 391–407, 2015.

Appendix A

Relevant Script and Template Files

A.1 Kubernetes

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: executor-dep
  labels:
    app: executor
spec:
  replicas: 1
  selector:
    matchLabels:
      app: executor
  template:
    metadata:
      labels:
        app: executor
    spec:
      containers:
      - name: executor
        image: shobuxtreme/ensembler.executor
        ports:
        - containerPort: 9000
        env:
        - name: RABBIT_HOST
          value: "rabbitmq-svc"
        args: ["--spring.rabbitmq.host=$(RABBIT_HOST)"]
---
apiVersion: v1
kind: Service
metadata:
  name: executor-svc
spec:
  type: LoadBalancer
  loadBalancerIP: "34.65.140.150"
  selector:
```

```
    app: executor
  ports:
    - protocol: TCP
      port: 9000
      targetPort: 9000
      nodePort: 30010
```

Listing A.1: k8s executor Deployment

```
#!/bin/sh
# Author : Shubhankar Joshi
# enseMbLer Master Thesis Project

# Deploy enseMbLer core services
kubectl apply -f mongo-secret.yaml
kubectl apply -f mongo.yaml
kubectl apply -f rabbit.yaml
kubectl apply -f datasvc.yaml
kubectl apply -f executor.yaml
kubectl apply -f gui.yaml

# Customize Factorizer
kubectl apply -f factorizer.benchmark.yaml

# Deploy worker
kubectl apply -f worker.adab.yaml
kubectl apply -f worker.deepl.yaml
kubectl apply -f worker.deeplj.yaml
kubectl apply -f worker.deepljs.yaml
kubectl apply -f worker.extratrees.yaml
kubectl apply -f worker.gboost.yaml
kubectl apply -f worker.rforest.yaml

# Deploy l1 workers
kubectl apply -f worker.l1stackxgb.yaml
kubectl apply -f worker.maxvote.yaml
```

Listing A.2: k8s *enseMbLer* Start Script

A.2 Docker Template Files

```
# Ensembler ML-worker template dockerfile
FROM ubuntu

# 2. Inject ensembler base worker msvc
ADD worker-install.sh .
RUN chmod +x /worker-install.sh
RUN /worker-install.sh
WORKDIR /usr/src/ensembler
COPY worker-0.0.1.jar .
ENTRYPOINT ["java","-jar","worker-0.0.1.jar"]

# 4. ----- BEGIN INSTRUCTIONS -----
```

```
13 # Your dockerfile instructions
14 #
15 # 4. ----- END OF INSTRUCTIONS -----
```

Listing A.3: Dockerfile.template

```
1 #!/bin/sh
2
3 # Installs java JRE for worker msvc
4 apt-get update
5 apt-get install -y \
6         openjdk-18-jre
```

Listing A.4: worker-install.sh

Appendix B

enseMbLer Graphical User Inerface

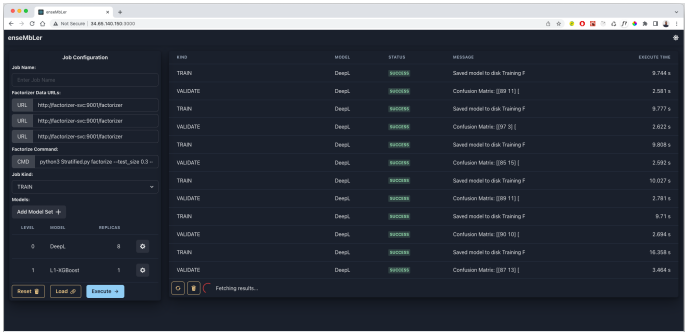

Figure B.1: GUI in dark mode.

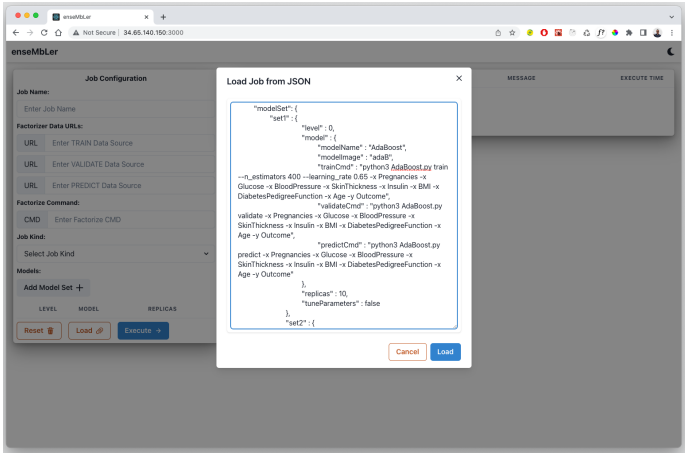

Figure B.2: GUI for loading JSON job configurations.

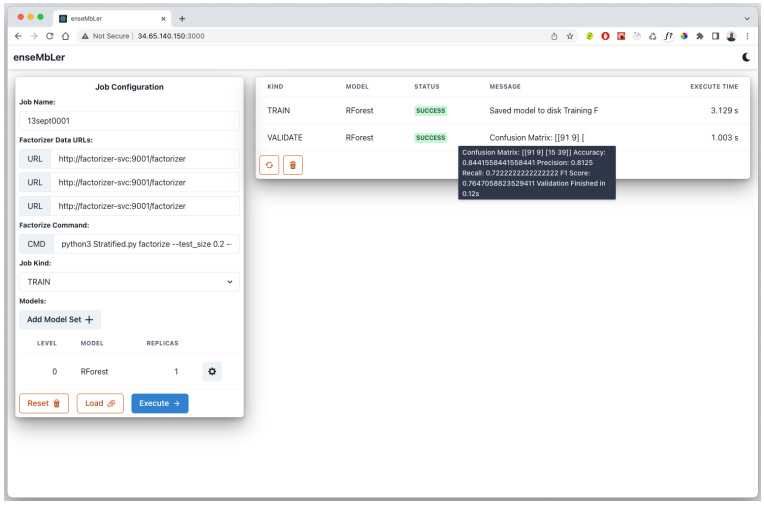

Figure B.3: GUI highlighting further details on mouse hover.

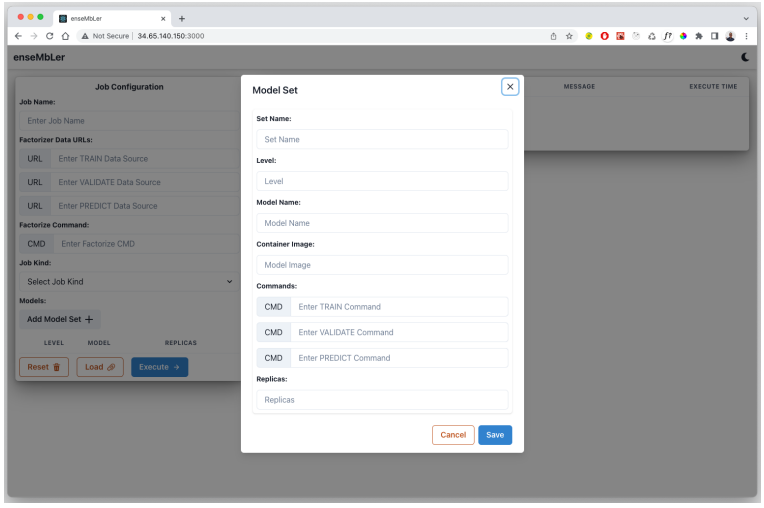

Figure B.4: GUI for adjusting model hyperparameters.