Noname manuscript No.
(will be inserted by the editor)

MLASP: Machine Learning Assisted Capacity Planning

An industrial experience report

Arthur Vitui · Tse-Hsun (Peter) Chen

Received: —/ Accepted: —

Abstract In industrial environments it is critical to find out the capacity of a system and plan for a deployment layout that meets the production traffic demands. The system capacity is influenced by both the performance of the system's constituting components and the physical environment setup. In a large system, the configuration parameters of individual components give the flexibility to developers and load test engineers to tune system performance without changing the source code. However, due to the large search space, estimating the capacity of the system given different configuration values is a challenging and costly process. In this paper, we propose an approach, called *MLASP*, that uses machine learning models to predict the system key performance indicators (i.e., KPIs), such as throughput, given a set of features made off configuration parameter values, including server cluster setup, to help engineers in capacity planning for production environments. Under the same load, we evaluate *MLASP* on two large-scale mission-critical enterprise systems developed by Ericsson and on one open-source system. We find that: 1) *MLASP* can predict the system throughput with a very high accuracy. The difference between the predicted and the actual throughput is less than 1%; and 2) By using only a small subset of the training data (e.g., 3% of the entire data for the open-source system), *MLASP* can still predict the throughput accurately. We also document our experience of successfully integrating the approach into an industrial setting. In summary, this paper highlights the benefits and poten-

Arthur Vitui
Software PErformance, Analysis, and Reliability (SPEAR) Lab, Concordia University, Montreal, Canada
Senior Solution Architect at Ericsson Canada Inc.

Tse-Hsun (Peter) Chen
Software PErformance, Analysis, and Reliability (SPEAR) Lab, Concordia University, Montreal, Canada
E-mail: arthur.vitui@mail.concordia.ca , peterc@encs.concordia.ca

tial of using machine learning models to assist load test engineers in capacity planning.

Keywords Load Testing, Capacity Testing, Performance Testing, Machine Learning, Deep Learning

1 Introduction

Modern large-scale systems, such as telecommunications systems (e.g., Ericsson) and e-commerce websites (e.g., Amazon.com), need to handle millions of concurrent user requests. Any unexpected load-related issues, such as the surge in loads during Black Friday causing the system to crash, may cost these companies millions or even billions of dollars (FastCompany, 2016). Thus, capacity planning, which estimates the amount of load that the system can handle and the expected performance, becomes a critical activity in the quality assurance process of such systems.

To determine system capacity, load testing is a key step in the overall process. The goal of load testing is to ensure that the system behaves correctly under load (e.g., simulated real world usage), and help load test engineers and developers evaluate system performance and capacity (Jiang and Hassan, 2015). Due to the complexity of modern large-scale systems and deployment methods, developers often provide configuration parameters for additional flexibility. Load test engineers would execute the same load against the system under different settings to analyze the system's behaviour and the expected performance (Chen et al., 2017). Developers or load test engineers can change the values of configuration parameters to adjust system performance (e.g., increase the size of thread pools) without changing the source code. When tuning the parameter values, load test engineers also need to consider different deployment (e.g., horizontal scaling by adding more worker nodes) or hardware configuration (e.g., vertical scaling by adding more computation power) settings that the customers employ.

However, it is impossible for load test engineers to cover all the configuration settings in load tests. Large-scale software systems may contain several components and each component may have up to hundreds of configurable parameters and deployment settings (Ha and Zhang, 2019; Li et al., 2018). Due to limited resources and time constraints before releases, it is impossible and impractical to perform load testing on all the possible combinations of configuration parameter values to provide a comprehensive overview of the system capacity (Guo et al., 2013, 2017; Ha and Zhang, 2019). In practice, load test engineers often only test a handful combination of configuration parameter values under the default deployment setting, and their selection process remains ad hoc (Li et al., 2018). For example, load test engineers often use a set of default configuration parameter values supplied by the developers, and may only *marginally* modify some values during load tests. Such configuration tuning is inefficient and largely depend on load test engineers' domain knowledge of the systems, and whether such knowledge is up to date as systems

evolve. Failure to understand the effect of the parameter values on the system capacity under a certain deployment setting may result in load-related issues (e.g., low response time or even crashes), underutilization of the resources, or violation of the service level agreement (Chen et al., 2016; Yin et al., 2011).

Many prior studies (Bao et al., 2018a; Guo et al., 2013, 2017; Ha and Zhang, 2019; Sayyad et al., 2013) have proposed techniques to find the optimal parameter values and maximize performance. However, there are major differences between parameter optimization and capacity planning. Parameter optimization often focuses on finding the optimal performance of a single instance of a system (e.g., on one machine) using a relatively smaller load (e.g., within seconds). Capacity planning, on the other hand, often focuses on finding the system's capacity, such as the key performance indicator (KPI), given a load, system configuration, and deployment setting. Customers or product managers may want to know the system's KPI when more worker nodes are added to the system (e.g., for service level agreement), and what is the system KPI under a specific set of parameter values (e.g., to provide a tuning guideline to customers). Therefore, capacity planning is often used in a larger test environment involving a production-like deployment setting and long running load.

In this paper, we propose an approach, called *MLASP*, that uses machine learning models for capacity planning and prediction. In particular, *MLASP* predicts the system capacity (i.e., throughput) given a set of configuration parameter values (including cluster specific information such as environment and deployment settings). *MLASP* is used to assist load test engineers at Ericsson in capacity planning and load testing. The results from the models also produce empirical responses to customers and developers, who may ask why certain values are being used in a particular configuration setting and to provide a performance estimation calculator (e.g., for service level agreement or system tuning guideline). To train the models, *MLASP* requires load test engineers to run a number of load tests, each with varied configuration parameter values (including cluster information) and under the same load. The set of configuration parameter values and the corresponding throughput are used as training set for the machine learning models. *MLASP* uses a non-intrusive approach to collect the input data to the model by only querying available application programming interfaces (i.e., using JMX or other existing API based interfaces), or through log parsing. Therefore, *MLASP* adds minimal overhead to the system and does not affect the load test results.

We evaluate *MLASP* on two large-scale mission-critical enterprise systems that are developed by Ericsson. The systems handle millions of concurrent users around the world on a daily basis. Due to the non-disclosure agreement (NDA), we can disclose only limited details of the experiment results as well as about what the enterprise systems are. Therefore, we also conduct our experience on an open-source system. For the open-source system, we use Apache Kafka (Apache, 2019), a popular open-source message streaming system. We selected Kafka due to its internal architecture design, as it can scale both vertically and horizontally, which is similar to the systems developed by Ericsson.

However, the Ericsson systems are much more complex and are a conglomerate of different applications. We execute the load tests in all studied systems with varied configuration parameter values. Our experiment results on the open source system show similar trends as the ones uncovered with our industrial partner. We also made the test results from the open-source systems publicly available online (MLASP, 2020).

We evaluate *MLASP* by answering the following research questions:

- **RQ1: What is the prediction accuracy on the system throughput given varied system configuration parameter values?**
  We evaluate the performance of six machine learning algorithms: random forest, XGBoost trees, Multi-Layer-Perceptron (MLP) Neural Networks, Convolutional Neural Networks (CNN), Long Short-Term Memory (LSTM) Neural Networks, and Linear Regression. We find that models such as XGBoost achieve a low mean absolute percentage error (e.g., 1% to 3%) when evaluating the models on a test dataset.
- **RQ2: What is the prediction accuracy by training the models using a small number of test runs with different configuration parameter values?**
  We find that by using a small subset of the training data (e.g., 3% of the data used in RQ1 for the open-source system), the models can still achieve a low mean absolute percentage error, with the best model achieving 1 to 2.7%. We also find that XGBoost gives the most stable results among all the evaluated models when the training data is small.

In summary, the paper makes the following contributions:

- We propose an approach, *MLASP*, to model the throughput of the system given difference sets of configuration parameter values. We use a non-intrusive data collection approach that makes *MLASP* easier to be adopted in practice and has minimal performance impact.
- We evaluate the performance of six machine learning algorithms. We find that algorithms such as XGBoost and Multi-Layer Perceptron give great prediction results, while the results vary for other algorithms. In particular, the results from XGBoost are the most stable among all evaluated algorithms.
- We discuss our experience on integrating *MLASP* in an industrial setting. The prototype of *MLASP* is now used by our industrial partner to assist load test engineers with capacity planning.
- We make our test execution data for the open source system publicly available (MLASP, 2020), which contains 900 test runs with a total machine time of over 18 days.

Paper organization. Section 2 provides motivating examples of our approach. Section 3 describes the studied systems and case study setup. Section 4 describes our methodology including model building and tuning. Section 5 presents the results of our case study by answering two research questions. Section 6 discusses the lessons that we learned and our experience on integrating our approach in an industrial setting. Section 7 discusses the threats to

the validity of our findings. Section 8 surveys related work. Finally, Section 9 concludes the paper.

2 Motivating Examples

In this section, we discuss the challenges that Ericsson faces in the process of production capacity planning by using motivating examples.

Dave is a load test engineer working at Ericsson Inc. Dave's task is to test and verify the capacity of the systems in a short period of time, since Ericsson has a short release cycle (i.e., six weeks[1]). The software development teams at Ericsson rely on Dave's test results to ensure that the new release can be deployed in production and provide an updated service level agreement (SLA) if needed. Hence, delays in Dave's test schedule may delay the release of the systems. However, it is difficult for Dave to run a comprehensive load test within the given time frame due to the complexity of the systems and hundreds of performance-related configurations.

The systems at Ericsson are composed of several components, where each component is maintained by many developers and has millions or hundreds of millions of lines of code. Depending on the situation, components may be grouped together under the same or across several application servers, which may be distributed across multiple different computing nodes, for redundancy and increased capacity (throughput) purposes. In some cases, the redundancy is geographically distributed across different data centres. To increase the flexibility of the systems, each component has various configuration parameters that may affect the systems' performance and capacity (both at single instance and cluster values). Dave needs to estimate the capacity of the system by taking into account both the hardware setting and configuration values to provide a performance guideline that fulfills the service level agreements expected by the customer.

On one hand, Dave does not know how each configuration parameter (or a combination of various configuration parameters) affects the overall system performance. He relies on the default values provided by the development teams. On the other hand, developers may not know what type of hardware (or virtualisation technology) will be used for running the software in production environments. As also discussed in a prior study (Li et al., 2018), developers sometimes may not even know the effects of a combination of configuration parameter values. Therefore, they may provide the values based on guessing and/or previous experience from similar projects. Given the time constraints for testing, Dave cannot afford to tune all the configuration parameters. Dave needs to rely on his experience for testing the most significant parameters. After choosing an initial set of configuration parameter values, Dave needs to further perform endurance testing to ensure that the systems perform normally under load, using the earlier determined configuration parameter values.

[1] https://www.ericsson.com/en/press-releases/2017/9/ericsson-offers-continuous-software-updates

At the same time, throughout the endurance testing, the software system is continuously monitored to verify its capacity (e.g., throughput) when the configuration values change.

A major drawback of Dave's approach is that when the systems are modified (i.e., can be any of the source code, deployment settings, or hardware configuration), the effects of the configuration parameters may also change - a problem known as the n-way feature interaction problem. Rerunning load tests to verify the effect of different combinations of the configuration parameters is very time consuming and costly. In addition, when a new component is added in the system, the number of configuration parameters may increase exponentially. This makes it impossible to test out a sufficiently wide range of values to say with confidence that a given combination of the configuration parameter values leads to a desired capacity that can sustain under heavy load the SLAs. Given the time constraints between releases, Dave could accelerate testing should more computing resources be available to execute tests in parallel on multiple testing environments. However, this approach not only drastically increases the costs of the project by requesting a significant amount of data centre resources, but it also may not be a feasible solution given the time it may take to deploy new instances for every component in the system. The cost of having wrongly, or poorly, configured software may lead to suboptimal resource utilization, or may even lead to enormous losses due to unexpected performance issues. Also, poorly configured software may lead to larger than required deployments, increasing the capital expenditure costs for hardware and electricity, etc. All these points highlight the criticality of proper and efficient capacity planning and configuration analysis for the large scale systems.

In this paper, we propose a machine learning based approach to assist load test engineers, such as Dave, to reduce the number of needed test runs for studying and verifying the capacity of a system. Based on a limited number of test runs, our approach can automatically predict the measured key performance indicators (i.e., throughput), given an unseen set of configuration parameter values. Our goal is to predict the system capacity (i.e., throughput) given the configuration parameter values and the deployment settings. Our approach may help provide recommendations to load test engineers and determine, in a shorter amount of time, a set of configuration parameters that can sustain a desired capacity for the software system. Our approach is now integrated into the testing process of the enterprise systems to help load test engineers with load testing and capacity planning.

3 Case Study Setup

In this section, we present the studied systems as well as the methodology for obtaining the data for evaluating our approach.

3.1 Studied Systems

We conduct our experiment on two sets of studied systems: enterprise and open-source.

Enterprise Systems. We evaluate our methodology on two large-scale enterprise systems built by Ericsson Inc. Due to the non-disclosure agreement (NDA) in place, we cannot give the exact details about the enterprise systems. Nonetheless, the systems have millions of lines of code and are maintained by a large number of software developers. These systems provide Business-2-Business and Business-2-Consumer telecommunication functions related to messaging, location based services, and payments. These systems are integrated to Ericssons in-house developed products, third party commercial products, as well as some open source products. Due to their criticality, the systems are deployed in high redundancy settings, both locally and geographically distributed. These systems process tens of millions of requests on a daily basis, and are used by millions of customers around the world for mission-critical operations. In our study, we consider various deployment settings of the studied systems.

Open-Source System. In order to provide a more detailed discussion on the result and to allow others to replicate and verify our findings, we also perform our experiment on an open-source system — Apache Kafka (Apache, 2019). Kafka is a distributed stream processing system that runs in clusters of one or more servers, also called brokers. A cluster may be formed in the same or across several data centres. A cluster stores streams of records, also called messages. The messages with similar contents are grouped by a "topic". The topics are then divided in partitions. The partitions may be distributed and replicated across any number of brokers within the cluster.

We use Kafka as our open-source studied system because it is a highly configurable system that supports non-intrusive measurement and dynamic configuration changes (i.e., using JMX), and its performance depends on both vertical and horizontal scaling settings. These features of Kafka are similar to the ones existing in our enterprise studied systems, and thus make it a good candidate for conceptual replication of the capacity planning process used by our commercial partner. We implement a system that uses Kafka to exchange messages in the Amazon Web Service (AWS) cloud with three different environment (i.e., deployment) settings:

- One broker with a one-partition topic, without replication.
- One broker with a two-partition topic, without replication.
- Two brokers with a two-partition topic, each broker has one active partition and the replica of the other one (cross replicated partitions).

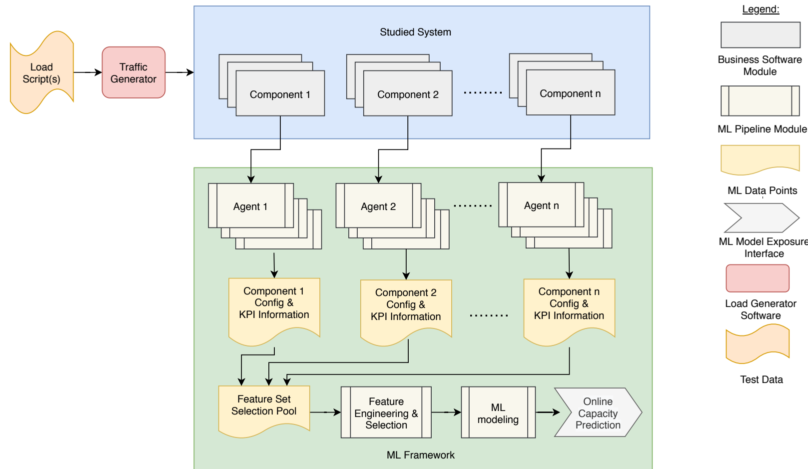

Fig. 1: A high-level overview of our approach and load test execution setup.

3.2 Running Load Tests

Figure 1 shows a high-level overview of the load test execution setup. For each studied system, we use load scripts to generate traffic and exercise the system. Since each system may contain multiple components, we collect the set of configuration parameters and the corresponding throughput in each component (details in section 4.1). Finally, we aggregate the collected information and use it as the input for our machine learning models for capacity planning and prediction. We use non-intrusive data collection mechanisms, which do not require loading additional profiling modules that may increase the system overhead. We implement this type of approach as it has the minimal performance impact, it can be easily automated in continuous integration (CI) pipelines, and it may be adopted and applied to other systems (Li et al., 2018). We rely on out-of-the-box capabilities of the system to provide information or to reconfigure itself either by using specialized APIs (web-based APIs, RESTful or SOAP, or Java Management Extensions - JMX), or periodic reloading of configuration files. Throughput information about the studied software system may also be collected from logs. Depending on the nature of the logging information, expert knowledge may be required in order to perform the assembly of the desired throughput information from logs.

Below, we describe the process of running load tests for the studied systems.

Enterprise Systems. The two enterprise systems are deployed at a large scale. Each of these systems contains several different products that run together to fulfill certain business requirements. Some of the systems offer out-of-the-box capabilities for querying dynamically collected performance metrics (i.e., for calculating the throughput), and others need further analysis to extract the necessary metrics. To cope with the distributed information, we

aggregated the metrics based on timestamps. In large-scale deployments, these different components are required to be highly synchronized. We used the synchronous property of the enterprise systems to extract information from each component at the same synchronized time value intervals. Once the data collection endpoints were in place, the load test engineers execute load tests by following defined in-house process. Due to the nature of the studied systems, the load test engineers use in-house custom scripts to generate and execute the load. The test generation and execution process are integrated as part of the CI pipeline. At the end of each test, we collect and calculate the throughput of the studied systems under the executed load, the given values of the configuration parameters, and the corresponding deployment setting.

Open-Source System. We consider three sets of configuration parameters in our study: environment settings, broker, and load-generator. We executed a total of 900 load tests, each test had the duration of 30 minutes, and 300 tests on each of the three above-mentioned environment settings (Section 3.1). For each environment setting, we vary the Kafka broker configuration parameters as well as the load-driver parameters using the same sequence of values from the selected range applicable to every parameter. Following recommendations from previous academic research focused on Kafka performance (Bao et al., 2018b; Le Noac'h et al., 2017), as well as from commercial support suppliers for Kafka (Cloudera Documentation, 2018; Confluent Blogs, 2017), we select and vary the most important Kafka broker configuration parameters in our tests (Apache, 2019):

- background.threads: The number of threads to use for various background processing tasks. The range of values we used in our experiments for this parameter was [5-30].
- num.io.threads: The number of threads that the server uses for processing requests, which may include disk I/O. The range of values we used in our experiments for this parameter was [4-16].
- num.network.threads: The number of threads that the server uses for receiving requests from the network and sending responses to the network. The range of values we used in our experiments for this parameter was [3-6].
- num.replica.fetchers: The number of fetcher threads used to replicate messages from a source broker. Increasing this value can increase the degree of I/O parallelism in the follower broker. The range of values we used in our experiments for this parameter was [1-2].

Note that the three above-mentioned environment settings concerning the Kafka cluster setup (i.e., the number of brokers and number of partitions per topic) are also part of the configuration parameters. From the load-generator's standpoint, we use message size and the number of client threads publishing messages into Kafka as input variables.

The measured target is the client side throughput in relation with the server side capabilities of the target Kafka environment. In other words, given a certain configuration tuple (i.e., broker, environment, load-generator), we

calculate the throughput as how many messages can be published in Kafka within 30 minutes. The load test results are aggregated per environment and are used for building the machine learning models for capacity prediction, which is described in the next section.

4 Predicting System Throughput Under Different Configuration Parameters

In this section, we describe our modelling approach to predict throughput as the key performance indicator (KPI) of a system under the same load with different configuration parameter values.

4.1 Collecting Data from System Execution

As described in Section 3.2, we run load tests and collect the throughput under different values of configuration parameters. In particular, we follow the steps below for the open-source system:

- We used the load-driver to run tests for a fixed duration (i.e., 30 minutes).
- At the beginning of each test run, we update the configuration of both the load-driver as well as of the Kafka brokers. We generate a total number of 300 distinct configurations.
- We executed load tests using the same 300 distinct configurations on each of the three different Kafka environments (as described in Section 3.1). In total, we executed 900 load tests with a total machine execution time of over 18 days.

We leveraged expert knowledge to decide which configuration parameters to tune on the Kafka broker side for each load test scenario (Bao et al., 2018b; Cloudera Documentation, 2018; Confluent Blogs, 2017; Le Noac'h et al., 2017). The range used to vary each configuration parameter was defined in a list of values described by the function $f(x) = nx$, where both n and x are positive integer numbers. The range contained values both higher and smaller than the default configuration value for the parameter x. The upper and lower bounds of the range (i.e., n) were selected based on domain knowledge for each configuration parameter considered in the test scenarios. As an example, in the open source environment the "BackgroundThreads" parameter values of the Kafka broker were modelled by the function $f(x) = 5x$, where $x \in [1, 6]$, and the 'NumIoThreads' parameter values of the Kafka broker were modelled by the function $f(x) = 4x$, where $x \in [1, 4]$.

We used a custom developed Java based command line application for the load driver. Within the load-driver, we could control the number of client threads used to send messages to the Kafka brokers, as well as the size of the messages being delivered. The load-driver application recorded in a log file the

configuration parameter values as well as the measured throughput (i.e., the KPI for the studied open-source system) for each test run.

On the Kafka broker side, we developed an agent that collects Kafka throughput metrics on a one-minute interval. For each test run, we aggregate the broker's throughput to correlate with the load-driver's recorded throughput for calculating the overall system throughput under a certain set of configuration parameter values. The models source code and test results of the open-source systems can be found online (MLASP, 2020).
Although we cannot disclose the details of the enterprise systems due to NDA, we follow a similar process in running the load tests with different configuration parameter val ues. We use customized scripts for test execution and data collection, which are integrated with the CI process used by Ericsson Inc.

4.2 Feature Engineering and Selection

Before building the machine learning models for capacity prediction, we apply feature engineering and selection techniques to preprocess the collected configuration parameter values and their corresponding throughput. The approaches that we use are similar for both the open-source and the enterprise systems. However, there are multiple KPIs in the enterprise systems whereas we only model the throughput in the open-source system.

Since the magnitude of the values may vary significantly among the configuration parameters (e.g., number of threads vs memory size in megabytes), we apply different scaling techniques to normalize the values. Prior studies also show that applying normalization can improve the performance of machine learning models (Singh et al., 2015; Sola and Sevilla, 1997). We use available scaling methods provided by the scikit-learn (SciKit-Learn, 2019) library, namely StandardScaler and MinMaxScaler techniques. Note that we retrain the models using two different scaling methods separately and only report the model that achieves the best performance. The StandardScaler subtracts the mean and scales the data to unit variance. The MinMaxScaler rescales the dataset so that the entire feature set is in the range [0, 1]. Both scaling methods may be sensitive to outliers. However, this is not a real concern in our experiments since our feature values are the configuration parameter values within the defined range.

Some configuration parameters may be highly correlated (e.g., CPU readings and application transaction per second rate - TPS). A high correlation among the features (i.e., configuration parameters) in machine learning models may result in multicollinearity that affects the stability of the model (e.g., overfitting) (Friedman and Wall, 2005; Harrell, 2006). Hence, we conduct a correlation analysis and remove one of the features if they have a correlation of over 0.8 (Garcia Asuero et al., 2006).

Noteworthy is the aspect that the result of feature selection is highly coupled with a given set of configuration parameters. This means that in the event where new non-constant parameters are added into the system, the model may

no longer provide accurate predictions, as it would not take into account the new parameters. Therefore, the model should be recreated and retrained using the updated feature set.

4.3 Applying Machine Learning Techniques

We apply various machine learning models to predict the throughput of the system given a set of configuration parameter values. In particular, we apply the following models:

Tree-based models:

- Random Forests is an ensemble learning algorithm that is based on constructing multiple decision trees. Random Forest is less sensitive to outliers and can automatically identify important features (Breiman, 2001).
- XGBoost is an optimized library for the Gradient Boosting machine learning method based on decision trees. XGBoost is very efficient for modelling regression problems (Chen and Guestrin, 2016; Montero-Manso et al., 2020; Pan, 2018).

Deep Neural Network models: Different types of neural network may capture different relationships between the features and the target variable (i.e., throughput in our experiment). Thus, we apply various types of neural networks and study how well can each network model the given data.

- Multi-Layer Perceptron (MLP) is a fully connected feed forward neural network that is able to model data that has a nonlinear relationship (Zaccone et al., 2017). MLP optimizes the weights of each neuron to minimize the training error given the input data, and uses the trained network of neurons for prediction.
- Convolutional Neural Networks (CNN) are regularized versions of MLP and automatically consider the hierarchical pattern in the training data (Zaccone et al., 2017). Prior research (Lathuilire et al., 2019) demonstrated that CNNs are very good at modelling regression problems, despite being originally created for image analysis type of problems (Giulli and Pal, 2017).
- Long Short Term Memory (LSTM) (Zaccone et al., 2017) is a recurrent neural network (RNN) variant, which captures and remembers the order of the data in the sequences during the training process (Ergen and Kozat, 2017; Wöllmer et al., 2009).

Traditional model:

- Linear Regression models the linear regression relationship between the features and the response variable (i.e., throughput). We use Linear Regression as a baseline model and compare its prediction performance with other more advanced models.

For each model, we use the same set of training and testing data. We train the model using a subset of results from the test runs, using the configuration parameter values as input features to predict the throughput (Section 5 discusses more on how we separate the training and testing data). We then apply the model on another set of test runs to predict the throughput in tests executed using different configuration parameter values.

Similar to the work by Ha and Zhang (Ha and Zhang, 2019), we devise an iterative method to determine the optimal depth for the neural network that will yield the best performance on the validation subset. We apply this iterative search for network architectures using different widths, such as 32, 64 and 128 neurons per layer respectively. We start the evaluation with no hidden layer and then the depth of the network is increased one layer at time, for a predetermined maximum number of layers (i.e., 15 in our case). We store, for each iteration, the following information:

- The model performance metrics of applying the trained network model on the test data (e.g., R2 score, MAE, etc.).
- The training data and the validation data values for each training iteration. We later visualize the difference between the training and validation history to look for signs of overfitting or underfitting (e.g., the training result is very different from the validation result).
- The predicted throughput for the test data.
- The file storing the details of the trained model (i.e., the file containing the saved model parameters with the best training score). The files allow us to reload the best fully trained model and deploy the model in the CI process.

The network depth expansion process may be either manual or automated. For the manual expansion process, ML model developers may follow an iterative approach to determine the maximum depth of the neural networks or trees. For example, in the first trial, the maximum depth may be set to N=5 and after reviewing the results, developers can decide whether the values are acceptable or more modeling is required, in which case the max depth may increase (e.g., N=10 or 15).

Although the manual process is easy to implement, it may not be the most efficient method from a training time perspective. To optimize the depth expansion, developers may also use an automated approach (Ha and Zhang, 2019), such as continuously increasing the depth as long as the results (i.e., the model performance metrics) increase over a certain threshold (e.g., stop if the result improvement is less than $x\%$ after adding N more layers, or stop if the results are worsened by $y\%$ after adding M more layers). In our experiments, we opted for a manual approach for the open-source system, while we use an automated approach for the enterprise system.

During this iterative process, we also tune other neural network specific parameters:

- Batch size: defines the number of samples (from the training set) the model should work with before adjusting any weights (internal model parameters).

– Epochs: defines the number of times the model will go through the dataset while learning. One epoch is a complete pass (forward/backward) through the entire dataset.
– Regularization values for L1 and L2 type of regularization (Ng, 2004).

Regularization is a process used to prevent overfitting. Regularization works by adding a penalty to all the parameters (except the intercept) with the goal for the model to generalize the data and avoid overfitting. This penalty is added to the loss function used by the model. Prior research (Ng, 2004) recommends verifying the model performance using two standard types of regularization procedures:

– Lasso regression (Tibshirani, 2011), or L1 norm, uses a penalty term so that the sum of the absolute values of the model parameters (weights) is small.
– Ridge regression (Nigam et al., 1999), or L2 norm, uses a penalty term so that the sum of the square of the parameters is small.

For the tree based algorithms, we also perform an iterative approach to determine the number of trees as well as the L1 and L2 regularization values in order to produce a model with better prediction result. We gradually increased the number of trees until we saw little or no more prediction improvements. Similarly, we gradually adjusted the L1 and L2 regularization values.

5 Case Study Results

In this section, we evaluates *MLASP* by answering two research questions. To comply with the non-disclosure agreements with Ericsson Inc, we cannot present the details about those systems, so we present only a brief summary of our findings.

5.1 Training Machine Learning Models

Before presenting the actual results of the experiments, we discuss some of the insights from the data collection process as well as the feature engineering and modelling process.

In our approach, we applied a sequential model hyper-parameter tuning, which may require a longer training time. In particular, we find that it may take a significant amount of time (e.g., hours for the open-source system) to search for optimal model hyper-parameters when training deep neural network models due to their complexity (e.g., contain many parameters that can be optimized). Even though deep neural network models in general give great prediction results when there is more training data (as shown in the results of RQ1), load test engineers may still choose to use algorithms such as XGBoost due to its short training time and good prediction results.

In our experience, as more test data becomes available, the optimal model architecture (e.g., number of neurons per layer in Neural Network models) or hyper-parameter values may change. For example, we find that using our model hyper-parameter tuning approaches, the optimal hyper-parameter values that we found in RQ1 and RQ2 are different. Therefore, in additional to online training, we need to retrain the models periodically to find the new optimal model hyper-parameter values. We also observe similar findings in the enterprise systems, which prompted our industrial collaborators into adding model retraining and hyper-parameter tuning as part of the CI pipeline when the number of new load tests exceed a certain number. By doing so, we find that we can achieve a more accurate prediction result and better capture the relationship between the subject system's configuration parameters and the throughput.

5.2 RQ1: What is the prediction accuracy on the system throughput given varied system configuration parameter values?

Motivation. Due to the complexity of modern large scale systems, there may be hundreds of different combinations of configuration parameter values, and environment and deployment settings. Running load tests to verify the effect of the configuration parameters and find the values that support a desired capacity will take a significant amount of time. Even worse, when developers make changes to the system or add new configuration parameters, load testers would need to re-run the tests, which leads to significant and undesirable increases in project costs. Therefore, being able to estimate the system throughput given a set of configuration parameter values is important for capacity planning and reducing the costs of load testing. In this RQ, we investigate if we can accurately predict the system throughput, given a new set of configuration parameter values, by using machine learning models.

Approach. For our experiments, we train and test the machine learning models using the complete dataset by following the Pareto principle (ALQahtani and Whyte, 2016; Xu and Goodacre, 2018). Namely, we split the available data into training, testing, and validation sets. We first split the data into training and testing sets using the 90% - 10% split. The training set is then split again into actual training and validation sets, following the 80%-20% splitting convention. The validation set is used to tune the model parameters that we discussed in Section 4.3. We report the following metrics for model evaluation:

- **Median Percentage Deviation** is the median of the measured percentage deviation between the predicted and actual target (system throughput) The percentage deviation between the actual and the predicted throughput is calculated as: $\frac{actual - predicted}{actual}$.
- **Mean Absolute Percentage Error** (MAPE), also known as mean absolute percentage deviation (MAPD), is a measure of prediction accuracy of a forecasting method and measures the size of the error in percentage terms.

Table 1: Model evaluation results when using the entire data with separated training and testing. We excluded the MAE, MSE and RMSE for the enterprise systems due to NDA.

System Type	Model	R2 Score	Median (%) Deviation	MAPE	MAE	MSE	RMSE
Open-Src.	XGBoost	0.99964	0.0865 %	0.8060 %	2,193	2.4028e7	4,901
	Random F.	0.99958	-0.1828 %	0.6617 %	2,266	2.8253e7	5,315
	MLP NN	0.99969	0.1648 %	0.6822 %	2,110	2.0594e7	4,538
	CNN	0.99949	0.1590 %	1.1940 %	3,555	3.4482e7	5,872
	LSTM NN	0.99902	0.2583 %	1.4341 %	5,586	6.6280e7	8,141
	Linear Regr.	0.90930	7.3951 %	41.3498 %	66,759	6.1744e9	78,579
Entprz. 1	XGBoost	0.99324	0.6323 %	2.3596 %	—		
	Random F.	0.95078	-0.1450 %	4.5820 %			
	MLP NN	0.99512	2.2929 %	2.3435 %			
	CNN	0.98317	-1.0807 %	3.5182 %			
	LSTM NN	0.92908	-16.0805 %	8.0132 %			
	Linear Regr.	0.98830	4.6561 %	21.3450 %			
Entprz. 2	XGBoost	0.99973	0.8091 %	4.5485 %	—		
	Random F.	0.99647	1.5526 %	67.5510 %			
	MLP NN	0.99991	1.0868 %	3.3082 %			
	CNN	0.99067	-8.1808 %	23.6534 %			
	LSTM NN	0.97589	-5.9882 %	62.2091 %			
	Linear Regr.	0.92218	4.3187 %	10.4637 %			

It is calculated by the formula: $MAPE = \frac{1}{n}\sum_{i=1}^{n} \mid \frac{actual_i - predicted_i}{actual_i} \mid$, given n points in the testing set.

- **Mean Absolute Error** (MAE) is the average of the absolute errors, which is the difference between the measured/predicted value and the actual value and it is defined by the following formula: $MAE = \frac{1}{n}\sum_{i=1}^{n} \mid actual_i - predicted_i \mid$, given n points in the testing set.
- **Mean Squared Error** (MSE) measures the squared average distance between the real data and the predicted data and it is defined by the following formula: $MSE = \frac{1}{n}\sum_{i=1}^{n}(actual_i - predicted_i)^2$, given n points in the testing set.
- **Root Mean Squared Error** (RMSE) is the square root of the mean squared error, thus defined as: $RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(actual_i - predicted_i)^2}$, given n points in the testing set.
- **R2 score**, or the coefficient of determination is a statistical measure of how close the data are to the fitted regression line and it is defined by the following formula: $R^2 = \frac{Variance_explained_by_the_model}{Total_variance} = \frac{SS_{tot} - SS_{res}}{SS_{tot}}$, where $SS_{tot} = sum_of_total_squares$ and $SS_{res} = sum_of_squares_of_residuals$ (i.e., the unexplained variation).

Results. ***The ML models can predict the throughput with a high accuracy. In particular, XGBoost and MLP Neural Network achieve the best prediction result and the findings are consist across the studied systems.*** Table 1 shows the throughput prediction results. We find

Table 2: Mean of model evaluation results when using the entire data with a 10-fold cross-validation. We excluded the MAE, MSE and RMSE for the enterprise systems due to NDA.

System Type	Model	R2 Score	Median (%) Deviation	MAPE	MAE	MSE	RMSE
Open-Src.	XGBoost	0.99967	0.0473 %	0.9716 %	2,360	2.2792e7	4,742
	Random F.	0.99963	0.0789 %	0.7673 %	2,295	2.5318e7	5,004
	MLP NN	0.99921	0.4544 %	1.8316 %	4,250	5.4934e7	7,320
	CNN	0.99181	-0.6284 %	6.8631 %	14,891	5.7416e8	22,665
	LSTM NN	0.99759	0.4520 %	1.9627 %	7,286	1.7208e8	11,423
	Linear Regr.	0.91294	4.2547 %	42.1498 %	64,867	6.0917e9	77,905
Entprz. 1	XGBoost	0.96281	1.4391 %	3.7316 %			
	Random F.	0.92250	-1.5916 %	7.0361 %		—	
	MLP NN	0.95681	3.3302 %	3.0288 %			
	CNN	0.91407	1.6746 %	5.8063 %			
	LSTM NN	0.90522	-18.5884 %	10.0907 %			
	Linear Regr.	0.91538	4.5912 %	19.1878 %			
Entprz. 2	XGBoost	0.94997	2.2235 %	8.1909 %			
	Random F.	0.90843	-1.3599 %	57.8177 %		—	
	MLP NN	0.95196	-3.4797 %	6.9008 %			
	CNN	0.91448	9.1220 %	33.5975 %			
	LSTM NN	0.93642	4.1062 %	69.3470 %			
	Linear Regr.	0.85118	4.5878 %	27.0634 %			

that the MAPE values are low (0.68% to 4.5%) for models such as XGBoost and MLP Neural Network, suggesting that the prediction results have a high accuracy. However, for other models such as random forest, CNN, LSTM, and linear regression, the MAPE values have a high variation in the studied systems (goes up to 62%). The finding suggests that some ML models may not be able to capture the relationship between the configuration parameter values and throughput in all studied systems.

The MAE for the open-source system shows the mean absolute difference between the actual and predicted total number of sent messages during the 30 minutes test interval. The MAEs for the open-source system range from 2,200 to 5,600 for most ML models (except for the linear regression model where the MAE is 66,759). Given that the number of sent messages ranges from 100K to 700K (as shown in Figure 2 and Figure 3), a MAE of around 2,000 to 5,000 is considered very small. Although we cannot disclose the MAE values for the enterprise systems, similar to the open-source system, the values are also very low for models such as XGBoost and MLP but higher for other models. One observation is that MLP Neural Network and XGBoost achieve both the minimal MAE and MAPE compared to all other models. The findings indicate that MLP Neural Network and XGBoost may be better at capturing the relationship between the configuration parameter values and the system throughput. As an example, Figure 2 and Figure 3 show the predicted vs actual throughput across various test runs from the MLP Neural Net model of the

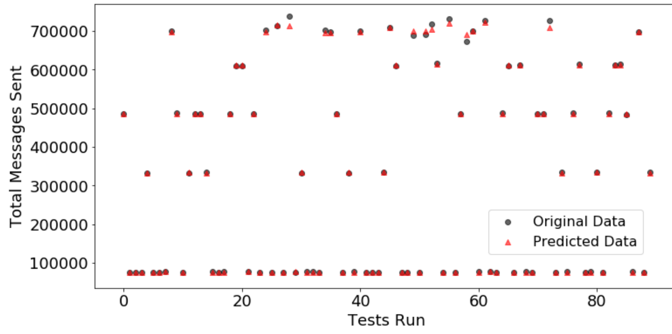

Fig. 2: An example of the predicted v.s. actual throughput for the open-source system from the MLP Neural Net model results built using the complete dataset.

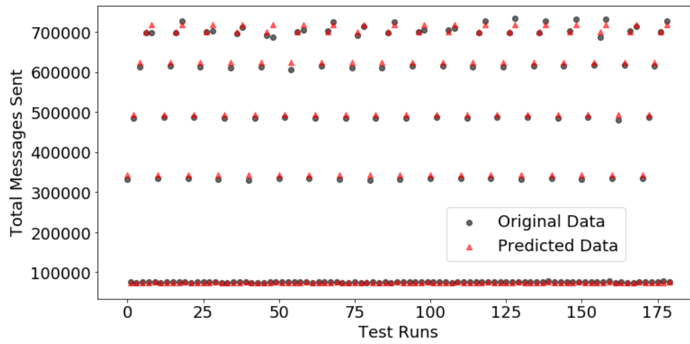

Fig. 3: An example of the predicted v.s. actual throughput for the open-source system from the MLP Neural Net model results built using a subset of the complete dataset.

open-source system. The results show that there is a very high overlap between the predicted and actual throughput (i.e., there is a near-perfect alignment between the two types of dots in the figure).

To better understand the models performance and stability (meaning the measured deviations between the actual throughput values and the predicted ones), Table 1 shows several model evaluation metrics including the median value of the percentage deviation between the predicted and the actual throughput of the open-source systems. Figure 4 further shows the distribution (i.e., each point is the deviation value for one load test result). The results show that the percentage deviation is small for all the models in the open-source systems, except for linear regression models. The finding indicates that, in the open source system, the model prediction is consistent across various load tests given different sets of configuration parameter values. We exclude the detailed distribution for the enterprise systems due to NDA. However, as shown in Ta-

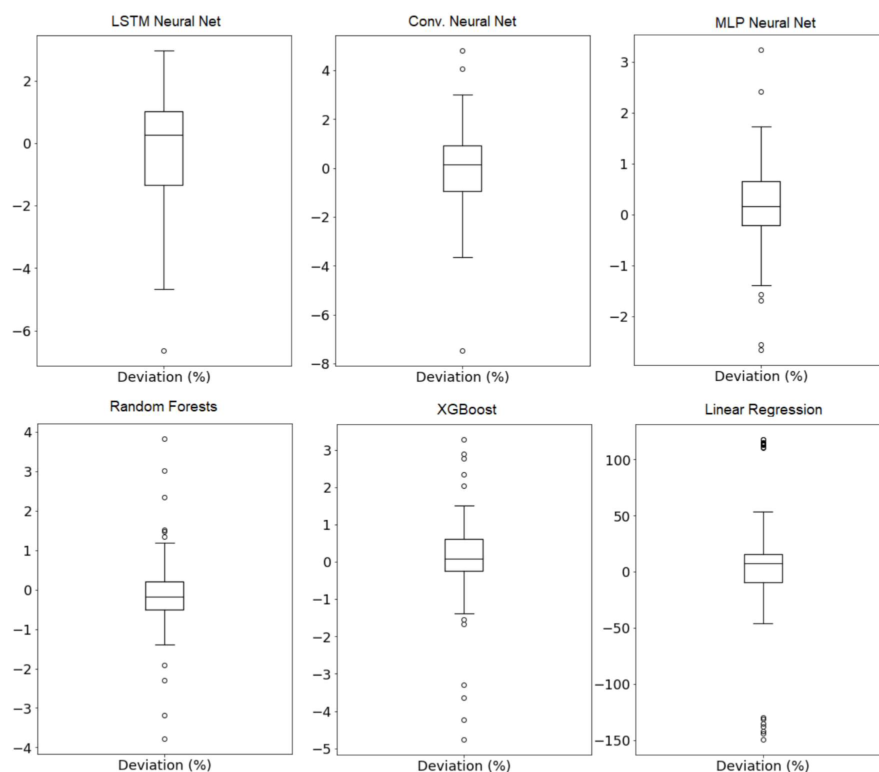

Fig. 4: Distributions of the percentage deviation between the predicted and actual value for the open-source system using the complete dataset with separated training and testing dataset.

ble 1, the median value of the percentage deviation is higher for models such as CNN, LSTM, and linear regression. Our findings show that CNN, LSTM, and linear regression may result in a higher deviation between the predicted and the actual throughput. In contrast, XGBoost and MLP Neural Net have the highest R2 and relatively low deviation in the prediction results.

To further study if our models suffer from any overfitting, we conduct a 10-fold cross validation. Table 2 shows the results of the mean of the metrics obtained after applying a 10-fold cross-validation (following the same data splitting and training process). Our findings show that using 10-fold cross validation, the models show similar prediction results as before. Figure 5 shows an example of the training v.s. validation loss for the MLP model obtained after training on the training set. The training and the validation loss converges, which shows that the model is not suffering from overfitting.

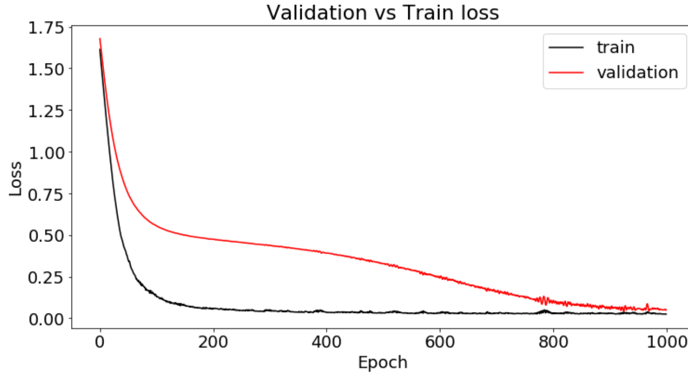

Fig. 5: An example of train v.s. validation loss for the open-source system from the MLP Neural Net model results built using a subset of the complete dataset.

> Our prediction models can predict the throughput of a system given a set of configuration parameter values with a very high accuracy. In particular, ML models such as XGBoost and MLP achieve the best prediction results, with a MAPE value that ranges from 0.81% to 8.2% and a low median percentage deviation. However, other ML models, such as linear regression, CNN, and LSTM have a lower prediction accuracy and a higher variation in the prediction results across the studied systems.

5.3 RQ2: What is the prediction accuracy by training the models using a small number of test runs with different configuration parameter values?

Motivation. As discussed in Section 2, running load tests can be resource intensive. Load testers may not be able to run a large number of tests with a wide range of configuration parameter values to build a capacity prediction model. Therefore, in this RQ, we investigate if we accurately predict the system throughput using the machine learning models that are trained using a smaller number of test runs. The results would give an insight on the test execution time that may be saved when using our approach to predict the system throughput given different configuration parameter values.

Approach. For the open-source system, we use a small subset (i.e., 30 test results) out of the total 900 test results. We randomly selected these 30 test results to build the models, and evaluated the models on several hundred of randomly selected test results from the remaining test results. We follow a similar process for the enterprise systems and use only a small subset of the test results to build models. To evaluate the models, the same as RQ1, we report the R2, MAE, and the percentage deviation between the predicted and actual throughput.

Table 3: Model evaluation results when using **a subset of the entire data** with separated training and testing. We excluded the MAE, MSE and RMSE for the enterprise systems due to NDA.

System Type	Model	R2 Score	Median (%) Deviation	MAPE	MAE	MSE	RMSE
Open-Src.	XGBoost	0.99889	0.5694 %	1.0672 %	4,154	7.8966e7	8,886
	Random F.	0.99893	0.5315 %	1.4713 %	5,539	7.6122e7	8,724
	MLP NN	0.99953	2.1227 %	2.6546 %	4,392	3.3086e7	5,752
	CNN	0.96168	-2.2649 %	15.5894 %	29,567	2.7289e9	52,239
	LSTM NN	0.96830	-13.0955 %	14.7153 %	37,675	2.1557e9	46,430
	Linear Regr.	0.89828	10.4876 %	36.4431 %	67,751	7.2433e9	85,108
Entprz. 1	XGBoost	0.99159	0.0262 %	2.7578 %		—	
	Random F.	0.94208	-0.6794 %	5.0949 %			
	MLP NN	0.98547	1.5053 %	5.7127 %			
	CNN	0.98269	1.3599 %	4.5767 %			
	LSTM NN	0.90235	6.5161 %	17.8877 %			
	Linear Regr.	0.96475	8.4154 %	14.4674 %			
Entprz. 2	XGBoost	0.99954	-1.0259 %	1.7781 %		—	
	Random F.	0.99367	-1.2040 %	7.8301 %			
	MLP NN	0.99192	7.1343 %	13.3365 %			
	CNN	0.94616	-4.2667 %	36.1945 %			
	LSTM NN	0.95059	-13.1491 %	13.9580 %			
	Linear Regr.	0.96084	2.5049 %	8.18635 %			

Results. ***When training the models using only a subset of the test results, XGBoost achieves the best prediction results with the lowest variability compared to other ML models.*** Table 3 shows the prediction results when using a subset of the test results to train the model. We find that, in general, the prediction accuracy has decreased compared to using the entire dataset. The MAE values for the open-source systems have increased to the range of 4,154 to 64,867, compared to 2,110 to 67,751 in RQ1. Similarly, the MAPE values for the open source systems range between 1.06% to 36.44% depending on the algorithm used to solve the regression problem, where lower values suggest a high accuracy of the predictions. In particular, XGBoost achieves a MAPE of 1.1% to 2.8%, which shows that the prediction result is comparable to when training the models using the entire data. Our findings show that even though the prediction performance has generally decreased, the models can still achieve good prediction results.

We find that some algorithms may be less stable (i.e., have a higher percentage deviation) compared to others (Table 3). In particular, LSTM models are the least stable and have the highest median percentage deviation across all studied systems (-13% to 6.5%). On the other hand, as shown in Figure 6, there are some test results in the open source system, where there is a very high percentage deviation between the predicted and the actual throughput (i.e., outliers in the box plot). Overall, XGBoost have the lowest MAPE and percentage deviation in the prediction results. Among all the models, XGBoost

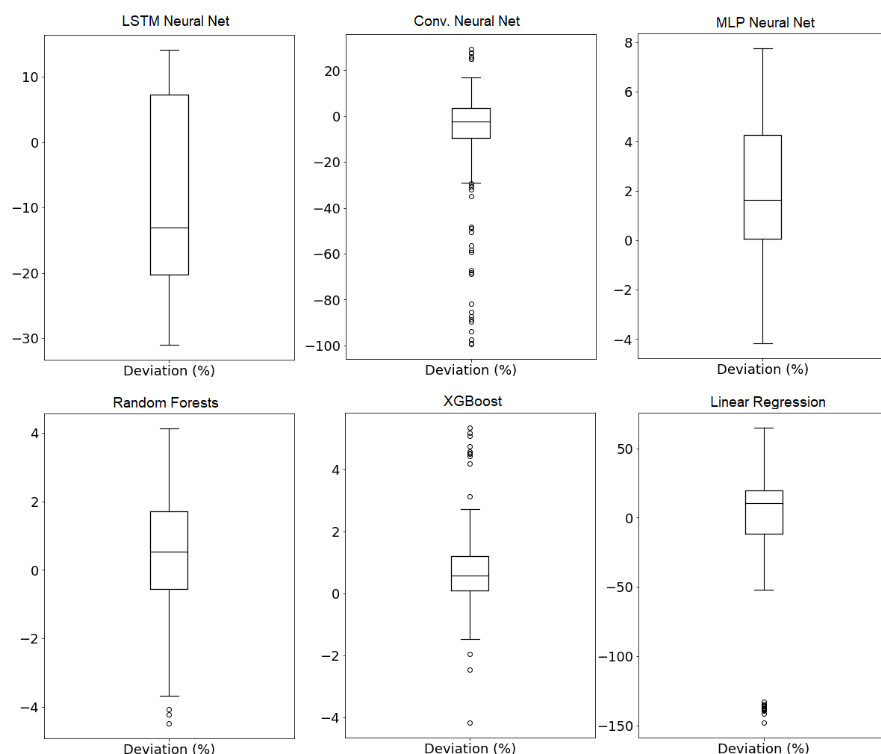

Fig. 6: Distributions of the percentage deviation between the predicted and actual value for the open-source system using a subset of data to train the models.

achieves the best results when using a subset of test data to train the models. Table 4 shows the results of the mean of the evaluation metrics when applying a leave-one-out cross-validation on the subset dataset. We find that, for some models (e.g., LSTM) the MAPE obtained in leave-one-out is very different from using a subset of training and testing. The findings show that, due to the smaller training and testing data, some models may have some issues of overfitting (i.e., different runs achieve varied results in leave-one-out). Nevertheless, XGBoost models achieve very similar results when using a separate training and testing dataset, and leave-one-out. Our finding shows that XGBoost models give the best prediction results and are the most stable when using a subset of data for training. Future studies and practitioners may choose XGBoost models for capacity prediction if there is only limited training data.

Table 4: Mean of model evaluation results when using **a subset of the entire data** with separated training and testing and leave-one-out cross-validation technique on the training dataset. We excluded the MAE, MSE and RMSE for the enterprise systems due to NDA.

System Type	Model	R2 Score	Median (%) Deviation	MAPE	MAE	MSE	RMSE
Open-Src.	XGBoost	0.99883	0.5632 %	1.0795 %	4,242	8.3032e7	9,092
	Random F.	0.99877	0.7213 %	1.5241 %	5,857	8.7052e7	9,252
	MLP NN	0.99660	-4.0275 %	7.8962 %	11,654	2.4191e8	14,454
	CNN	0.88445	-4.8323 %	23.8084 %	51,175	8.2286e9	87,672
	LSTM NN	0.83048	-33.5486 %	47.4998 %	73,846	12.149e9	90,930
	Linear Regr.	0.89742	9.2974 %	36.4756 %	67,810	7.3047e9	85,466
Entprz. 1	XGBoost	0.96354	0.0879 %	3.6777 %			
	Random F.	0.89339	0.7440 %	7.3675 %			
	MLP NN	0.91848	1.9285 %	5.7111 %		—	
	CNN	0.88238	2.6036 %	7.0554 %			
	LSTM NN	0.89979	7.2277 %	25.8928 %			
	Linear Regr.	0.86504	9.6058 %	21.4257 %			
Entprz. 2	XGBoost	0.98516	0.5902 %	2.0413 %			
	Random F.	0.94357	0.7911 %	13.3980 %			
	MLP NN	0.92409	8.1647 %	19.3214 %		—	
	CNN	0.88860	-6.4085 %	48.8870 %			
	LSTM NN	0.92140	-15.6541 %	21.1542 %			
	Linear Regr.	0.82025	3.0043 %	11.4730 %			

MLASP can still achieve great prediction results on the throughput when trained using a small subset of data (i.e., 3% of the open source data and a subset of the enterprise systems data). We also find that XGBoost gives the best prediction results (MAPE between 1% to 3% for all studied systems) with the most stable performance when the training data is smaller.

6 Discussions

In this section, we discuss the lessons that we learned from conducting the experiments and integrating the approach in the enterprise systems.

6.1 Understanding and Verifying the Effect of Configuration Parameters

As also discussed in prior research (Li et al., 2018), we find that developers may not know the effect of a configuration parameter once the system is deployed in a distributed and complex setting. As a result, load test engineers need to spend a significant amount of time testing the system under different configuration parameter values, which can be costly or even impossible. We find that by building the models, we could understand the importance of a

certain feature (i.e., configuration parameter) in relation to the target variable (i.e., KPI), and detect upper boundaries that would produce the same results in the target (i.e., continue to increase the values would not have a significant effect on the KPI). For example, in the open-source system, we see that the throughput remains unchanged, once the number of background threads in a Kafka broker reaches a certain value. After examining the feature importance in the model, we find that the combinational effect of the configuration parameter is more important than a single configuration parameter. In addition to the number of threads, the model results also show that the message size is also an important feature. Our investigation finds that the overall input/output operations per second (IOPS) of the disk and network will greatly influence the total number of messages the system can process, regardless of how high we go with the number of background threads. By using *MLASP*, we can model the combinational effect of the configuration parameters and environment/deployment settings by running fewer load tests, and provide more suggestions to load test engineers and developers on the system capacity.

We found similar cases applicable for the enterprise systems. Namely, *MLASP* helps not only in the knowledge gain about the importance of the configuration parameters but also provide strong evidence on fine-tuning the scaling strategy. For example, knowing the IOPS limits and requirements for a certain system helps virtual machine allocation and relocation strategy for multi-tenant hardware inside the private cloud and data centre. The models can also help in the test automation procedure by reducing the number of tests that involve tuning/verifying less important configuration parameters, leading to an overall reduction of the load testing time. Overall, we find that *MLASP* can help with significant cost savings, and can also help make the CI process more streamlined and ensure more efficient load testing activity.

6.2 Integrating *MLASP* in Industrial Settings

There are different approaches to integrate *MLASP* in industrial settings, such as modifying the source code to include the self-monitoring and self-tuning functionalities. In the end, *MLASP* uses a non-intrusive approach for the data collection procedure and configuration parameter tuning. As discussed in Section 3, we calculate system throughput by leveraging readily available information. Our approach ensures that we need no additional changes on the system source code, and does not affect the testing activities that are part of the software development process of the enterprise systems. This non-intrusive approach was highly regarded by the project management team since it avoided adding additional cost in the project. This in turn avoided jeopardizing timely delivery plans and release roadmaps, which ultimately led to the integration of *MLASP*. Future studies may consider such a non-intrusive approach to increase the adoption of the developed approaches.

In addition to the great prediction results, *MLASP* also helps load test engineers identify the combination of configuration parameter values that yield

a certain output. When integrating our approach with the industrial system, we discussed with our industrial partner (with project management and with technical personnel) on whether the models can be used for capacity planning, in addition to capacity prediction. For example, our industrial partner may be interested in knowing what would be the expected throughput if the number of deployed nodes decreases by two. If the model is able to make a good prediction result, by giving the trained models a set of variable values that one wishes to inquire about, the model can give an accurate estimated outcome. Namely, *MLASP* helps load test engineers model the scalability of the software system (e.g., the types and amount of resources that are needed to achieve a desired throughput, using a forecast on future traffic needs). Such inverse prediction/classification is commonly used in statistics and machine learning to understand the effect of the variable and help make business decisions (Aggarwal et al., 2010; Chen et al., 2012; Li et al., 2018). *MLASP* also helps load test engineers identify possible physical limitations of the system hardware by studying the correlation between the features (i.e., configuration parameters) and the throughput, as discussed in Section 6.1. This information was also very helpful for reporting purposes to both management and development teams when simulating "what-if" scenarios. The results from these simulations pointed out the relationship between different parameters (underlying infrastructure and software configurations) and provided important insight to development teams on where improvement efforts could be focused in the next iteration. Project management also had a clearer perception of the roadmap and were able to create risk mitigation plans much faster (e.g. when to order additional hardware, and when and what communications should be given to the customer with respect to the roadmap).

Given the high interest from our industrial partner, we also ported this tool to use it with the open source system. We added the ported code to the public GitHub repository, sharing the data and algorithms (MLASP, 2020).

During our integration process, we also provided enhancements to the existing load testing process by improving automation. In particular, we integrated the earlier-mentioned random configuration generator tool and the machine learning model training capability. Due to the NDA, we cannot disclose which tools are used in each phase by our industrial partner. Nevertheless, we provide a general depiction of the overall process and discuss a wide range of options for tools in each stage. The process diagram is presented in Figure 7.

The automated load testing portion is controlled by a random configuration generator which may be a script or an application written in any programming language. Alternatively, configurations may be retrieved from a persistent storage system that may be a database (e.g., MySQL, PostgreSQL, or NoSQL databases) or a file versioning system. The configurations are pushed by specialized tools (e.g., Ansible, Puppet, or Chef) or automated scripts to the controlled load driver and the test systems. The load is generated by using either custom developed applications or specialized tools (e.g., JMeter, Jenkins, or Soap-UI). Data is then collected in a non-intrusive fashion to a central location using specialized tools (e.g., Prometheus, InfluxDB, Logstash,

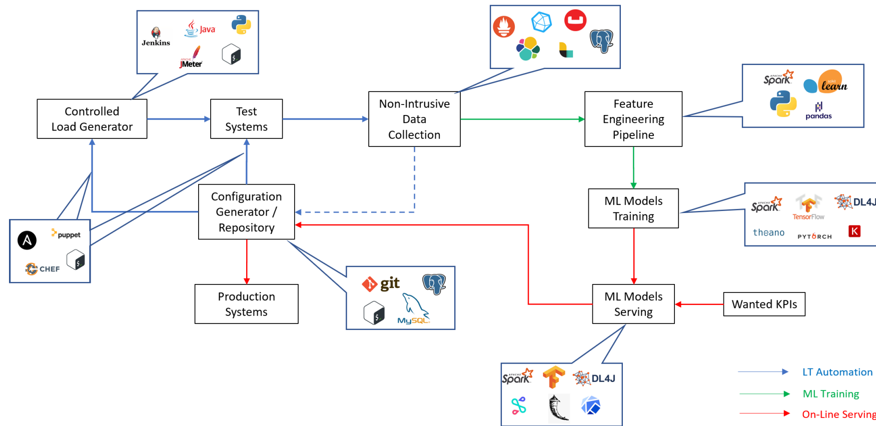

Fig. 7: MLASP Complete Process with its three stages: (1) Automated Load Testing, (2) Feature Engineering and ML Training, and (3) Model On-Line Serving.

or Elasticsearch). The aggregations may be permanently stored in databases. The feature engineering pipeline shall extract and preprocess the data collected from the data lake. The preprocessed data will be used for training the machine learning models. Various tools are available for data collection, pre-processing, and model training: Apache Spark, DeepLearning4J, keras, scikit-learn, pandas, pytorch, tensorflow, theano, etc. Once a model is trained, the model may be deployed using frameworks such as Apache Spark, DeepLearning4J, TensorServing, Seldon, Flask, KFServing, etc.

7 Threats to Validity

Internal validity. Since the enterprise systems are continuously evolving, changes in the source code may affect the throughput of a system. Therefore, we tried to use the same release of the system throughout the process to minimize the effect of new code changes. We did not use any system KPI metrics for the open-source system (e.g., CPU), as both the underlying hardware and virtual servers capabilities may experience some noises (e.g., context switch or garbage collection overhead). Therefore, we choose to model the KPI that may better reflect users' perception of a system (i.e., throughput).

External validity. We conducted our experiments on both open-source and enterprise systems. In the end, *MLASP* is well received by our industrial partner and the prototype is integrated with the software development process. Although we find that the results are similar for the studied systems, our results may not generalize to other systems.

Construct validity. As found by prior studies (Ha and Zhang, 2019; Ng, 2004), parameters in machine learning models may affect the model performance. Therefore, in our experiments, we applied semi-automated analysis to tune a wide range of model parameters using a validation set. Other possible issues with machine learning models are multicollinearity and overfitting. To mitigate the risk, we split the data into training, validation, and testing set to avoid training biases. We also applied different regularization methods to reduce the possibility of overfitting. Our prediction results on external test datasets show that our prediction results are still very good and are similar to the results of the models when applied on the validation set. Thus, the models are not suffering from the problem of overfitting.

8 Related work

Tuning configuration parameter values for software systems used in open-source and commercial solutions has been of great interest in the research community over the years. Previous research can be grouped in the following categories (Bao et al., 2018a):

- Analytical optimizations - using mathematical models to calculate the effect of configuration parameters, often used in the early development cycles of a software system (Chen et al., 2016; Sayyad et al., 2013).
- Measurement based configuration optimization - relying on statistical approaches (Guo et al., 2013, 2017).
- Search based configuration optimization - black box optimization problem using search algorithms (Li et al., 2018).
- Learning based configuration optimization - make use of various machine learning techniques (Bao et al., 2018a; Ha and Zhang, 2019; Sayyad et al., 2013).

Prior studies by Guo et al. (Guo et al., 2013, 2017) use statistical learning approaches to predict system performance given a random sample of various sets of system configuration parameter values. However, these methods focus on tuning binary system configuration parameters. With recent advances in Neural Networks, some of earlier state of the art approaches have been tested against more modern algorithms. Ha and Zhang (Ha and Zhang, 2019) have proposed a novel approach, called DeepPerf, for performance prediction using Deep Sparse Neural Networks. Alongside the novel approach, which includes a strategy for hyper-parameter search, they also compare the performance of their approach with existing state of the art methodologies on 11 open-source systems. The proposed method outperforms other existing approaches. In our paper, we focus on the aspects of large-scale load testing and the predicted capacity of the system which is formed of many different components, given a set of configuration parameter values. Similar to DeepPerf, our configuration parameters include both numeric and binary. In contrast with our work, DeepPerf, focuses on finding optimal configuration parameter values for a single instance of a component, where we focus on predicting system capacity at

a larger scale (e.g., our tests may take hours to run). We also consider various environment and deployment settings as part of the configuration. More importantly, we conducted our experiment in an industrial setting, and reported our experience on adopting *MLASP* in practice, instead of focusing only on open-source systems. It is noteworthy to mention that in an industrial setting the source code of the software systems is not always available. In addition, profiling or sampling source code execution is not possible due to significant performance overhead. Although design documents are sometimes available, the accent is on system integration aspects (e.g., block diagrams). Lower level design documents (i.e., software design documents) are often not available. Considering these aspects, a white-box approach is not always possible, or possible only to a certain extent.

Other work focused on determining the selection of optimal configuration parameter values. Sayyad et al. (Sayyad et al., 2013) proposed an approach called IBEA (Indicator-Based Evolutionary Algorithm) for finding the optimum configuration models for very large software systems with thousands of parameters (e.g. the Linux kernel). They used heuristics to determine a subset of configuration affecting a part of a system. Chen et al. (Chen et al., 2016) analyze logs to uncover system execution. They model the execution using Petri net to recommend caching configuration. Bao et al. (Bao et al., 2018a) proposed an approach called AutoConfig for automated configuration of distributed message systems (DMS). Although their work is centred around DMS such as Apache Kafka (Apache, 2019) and RabbitMQ (Rabbit MQ, 2020), the algorithm may be extended for other systems. Li et al. (Li et al., 2018) share their experience on working with an industrial partner to include autonomic computing capabilities to reduce human intervention on performance configuration tuning. Their approach can find the optimal configuration parameter values dynamically in real-time. Different from prior studies, our approach consider the entire set of configuration parameters and we focus on assisting load testers with capacity prediction, rather than finding optimal configuration for every component making the software system. In addition, we also propose a blueprint for test pipelines integration.

9 Conclusions

The goals of load testing are to ensure that the system behaves correctly under load (e.g., simulated real world usage) and help load test engineers and developers evaluate system performance profile (i.e., capacity). As a result, an important task for load test engineers is to understand the system capacity under different configuration settings. However, there may be hundreds of different configuration parameters in a large-scale system, which makes it impossible for load test engineers to cover all combinations of the configuration parameter values. In this paper, we propose an approach, called *MLASP*, to help load test engineers determine the key performance indicators (KPIs), such as throughput, given a set of configuration parameter values and en-

vironment settings. *MLASP* leverages machine learning models and use the values of configuration parameters as input features and predict the expected throughput, used in capacity planning for production environments. *MLASP* uses a non-instructive approach to collect data from the systems, which in turn helps with the adoption of the *MLASP* in practice. We evaluated *MLASP* on two large-scale enterprise systems developed by Ericsson Inc., and one popular open-source system (Apache Kafka). We find that *MLASP* can predict the system throughput with a very high accuracy (with R2 values above 0.9 and the percentage deviation between the predicted and the actual throughput value is less than 1%). We also find that by using only a small subset of data (e.g., 3% of the open-source test results), we can still achieve a great prediction accuracy. Our approach is well received by our industry partner and the prototype is being integrated in an industrial setting. We also document our experience on applying *MLASP* at Ericsson and the lessons that we learned.

Acknowledgement

We want to thank Ericsson for providing access to the enterprise systems that we used in our case study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of Ericsson and/or its subsidiaries and affiliation. Our results do not in any way reflect the quality of Ericsson's products.

References

Aggarwal C, Chen C, Han J (2010) The inverse classification problem. Journal of Computer Science and Technology 25:458–468

ALQahtani AH, Whyte A (2016) Estimation of life-cycle costs of buildings: regression vs artificial neural network

Apache (2019) Apache kafka - a distributed streaming platform. Https://kafka.apache.org/

Bao L, Liu X, Xu Z, Fang B (2018a) Autoconfig: Automatic configuration tuning for distributed message systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018

Bao L, Liu X, Xu Z, Fang B (2018b) Autoconfig: automatic configuration tuning for distributed message systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, pp 29–40

Breiman L (2001) Random forests. Machine learning 45(1):5–32

Chen T, Guestrin C (2016) Xgboost: A scalable tree boosting system. In: Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, pp 785–794

Chen TH, Thomas SW, Nagappan M, Hassan AE (2012) Explaining software defects using topic models. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, MSR '12, p 189198

Chen TH, Shang W, Hassan AE, Nasser M, Flora P (2016) Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, p 666677

Chen TH, Syer MD, Shang W, Jiang ZM, Hassan AE, Nasser M, Flora P (2017) Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp 243–252

Cloudera Documentation (2018) Configuring apache kafka for performance and resource management. Https://docs.cloudera.com/documentation/kafka/latest/topics/kafka_performance.html

Confluent Blogs (2017) Optimizing your apache kafka deployment. Https://www.confluent.io/blog/optimizing-apache-kafka-deployment/

Ergen T, Kozat SS (2017) Online training of lstm networks in distributed systems for variable length data sequences. IEEE transactions on neural networks and learning systems 29(10):5159–5165

FastCompany (2016) How one second could cost Amazon 1.6 billion sales. http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales, last accessed March 3 2016.

Friedman L, Wall M (2005) Graphical views of suppression and multicollinearity in multiple linear regression. The American Statistician 59:127–136, DOI 10.1198/000313005X41337

Garcia Asuero A, Sayago A, Gonzalez G (2006) The correlation coefficient: An overview. Critical Reviews in Analytical Chemistry - CRIT REV ANAL CHEM 36:41–59, DOI 10.1080/10408340500526766

Giulli A, Pal S (2017) Deep Learning with Keras. Packt Publishing Ltd.

Guo J, Czarnecki K, Apel S, Siegmund N, Wasowski A (2013) Variability-aware performance prediction: A statistical learning approach. 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE) pp 301–311

Guo J, Yang D, Siegmund N, Apel S, Sarkar A, Valov P, Czarnecki K, Wasowski A, Yu H (2017) Data-efficient performance learning for configurable systems. Empirical Software Engineering 23:1826–1867

Ha H, Zhang H (2019) Deepperf: Performance prediction for configurable software with deep sparse neural network. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 19, pp 1095–1106

Harrell FE (2006) Regression Modeling Strategies. Springer-Verlag, Berlin, Heidelberg

Jiang ZM, Hassan AE (2015) A survey on load testing of large-scale software systems. IEEE Transactions on Software Engineering 41(11):1091–1118

Lathuilire S, Mesejo P, Alameda-Pineda X, Horaud R (2019) A comprehensive analysis of deep regression. IEEE Transactions on Pattern Analysis and

Machine Intelligence pp 1–1
Le Noac'h P, Costan A, Boug L (2017) A performance evaluation of apache kafka in support of big data streaming applications. In: 2017 IEEE International Conference on Big Data (Big Data), pp 4803–4806
Li H, Chen THP, Hassan AE, Nasser M, Flora P (2018) Adopting autonomic computing capabilities in existing large-scale systems: An industrial experience report. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '18, pp 1–10
MLASP (2020) Mlasp - open source system experimental data. Https://github.com/SPEAR-SE/mlasp
Montero-Manso P, Athanasopoulos G, Hyndman RJ, Talagala TS (2020) Fforma: Feature-based forecast model averaging. International Journal of Forecasting 36(1):86–92
Ng AY (2004) Feature selection, l1 vs. l2 regularization, and rotational invariance. In: Proceedings of the Twenty-First International Conference on Machine Learning, Association for Computing Machinery, New York, NY, USA, ICML 04, p 78, DOI 10.1145/1015330.1015435, URL https://doi.org/10.1145/1015330.1015435
Nigam K, Lafferty J, McCallum A (1999) Using maximum entropy for text classification. In: IJCAI-99 workshop on machine learning for information filtering, Stockholom, Sweden, vol 1, pp 61–67
Pan B (2018) Application of xgboost algorithm in hourly pm2.5 concentration prediction. IOP Conference Series: Earth and Environmental Science 113:012127, DOI 10.1088/1755-1315/113/1/012127
Rabbit MQ (2020) Rabbit mq - an open source message broker system. Https://www.rabbitmq.com/
Sayyad AS, Ingram J, Menzies T, Ammar H (2013) Scalable product line configuration: A straw to break the camels back. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, ASE13, p 465474
SciKit-Learn (2019) Scikit learn - machine learning in python. Https://pypi.org/project/psutil
Singh BK, Verma K, Thoke AS (2015) Investigations on impact of feature normalization techniques on classifier's performance in breast tumor classification. International Journal of Computer Applications 116:11–15
Sola J, Sevilla J (1997) Importance of input data normalization for the application of neural networks to complex industrial problems. Nuclear Science, IEEE Transactions on 44:1464 – 1468, DOI 10.1109/23.589532
Tibshirani R (2011) Regression shrinkage selection via the lasso. Journal of the Royal Statistical Society Series B 73:273–282, DOI 10.2307/41262671
Wöllmer M, Eyben F, Schuller B, Douglas-Cowie E, Cowie R (2009) Data-driven clustering in emotional space for affect recognition using discriminatively trained lstm networks. In: Proc. Interspeech 2009, Brighton, UK, pp 1595–1598
Xu Y, Goodacre R (2018) On splitting training and validation set: A comparative study of cross-validation, bootstrap and systematic sampling for

estimating the generalization performance of supervised learning. Journal of Analysis and Testing 2, DOI 10.1007/s41664-018-0068-2

Yin Z, Ma X, Zheng J, Zhou Y, Bairavasundaram LN, Pasupathy S (2011) An empirical study on configuration errors in commercial and open source systems. SOSP 11, p 159172

Zaccone G, Karim MR, Menshawy A (2017) Deep Learning with TensorFlow. Packt Publishing Ltd.