h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

– Fachbereich Mathematik und Naturwissenschaften–

Time Series Analysis of Server Performance Simulation for Large Language Model Deployment

Abschlussarbeit zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

vorgelegt von

Kevin Bernardo

Matrikelnummer: 764124

Referent	:	Prof. Dr. Markus Döhring
Koreferent	:	Prof. Dr. Sebastian Döhler

DECLARATION

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 04 January 2025

Kevin Bernardo

ABSTRACT

In Germany, there are demands to host large language models (LLMs) on internal servers by banks to satisfy the related data protection requirements on confidential information. For deploying an LLM on an internal server, monitoring the system and recognizing if a high rate of request failures occurs are important to ensure the system can answer every request before reaching its timeout, which can be done by forecasting with a machine learning model. Before bringing the system into production, it is important to test whether the time series features generated from the LLM server are relevant for training the model for forecasting and find some insights from it.

This study used a simulation that reflects a real-world situation to test whether the features extracted from an LLM deployment server are relevant for LLM server performance forecasting and afterward find the key features for the forecasting process. First, the simulation dataset was generated based on BurstGPT and then run on the server. After that, the request and server performance metrics data were extracted and used to forecast the server performance, i.e., the server's failure rate of processing requests at specific time intervals. An XGBoost model was compared for the forecasting process with ARMA, VAR, and univariate XGBoost model and then interpreted using Shapley additive explanations (SHAP) to find useful features for forecasting the server's failure rate.

Based on an example use case of a chatbot project, the XGBoost model had the best performance for forecasting the server's failure rate, beating other comparison models. The performance was determined based on MAE and RMSE metrics on the test data with rolling origin evaluation setups. The interpretation of the XGBoost model with SHAP revealed that request duration was the most important feature of the forecasting process. This result was consistent with the XGBoost feature importance method with weight, gain, and coverage.

The prompt sampling method from BurstGPT with Zipf distribution was also compared with a sampling method based on the empirical prompt length distribution of the prompt dataset. Both prompt sampling methods were compared based on the duration of the simulation run, the number of failed requests, and the feature distributions. It was determined that differences existed regarding those aspects. This outcome indicated that considering the prompt length distribution before choosing the prompt sampling method for the simulation is crucial, as it might offer a different result on the simulation, forecasting, interpretation, and resulting parameters to be used

in production.

Keywords: LLM, simulation, time series analysis, BurstGPT, XGBoost, rolling origin evaluation setups, SHAP, Zipf distribution.

ZUSAMMENFASSUNG

In Deutschland gibt es Nachfragen im Bankwesen, Large-Language-Models (LLMs) auf internen Servern zu hosten, um Datenschutzanforderungen für vertrauliche Informationen zu erfüllen. Für die Überwachung eines LLMs auf einem internen Server ist es wichtig zu erkennen, ob und wann eine hohe Rate von Anfrageausfällen auftritt. So kann sichergestellt werden, dass das System jede Anfrage beantworten kann, bevor es zu Zeitüberschreitungen kommt. Um das zu erreichen, erfolgt eine Vorhersage mit einem maschinellen Lernmodell. Dazu werden vom LLM-Server ermittelten Zeitreihenmerkmale für das Training des Analyse-Modells benutzt. Es wird dann getestet, inwieweit diese Merkmale relevant für die Vorhersage sind und welche die Schlüsselmerkmale sind.

In dieser Studie wurde eine Simulation verwendet, die eine reale Situation widerspiegelt, um zu testen, ob die von einem LLM-Server extrahierten Merkmale für die Leistungsprognose des LLM-Servers relevant sind, und um die Schlüsselmerkmale für den Prognoseprozess zu finden. Zunächst wurde der Simulationsdatensatz auf der Grundlage von BurstGPT erstellt und dann auf dem Server verarbeitet. Danach wurden die Anfragedaten und die Leistungsmetriken des Servers extrahiert und zur Vorhersage der Serverleistung verwendet, d. h. der Ausfallrate des Servers bei der Bearbeitung von Anfragen in bestimmten Zeitintervallen. Ein XGBoost-Modell wurde für den Vorhersageprozess mit ARMA-, VAR- und univariaten XGBoost-Modell verglichen und dann mit Hilfe von Shapley-Additive-Explanations (SHAP) interpretiert, um nützliche Merkmale für die Vorhersage der Ausfallrate des Servers zu finden.

Basierend auf einem Anwendungsfall eines Chatbot-Projekts zeigte das XGBoost-Modell die beste Leistung bei der Vorhersage der Ausfallrate des Servers und übertraf die anderen Modelle. Die Vorhersagegüte wurde auf der Grundlage der MAE- und RMSE-Metriken für die Testdaten mit Rolling-Origin-Evaluierungs-Setups ermittelt. Die Ergebnisse des XGBoost-Modells mit SHAP ergaben, dass die Anfragedauer das wichtigste Merkmal des Vorhersageprozesses war. Dieses Ergebnis stand im Einklang mit der XGBoost-Feature-Importance-Methode mit Weight, Gain, und Coverage.

Die Prompt-Sampling-Methode von BurstGPT mit Zipfscher Verteilung wurde auch mit einer Sampling-Methode verglichen, die auf der empirischen Prompt-Längenverteilung des Prompt-Datensatzes basiert. Beide Prompt-Sampling-Methoden wurden anhand der Dauer des Simulationslaufs, der Anzahl der fehlgeschlagenen Anfragen und der Merkmalsverteilungen verglichen. Es wurde festgestellt, dass hinsichtlich dieser Aspekte Unterschiede

bestehen. Dieses Ergebnis deutet darauf hin, dass die Berücksichtigung der Prompt-Längenverteilung vor der Wahl der Prompt-Sampling-Methode für die Simulation von entscheidender Bedeutung ist, da sie zu unterschiedlichen Ergebnissen bei der Simulation, Vorhersage, Interpretation und den daraus resultierenden Parametern, die in der Produktion verwendet werden sollen, führen kann.

Schlüsselwörter: LLM, Simulation, Zeitreihenanalyse, BurstGPT, XGBoost, Rolling-Origin-Evaluierungs-Setups, SHAP, Zipfsche Verteilung.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABKÜRZUNGSVERZEICHNIS

API Application Programming Interface
ARIMA Autoregressive Integrated Moving Average
ARMA Autoregressive Moving Average
CSV Comma-Separated Values
ECDF Empirical Cumulative Distribution Function
GBDT Gradient-Boosted Decision Trees
GDPR General Data Protection Regulation
GPT Generative Pre-trained Transformer
HTTP Hypertext Transfer Protocol
iid independent and identically distributed
JSON JavaScript Object Notation
KDE Kernel Density Estimation
KV Key-Value
LLM Large Language Model
LoRA Low-Rank Adaptation
LSTM Long Short-Term Memory
MAE Mean Absolute Error
MAPE Mean Absolute Percentage Error
MSE Mean Squared Error
NLP Natural Language Processing
RAG Retrieval-Augmented Generation
RMSE Root Mean Squared Error
RMSSE Root Mean Squared Scaled Error
SHAP Shapley Additive Explanation
VAR Vector Autoregression
VRAM Video Random-Access Memory

Part I

THESIS

1 INTRODUCTION

This introduction chapter discusses the motivation behind the master thesis, its objectives, research questions, and how the thesis is structured.

1.1 MOTIVATION

AI models are recently extensively used in Natural Language Processing (NLP). The transformer architecture, proposed by Vaswani et al. [Vas+17], has grown into more complex and robust models, e.g., Large Language Model (LLM). The applications of LLM for solving NLP problems are also impacting the banking and finance sector. For example, Liu et al. [Liu+23] developed FinGPT, an open-source framework to train an LLM using financial texts. Loukas et al. [Lou+23a; Lou+23b] explored the possibility of using LLM for intent classification on a banking dataset. The possibility of using LLM for credit scoring was researched by Feng et al. [Fen+24], including its potential bias risk.

There are many developments in large language models on the market. US companies, such as OpenAI, Microsoft, and Google, are some of the dominant players in offering their closed-source LLMs as a service. In particular, OpenAI offers its Generative Pre-trained Transformer (GPT) models[1], and Microsoft Azure offers its cloud infrastructure for enterprises to run OpenAI GPT models[2]. Nevertheless, banks provide a lot of confidential information, particularly regarding their customer data. The General Data Protection Regulation (GDPR) also restricts banks in Germany to ensure that customer data is handled safely. Regarding the use case of LLM in banking, using closed-source LLMs requires possibly sending confidential information to third parties, which makes it complicated for banks to comply with GDPR. Therefore, there are demands to host the LLM by themselves to uphold these data protection requirements. Bank employees send requests to the server within their infrastructure, which may contain sensitive information about their clients to be processed. The internal LLM server should be able to handle the requests, regardless of how many requests are sent to the server.

There are some possibilities of obtaining LLM for internal use, e.g., fully training the LLM internally, using pre-trained LLM as is, or further fine-tuning them. In the end, the obtained LLM must be deployed and run on a server using serving frameworks, such as vLLM [Kwo+23]. During deployment, some

1 https://platform.openai.com/docs/models, Last accessed: 17 October 2024
2 https://azure.microsoft.com/en-us/products/ai-services/openai-service, Last accessed: 17 October 2024

of the important factors are monitoring the system and recognizing if a high rate of request failures occurs at a certain point to guarantee the model can answer every request before reaching its timeout, which otherwise triggers the request failure. One way to achieve this is by using time series analysis to forecast the model performance. For example, time series features extracted from the system are used to train a machine learning model to forecast the failure rate of the requests.

Before bringing the system into production, it is important to test whether the features used are relevant for training the model for forecasting. The test can be done using a simulation that reflects a real-world situation on how the requests are sent to the server. The simulation result is then analyzed and used for modeling to test the server's features, whether they are capable of accurately forecasting the server performance. The result may also give some insights into what features are truly useful for perceiving high request failure rates during production.

1.2 OBJECTIVES AND RESEARCH QUESTIONS

In cooperation with PPI AG, the purpose of this study was the usage of simulation for testing an LLM deployment server, analyzing the simulation result for server performance forecasting, and finding crucial features for forecasting. First, the server system was tested using LLM simulation based on BurstGPT [Wan+24]. Mathematical distributions were used to create multiple requests that were sent to the deployment server. The LLM server behavior during the simulation was then recorded as time series data based on the server's response (e.g., the response length and duration) and the internal server metrics generated by the serving framework (e.g., number of tokens generated per second and number of requests currently processed). These data served as the basis for forecasting the server performance. The data was used in the next step to analyze the system and forecast the server performance. The time series data was aggregated, explored, and selected to train an XGBoost model [CG16] to forecast the server's failure rate of processing incoming requests at specific points in time. The trained XGBoost model was then interpreted to find useful features for forecasting the server's failure rate, mainly using SHAP values [LL17]. The abovementioned approach was implemented based on an example use case of a chatbot project.

The research questions to be answered in this study were:

1. Can the time series data generated from the LLM simulation be used to create a meaningful forecast of the LLM server performance by training a machine learning model?
2. Can comprehensible key features/factors that are essential for forecasting the LLM server performance be found based on the trained machine learning model and the generated time series data?

The result of this study would benefit the company, both for internal use (e.g., deploying an LLM server in the company) and as a knowledge in projects with PPI AG customers.

1.3 STRUCTURE

This thesis is divided into several chapters. Chapter 2 details the crucial concepts used in this work. Other research and papers related to this thesis are explained and discussed in Chapter 3. Chapter 4 and Chapter 5 explain the steps and implementation details of the experiment's simulation and analysis. The simulation and analysis results are shown in Chapter 6, followed by their discussion in Chapter 7. Finally, Chapter 8 concludes the work with a summary and some possible works in the future.

2

BACKGROUND

This chapter provides some background information regarding this work. The concepts are divided into two parts. The first part details the concepts used for the simulation part of this study. The second part explains the methods for analysis, modeling, and interpreting the trained machine learning model.

2.1 SIMULATION

This section explains the benchmarking with BurstGPT and Zipf distribution used in the simulation process.

2.1.1 *BurstGPT*

BurstGPT is a dataset collected by Wang et al. [Wan+24], which reflects the real-world workload of an LLM serving system. This dataset was collected by Wang et al. for two months within a campus-sized region, consisting of around 1.5 million traces of timestamps, request length, and response length from ChatGPT and GPT-4 models [Wan+24]. It was found that the request patterns of the models are bursty, which means there are sudden increases in requests sent to the server [AE+14]. More details on the characteristics of this dataset can be found in their paper [Wan+24]. Based on the bursty characteristic of the BurstGPT dataset, Wang et al. created a benchmark method that can test LLM serving systems with bursty workload.

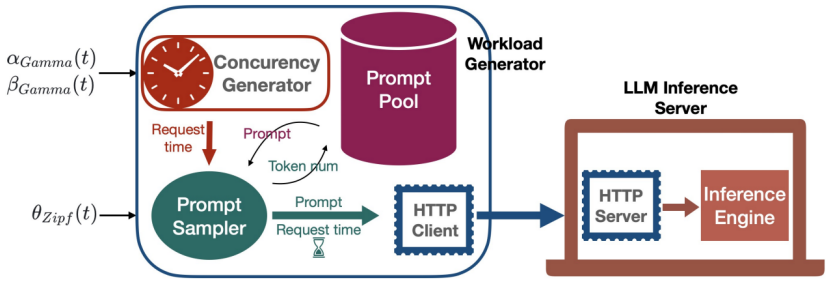

Figure 2.1: Diagram of benchmarking with BurstGPT [Wan+24].

Figure 2.1 shows how the benchmark method functions. There are several components, which are divided into the client and server sides. The server runs the LLM serving system, where the LLM reads the prompt tokens from the received requests and generates the output tokens as the response. The client or workload generator creates multiple concurrent requests, reflecting

bursty workload, using concurrency generator, prompt pool, and prompt sampler.

The concurrency generator determines when each request will be sent to the server. The result of the concurrency generator is the time a request has to wait before it is sent via Hypertext Transfer Protocol (HTTP) to the server. The randomness of the concurrency generator is controlled by using Gamma distribution, which is represented by shape parameter α and scale parameter β [Wan+24]. The Gamma distribution parameters do not always stay constant but change over time to simulate different patterns of burstiness. The shape parameter α is changed by quadratic function, and the scale parameter β is changed using linear function [Wan+24].

The prompt pool and prompt sampler work together to decide which prompt texts should be sent to the server for each request. The prompt pool contains a list of possible prompts that can be sampled for each request. The list of prompts could be gathered from existing prompt datasets. The prompt sampler samples the prompt from the prompt pool by specifying a random request length based on Zipf distribution [Wan+24], which fits the distribution of the request length in the BurstGPT trace. The θ parameter of Zipf distribution controls how often the shorter requests are selected [Wan+24]. The prompt pool finds a prompt with the specified request length and returns it to the prompt sampler. The result of this interaction between both components is a prompt for each request, which adheres to Zipf distribution, to be sent after the request's waiting time ends.

The basic workflow of the benchmark starts with generating multiple waiting times by the concurrency generator based on Gamma distribution with shape parameter α and scale parameter β. After that, a prompt is sampled for each generated request by the prompt sampler from the prompt pool, which adheres to Zipf distribution with a specific parameter θ. Every request waits until its waiting time ends, and the prompts are sent via HTTP. These processes happen within the workload generator. The LLM server accepts and handles the HTTP requests from the workload generator. The generated requests from the workload generator create a concurrency pattern, which can be considered as time series data.

2.1.2 *Zipf Distribution*

Zipf distribution was named after a Harvard University professor, Dr. George Kingsley Zipf [Qiu+17]. For a certain corpus of texts, the frequency of words (F_r) that appear in the corpus can be counted, sorted in descending order, and given ranks (r) for each word, with rank 1 representing the most frequent word ($r \in [1, L], r \in N$) [Qiu+17]. Zipf's law states that F_r and r have the following relation:

$$F_r \cdot r = C, \tag{2.1}$$

where C is a non-absolute constant that varies around a certain value [Qiu+17]. The formula suggests that word frequency and its rank are inversely proportional. This relation can be shown graphically in Figure 2.2 as an example. The most frequent word (w_1) has a frequency of 1,000. The second most frequent word has half of the frequency of w_1. The third most frequent word has a third of the frequency of w_1, and so forth.

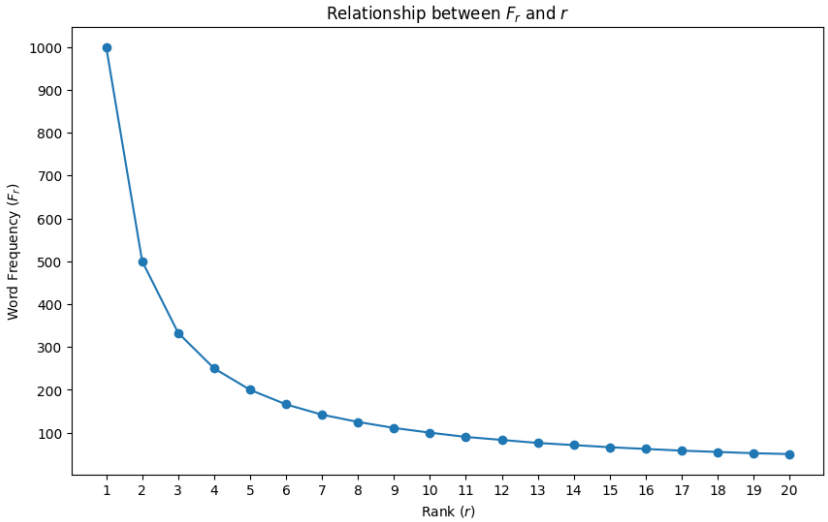

Figure 2.2: Example of relationship between word frequency and rank in Zipf's law.

The Zipf distribution of the words can also be determined for each rank. Given the number of unique words appearing in the corpus (L), the probability of a word with rank r appearing in the corpus (P_r) can be calculated using the following formula:

$$P_r = C \cdot r^{-1}, C = \frac{1}{\sum_{r=1}^{L} r^{-1}}, \tag{2.2}$$

where C represents the probability of the most frequent word [Qiu+17]. Based on Equation 2.2, the sum of all probabilities equals 1 [Qiu+17].

$$\sum_{r=1}^{L} P_r = \sum_{r=1}^{L} C \cdot r^{-1} = C \cdot \sum_{r=1}^{L} r^{-1} = \frac{1}{\sum_{r=1}^{L} r^{-1}} \cdot \sum_{r=1}^{L} r^{-1} = 1 \tag{2.3}$$

After findings from Zipf, M. Joos proposed a correction to Zipf's formula by adding parameter θ in the formula:

$$P_r = C \cdot r^{-\theta}, \tag{2.4}$$

where $\theta > 0$ [Qiu+17]. To make sure that the sum of all probabilities equals 1, C is then defined as follows:

$$C = \frac{1}{\sum_{r=1}^{L} r^{-\theta}} \tag{2.5}$$

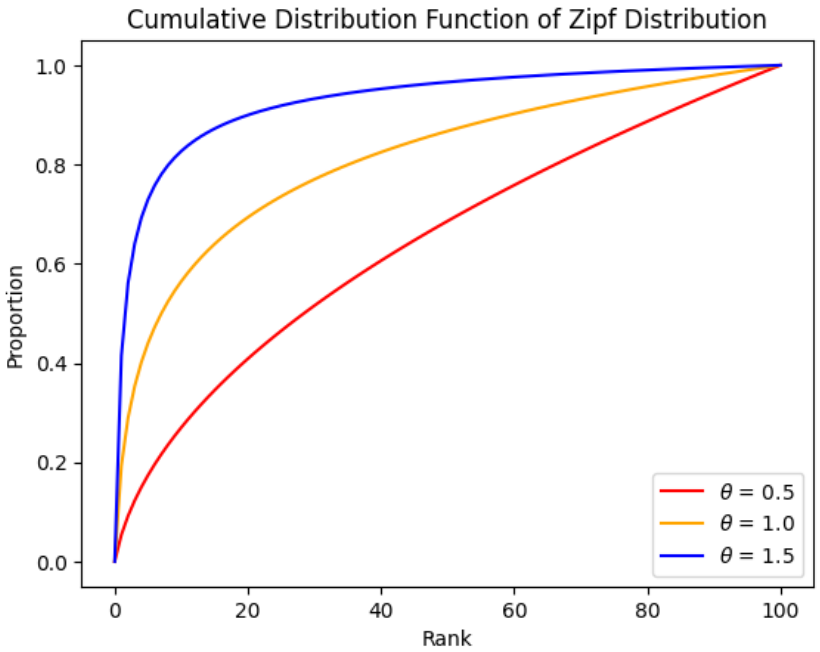

Figure 2.3: Different effects of the θ parameter on the Zipf distribution with $L = 100$.

Figure 2.3 shows the effect of parameter θ on the Zipf distribution in a corpus with 100 unique words. The new formula is equivalent to the formula proposed by Zipf for $\theta = 1$ [Qiu+17]. A smaller value of $0 < \theta < 1$ contributes to a smaller proportion of the most frequent words in the corpus. On the other hand, most frequent words appear more often in comparison to least frequent words for higher values of $\theta > 1$.

Zipf distribution has been found in multiple fields, such as word frequency, publication number of scientific literature, and urban population [Qiu+17]. In the context of simulation with BurstGPT, the Zipf distribution can be used to model the distribution of request length sent to the server. The number of requests with less tokens in the BurstGPT dataset dominates the dataset, which adheres to the Zipf distribution [Wan+24].

2.2 ANALYSIS, MODELING & INTERPRETATION

This section contains the main idea of time series analysis methods for data preparation and forecasting, the XGBoost algorithm for forecasting, and

Shapley Additive Explanation (SHAP) for interpreting the prediction results of machine learning models.

2.2.1 *Time Series Analysis*

This section contains some short explanations of the time series concepts that are used in this work. A time series consists of data observations x_t, where t is the specific time when the data is recorded [BD16]. The main goal of time series analysis is to create a mathematical model to describe time series data [SS17c]. There are multiple ways to explain univariate time series data observation, such as using autoregressive and moving average models.

The main idea of autoregressive models is explaining the current observation x_t from a time series using previous observations [SS17a]. Given p previous observation used for describing the current observation, the autoregressive model $AR(p)$ can be defined as follows:

$$x_t = \alpha + \phi_1 x_{t-1} + \phi_2 x_{t-2} + \ldots + \phi_p x_{t-p} + w_t, \tag{2.6}$$

where $\alpha, \phi_1, \phi_2, \ldots, \phi_p$ are constants, and all $\phi_p \neq 0$ [SS17a]. The values of $x_{t-1}, x_{t-2}, \ldots, x_{t-p}$ can also be called lagged values, with x_{t-p} representing lag p of the time series data. The value of w_t corresponds to white noise with the mean o and variance σ_w^2, where $w_t \sim wn(0, \sigma_w^2)$ [SS17a]. The autoregressive model can also describe current observations by combining multiple features, i.e., multivariate. This model is called Vector Autoregression (VAR) model, which uses matrices and vectors to represent the current observation x_t instead of a simple equation.

The moving average model is another way to describe current observation x_t using the linear combination of white noises [SS17a]. Given q previous white noises used for describing the current observation, the moving average model $MA(q)$ can be defined as follows:

$$x_t = w_t + \theta_1 w_{t-1} + \theta_2 w_{t-2} + \ldots + \theta_q w_{t-q}, \tag{2.7}$$

where $\theta_1, \theta_2, \ldots, \theta_q$ are model parameters and all $\theta_q \neq 0$ [SS17a]. The value of w_t corresponds to white noise with $w_t \sim wn(0, \sigma_w^2)$ [SS17a].

Both of the two previous models can be combined into a single model, which is called the Autoregressive Moving Average (ARMA) model. This new $ARMA(p, q)$ model can be described with the following formula [SS17a]:

$$x_t = \alpha + \phi_1 x_{t-1} + \ldots + \phi_p x_{t-p} + w_t + \theta_1 w_{t-1} + \ldots + \theta_q w_{t-q} \tag{2.8}$$

Equation 2.8 shows the current observation x_t as the combination of Equation 2.6 and Equation 2.7, with $w_t \sim wn(0, \sigma_w^2)$ [SS17a]. The ARMA model is also a generalization of both autoregressive and moving average models since $p = 0$ or $q = 0$ resulted in the moving average and autoregressive model, respectively. [SS17a]

The important assumption of all models mentioned above is the stationarity of the data. Stationarity is a time series data property, which simply means that the time aspect do not influence the statistical properties [WKP98]. A time series is considered stationary when the following conditions are met [SS17c]:

1. The mean value of the time series is constant, i.e., for a given observation x_t at time t, the mean value μ_{x_t} is equal to:

$$\mu_{x_t} = \mu_t = E(x_t), \tag{2.9}$$

   with $E(x_t)$ defined as the expected value of x_t [SS17c].

2. Given two observations x_t at time t and x_s at time s, the autocovariance value of the time series relies only on the absolute difference of s and t ($|s - t|$). The autocovariance function between x_t and x_s can be described as:

$$\gamma_x(s,t) = \gamma(s,t) = cov(x_s, x_t) = E[(x_s - \mu_s)(x_t - \mu_t)], \tag{2.10}$$

   with $\gamma_x(s,t) = \gamma_x(t,s)$, to measure the linear dependence of two observations at different times [SS17c].

A time series can be defined further as strictly stationary when the probabilistic behavior of a set of data observations $\{x_{t_1}, x_{t_2}, \ldots, x_{t_k}\}$ and a set of shifted data observations $\{x_{t_{1+h}}, x_{t_{2+h}}, \ldots, x_{t_{k+h}}\}$ for all time points $t_1, t_2, \ldots, t_k$ and all time shifts $h = 0, \pm 1, \pm 2, \ldots$ is similar [SS17c]. However, this property is more complicated to evaluate [SS17c].

An example of a stationary time series is white noise [SS17c]. μ_t of white noise is 0 and its $\gamma_w(s,t)$ for $s = t + h$, which can be written as $\gamma_w(h)$, equals to σ_w^2 and 0 for $h = 0$ and $h \neq 0$, respectively [SS17c]. Both values do not depend on the time component. Another example is the autoregressive model. Given an $AR(1)$ model based on Equation 2.6 with $x_t = \phi x_{t-1} + w_t$, the model is only stationary when $|\phi| < 1$ [SS17a].

The Dickey-Fuller test is a method used to test if a time series has $|\phi| < 1$ [SS17b]. Given an $AR(1)$ model with $x_t = \phi x_{t-1} + w_t$ and $w_t \sim$ independent and identically distributed (iid) $N(0, \sigma_w^2)$ [SS17c; SS17b], the Dickey-Fuller test is done with a null hypothesis $H_0 : \phi = 1$ and an alternative hypothesis $H_1 : |\phi| < 1$. The test does not use the equation of x_t directly to calculate the test statistic. Instead, it uses the following equation [SS17b]:

$$\begin{aligned}\nabla x_t &= x_t - x_{t-1} = \phi x_{t-1} + w_t - x_{t-1} = (\phi - 1)x_{t-1} + w_t \\ &= \gamma x_{t-1} + w_t\end{aligned} \tag{2.11}$$

The Wald statistic t_γ is then calculated based on regression to find γ, where:

$$t_\gamma = \frac{\hat{\gamma}}{se(\hat{\gamma})}, \tag{2.12}$$

[SS17b], $\hat{\gamma}$ as the estimated value of γ, and $se(\hat{\gamma})$ defined as the standard error of $\hat{\gamma}$. Finally, the value of t_γ is compared to the Dickey-Fuller distribution to test if the null hypothesis should be rejected. The test can also be extended to an $AR(p)$ model with multiple lags, which is called the augmented Dickey-Fuller test [SS17b].

2.2.2 *XGBoost*

XGBoost is a machine learning algorithm proposed by Chen and Guestrin [CG16], which uses the gradient boosting algorithm to create a tree ensemble for prediction. This algorithm can achieve state-of-the-art results, is highly scalable, and performs fast even with a large amount of data due to its implemented optimizations [CG16]. The following explanations are the basic mathematical concepts of the XGBoost algorithm with regression trees by Chen and Guestrin [CG16].

Given a dataset $D = \{(x_i, y_i)|i \in 1, 2, \ldots, n\}$, with $x_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}$, the prediction output of XGBoost can be defined as:

$$\hat{y}_i = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), \tag{2.13}$$

where the output $\hat{y}_i$ is based on the sum of the output from K tree models ($f_k(x_i)$).

The XGBoost algorithm uses an objective function to train every model f_k for the prediction. The objective function consisted of 2 parts: a differentiable convex loss function l and a regularization term Ω.

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \tag{2.14}$$

The loss function is used to calculate the residual between the predicted value $\hat{y}_i$ and the true value y_i. At the same time, the regularization term tries to reduce the complexity of every tree to make the model generalizable. This

objective function makes the models simple and able to create robust predictions.

The XGBoost algorithm creates a new tree f_t based on the output of the previous trees $\hat{y}_i^{(t-1)}$ to create a new prediction $\hat{y}_i^{(t)}$. This process is done by optimizing the objective function based on Equation 2.14 with the second-order Taylor approximation for the current tree f_t.

$$\begin{aligned}\mathcal{L}^{(t)} &= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) \\ &\simeq \sum_{i=1}^{n} [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \\ &\approx \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) = \tilde{\mathcal{L}}^{(t)},\end{aligned} \tag{2.15}$$

where $g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$ and $h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$. Since $l(y_i, \hat{y}_i^{(t-1)})$ is constant, the term can be omitted from the last line in Equation 2.15 to simplify the objective function into $\tilde{\mathcal{L}}^{(t)}$.

Given that the function f_t is a tree model, the structure of the model and the data points that belong to a leaf j can be represented by the symbol q and $I_j = \{i | q(x_i) = j\}$, respectively. The formula from Equation 2.15 can be transformed further to be more suitable for the tree structure of f_t. The sigma sign can be rewritten to iterate all sets of data points I_j in all T terminal nodes or leaves in the tree, with w_j as the weight of each leaf. The term Ω can also be expanded.

$$\begin{aligned}\tilde{\mathcal{L}}^{(t)} &= \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T\end{aligned} \tag{2.16}$$

The Equation 2.16 introduces two regularization parameters. The parameter γ controls how extensive the tree grows by pruning the tree leaves. The parameter λ controls the weight of the tree leaves.

To find the minimum of the loss function, the optimal weight w_j^* is defined by calculating the derivative of $\tilde{\mathcal{L}}^{(t)}$ with respect to w_j and setting it equal to 0.

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \tag{2.17}$$

By substituting the optimal weight w_j^* to Equation 2.16, the optimal value of the loss function for a fixed tree structure q can be defined as:

$$\tilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2}\sum_{j=1}^{T}\frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma T \tag{2.18}$$

This function serves as the scoring function to evaluate the tree. Notice that the parameter λ controls the value of w_j^* in Equation 2.17 and both parameters γ and λ control $\tilde{\mathcal{L}}^{(t)}(q)$ in Equation 2.18.

The XGBoost algorithm uses a greedy algorithm to build the tree from the root node down to the leaf nodes by adding new branches to the tree iteratively. The algorithm iterates every feature to find the best split value from a given terminal node I that creates two disjunctive left node I_L and right node I_R and reduces the loss function based on the following equation:

$$\mathcal{L}_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma, \tag{2.19}$$

The algorithm finds the split that reduces the loss most significantly. Notice that both parameters γ and λ control the loss reduction in Equation 2.19.

Like gradient boosting, the XGBoost algorithm uses shrinkage or learning rate parameter to reduce the gradient descent step of each tree [BCMM21]. After training the current tree f_t, the prediction output can be summarized by adding the prediction from all previous trees with the output from f_t multiplied by the learning rate η.

Chen and Guestrin [CG16] focus not only on the gradient boosting and the tree training side of the algorithm, but also on parallelization and computation methods to enhance the performance of XGBoost. For example, XGBoost uses the approximate algorithm, weighted quantile sketch, and sparsity-aware method to efficiently find splits for efficient split finding, parallel training, and handling missing data. The author also optimizes the system by using cache-aware access, column block, and out-of-core computation to use the computer system for training efficiently.

2.2.3 *Shapley Additive Explanation (SHAP)*

For a simple model, such as linear regression, the model itself can be used directly for prediction and explaining how the model works or comes to the prediction it calculates. The explainability factor is more complicated for complex models. For example, an ensemble model consists of a group of models, which makes interpreting the whole model difficult, even when it consists of multiple simple models. SHAP is a method that can explain the

importance of features for a machine learning prediction with tabular data, even for complex models. This method was proposed by Lundberg and Lee [LL17], which is based on Shapley values. The following SHAP explanations are based on the information from [LL17].

The importance calculated by SHAP satisfies the properties of local accuracy, missingness, and consistency [LL17]. Given an original model f and a single input x for local explanation method, a simplified input x' can be created. This input can be mapped to the original input x using a certain function $x = h_x(x')$. The function h_x creates a mapping for the values 0 and 1 to the original feature space. 0 means the input feature is excluded in the model, and vice versa for 1. For local accuracy properties to be fulfilled, the simpler explainable model should be able to at least match the output from the original model, hence:

$$f(x) = g(x') = \phi_0 + \sum_{i=1}^{M} \phi_i x'_i, \tag{2.20}$$

with M defined as the number of the simplified input features and ϕ_i as the influence of each feature on the prediction. ϕ_0 refers to the predicted value without any known features. The explanation model g was created using additive feature attribution methods. Missingness property implies that a feature does not have any influence if the feature is not included:

$$x'_i = 0 \implies \phi_i = 0 \tag{2.21}$$

Consistency property simply means that if the contribution of a simplified input increases or stays the same despite other features, the value of ϕ_i should at least remain the same or increase.

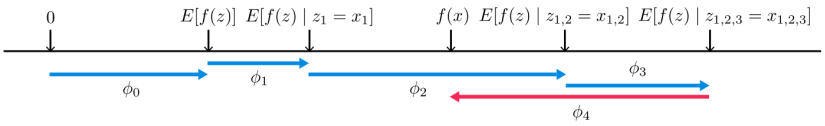

Figure 2.4: An example of SHAP values [LL17].

Figure 2.4 shows an example of how the SHAP values can be interpreted for one instance, with the assumption that the predicted value $f(x)$ is a probability between 0 and 1. It starts with ϕ_0, which represents the base value without known features. After that, the values of influence/SHAP values from the features are added, and it lands at the predicted value of the original model $f(x)$. The values of ϕ_1, ϕ_2, and ϕ_3 are positive, which means the values of x_1, x_2, and x_3 have a positive influence by increasing prediction probability. Meanwhile, adding the value of ϕ_4 reduces the probability, which means the value of x_4 has a negative effect. The absolute SHAP value shows the intensity of the influence. x_2 and x_4 have more influence on the

prediction compared to x_1 and x_3, which makes them important features for the prediction.

The example above shows the important benefit of using SHAP, which is the ability to transparently explain the influence of each feature for each data entry (local explanation) [Dwi+23]. SHAP also offers the possibility to explain the influence of each feature across all data entries (global explanation) [Dwi+23]. Nevertheless, SHAP is not a perfect solution. SHAP value does not quantify the uncertainty of the explanation, which emerges from the uncertainty in the data [MCB20]. SHAP also does not guarantee a causal interpretation of the prediction [MCB20]. Feature dependence can also extrapolate the result, which might be misleading the interpretation [MCB20]. The calculation of SHAP values is also time-consuming [Dwi+23]. In the case of tree-based models (e.g., XGBoost), the SHAP values can be calculated with the TreeSHAP algorithm [Lun+20].

3 RELATED WORKS

This chapter contains an in-depth review of works relevant to this study, followed by some comparisons between the chosen related works. It is divided into two sections. The first part dives into related works for simulation and benchmarking LLM models and systems. The second part focuses more on methods for forecasting time series data. Both parts also include discussions about the differences between the related works with tables.

3.1 LLM BENCHMARK

The works selected for this section were based on their contribution for benchmarking LLM models and systems with performance evaluations. Three works were picked and discussed based on this criterion. One selected work is BurstGPT, whose detailed explanations are presented in Section 2.1.1.

Sun et al. proposed a method to benchmark the performance and cost-effectiveness of a locally deployed LLM pipeline called CEBench [Sun+24]. Their proposed benchmarking method focuses on the benchmarking of Retrieval-Augmented Generation (RAG) pipelines on multiple benchmarking scenarios, e.g., the LLM effectiveness benchmarking by testing different hyperparameters, end-to-end benchmarking for RAG pipelines from document retrieval to the generated output, prompt engineering benchmarking with different prompting strategies, and multi-objective evaluation for quality, time, and cost assessment with recommendations.

Figure 3.1 illustrates the components and the workflow of CEBench. The configuration files manage the benchmark settings. The dataloader prepares the data by merging prompt templates and queries and preparing chunks for the document retrieval process. The query execution engine then uses the prompts to benchmark the system, whose performance is monitored and logged. CEBench tracks several metrics for evaluation, such as memory usage, query latency, output quality (e.g., Mean Absolute Error (MAE) and F1-score), and estimated cost per prompt. The plan recommender suggests optimal plans based on the benchmarking result for cost and effectiveness. The paper also provided two use cases, mental health LLM assistant and contract reviewing, to demonstrate the application of CEBench. This benchmarking method could help test locally deployed RAG systems with limited budgets to find the best cost-effective solution.

Tuggener et al. explored the possibility of benchmarking LLM with quantization and low-ranking approximation methods [Tug+24]. The benchmark-

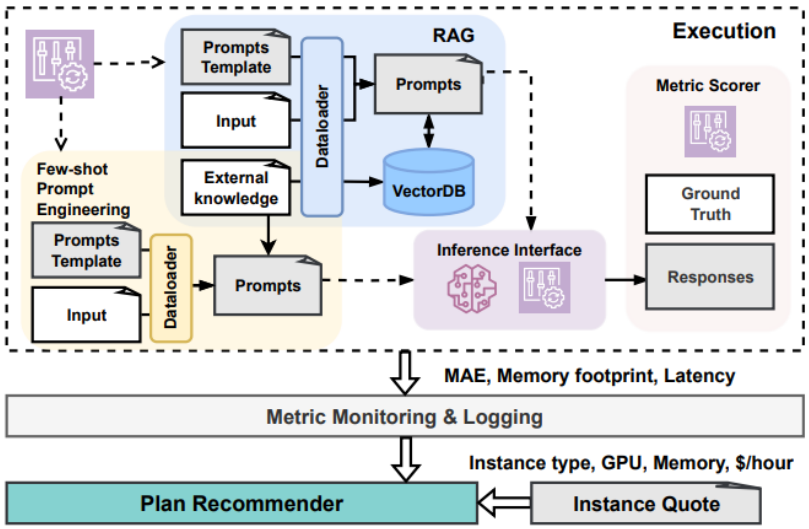

Figure 3.1: CEBench workflow [Sun+24]. The components consist of configuration, dataloader, query execution, metric monitoring & logging, and plan recommender.

ing method focuses on LLM inference and fine-tuning with limited resources using quantization and Low-Rank Adaptation (LoRA). The benchmark measures the resources used for the computation during inference or fine-tuning and the generated output quality. The metrics used to measure the resources are Video Random-Access Memory (VRAM) usage, generated tokens per second, processed batches per second, and power consumption in watt-hour. The output quality is measured using quantitative and qualitative evaluation based on questions by MT-bench [Zhe+24]. The quantitative evaluation is based on the average score generated from the OpenAI GPT-4 model to assess the output quality on a scale from 1 to 10. It also compares the output of a tuned model with its base model by showing them to the GPT-4 model. Then, the GPT-4 model selects which model performs better or a tie. Human evaluation is also proposed to qualitatively evaluate the output and support the quantitative evaluation. Five questions with different topics selected from MT-bench were used for evaluation by averaging scores ranging from 1 to 5 from eight humans. This benchmarking method has the potential for real-world application to measure the quality of LLM that applied efficiency methods for low resource application.

3.1.1 *Discussion*

Table 3.1 shows the summary and comparison between the three benchmarking methods selected in this study.

The focuses and possible real-world applications of the selected works cover a broad spectrum, ranging from high performance application in burst situations in [Wan+24] to low resource application in [Tug+24]. The bench-

Aspect	Wang et al. (BurstGPT) [Wan+24]	Sun et al. (CEBench) [Sun+24]	Tuggener et al. [Tug+24]
Focus	LLM deployment in burst situations	Cost-effective LLM deployment	LLM deployment with limited resource
Performance Evaluation	Request failure rate, token latency, system throughput	Memory usage, query latency, estimated cost per prompt	VRAM usage, generated tokens per second, processed batches per second, power consumption (watt-hour)
Output Quality Evaluation	-	MAE, F1-score	Quantitative (using GPT-4 model evaluation), qualitative (using human evaluation)
Real-world application	LLM test for high performance	LLM test for cost-effective solution	LLM test for low resource application

Table 3.1: Summary of the benchmarking approaches used in Section 3.1.

marking method in [Sun+24] shows more focus in between.

All selected papers had metrics to measure the performance for benchmarking. [Wan+24] purely focused on the performance metrics, such as request failure rate, system throughput, and token latency. [Sun+24] and [Tug+24] also considered the performance factor, e.g., by measuring latency and throughput. However, they also considered the resource consumption aspects, such as memory usage, power consumption, and estimated cost per prompt. Only [Sun+24] and [Tug+24] considered the generated output quality in their benchmark, either by using only scoring metrics or with a combination of human evaluation. [Wan+24] needs a separate process to evaluate the output quality.

Based on the descriptions of the benchmarking methods, [Tug+24] has a more straightforward benchmarking setup. [Wan+24] and [Sun+24] methods consisted of multiple components for the benchmarking process. Meanwhile, [Tug+24] only used the model performance and questions from MT-bench

for evaluation. Overall, the methods from [Wan+24] and [Tug+24] are more general in comparison to [Sun+24]. [Wan+24] implementation can theoretically be used to benchmark the performance of different use cases. On the other hand, [Sun+24] focuses more on RAG implementation based on its architecture and use case examples. [Tug+24] could also be used for general purposes. Its scoring method to evaluate the output quality can be used for various use cases, while the metrics from [Sun+24] are only specific for certain problems. However, the scalability problem of the test needs to be addressed since it uses human evaluation.

3.2 FORECASTING TIME SERIES DATA

This section focuses on related works selected based on using machine learning models to forecast time series data, specifically the methods used in the selected works for handling time series data for forecasting with XGBoost. Four works were selected and discussed based on this criterion. This section shows the versatility of the XGBoost algorithm in forecasting time series data for different use cases, such as in sales, financial, and healthcare domains.

Niu built a sales forecast model for Walmart sales dataset based on XGBoost model to forecast the sales data of the next 28 days [Niu20]. The training data consisted of 1,941 days, with 3,049 products in 10 Walmart stores. Due to the vast amount of data, the time series data were preprocessed to simplify the data further. The first step was compressing the data into a smaller representation with less memory. To ensure that XGBoost could handle the time series data, temporal characteristics from the timestamps, e.g., month, day of week, weekend, etc., and statistical properties from a certain period of time, such as lag feature, minimum, maximum, etc., were extracted. It also recursively eliminated features using cross validation to remove unimportant features during training. Niu also compared the XGBoost with logistic and ridge regression based on the Root Mean Squared Scaled Error (RMSSE) metric. XGBoost achieved the lowest RMSSE value of 0.652, followed by ridge (0.765) and logistic regression (0.793). This result made XGBoost the best performing model in the experiment. The feature importance of the top 20 features extracted from the XGBoost model was also shown for interpreting the forecasting result and possibly using the important features for forecasting in the future.

Jabeur, Mefteh-Wali, and Viviani forecasted gold price in US dollars using XGBoost and interpreted the forecast using SHAP [JMWV24]. The gold price data ranged from January 1986 to December 2019, with a total of 408 monthly observations. The other seven features used to forecast the gold price were collected from different freely available sources, also in a monthly period. These features included silver price, oil price, inflation rate in the USA, and S&P 500 index. The data was divided into 80% training and 20% test data for training and model evaluation. Then, six machine learning algo-

rithms were trained to forecast the gold price data: linear regression, neural networks, random forest, LightGBM, CatBoost, and XGBoost. These models were compared using Root Mean Squared Error (RMSE), Mean Squared Error (MSE) , MAE, and R^2 evaluation metrics. XGBoost was identified as the best model based on these metrics, with the lowest RMSE (34.921), MSE (1219.500), and MAE (21.968) score, and the highest R^2 (0.994) score. After training and evaluation, some visualization plots based on SHAP values were generated and analyzed to understand the forecasting result from the XGboost model, such as summary and dependence scatter plots.

Zhang et al. applied the XGBoost algorithm to forecast the sales volume of two milk-tea stores in Beijing based on sales and weather data [Zha+21]. Two sales data sets were used to train the model and test the forecast. The first data contained almost 1.2 million orders from January 2019 to January 2020. The second data had fewer data, with 89,109 entries from July 2019 to January 2020. The weather data consisted of weather information every day from January 2019 to January 2020. Since the trained model was expected to forecast the sales volume every 15 minutes, the previously mentioned data entries were aggregated to create the sales volume (based on order value) and order volume (based on the number of orders). The temporal information, such as month, day of month, up to the minute of the entries, and historical transaction from the 7-day period, were used for training. Holiday information, air quality, temperature, rain, and haze information were also added to the sales data. An XGBoost model was then trained based on 75% of the data. The model performance was evaluated on the two sales data using MAE and RMSE metrics. The trained XGBoost model was also compared with other models, such as Autoregressive Integrated Moving Average (ARIMA), Long Short-Term Memory (LSTM), Prophet, and Gradient-Boosted Decision Trees (GBDT). The XGBoost and GBDT models outperformed all other models by achieving the lowest MAE and RMSE on both sales data. However, the XGBoost model still outperformed the GBDT model with a slight margin on both sales data and only a third of the iterations needed for GBDT to achieve almost the same result.

Fang et al. forecasted the occurrence of COVID-19 cases and compared the performance of the seasonal ARIMA algorithm with the XGBoost algorithm in their work [Fan+22]. The data used for training was based on the COVID-19 cases and vaccination data in the USA from December 13, 2020 to June 16, 2021. The goal was to forecast the number of COVID-19 cases for the next 14 days until 30 June 2021. Fang et al. added seven lag features to capture the seasonal trend in the data for XGboost training. Weekday information was also included to handle the temporal information during training. The XGBoost model used one-step ahead prediction as the forecasting method in the experiment. This method used all data before and at time t to forecast the number of cases at time $t + 1$. Mean Absolute Percentage Error (MAPE), MAE, and RMSE were the chosen metrics for model evaluation. The XGBoost

model performed better on the test data, with more than half of the seasonal ARIMA model's MAPE, MAE, and RMSE values. The work also tried to find the important features using information from the XGBoost model.

3.2.1 *Discussion*

Table 3.2 shows the summary and comparison between the four forecasting methods selected in this study.

Aspect	[Niu20]	[JMWV24]	[Zha+21]	[Fan+22]
Predicted value	Walmart sales data	Gold price	Milk-tea stores sales volume	COVID-19 cases
Data	Walmart sales dataset	Silver price, oil price, exchange rates, inflation, S&P 500, iron ore price	Sales data, weather data	US COVID-19 cases and vaccination data
Temporal features	Temporal information, lagged features	Monthly information	Temporal information, lagged features	Temporal information, lagged features
Evaluation metrics	RMSSE	RMSE, MSE, MAE, R^2	MAE, RMSE	MAPE, MAE, RMSE
Comparison models	Logistic regression, ridge regression	Linear regression, neural networks, random forest, LightGBM, CatBoost	ARIMA, LSTM, Prophet, GBDT	Seasonal ARIMA
Explanation method	Information from XGBoost model	SHAP	-	Information from XGBoost model

Table 3.2: Summary of the forecasting approaches used in Section 3.2.

Most of the selected works combined several time series data from different sources to support the forecasting process, with [JMWV24] combining the most time series data from different sources for the forecasting. Different data sources make the data preprocessing more tedious since they may have different representations of the time series data, which requires more effort to standardize the data before combining them. The type of data and the main preprocessing step from [Zha+21] are the most similar to what this work had done, which is aggregating the time series data with irregular intervals into regular intervals before combining it with another time series data.

Most of the reviewed works also created new temporal features during feature engineering processes since the XGBoost algorithm does not consider time components in the data by default. Compared to the related works, this work only used the lagged information derived from existing features. [Niu20], [Zha+21], and [Fan+22] created temporal information as features, ranging from year information down to minute information, as well as lagged features. [JMWV24] only used the monthly information of each feature for the forecasting.

The selected evaluation metrics used for model comparison were also varied. Most of the works used RMSE and MAE for their model evaluation, which is also mainly used in this work. [Niu20] chose the RMSSE metric for the comparison and evaluation. [JMWV24] used MSE and R^2 as additional metrics to evaluate the model performance. However, using both MSE and RMSE might be redundant since RMSE is only the root of MSE. [Fan+22] chose MAPE as an additional metric. None of the selected papers mentioned explicitly any information regarding the objective function used during the training of the XGBoost model. There is also no concrete information about the forecasting method used for the XGBoost model, except [Fan+22], which used one-step prediction. This forecasting method was also used in this work.

The related works compared the XGBoost model with different types of machine learning models. Only [Zha+21] and [Fan+22] provided a comparison with machine learning models that specifically handle time series data, ranging from simple linear models (ARIMA) to complex deep learning models (LSTM). [Niu20], [JMWV24], and [Zha+21] used traditional linear models as comparison models, with [JMWV24] and [Zha+21] also offering comparisons with other complex models. In comparison, this work used time series models, such as ARMA and VAR, and an XGBoost model that was only trained with time series data of the predicted value as comparison models. The related works, except for [Fan+22], should also give enough attention to the comparison models since they lack comprehensive information on their training.

Three of the four selected works explored ways to find the essential predictors for forecasting. [JMWV24] used SHAP to find the important features and explain the correlation between the features and prediction values. This method was also similar to this work. [Niu20] and [Fan+22] tried to find the important predictors by extracting information from the trained XGBoost model. The XGBoost model offers three ways to show the important features, which are explained in Section 4.2.4. Nevertheless, no clear information about which method was chosen in [Niu20] and [Fan+22].

4

APPROACH

This chapter explains the details of the general steps that were carried out to conduct the experiment. The experiment was divided into two main parts: simulation and forecasting.

4.1 SIMULATION

This section mainly used the BurstGPT benchmarking method to simulate the requests. The details of BurstGPT can be found in Section 2.1.1. This section can be divided further into two steps: generating the simulation dataset and simulation run.

4.1.1 *Generating the Simulation Dataset*

This step aimed to create and save the simulation dataset into Comma-Separated Values (CSV) files. The first thing to do was finding an existing prompt dataset for the prompt sampling. The chosen prompt dataset should be suitable for the use case on which it is tested. If the project had a suitable prompt dataset for its use case from its own system or database, the searching step for another dataset could be skipped.

After finding the suitable dataset, the prompts in the dataset were tokenized to fill the prompt pool. It is crucial to ensure that the tokenizer used for the tokenization process corresponds to the LLM model used in the server. For each request, the number of tokens created from this tokenization process was counted and defined as the request length. The prompt pool was implemented using a dictionary data structure, with the request length as keys and the list of texts with the corresponding length as values. This structure indicates that multiple prompts might have the same request length. Based on the request length information from the prompt pool, the probability for each request length can be defined following the Zipf distribution with parameter θ. Given L different prompt lengths in the prompt pool, the shortest prompt length was labeled with rank 1, the second shortest prompt was labeled with rank 2, and so on, with the longest prompt length labeled with rank L. More detail on how the probability was determined can be found in Section 2.1.2. This probability served as the basis for the prompt sampler to sample the prompt.

The next step was to create a simulation dataset based on the mathematical concepts mentioned in BurstGPT. The simulation dataset in this experiment was directly divided into training, validation, and test data. This approach

can be done since BurstGPT can create arbitrary amounts of requests based on given mathematical distribution parameters.

To create the training data, the number of the simulated burst situation was determined. After that, lists of shape parameters α and scale parameters β of the Gamma distribution were specified for the burst situations. The number of elements in each list is equivalent to the determined number of burst situations. The values of shape parameters α were modeled based on a quadratic function, and the values of scale parameters β were modeled based on a linear function, similar to BurstGPT. The list values for both parameters were ordered, either in ascending or descending order, to simulate gradual changes in the Gamma distribution. A pair of shape and scale parameters were then selected from both ordered lists for each burst situation. The first elements from both lists were selected for the first burst situation, the second elements for the second burst situation, and so on, up to the last elements. Each burst situation could create multiple requests based on these Gamma distribution parameters.

The first step in creating requests for each burst situation was to determine the number of requests to be sent based on discrete uniform distribution. Each request consisted of a request prompt and a waiting time before it was sent to the server. The waiting times for the requests were generated based on Gamma distribution. The generated waiting times were then normalized into a longer given time range in seconds, as using the waiting time directly generated from the Gamma distribution was too short. The time range was also randomly selected based on discrete uniform distribution. After generating the waiting time, the request prompt was sampled for each request using the prompt sampler and prompt pool. The prompt sampler selected a random request length based on the calculated probability following the Zipf distribution. If the selected request length had only one prompt, this prompt was automatically selected for the request. Otherwise, the prompt was sampled from the list of prompts based on uniform distribution.

When the waiting time was specified and the request prompt was selected for each request, the requests for each burst situation were saved in CSV files. Each CSV file only contained information on one burst situation. Each CSV file consisted of three rows with information regarding the request's waiting time, prompt, and prompt length, along with three rows with information regarding which burst situation it belongs to, the shape parameter α, and the scale parameter β.

Even though the previous paragraphs explain the steps for generating the training data, they also apply to the validation and test data, with two key differences. First, the requests in training data had the information on which burst situation they belong to, e.g., Burst 1, Burst 2, etc., where the validation and test data did not. Second, the order of the shape parameter α and scale

parameter β were randomized in validation and test data. This randomized process was done to simulate uncertainty during the simulation.

4.1.2 *Simulation Run*

Feature name	Description
request_start	Point in time the request started based on $t = 0$s.
request_end	Point in time the request ended based on $t = 0$s.
request_duration	The difference between 'request_start' and 'request_end'.
waiting_time	The waiting time before the request was sent.
request_type	Which burst situation the request belongs to, e.g., 'Burst n' (for training data), 'Validation', or 'Test'.
request	Request prompt string sampled from prompt pool.
request_length	The number of tokens of the request prompt.
response	Response string generated from the LLM inference server.
response_length	The number of tokens of the response.
is_ok	Only true, if there was no problem during sending the request and receiving the response string.
exception_type	Type of the exception, when 'is_ok' is false.
shape_gamma	Shape parameter α of the Gamma distribution to generate the corresponding waiting time.
scale_gamma	Scale parameter β of the Gamma distribution to generate the corresponding waiting time.

Table 4.1: Description of the data collected from the simulation run for each request.

This step aimed to test the LLM server based on the simulation dataset generated in the previous section. First, all requests from all burst situations, namely training, validation, and test data, were loaded from the CSV files for the simulation run. The simulation began from a start point in time $t = 0$ seconds and was run by sending requests from each burst situation one at a

time. The following burst situation was only executed if all requests in the current burst situation had finished waiting or receiving the response from the server.

For each burst situation, the requests were sent via HTTP asynchronously and concurrently to the server. Each request waited for the time to send the prompt based on their respective waiting time. After the waiting time was up, the prompt was sent to the LLM inference server using the server's Application Programming Interface (API). The request then waited until it got the response from the LLM server or the request timed out. Each request had only a limited time to wait for the response from the server before it was deemed a failed request. During the simulation run, the metrics generated from the LLM inference server were also recorded at regular intervals of under one minute to capture the server performance information as time series data.

After the simulation with all burst situations was finished, the important response data from the LLM inference server were extracted for each request and saved in tabular format. Table 4.1 shows the description of the extracted information for each request. The collected data from the requests and the server performance metrics were saved in two separate CSV files, which served as the result of the overall simulation process and the raw data for the forecasting process.

4.2 FORECASTING

This section mainly focuses on data preparation and forecasting processes, followed by interpretation of the trained model. This section can be divided further into several steps.

4.2.1 *Data Preparation*

The raw data used for the experiment was based on the two CSV files created from the previous section, namely the requests and LLM server performance metrics data. The data from these files was used for further data preparation processes for the forecasting. Please note that the explanation in this section focuses on the preparation step of the request data with only a general explanation for the server performance metrics since the server performance metrics are relevant specifically to what technology or serving framework is used in the use case. The details of the performance metrics and the data preparation process in this experiment will be described later in Section 5.1 and Section 5.5, respectively.

After loading the request data, the 'request' and 'response' columns were removed since they are not helpful for the forecasting process, and analyzing the generated result quality is outside the scope of this experiment. The

request data is overall a time series data, but not in irregular intervals due to randomness in waiting time. When a request did not receive any response from the server, the value of 'response_length' was null. Therefore, these null values were filled with zeroes.

After filling in the null values, the data was sorted based on the point in time when the requests finished waiting or receiving the response from the server, i.e., the column 'request_end'. Then, a time index was created based on this feature. A new feature was also created by inverting the value from column 'is_ok' and naming it 'fail'.

The next step was to create time series data with regular intervals. The features of the sorted data were aggregated into regular intervals of one minute. For example, requests under 1 minute from $t = 0$s were aggregated together, requests between 1 and 2 minutes from $t = 0$s were aggregated together, and so on. The selected features for the aggregation process with the corresponding aggregation function can be found in Table 4.2.

Feature name	Aggregation function
request_duration	Mean
request_type	Maximum
request_length	Mean
response_length	Mean
fail	Mean

Table 4.2: Request data features selected for aggregation with their corresponding aggregation function.

Notice that after the aggregation process, the column 'fail' contained the failure rate of the requests in every interval with values between 0 and 1. The value 0 indicated that no request failure occurred in the time interval, and the value 1 suggested that all requests failed in the time interval. This feature served as the target variable of this experiment. Other than the request data, the data points of the LLM server performance metrics were also aggregated in one-minute intervals. Information regarding the features selected for the aggregation with their corresponding aggregation functions specific to this experiment can be found in Table 5.2 and will be discussed later in Section 5.5.

The next step of the data preparation process was to check the stationarity of the features in the time series data, as some trained models assume that the time series are stationary. The stationarity property was checked on all features using the augmented Dickey-Fuller test with significance level $\alpha = 0.05$. Information about the Dickey-Fuller test can be found in Section 2.2.1.

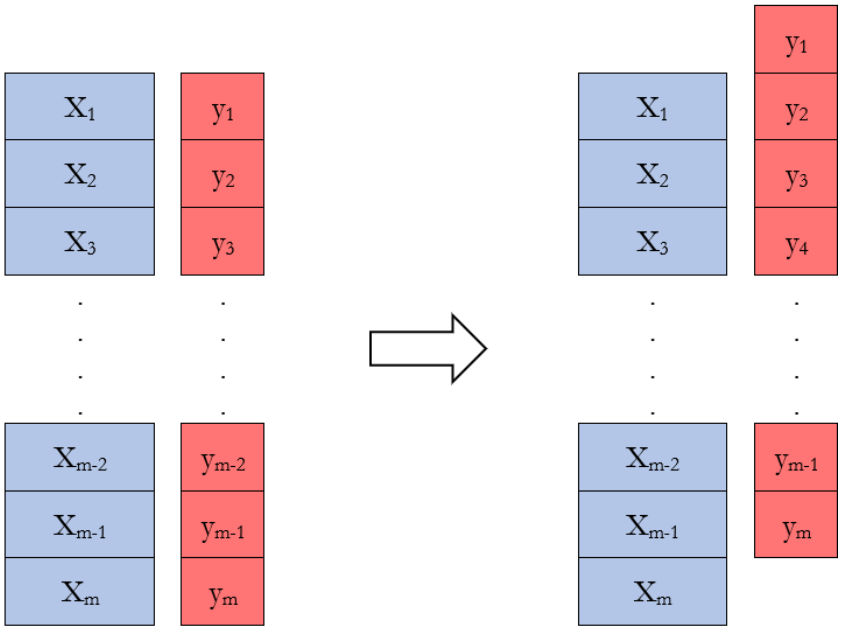

Figure 4.1: Row shifting of the 'fail' column (y column) for forecasting.

As both request and metrics data had the same intervals after aggregation, the data were combined row-wise. For forecasting the failure rate of the next interval, the rows of the 'fail' feature were shifted, which is displayed in Figure 4.1. The idea of row-shifting was to ensure that the machine learning model that does not consider time series components by default can still forecast the future value of the target variable using only known values from the past. All predictors at time t (X_t) were used to forecast the failure rate of requests at time $t+1$ (y_{t+1}). The shifting made the predictors X_t the lag 1 features for the forecasting. The first and the last rows of the shifted data in Figure 4.1 were removed, as they were either missing the predictors or the target variable.

The result of the data processing step was combined time series data with multiple features from request and LLM server performance metrics to forecast the failure rate of requests that did not receive a response from the server.

4.2.2 *Evaluation Metrics*

There were two metrics used to evaluate the trained models in this study, MAE and RMSE. The formulas of the evaluation metrics are:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|, \tag{4.1}$$

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}, \tag{4.2}$$

with n representing the number of observations, y_i representing the true value, and $\hat{y}_i$ representing the predicted or forecasted value. Both metrics have values ≥ 0, with 0 representing the best possible value.

Both metrics could be used simultaneously for evaluating the model performance with test data. However, only one evaluation metric could be used for hyperparameter tuning with training and validation data. The chosen metric must be dependent on the characteristics of the target variable. Hewamalage, Ackermann, and Bergmeir explored this topic further in their work [HAB23].

According to Hewamalage, Ackermann, and Bergmeir, MAE is a measure that optimizes for the median, while RMSE is a measure that optimizes for the mean [HAB23]. It was suggested that MAE and RMSE could be used equally for different types of time series characteristics, except for intermittence and outliers [HAB23]. In intermittent time series where most of the data contain zeroes, RMSE is more suitable than MAE to be chosen as the evaluation metric, as MAE might constantly view zeros, i.e., the median, as the best prediction [HAB23]. On the other hand, MAE is more robust in handling outliers in the time series data in comparison to RMSE since MAE optimizes for the median, which is more robust against outliers [HAB23].

Therefore, the target variable should be examined before deciding the evaluation metric for hyperparameter tuning. If outliers appear in the data, MAE should be chosen. If the data are dominated with zero values, RMSE should be used for evaluation. Otherwise, either evaluation metric can be chosen.

4.2.3 *Modeling*

This step aimed to train machine learning models with the time series data from the data preparation step to find a model with the best forecast. Before starting the training process, the time series data was split into training, validation, and test data. The split was implemented by filtering the rows based on the 'request_type' column. Rows with column values 'Validation' and 'Test' served as the validation and test data, respectively. Otherwise, they were selected as the training data. The column 'request_type' was only used for splitting the dataset and not for forecasting. Therefore, the column was removed after the split.

The main approach for the modeling is to find the best hyperparameter for the model from a set list of possible hyperparameters based on the evalu-

ation with validation data and use the result from hyperparameter tuning to evaluate the test data. First, the possible hyperparameters were determined for the test. Next, the model used one of the hyperparameter combinations to train the model and forecast all values on the evaluation data simultaneously. A suitable evaluation metric was used to measure how good the forecast was on the evaluation data. The training was repeated with all possible hyperparameter combinations, and the hyperparameter combination with the best evaluation metric value was chosen for the test data.

The model was then trained with the training and validation data with the selected hyperparameter combination and evaluated on the test data. The evaluation of the test data was different from that of the validation data. The evaluation of the test data used the rolling origin evaluation [HAB23]. Figure 4.2 and Figure 4.3 illustrate the two different rolling origin evaluation setups: the rolling window and the expanding window.

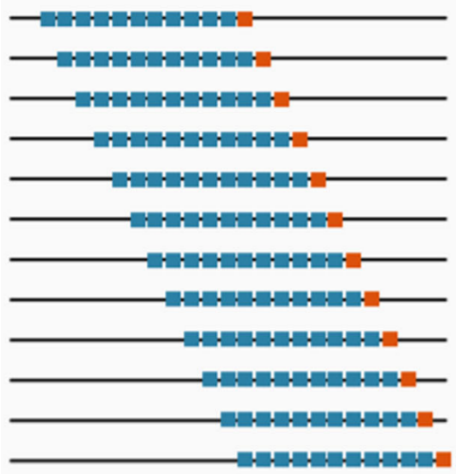

Figure 4.2: Rolling window evaluation setup [HAB23].

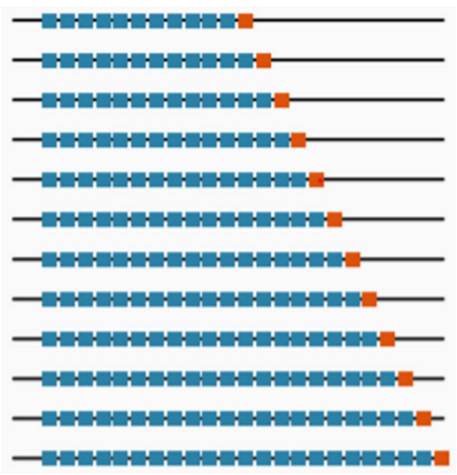

Figure 4.3: Expanding window evaluation setup [HAB23].

The main idea of both evaluation setups was to forecast the next value one step at a time using previous data. The machine learning model used multiple observations until time t, which are marked with blue boxes, to forecast the value at time $t+1$, which are marked with orange boxes. The main difference was the data volume used for the model training. Rolling window used only a fixed amount of data to train the model by adding the newest data and removing the oldest data from the rolling window for every forecasting step while expanding window concatenated the data for every step, which made the number of data used for retraining grow over time. In comparison, the evaluation for hyperparameter tuning used the fixed origin setup, using the same data for every forecasting step [HAB23]. The rolling origin evaluation setup ensures that the forecast errors are not influenced by only specific patterns observable in a particular period [HAB23]. Therefore, the rolling origin evaluation is useful in this experiment, as the result from the model training will be used in production for a longer period of time, which makes

capturing the dynamic in the system over time better than only capturing certain patterns in a certain time period, and the model was retrained for every forecasting step with fresh data.

The primary model used in this experiment was XGBoost, which used multiple features as predictors. More details on the XGBoost algorithm can be found in Section 2.2.2. XGBoost was chosen as the primary model since it is versatile for different use cases and performs well, as shown by the related works in Section 3.2. It is also scalable and fast [CG16], which may be helpful for monitoring the server in production in real time. Temporal features were created to train the XGBoost model since it does not consider time components in the data by default. The XGBoost model could be directly trained using the data from the steps in Section 4.2.1. Nevertheless, the data only contained lag 1 predictors for the forecasting. It also did not contain lagged features from the target variable itself. Therefore, one of the hyperparameters to be tuned for the XGBoost model was the number of lags of the predictors and target variable (p).

Hyperparameter	Description
Learning rate	Shrinkage factor of the leaf weights.
Maximum depth	Maximum depth of an estimator tree.
α (Regularization)	L1 regularization term.
λ (Regularization)	L2 regularization term.
Number of estimator	The number of tree estimators of the XGBoost model.

Table 4.3: Hyperparemeter of the XGBoost model.

Table 4.3 shows the hyperparameters of the XGBoost model used in this experiment. For every possible value of p, all possible XGBoost hyperparameter combinations were iterated to find the best XGBoost model. The XGBoost model in this study used tree models for the estimators. It also used logistic regression for the regression learning objective, as the target variable ranged from 0 to 1. The model also used L1 and L2 regularization mutually exclusively. The suitable maximum tree depth parameter, α regularization parameter, and λ regularization parameter were tuned to control the tree growth for each estimator. The number of estimators was found with early stopping. The early stopping method was chosen, as using a fixed number of estimators during training may accidentally raise the error value on the evaluation data due to overfitting. After the XGBoost algorithm trained a new estimator, it evaluated the model with the evaluation data (in this case, the validation data) and calculated the prediction quality using an evaluation metric. In this experiment, if the model did not improve with a difference of at least 10^{-5} on the evaluation metric after 50 training steps, the model training would end, and the best number of estimators trained was given. This process was done to find the ideal amount of trees based on the evaluation

data.

Three models were also trained as comparisons with the XGBoost model, two linear time series models, and one univariate XGBoost model.

1. *ARMA*. This model used only the target variable for forecasting. The hyperparameters used for the ARMA model were p and q. More information about this model and its hyperparameters can be found in Section 2.2.1.

2. *VAR*. This model could use multiple features to forecast the target variable. This model used the same columns as the XGBoost model for training. However, this model did not use a shifted target variable at the end of the data preparation step for the forecasting. The hyperparameter used for the VAR model was p, which indicates the number of lags.

3. *Univariate XGBoost model*. The training process of this model is similar to the XGBoost model explained above but with one key difference. This model used only the target variable for the forecasting process.

Since the target variable value ranged from 0 to 1 and the output of the linear time series models are real numbers, the target variable was first transformed with a logit function for the training process. The transformed values of 0 and 1 with the logit function were set to -30 and 30, respectively. Those values were chosen since their logistic function outputs are close to 0 and 1, without using an extremely large or small value that may affect the regression process. The model output for the forecasting was transformed again into a probability with a logistic function.

4.2.4 *Interpretation*

After training and finding the best XGBoost model, the last step was to interpret the model and find the important features for the forecasting. Two different approaches were used to find the important features.

The first approach was to use the XGBoost model itself. The XGBoost model offers three ways to define the feature importance using weight, gain, and coverage [Shi22]. *Weight* measures how often a feature appears in the tree estimators, *gain* measures the average training loss decrease using a feature, and *coverage* measures the number of observations impacted by a feature [Shi22]. The feature importances were visualized with a bar graph. This approach considered the training and validation data for interpreting the model in this experiment.

The second approach was to use SHAP values to explain the forecasting result on the test data. More details on how SHAP works can be found in Section 2.2.3. SHAP offers multiple ways to visualize the feature importance,

either for local or global explanation. In this experiment, this approach considered the training, validation, and test data to interpret the forecast of the test data.

Both approaches can be used to find the important features, as there is no concrete definition of interpretability and no standard way to quantify the correctness of an explanation [MCB20]. Nevertheless, this work preferred the SHAP method for finding the important features. The feature importance from XGBoost can be defined in three different ways, which makes it confusing to choose the definitive method, as they may give different results. Also, the feature importance with XGBoost only considered the trained model with training and validation data. SHAP used all the data for the explanation since it interpreted the result of the test data using the model trained with training and validation data. This approach also made SHAP similar to the evaluation with evaluation metrics. Nonetheless, the results of both approaches were still shown in this experiment.

5 IMPLEMENTATION

This chapter covers the implementation details of this study, including the example use case for the experiment, the dataset chosen based on the example use case, technical specifications, simulation and forecasting details specific to this experiment, and the hypotheses related to the research questions.

5.1 USE CASE: CHATBOT PROJECT

It was assumed in this experiment that there was a new chatbot project for English conversations to be tested. The project had no available dataset to test the LLM server and no concrete idea about the pattern of how the requests were sent to the server. This project used the open-source Mistral-7B-Instruct-v0.2 model[1] as the LLM for generating the chatbot responses.

The project also used the vLLM serving framework, which was developed by Kwon et al. [Kwo+23], to run its LLM on an internal server. The vLLM framework is open-source and provides the production metrics[2] generated from the framework for monitoring. The production metrics were pulled every 5 seconds as time series data using Prometheus[3], an open-source monitoring toolkit. Table 5.1 shows the pulled metrics from the vLLM framework in the project.

Each request from a user was given a maximum timeout of 200 seconds. It was also given a limit of 2,048 tokens for the sum of prompt and generated tokens. Therefore, the number of generated tokens was determined by subtracting 2,048 with the number of tokens in the prompt. For prompts with more than 2,048 tokens, the generated output was limited to 10 tokens.

The project members wanted to know if it is possible to forecast the request failure rate based on the available time series data for server monitoring or if more data should be collected. The project members were also interested to know which features are crucial to forecast the failure rate. This study worked based on this example use case for the details of the experiment.

1 https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2, Last accessed: 27 November 2024
2 https://docs.vllm.ai/en/v0.5.3.post1/serving/metrics.html, Last accessed: 27 November 2024
3 https://prometheus.io/, Last accessed: 27 November 2024

Feature name	Description
vllm:avg_generation_throughput_toks_per_s	Average throughput of generated tokens in tokens/second.
vllm:avg_prompt_throughput_toks_per_s	Average throughput of tokens processed for prefill in tokens/second.
vllm:generation_tokens_total	Number of generated tokens in total.
vllm:prompt_tokens_total	Number of tokens processed for prefill in total.
vllm:cpu_cache_usage_perc	Key-Value (KV) cache usage on the CPU.
vllm:gpu_cache_usage_perc	KV cache usage on the GPU.
vllm:num_requests_waiting	Number of requests waiting to be processed until the resource on the GPU is available.
vllm:num_requests_swapped	Number of requests swapped to the CPU.
vllm:num_requests_running	Number of requests processed at the moment on the GPU.

Table 5.1: Description of the metrics pulled from the vLLM framework from the vLLM documentation.

5.2 DATASET

Based on the assumption mentioned above, a conversation dataset was explicitly looked for since the use case was based on chatbot implementation for prompt sampling. Other appropriate datasets should be used for other use cases instead.

The chosen dataset for this experiment was the OASST2 dataset[4], an open-source conversational dataset. The experiment used only the English conversation on the dataset from the training and validation splits. Even though the dataset contains message trees with multiple replies, this experiment only used the prompt from the prompter on the root node of the conversation, i.e., the first prompt in the message tree from the prompter, to simplify the experiment. The list of possible prompts was reduced further by removing prompts with token lengths smaller than ten since they are too short and lack important context. The length of a given prompt was calculated by counting the number of tokens generated by the LLM tokenizer minus one,

4 https://huggingface.co/datasets/OpenAssistant/oasst2, Last accessed: 27 November 2024

as the tokenized text always starts with the token <s> by the chosen LLM.

The result was a list with 4,607 different prompts. Figure 5.1 and Figure 5.2 show the histogram and Empirical Cumulative Distribution Function (ECDF) of the prompt lengths. The prompt length ranged from 10 to 3,085 tokens. Based on both figures, the distribution of the prompt lengths concentrated on prompts with a smaller number of tokens. More than 80% of the prompts had less than 64 tokens, with very few prompts having more than 1,000 tokens.

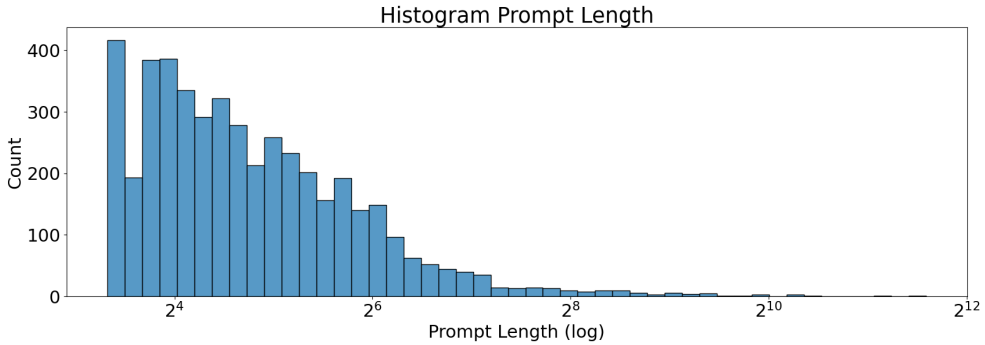

Figure 5.1: Histogram of the prompt lengths with logarithmic scale on prompt length.

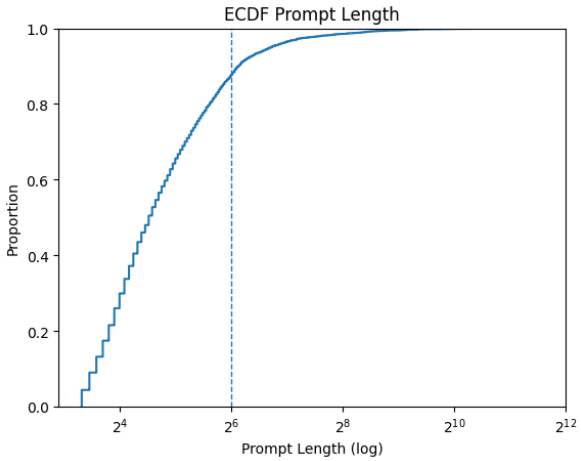

Figure 5.2: ECDF of the prompt lengths with logarithmic scale on prompt length. The dashed line clarifies the proportion on prompt length equals 64.

This prompt list was then used to create the prompt pool and prompt sampler. The list had 247 different prompt lengths, corresponding to 247 keys in the prompt pool. The top five prompt lengths with the most amount of prompts were 11 (212 prompts), 15 (206 prompts), 10 (204 prompts), 13 (195 prompts), and 12 (193 prompts). Many prompt lengths had only one corresponding prompt, mainly prompts with more tokens.

As there were 247 different prompt lengths, the prompt sampler created a Zipf distribution with 247 ranks, with rank 1 corresponding to prompt

length 10, and rank 247 corresponding to prompt length 3,085. The parameter θ for the Zipf distribution can be arbitrarily chosen. Nevertheless, the parameter θ was chosen in this experiment by finding the suitable fit for the prompt length distribution based on Q-Q plot. The chosen value for parameter θ in this experiment was $\theta = 1.38$. The Q-Q plot and the CDF comparison between the theoretical and empirical distribution can be found in Figure 5.3 and Figure 5.4. Based on the figures, the Zipf distribution matched the distribution of prompt lengths with ranks larger than 100. However, the Zipf distribution had a higher probability for shorter prompts compared to the empirical distribution. The differences between both distributions will be discussed further in Section 7.1.

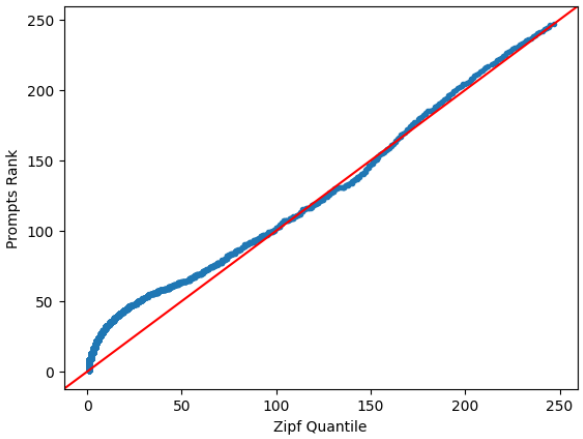

Figure 5.3: Q-Q plot of Zipf distribution with $\theta = 1.38$.

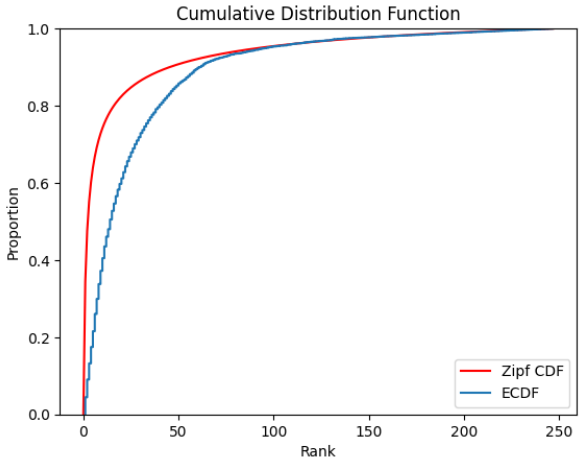

Figure 5.4: CDF comparison between the empirical prompt length data ranks and theoretical Zipf distribution with $\theta = 1.38$.

5.3 COMPUTER SPECIFICATION AND PYTHON MODULES

This experiment used docker infrastructure to simulate the use case. The docker infrastructure consisted of one vLLM and one Prometheus container.

The vLLM container was an OpenAI-compatible server with version 0.5.3.post1, which provided the implementation for OpenAI completion and chat API. During the experiment, the requests were sent using the chat API to the vLLM server. The vLLM container was run on an internal server within PPI AG with default settings. The internal server was equipped with an AMD EPYC 7343 16-core Processor CPU with 129 GB of RAM and an NVIDIA RTX A6000 GPU with 48 GB of VRAM. The Prometheus container with version 2.55.0 pulled the server performance metrics from the vLLM container every 5 seconds during the experiment.

For the technical implementation of this experiment, Python version 3.11.2 was used. The Python modules used in this experiment, along with their versions, were:

- `aiohttp` version 3.9.3
- `datasets` version 2.18.0
- `jupyter` version 1.0.0
- `matplotlib` version 3.8.4
- `notebook` version 7.1.2
- `numpy` version 1.26.4
- `pandas` version 2.2.1
- `scikit-learn` version 1.5.0
- `scipy` version 1.14.0
- `seaborn` version 0.13.2
- `shap` version 0.45.1
- `statsmodels` version 0.14.2
- `transformers` version 4.39.3
- `xgboost` version 2.0.3

5.4 SIMULATION

The settings of the parameter can be arbitrarily chosen for the simulation run. Nevertheless, the chosen settings for the implementation of this experiment were determined to ensure that the simulation could run smoothly on the internal server and did not exceed the server connectivity time limit in the system infrastructure. Based on explanations in Section 4.1.1, the shape parameters α and scale parameters β of the Gamma distribution were modeled with quadratic and linear function, respectively. The quadratic function for modeling the shape parameter was defined as follows:

$$\alpha = 0.07375 \cdot (x_\alpha - 10)^2 + 0.125 \tag{5.1}$$

With $x_\alpha \in [0, 10]$, the shape parameter α ranged from 0.125 to 7.5. The scale parameter β was defined by a linear function $\beta = x_\beta$, with $x_\beta \in [1, 10]$.

For the training data, the domain of x_α was divided into 22 evenly spaced intervals and used as the input for the given quadratic formula. The output of the quadratic function was copied into a new list. Its elements were reversed and concatenated with the original output list. The domain of x_β was also divided into 22 evenly spaced intervals. The list of x_β was also concatenated with its copy without any reversing. The validation and test data were also created in a similar fashion, but with only 11 evenly spaced segments instead of 22 and shuffling the list elements randomly after the concatenation. In total, there were 88 burst situations in the experiment, with 44 burst situations for training data, 22 burst situations for validation data, and 22 burst situations for test data. The number of requests in each burst situation was randomly selected with a discrete uniform distribution ranging from 10 to 2,000 requests. Also, the time range for the normalization was randomly selected with a discrete uniform distribution ranging from 50 to 400 seconds.

During the simulation run, the experiment used `aiohttp` Python library for the HTTP client and the sleep function from `asyncio` to simulate the waiting time. The parameters for the OpenAI chat API were also set to default with temperature 0 for reproducibility. The response from the server was a JavaScript Object Notation (JSON) object with information about the generated response for the request prompt, which was extracted after the simulation into tabular data. The time series data collected by the Prometheus container was also requested at the end of the simulation and saved into a CSV file.

5.5 DATA PREPARATION FOR SERVER PERFORMANCE METRICS

The server performance metrics data contained information from the vLLM server pulled by the Prometheus container. The time interval between the data entries was regular (5 seconds interval) compared to the request data. The data entries were already sorted based on the column 'Timestamp', which was also the time index.

Two new features were created for the server performance metrics since the metrics 'vllm:avg_generation_throughput_toks_per_s' and 'vllm:avg_prompt_throughput_toks_per_s' were deprecated based on the vLLM documentation page[5]. The new features were created by calculating the dif-

5 https://docs.vllm.ai/en/v0.5.3.post1/serving/metrics.html, Last accessed: 27 November 2024

ference between the current row and the previous row value for the features 'vllm:generation_tokens_total' and 'vllm:prompt_tokens_total'. These new features were called 'vllm:generation_tokens_num' and 'vllm:prompt_tokens_num', and both features represented the number of tokens generated and processed for prefill every 5 seconds, respectively.

The server performance metric data was aggregated further into regular intervals of one minute, similar to the request data. The selected features for the aggregation process with the corresponding aggregation function can be found in Table 5.2.

Feature name	Aggregation function
vllm:gpu_cache_usage_perc	Mean
vllm:num_requests_waiting	Mean
vllm:num_requests_running	Mean
vllm:generation_tokens_num	Sum
vllm:prompt_tokens_num	Sum

Table 5.2: Server performance metric features selected for aggregation with their corresponding aggregation function.

5.6 MODELING

The possible hyperparameters for the XGBoost model can be found in Table 5.3. The list applied to both the multivariate and univariate XGBoost models. The number of estimators was set to a large number deemed sufficient to find the suitable amount of trees with early stopping.

Hyperparameter	Values
Number of lags (p)	$1 - 20$
Learning rate	0.001, 0.005, 0.01, 0.05, 0.1
Maximum depth	4, 5, 6, 7, No limit
α (Regularization)	0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 0, 1, 5, 10
λ (Regularization)	0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 0, 1, 5, 10
Number of estimator	100,000

Table 5.3: Values for hyperparameter tuning of the XGBoost models.

For the ARMA model, the possible values for p and q were ranged from 1 to 20. This range was also applied to the possible values of p in the VAR model. This value range was similar to the parameter p in XGBoost training and was chosen to provide an equal opportunity for the comparison with the XGBoost model.

5.7 HYPOTHESES

There are two hypotheses associated with the research questions.

1. The time series data generated from the LLM simulation might be useful in creating a meaningful forecast of the LLM server performance with the XGBoost model. The XGBoost model might also outperform the comparison models based on the evaluation metrics on the test data.

2. The model interpretation methods might be able to find the comprehensible key features essential for forecasting the LLM server performance. The result from the SHAP method might differ from the result based on the XGBoost feature importance method.

6 RESULTS

This chapter presents the experiment's results, including the simulation and forecasting results.

6.1 SIMULATION RESULTS

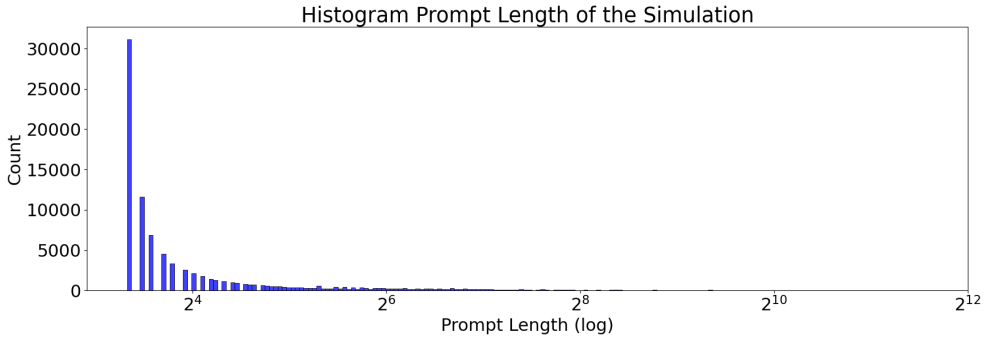

Figure 6.1: Histogram of the prompt lengths of the simulation with logarithmic scale on prompt length.

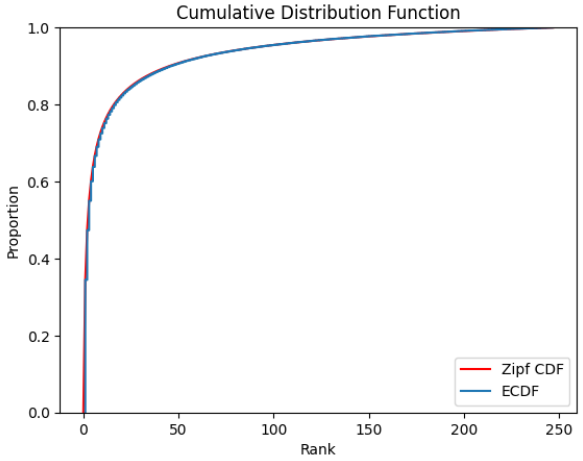

Figure 6.2: CDF comparison between the empirical prompt length data ranks of the sampled prompts and theoretical Zipf distribution with $\theta = 1.38$.

There were 90,322 requests created for all 88 burst situations in the simulation. There were 4,605 unique prompts in all requests, two prompts less than the number of unique prompts in the prompt pool. All different prompt lengths were represented in the simulation. Figure 6.1 shows the prompt length distribution of the simulation, which followed the Zipf distribution with $\theta = 1.38$. This similarity was proven in Figure 6.2, where the ECDF over-

lapped with the theoretical Zipf distribution. Some prompts were used only once in the simulation, and a unique prompt was used at most 181 times. The simulation run lasted 8 hours and 5 minutes.

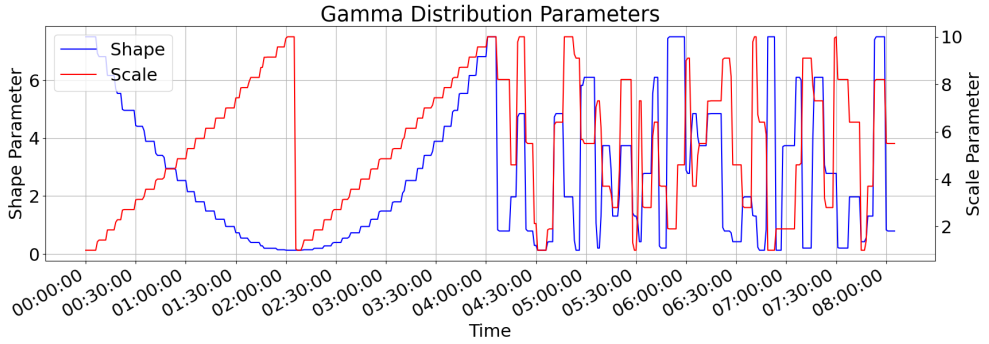

Figure 6.3: Time series of the Gamma distribution parameters.

Figure 6.3 visualizes the progression of Gamma distribution parameters during the simulation. Notice that the parameters were ordered in a certain way in the first half of the simulation and randomized afterward. The training data simulation ran for around 4 hours and 7 minutes, the validation data simulation ran for around 1 hour and 56 minutes, and the test data simulation ran for around 2 hours and 2 minutes.

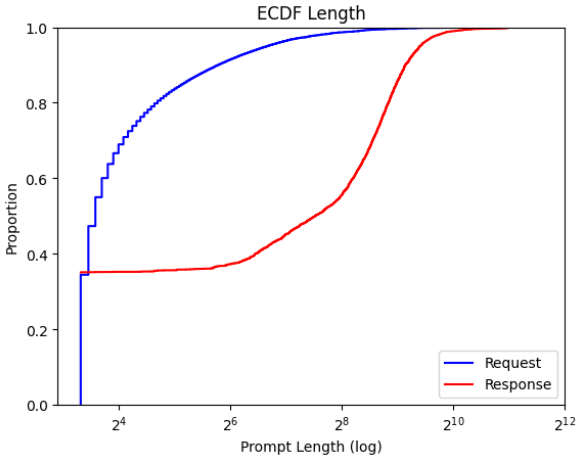

Figure 6.4: ECDF of the request and response lengths with logarithmic scale on prompt length.

Figure 6.4 and Figure 6.5 show the comparison between the request and response length distribution with ECDF and histogram. Notice that on the ECDF plot, the proportion of the response lengths did not start at zero. This jump indicates that around 35.11% of the data was filled with zeros, which indicates the number of failed requests in the simulation. Most of the errors happened due to 'TimeoutError', with only two requests experiencing 'ServerDisconnectedError'. Based on the ECDF and histogram, the distribution of the response lengths was concentrated on longer number of tokens compared to the request lengths.

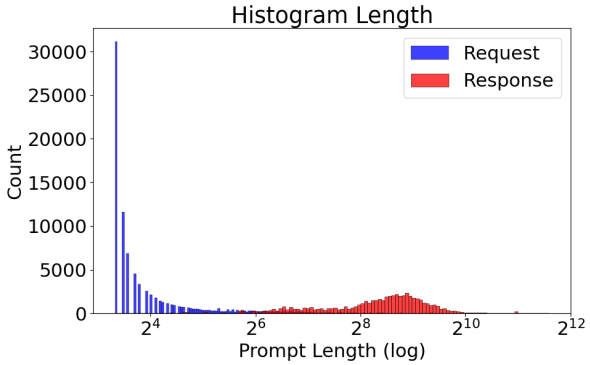

Figure 6.5: Histogram of the request and response lengths with logarithmic scale on prompt length.

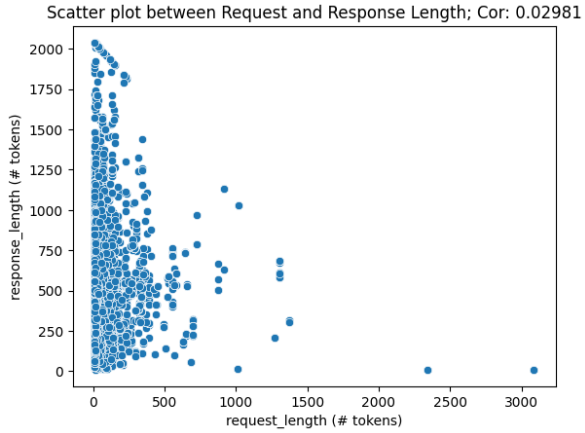

Figure 6.6: Scatter plot of the request and response lengths of successful requests.

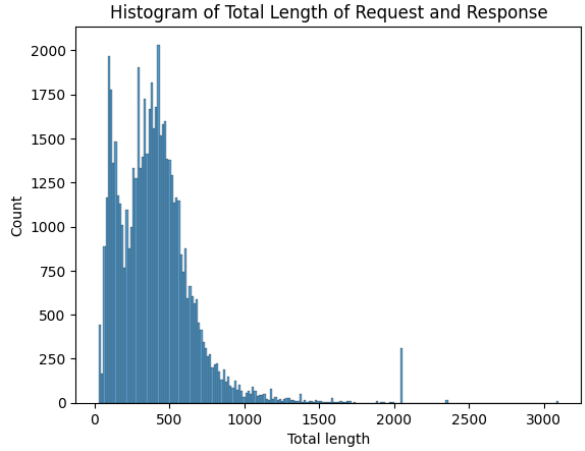

Figure 6.7: Histogram of the total length (request and response length) of successful requests.

Figure 6.6 visualizes the interaction between request and response length of successful requests. The Pearson correlation of both features was 0.02981, which shows almost no linear correlation between both features. However, the maximum limit of tokens generated from the server was influenced by the number of tokens in the prompt. This case can be seen on the top left side of the scatter plot, which shows a downward trend on the maximum amount of tokens generated. Nevertheless, Figure 6.7 shows that this case is only limited. The plot shows the histogram of the total length of successful requests, which is the sum of request and response length. The plot proves that most requests had a total length smaller than the limit of 2,048 tokens, with some requests achieving this limit or more, as shown by the area at and after the peak at the right side of the plot. After the peak are total prompts where the prompt lengths are longer than 2,048 tokens. Both plots show almost no correlation between request and response length, which means the responses were possibly generated based on the prompt context.

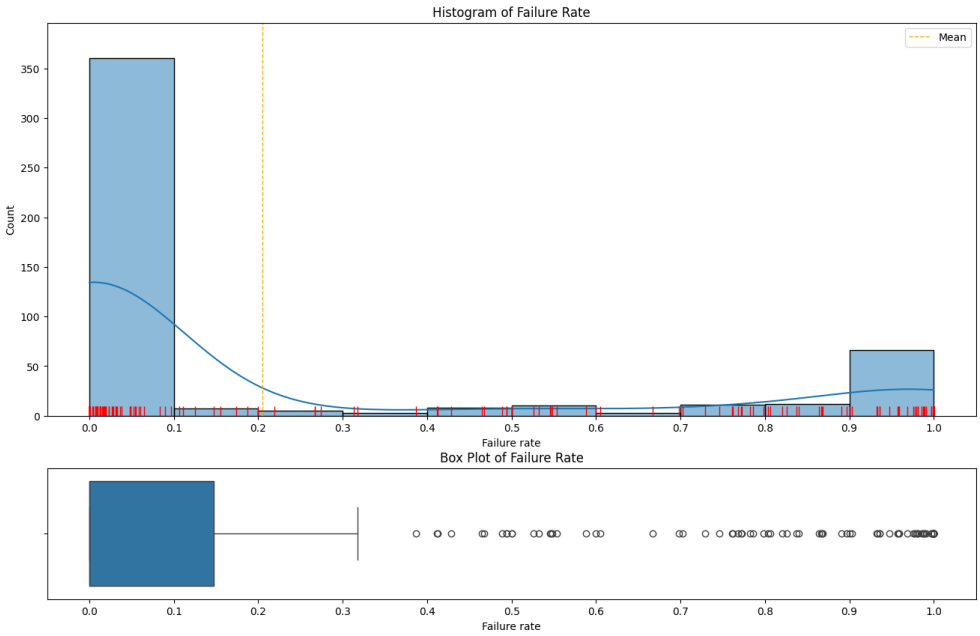

Figure 6.8: Histogram and box plot of the 'fail' time series.

Figure 6.8 and Table 6.1 show the distribution and statistical properties of the target variable 'fail' time series. Based on the information, the target variable is an intermittent time series, as almost 65% of the data were zeroes. This property can be seen on the histogram and box plot of the 'fail' time series. The histogram shows a high frequency around the value 0.0, where the box plot shows that 0.0 is the median of the data. Both plots also illustrate a quite high frequency of values around 1.0, indicating high failure rates of processing the requests. This property makes RMSE the better option for the evaluation during hyperparameter tuning based on criteria in Section 4.2.2. Figure 6.9 shows the bar plot of the top 20 request lengths with the highest and lowest failure rates. Based on the bar plot, there was no indication if

Statistic	Value
Mean	0.20497
Standard deviation	0.36891
Min	0.00000
25%-Quantile	0.00000
Median	0.00000
75%-Quantile	0.14706
Max	1.00000

Table 6.1: Statistical properties of the 'fail' time series.

shorter or longer request prompts caused a higher or lower failure rate.

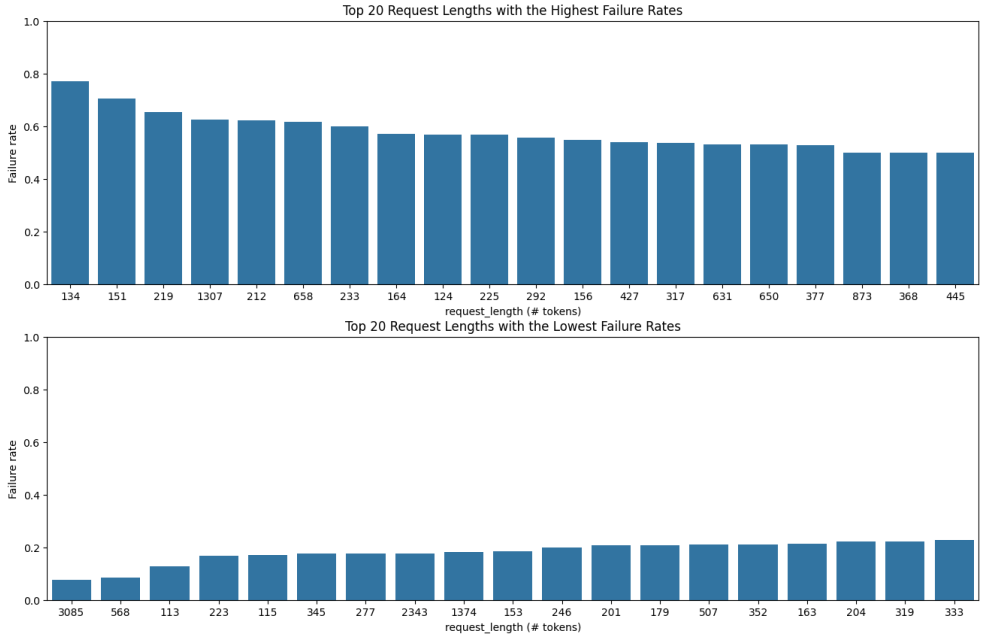

Figure 6.9: Bar plot of the top 20 request lengths with the highest and lowest failure rates, respectively.

Table 6.2 shows the p-value of the augmented Dickey-Fuller test of all features. Based on the p-values, the null hypothesis was rejected in all features, including the transformed target variable 'fail' with logit function. The result means that all features were stationary.

Appendix A contains the time series visualizations of each possible predictor with the target variable 'fail'. Based on Figure A.6 and Figure A.7, more tokens were generated than tokens processed for prefill. Since the values of both 'vllm:generation_tokens_total' and 'vllm:prompt_tokens_total' rose over time, both features were not used for forecasting directly but by creating derived features from them. Figure A.8 and Figure A.11 show that the

Feature name	p-Value
request_duration	$6.30224 \cdot 10^{-16}$
request_length	0.00000
response_length	$2.47135 \cdot 10^{-13}$
fail	$7.43469 \cdot 10^{-15}$
fail (logit)	$1.03110 \cdot 10^{-11}$
vllm:gpu_cache_usage_perc	$5.72207 \cdot 10^{-16}$
vllm:num_requests_waiting	$1.59305 \cdot 10^{-20}$
vllm:num_requests_running	$4.21031 \cdot 10^{-20}$
vllm:generation_tokens_num	$1.17537 \cdot 10^{-15}$
vllm:prompt_tokens_num	$1.60299 \cdot 10^{-20}$

Table 6.2: p-Value of the augmented Dickey-Fuller test of all features ($\alpha = 0.05$).

CPU was not utilized during the experiment. Therefore, the features 'vllm:cpu_cache_usage_perc' and 'vllm:num_requests_swapped' were not used as predictors.

6.2 FORECASTING RESULTS

Hyperparameter	Multivariate	Univariate
Number of lags (p)	2	3
Learning rate	0.05	0.1
Maximum depth	5	4
λ (Regularization)	0	0.05
Number of estimator	101	80

Table 6.3: The chosen hyperparameter of the XGBoost models.

Table 6.3 shows the chosen hyperparameters for the multivariate and univariate XGBoost model. The hyperparameter tuning result determined that the L2 regularization was chosen for the objective. The chosen hyperparameters for the ARMA model were $p = 19$ and $q = 15$. Meanwhile, the hyperparameter for the VAR model was set to $p = 10$.

Table 6.4 shows the forecasting quality of each model using the rolling origin evaluation setup. The bolded numbers indicate the lowest value of the evaluation metric in each column, which implies the best performance of all models. The XGBoost model outperformed all the comparison models on rolling and expanding window evaluation based on the MAE and RMSE metrics. The rolling window evaluation was especially more suitable for the

XGBoost evaluation than the expanding window, as it had the best MAE and RMSE value between the two evaluation methods.

Model	Rolling		Expanding	
	MAE	RMSE	MAE	RMSE
XGBoost (multivariate)	**0.07900**	**0.17835**	**0.08792**	**0.18668**
ARMA	0.17644	0.37356	0.17004	0.36475
VAR	0.11336	0.28558	0.11261	0.28718
XGBoost (univariate)	0.17223	0.29040	0.16625	0.28541

Table 6.4: The performance of the models on the test data.

The multivariate models (XGBoost and VAR) also performed better in general than the univariate models (ARMA and univariate XGBoost). The difference between the best MAE value of the multivariate XGBoost and VAR model was small, with a difference of 0.03361. However, the difference between the best RMSE values was 0.10723, which is bigger than the difference in MAE. This result suggested that the forecasts from the XGBoost model were more similar to the true value, as RMSE penalizes larger differences harder with the quadratic equation than MAE. This case also happened for the univariate model, as the difference of MAE and RMSE between the ARMA and univariate XGBoost model was 0.00379 and 0.07934, respectively. The table implies that the univariate XGBoost model was the best univariate model in this experiment.

However, based on the visualization of the forecasted and the true value, the univariate XGBoost performed worse than other models. Based on Figure B.4 and Figure B.8, the model constantly created forecasts of more than zero when the true values were zero. This prediction may cause false alarms when forecasting the LLM server performance. Other models did not have this problem. The forecast visualization of all models can be found in Appendix B.

A new XGBoost model was trained as the representative model to interpret the result since the rolling origin evaluation setup created multiple models for the forecasts, which makes it hard to choose one model as the representative for the interpretation. The new model was trained with fixed origin, which means training the model on the training and validation data with the chosen hyperparameter to forecast all test data simultaneously. This new model had an MAE value of 0.09163 and an RMSE value of 0.20099, almost similar to the best multivariate XGBoost forecast, with a difference of 0.01263 and 0.02264 for MAE and RMSE, respectively.

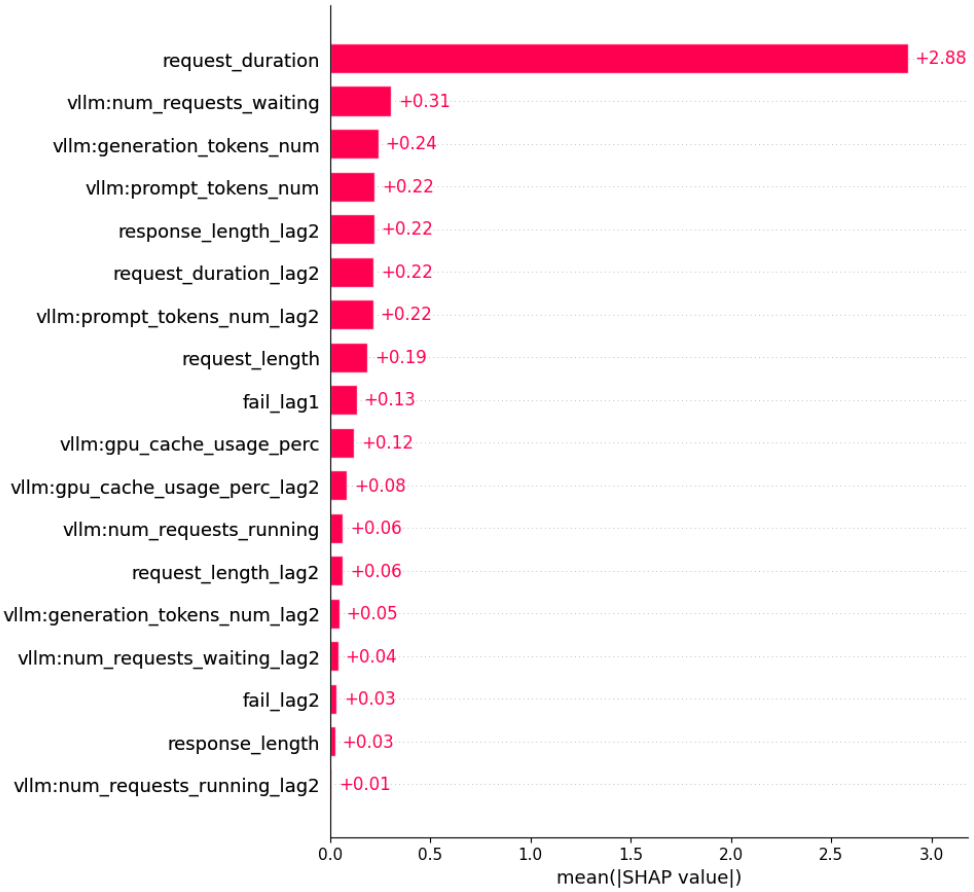

Figure 6.10: SHAP bar plot.

Figure 6.10 shows the SHAP bar plot from the representative model for interpretation, sorted in descending order based on the mean of the absolute SHAP value. The mean of the absolute SHAP value indicates the average influence of a feature for forecasting the data points in test data. There were features that did not include the lag information and features that included this information with suffixes "_lag1" and "_lag2". The features without this information were lag 1 features by default due to the row shifting shown in Figure 4.1. Based on the plot, the feature 'request_duration' outperformed all features with a considerable margin based on the mean of the absolute SHAP value, followed by 'vllm:num_requests_waiting' and 'vllm:generation_tokens_num'. This result implies that the feature 'request_duration' was by far the most important feature for the forecast based on SHAP value, with a mean value of 2.88. The mean value of this feature was 2.57 larger than the second most important feature, 'vllm:num_requests_waiting'. The top 10 most important features based on the plot were dominated by features with lag 1 (7 features), followed by features with lag 2 (3 features), which indicates that information from the last minute is more important for forecasting.

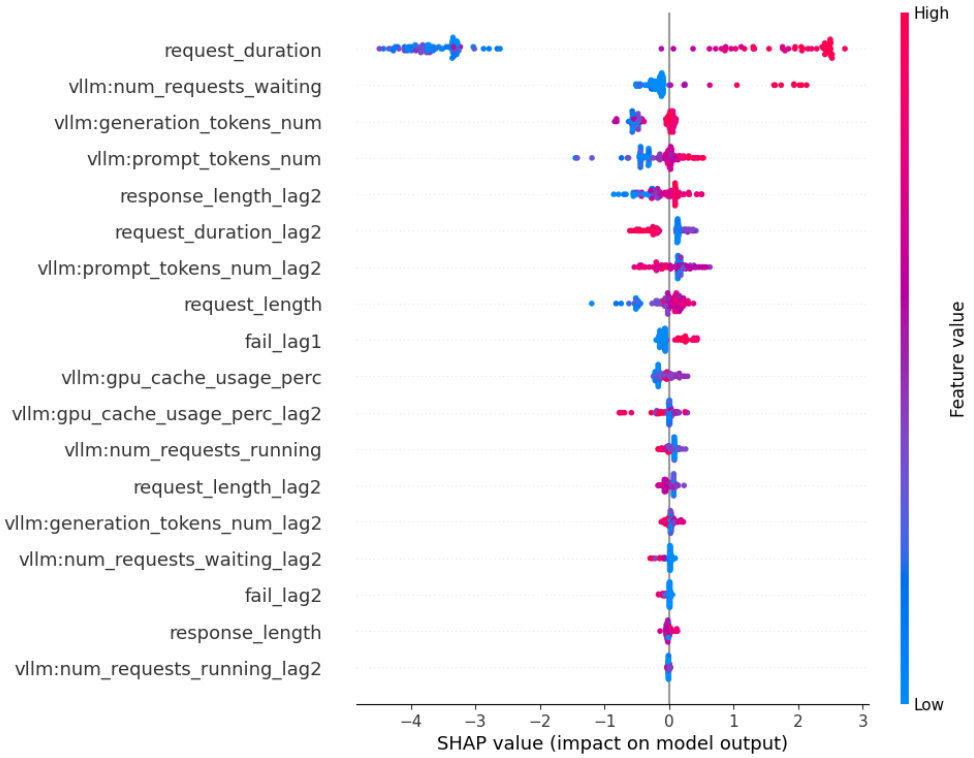

Figure 6.11: SHAP beeswarm plot.

The interaction between the SHAP values and the feature values can be seen with SHAP beeswarm plot in Figure 6.11. For each feature, each point represents an observation of the test data. Each point is plotted based on their SHAP value for the given feature and colored based on the value of the given feature. The color red means a high feature value, and the color blue represents a low feature value. The feature 'request_duration' had the largest dispersion of the observations' SHAP values. The relationship between the SHAP value of 'request_duration' and its feature value was directly proportional, where a high 'request_duration' value means a high SHAP value and vice versa. This relationship may imply that the higher the average request duration for a given time interval of one minute, the higher the failure rate for the given requests of the next time interval. Since this feature was by far the most important based on SHAP values, this information can be solely used to give an alert by the system if the value of this feature exceeds a particular value. This alert system can be implemented if the system infrastructure is not capable of running a machine learning pipeline for real-time forecasting and, therefore, using a simple alerting system as a solution. The threshold value for alerting can be determined using the scatter plot by SHAP, which is shown in Figure 6.12.

Similar to the beeswarm plot, each dot represents an instance in the test data. Each point was plotted based on the SHAP value and the feature value of 'request_duration' on the scatter plot. The red dots indicate instances with

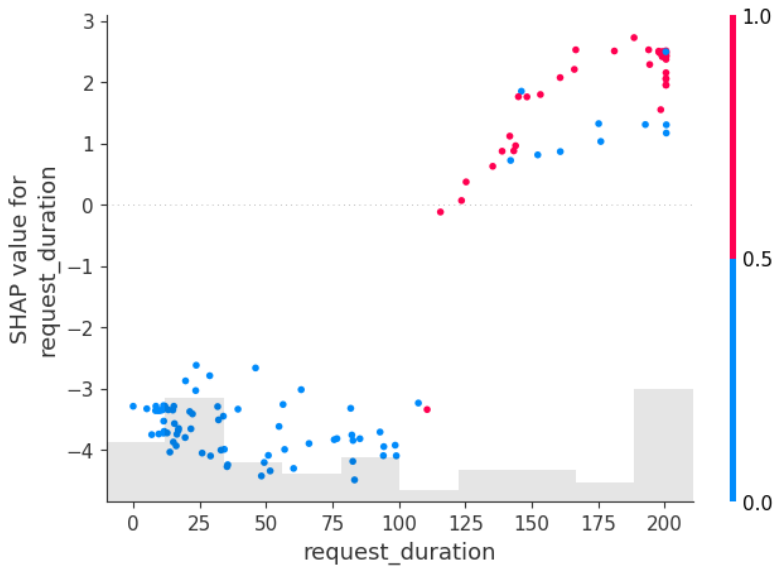

Figure 6.12: Scatter plot between the SHAP value and feature value of 'request_duration'. The red dots indicate instances with a failure rate (true value) of more than 0.0, and the blue dots indicate instances with a failure rate (true value) equal to 0.0.

a failure rate (true value) of more than 0.0, and the blue dots indicate instances with a failure rate (true value) equal to 0.0. Based on this plot, instances with an average request duration of more than 125 seconds had a higher SHAP value and failure rate. This result indicates that when the request duration of a particular one-minute interval exceeds 125 seconds, the LLM system should give an alert to notify a possible chance of a higher failure rate in the next minute in production.

Other features only made minor contributions to the forecast and, therefore, are not used for alerting to avoid confusion. Based on Figure 6.11, features such as 'vllm:num_requests_waiting', 'vllm:generation_tokens_num', 'vllm:prompt_tokens_num', 'response_length_lag2', and 'request_length' were similar to 'request_duration', where the relationship between the SHAP value and its feature value was directly proportional. On the other hand, this relationship in 'request_duration_lag2' and 'vllm:prompt_tokens_num_lag2' was inversely proportional, where a high 'request_duration' value means a low SHAP value and vice versa. This relationship may imply that after a high feature value occurs, the LLM server may regain control to normally process the incoming requests after two minutes. Other features after 'request_length' either did not have a considerable dispersion or a high mean of the absolute SHAP value.

The SHAP value interaction between the features for each instance can be shown using heatmap in Figure 6.13. The color in the plot represents the SHAP value of each feature, with the color red meaning a high SHAP value and the color blue representing a low SHAP value. The darker the color, the

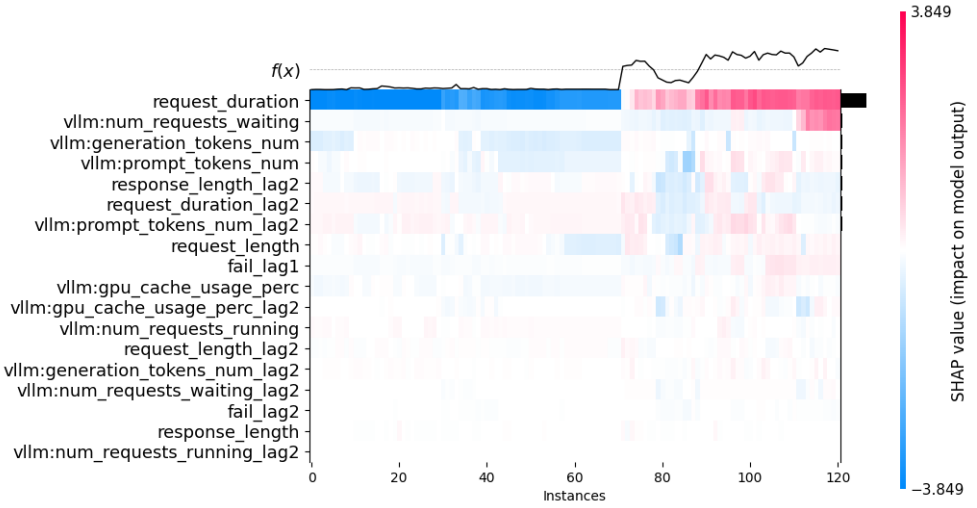

Figure 6.13: SHAP heatmap plot.

higher the absolute SHAP value of that particular feature. The term $f(x)$ at the top of the heatmap represents the corresponding model output of a particular instance. Based on the plot, the feature 'request_duration' heavily influenced the forecast of the failure rate for each time interval, whereas other features only made minor contributions to the forecast.

Similar to the SHAP values, the XGBoost feature importance method also determined that 'request_duration' was the most important feature for forecasting. This result was consistent on weight, gain, and coverage methods. The three feature importance plots from XGBoost with weight, average gain, and average coverage methods can be found in Appendix C. However, the ranks of the important features other than 'request_duration' were inconsistent on all plots. For example, features that were used more often to create a split did not mean that they had a high average gain value, e.g., 'vllm:gpu_cache_usage_perc' and 'vllm:generation_tokens_num_lag2', and vice versa, e.g., 'vllm:prompt_tokens_num'. Also, features that were used more often to create a split and had a high average gain value did not imply that they had a high average coverage value, e.g., 'vllm:gpu_cache_usage_perc_lag2', and vice versa, e.g., 'vllm:generation_tokens_num'. These inconsistencies between the feature ranks made it difficult to choose which method should be used to interpret the model. The complete ranking order of the SHAP and XGBoost feature importance method can be found in Table 6.5.

When considering the top 10 most important features based on the XGBoost feature importance with weight method, seven of those features were also in the top 10 most important features based on the mean of the absolute SHAP value. With the same comparison, the number of those features rose to eight and nine with gain and coverage method, respectively. Based on these

comparisons, the coverage method was the most similar to the SHAP value method.

Feature	SHAP	XGBoost (weight)	XGBoost (gain)	XGBoost (coverage)
request_duration	1	1	1	1
vllm:num_requests_waiting	2	3	6	7
vllm:generation_tokens_num	3	12	14	6
vllm:prompt_tokens_num	4	10	2	2
response_length_lag2	5	8	5	9
request_duration_lag2	6	11	7	12
vllm:prompt_tokens_num_lag2	7	9	8	3
request_length	8	2	3	5
fail_lag1	9	15	10	8
vllm:gpu_cache_usage_perc	10	5	13	10
vllm:gpu_cache_usage_perc_lag2	11	6	4	15
vllm:num_requests_running	12	13	11	14
request_length_lag2	13	14	9	17
vllm:generation_tokens_num_lag2	14	4	16	16
vllm:num_requests_waiting_lag2	15	16	15	11
fail_lag2	16	18	12	13
response_length	17	7	17	4
vllm:num_requests_running_lag2	18	17	18	18

Table 6.5: Feature importance rankings for each interpretation method.

7 DISCUSSION

Based on the results from the previous chapter, the research questions of this study can be answered and the hypotheses can be confirmed. The result from Table 6.4 can be used to answer the first research question: the time series data generated from the LLM simulation can be used to create a meaningful forecast of the LLM server performance using an XGBoost model. The XGBoost model achieved the lowest MAE and RMSE metrics values compared to all the comparison models on rolling and expanding window evaluation. Since the LLM simulation was based on BurstGPT [Wan+24], which reflects a real-world workload of an LLM serving system, the result means that the data collected from the server and the requests can be used for the use case in production to forecast the LLM server performance, i.e., failure rate of the request for a given time interval. The chosen XGBoost model with rolling window evaluation setup outperformed all other comparison models with an MAE value of 0.07900 and an RMSE value of 0.17835, which confirmed the first hypothesis. The VAR model was the closest model to challenge the XGBoost model, with a close score on MAE but a more significant difference on RMSE. The chosen XGBoost hyperparameter from the result should be used according to the findings for the use case in production as a starting point to forecast the failure rate of the next time interval by retraining a new XGBoost model with new data in a fixed time window.

The second research question can be answered based on the model interpretation explained in Section 6.2 using SHAP values: the key feature essential for forecasting the LLM server performance can be found based on the representative XGBoost model and the generated time series data with SHAP values. This representative model was chosen with an almost similar performance to the rolling window evaluation setup based on MAE and RMSE metrics. The most important feature based on the SHAP values from the XGBoost model and the generated data was 'request_duration'. This result was also consistent with the XGBoost feature importance method. This similarity makes sense, as the feature was used the most for creating a split and had the highest average gain and coverage value in the trained XGBoost model. Therefore, this feature was crucial for the forecast of the test data. Even though the result was consistent for the feature 'request_duration' that it was ranked first for the importance of both methods, both methods had different ranking order for other features, which confirmed the second hypothesis. The ranking order of both methods can be found in Table 6.5. With visualizations based on the SHAP method, the relationship between the SHAP and the feature values can be examined, which in turn may be helpful in understanding the LLM server performance behavior based on the feature

values. For example, the SHAP scatter plot was used to determine the threshold for 'request_duration' to send an alert about high failure rate. This value may be used for the use case in production to allocate more resources and prevent request failure, if the request duration during a certain time interval exceeds the threshold. Nevertheless, using this feature alone for the forecasting process may be insufficient. A new XGBoost model was trained using only 'request_duration' as the predictor from the same data as the trained XGBoost model in Section 6.2. By applying the same evaluation setup and metrics on the test data, the best result for the forecasting was 0.16752 for MAE and 0.27274 for RMSE, using the expanding window evaluation setup. The differences with the best XGBoost forecast from Table 6.4 were 0.08852 and 0.09439 for MAE and RMSE, respectively, a notable performance degradation. This result indicates that even though 'request_duration' is the most important feature for the forecasting process, this feature is just a part of all the predictors, whereas other features also contribute to the forecasting. Furthermore, these interpretation results must also be monitored carefully in production since the correlation obtained from the SHAP values does not guarantee causal interpretation [MCB20].

Since the simulation setup in this experiment was similar to BurstGPT [Wan+24], the differences with other benchmarking setups from Section 3.1 also apply to the setup in this work. This work did not use the evaluation processes from the three benchmarking methods because the main focus of this work was the simulation, not the scoring method. More information about the differences between the benchmarking methods of the selected related works can be found in Section 3.1.1. Even though this work mainly used BurstGPT to simulate the requests, this study had two differences with BurstGPT regarding the simulation implementation. The first difference was the separation between creating the simulation dataset generation and the simulation run. BurstGPT does not separate both processes, while this study first saved the generated requests in a CSV file and then ran the simulation. The first reason for this separation in this work was to ensure that the simulation may run smoothly without any intervention of additional workload, for example, giving the CPU control back and forth between sending the requests after waiting and selecting prompts for new requests with the prompt sampler. Instead, the simulation run only focused on sending requests and receiving responses. The second reason was to give the possibility to check the distribution of the sampled prompts before running the simulation instead of waiting for the simulation to be over first. The second difference between this simulation and BurstGPT was additional randomness in the number of requests and duration per burst situation. This randomness was added during the simulation dataset generation to test the LLM server with additional uncertainty.

The result from this experiment also coincides with the result of all related works in Section 3.2 for forecasting time series data. Like all of the selected re-

lated works, the XGBoost model outperformed all comparison models based on the chosen evaluation metrics. Unlike all related works regarding forecasting time series data, this work used row shifting in the data preparation process. The row shifting was done to forecast the target variable based on lagged features, similar to linear time series models. The related works did not mention this method specifically. Nevertheless, it was assumed that by creating lagged features, three of the four related works already considered using them to forecast future values. The usage of the row shifting method in this study was also possible due to the type of temporal features used for the forecasting. Three out of four related works created both the lagged features and temporal information, e.g., month, day of week, up to the minute of the entries. This work used no temporal information, as the simulation run was not long enough due to the server connectivity limitation in the system infrastructure. Therefore, the temporal information features would have insufficient attributes if created. For example, Zhang et al. created a feature containing the day of week information [Zha+21]. This feature was possible to use, as the data has a long period of several months, where day of week information can be crucial to forecast the sales. However, the experiment only ran for around eight hours in this study. Adding the similar day of week information as a feature may result in a feature with one attribute value. Adding the hour information as a feature, for example, only creates eight out of the 24 possible attributes in the feature. Hence, shifting the row of the target variable when temporal information columns are available may result in errors in the forecasting due to the mismatch between them. Since this study only used lagged features, shifting the row of the target variable was the simple solution to ensure that the lagged values were used to forecast the future value.

The modeling in this study separated validation and test data for evaluation. The other related works used only two sets for training and evaluation. These three splits were created in this experiment to ensure that the trained model does not overfit and performs better when encountering unknown data. Similar to the majority of the related works, the evaluation also used MAE and RMSE for the test data. Furthermore, this work considered the intermittent time series characteristic of the target variable to choose RMSE as the suitable metric for hyperparameter tuning, compared to the other related works, which only mentioned the metrics for test data evaluation. This work also showed the forecast visualizations to check the forecast quality. With these visualizations, it was determined that the univariate XGBoost model performed the worst. This result proved that using the lagged values from the failure rate alone as predictors was insufficient. However, forecast visualizations should be used carefully since they might be misleading and thus should be accompanied by error measures for evaluation [HAB23]. [JMWV24] and [Fan+22] also showed the forecast visualization in their works. In the related works, the model used as comparison models also included linear models. These models were either traditional linear

models (linear regression, logistic regression, ridge regression) or univariate linear time series models (ARIMA, seasonal ARIMA). This work also used the ARMA model as the univariate linear time series model. Compared to those works, this study included the VAR model to compare the performance of the XGBoost model with a linear time series model that considered multiple features for the prediction. This work also used both SHAP and XGBoost feature importance methods to interpret the forecasting result, compared to the other related works that used either only one of the methods or none. The results from both methods were also compared in this study, and the plots based on the SHAP values were also used to find important information that might be useful in production.

7.1 EMPIRICAL DISTRIBUTION FOR PROMPT SAMPLING

As mentioned in Section 4.1.1, the sampled prompts were determined based on Zipf distribution of the request length, according to Wang et al. in their paper [Wan+24] for their benchmarking setup. If the number of requests created for the simulation is larger than the number of unique prompts for the sampling, there is a chance that double occurrences happen on at least one prompt. This scenario happened in this experiment. Instead of using a more complex approach for prompt sampling, it is possible and more straightforward to sample the prompts directly from a list of possible prompts multiple times, i.e., using the empirical prompt length distribution of the prompt list. This other method raised the question of how different distributions for prompt sampling may impact the simulation outcome. Thus, the simulation was run once more with the empirical distribution, and the simulation result was compared with the one from Zipf distribution.

The CSV files of the generated simulation dataset with Zipf distribution were used as the input for testing the empirical distribution. The information in all CSV files was modified by changing the prompt in each request with a new prompt, which also included changing the prompt length information. Other information in each request remained the same, such as the request's waiting time, type, and Gamma distribution parameter. This step was possible due to the separation between the simulation dataset generation and the simulation run. Given a list of all possible prompts where each prompt had the same probability to be sampled, a new prompt was selected randomly for each request without returning them to the list. This process was done one at a time until the list was empty. If the list was empty and there were still available requests for sampling, the list was filled again with the same possible prompts, and the sampling process was resumed. The sampling was done until all requests in all burst situations obtained a newly sampled prompt. The prompt length distribution of the newly sampled prompt should be similar to the prompt length distribution in the dataset. Given the scenario mentioned in the previous paragraph with n unique prompts in the dataset and N requests in all burst situations for the simulation, each

unique prompt was sampled at least $\lfloor \frac{N}{n} \rfloor$ times. Some unique prompts were sampled $\lfloor \frac{N}{n} \rfloor + 1$ times if N is not the multiple of n.

The modified requests were then saved into multiple CSV files, similar to the process in Section 4.1.1. Following that, the simulation run and data preparation were performed as explained Section 4.1.2 and Section 4.2.1, respectively. The resulting features from the data preparation of both distributions were then compared. The features included the predictors and the target variable. For each feature from the predictors and target variable of the simulation result based on Zipf distribution, the identical counterpart from the simulation result based on the empirical distribution was chosen for the comparison using the Kolmogorov-Smirnov test with significance level $\alpha = 0.05$. This test was used to examine if the distributions of both features were different, which meant the null hypothesis was rejected. Additionally, each feature of both distributions was also compared visually with Kernel Density Estimation (KDE) and ECDF plots.

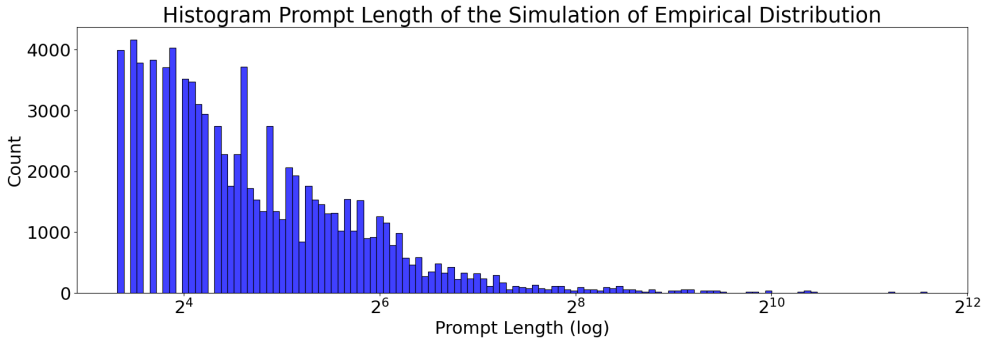

Figure 7.1: Histogram of the prompt lengths of the simulation of the empirical distribution with logarithmic scale on prompt length.

Figure 7.1 shows the sampled prompt length distribution of the new simulation, which follows the prompt length distribution of the prompt dataset. Due to the new sampling implementation, all unique prompts were used in all requests. Given 90,322 requests and 4,607 unique prompts in the simulation dataset, each unique prompt was sampled at least 19 times, with some of them sampled 20 times. Compared to the simulation with Zipf distribution, the new simulation ran 8 hours and 28 minutes, 23 minutes longer than the simulation with Zipf distribution.

The comparison between the request and response length distribution of the empirical prompt length distribution was visualized with ECDF plot and histogram in Figure 7.2 and Figure 7.3, respectively. Based on the ECDF plot, the ECDF of the sampled request prompts was similar to the ECDF of the prompt length list they were sampled from in Figure 5.2. The ECDF of the response lengths indicated that the number of failed requests of the simulation with empirical distribution was higher than with Zipf distribution. This number was close to 42.95%, an almost 8.00% difference with the Zipf distribution. The longer simulation duration of the empirical distribution might

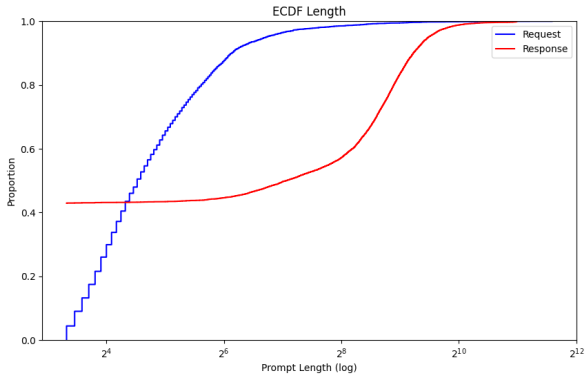

Figure 7.2: ECDF of the request and response lengths of the empirical distribution with logarithmic scale on prompt length.

be due to the higher failure rate. A request was deemed a failure if it did not receive a response from the LLM server, mainly due to timeout. The timeout was set at 200 seconds, which meant more requests waited longer for the server response, making the simulation run longer.

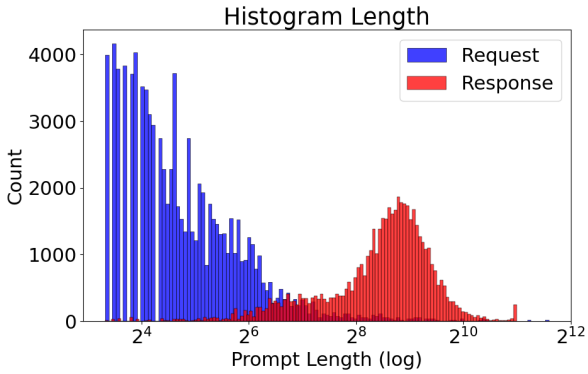

Figure 7.3: Histogram of the request and response lengths of the empirical distribution with logarithmic scale on prompt length.

The p-value of the Kolmogorov-Smirnov test and the visualizations of all features can be found in Table 7.1 and Appendix D, respectively. The distributions in five out of nine features were considered different based on the Kolmogorov-Smirnov test, where the null hypothesis was rejected. The difference makes sense for the features 'request_length' and 'vllm:prompt_tokens_num' since the distribution of the sampled prompts differed. The KDE comparison of both features (Figure D.3 and Figure D.17) shows that the feature values from Zipf distribution tended to be smaller than the empirical distribution. The sampling method using Zipf distribution favored shorter prompts to be sampled than longer prompts, compared to using the empirical distribution. This tendency can be seen in Figure 5.4, as the theoretical Zipf distribution has a higher proportion on shorter prompts than the empirical prompt length data ranks.

Feature name	p-Value
request_duration	0.07044
request_length	**0.00000**
response_length	**0.00000**
fail	0.15649
vllm:gpu_cache_usage_perc	**0.00000**
vllm:num_requests_waiting	0.31214
vllm:num_requests_running	**0.00114**
vllm:generation_tokens_num	0.15156
vllm:prompt_tokens_num	**0.00000**

Table 7.1: p-Value of the Kolmogorov-Smirnov test of all features ($\alpha = 0.05$). The bolded numbers indicated p-values lower than the significance level α.

The empirical and Zipf distributions also differed in feature 'vllm:gpu_cache_usage_perc'. This difference might be due to the difference in the number of tokens processed for prefill, i.e., 'vllm:prompt_tokens_num', where the distribution of the number of generated tokens ('vllm:generation_tokens_num') remained the same on the empirical and Zipf distributions. The KV cache usage on the GPU in the simulation with the empirical distribution tended to be higher than with Zipf distribution based on Figure D.9. The distributions in 'response_length' were also varied, with more shorter and less longer responses on the empirical distribution (see Figure D.5). This might be due to more failed requests happening during the simulation with the empirical distribution, which had a response length of zero. The simulation result for the feature 'fail' with the empirical distribution was identified as similar to the one from the Zipf distribution. This result indicated that the distribution of failure rates in each time interval was similar in both prompt sampling distributions.

Based on the distribution comparison results, there were some similarities and differences between the empirical and Zipf distribution for prompt sampling. Nevertheless, these differences might impact forecasting, interpretation, and resulting parameters that can be used in production. Both distributions can be helpful for prompt sampling depending on the assumption of the simulation. Zipf distribution was chosen by Wang et al. based on their analysis of the BurstGPT dataset [Wan+24]. The experiment steps for the simulation in Section 4.1 used Zipf distribution based on this observable assumption for the prompt sampling. However, it was also mentioned in Section 4.1.1 that using a suitable prompt dataset for the specific use case of the project from its own system or database is possible. This specific dataset may have a distinct prompt length distribution from Zipf distribution, which may be unique and more relevant to the use case. Instead of using the observation from Wang et al. [Wan+24], using the empirical prompt length

distribution for prompt sampling may give more valuable insights for forecasting and interpretation. Since the example use case in Section 5.1 had no available dataset for the simulation, the Zipf distribution assumption was applied. Thus, assessing the assumption before running the simulation was crucial for gaining a proper understanding of the LLM server performance.

8

CONCLUSION

This study aimed to test whether the request and server performance metric features extracted from an LLM deployment server are relevant for LLM server performance forecasting using a simulation that reflects a real-world situation. The trained machine learning model was also interpreted afterward to find the important features essential for forecasting. The experiment in this study was done with a chatbot project as an example use case and divided into simulation and forecasting steps. The goals of the simulation steps were to create a simulation dataset that reflects a real-world situation and use it for testing the LLM server. The simulation dataset was generated based on the method from BurstGPT. In the forecasting steps, the simulation result was used to model the machine learning model for forecasting the server's failure rate of processing requests and interpretation. The machine learning model XGBoost was chosen for the modeling due to its versatility, scalability and fast performance. It was also compared with several other comparison models, such as ARMA, VAR, and univariate XGBoost. The models applied the rolling origin evaluation setup to evaluate with the test data using MAE and RMSE metrics. The XGBoost model was used for analysis and interpretation to find the key features useful for forecasting using SHAP values and XGBoost feature importance methods.

The experiment result can be outlined into three main findings:

1. The time series data generated from the LLM simulation was appropriate for the use case to create a meaningful forecast of the LLM server performance with the XGBoost model. With the rolling and expanding window evaluation setup, the XGBoost model outperformed all comparison models on the test data, achieving the lowest value on MAE and RMSE metrics.

2. The model interpretation method with SHAP values was able to identify the comprehensible key features essential for forecasting the LLM server performance based on the use case. The most important feature by far was 'request_duration', supported by the SHAP values and XGBoost feature importance methods. The threshold value of 125 seconds may be useful for sending an alert about high failure rates in production based on SHAP values if the system infrastructure cannot run a machine learning pipeline for real-time forecasting. The importance rankings of other features differed in both interpretation methods. However, the XGBoost feature importance method with coverage was the most similar to the SHAP value method based on the top 10 most important features.

3. Using of empirical prompt length distribution for prompt sampling resulted in a different outcome of the simulation run compared to the sampling with Zipf distribution. These differences included the duration of the simulation run, the number of failed requests, and the feature distributions after data preparation. Evaluating the prompt length distribution assumption before choosing the prompt sampling distribution is important, as it might offer a different result on the simulation, forecasting, interpretation, and resulting parameters to be used in production.

In the future, other use cases with different LLM serving frameworks should be tested to test the generality of the approach. The simulation should also be run longer enough to test the influence of temporal information, e.g., day of week, hour, and minute of the entries. The possibility of testing the result from this experiment in a production environment should be explored to check if the result and understanding from the simulation can be translated into production. An infrastructure to monitor the server in production in real time will be needed to test this possibility. This infrastructure should be able to provide fresh data in real time using a data pipeline for training the machine learning model. The forecasting result will then be used to dynamically allocate the server resources to handle and prevent higher failure rates.

Part II

APPENDIX

A

TIME SERIES VISUALIZATION

The following plots visualize the time series of each predictor with the target variable 'fail'. The red line illustrates the time series data of the predictor, and the blue line shows the time series data of the target variable 'fail'.

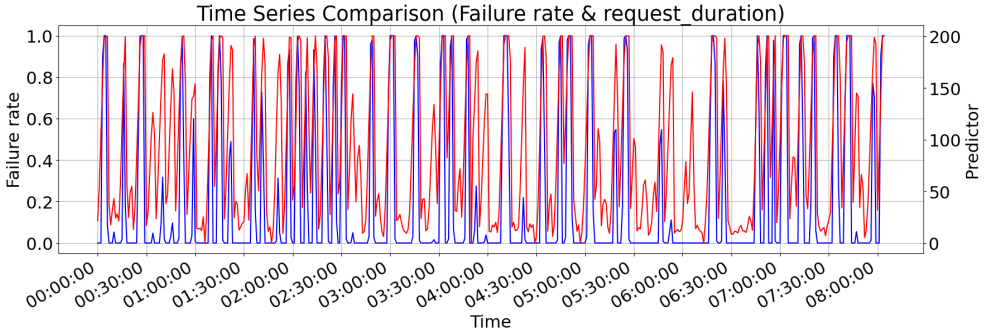

Figure A.1: Time series of 'request_duration' with the target variable 'fail'.

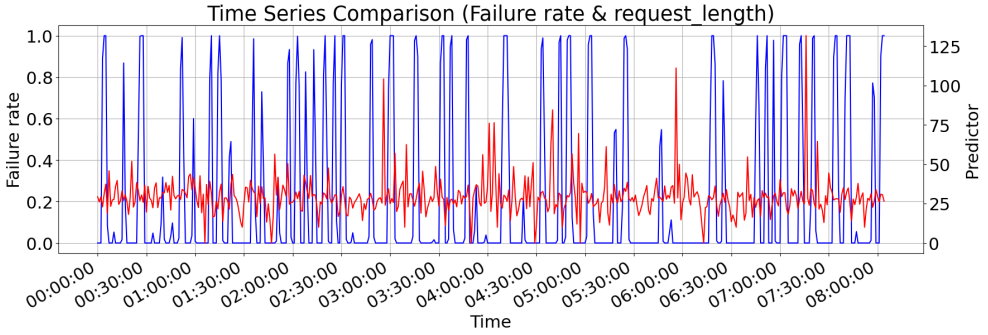

Figure A.2: Time series of 'request_length' with the target variable 'fail'.

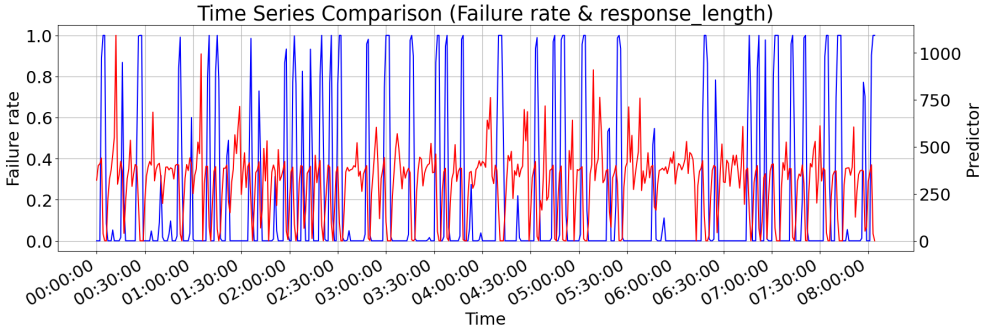

Figure A.3: Time series of 'response_length' with the target variable 'fail'.

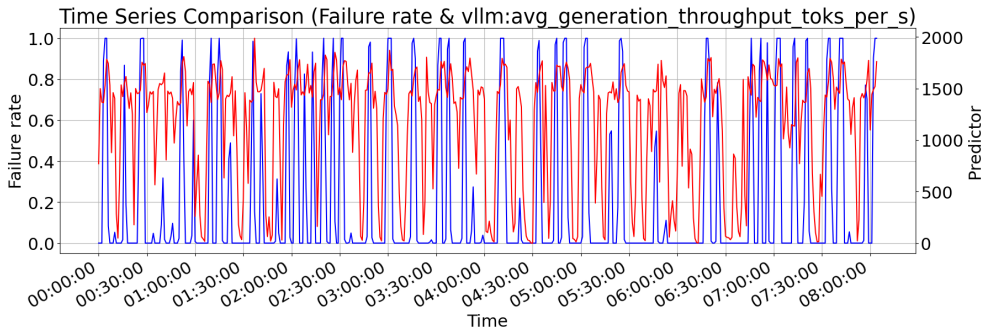

Figure A.4: Time series of 'vllm:avg_generation_throughput_toks_per_s' with the target variable 'fail'.

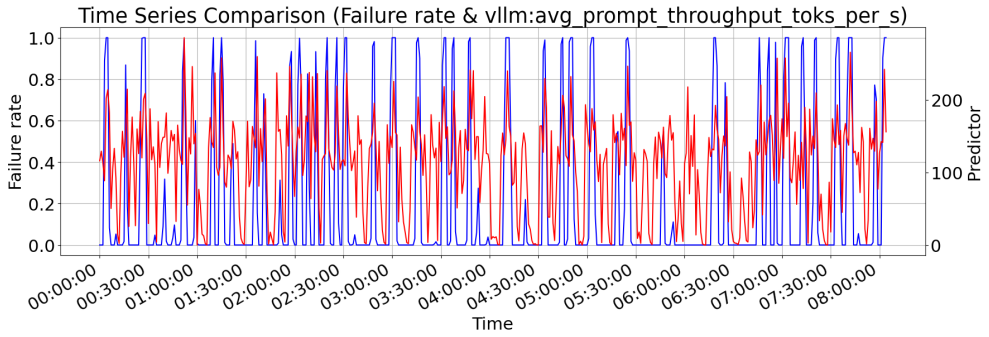

Figure A.5: Time series of 'vllm:avg_prompt_throughput_toks_per_s' with the target variable 'fail'.

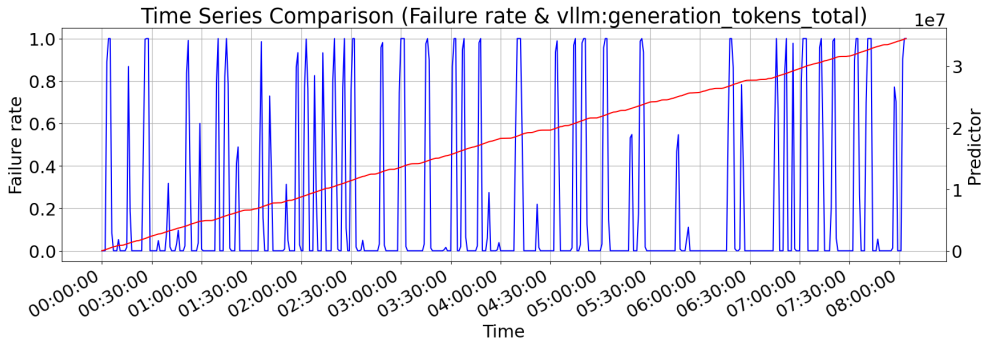

Figure A.6: Time series of 'vllm:generation_tokens_total' with the target variable 'fail'.

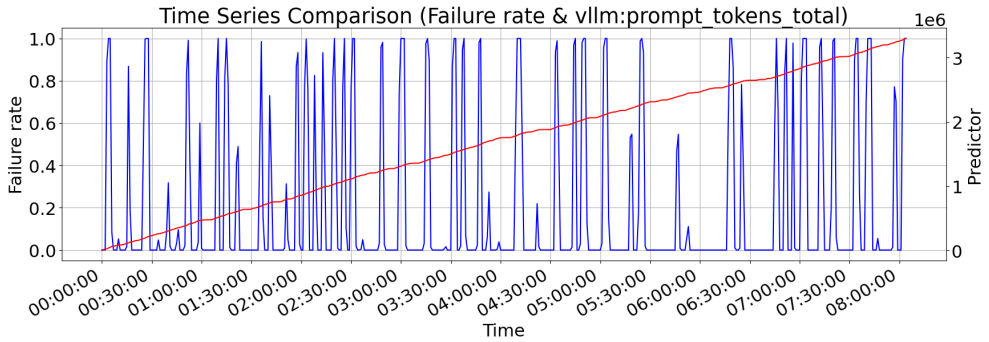

Figure A.7: Time series of 'vllm:prompt_tokens_total' with the target variable 'fail'.

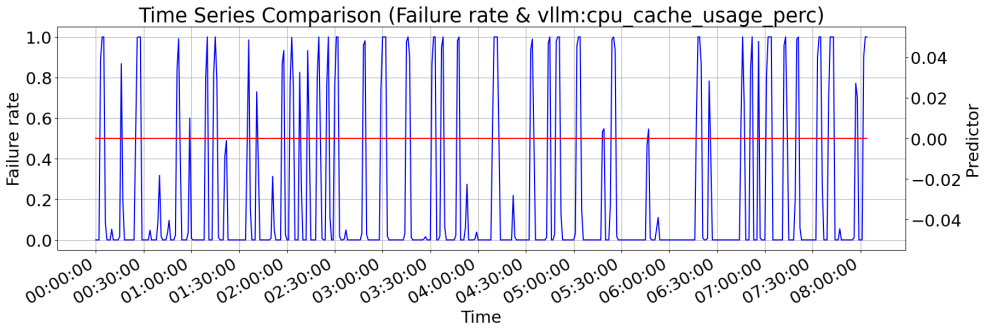

Figure A.8: Time series of 'vllm:cpu_cache_usage_perc' with the target variable 'fail'.

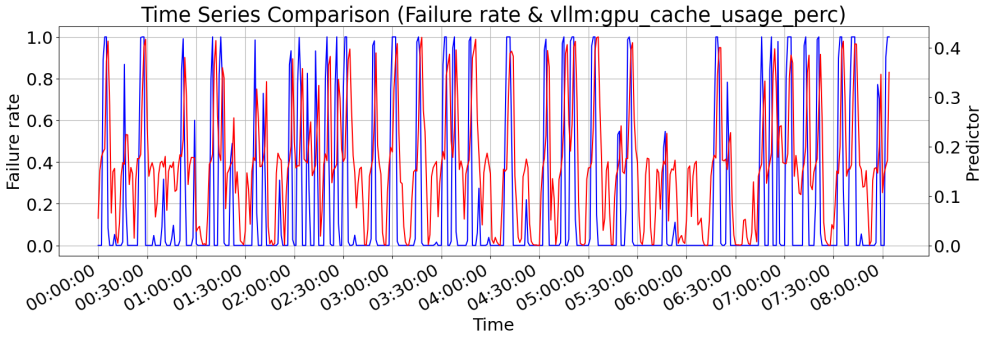

Figure A.9: Time series of 'vllm:gpu_cache_usage_perc' with the target variable 'fail'.

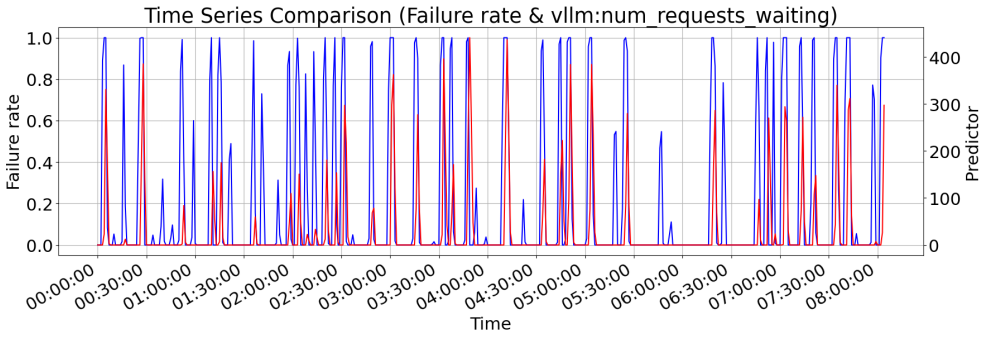

Figure A.10: Time series of 'vllm:num_requests_waiting' with the target variable 'fail'.

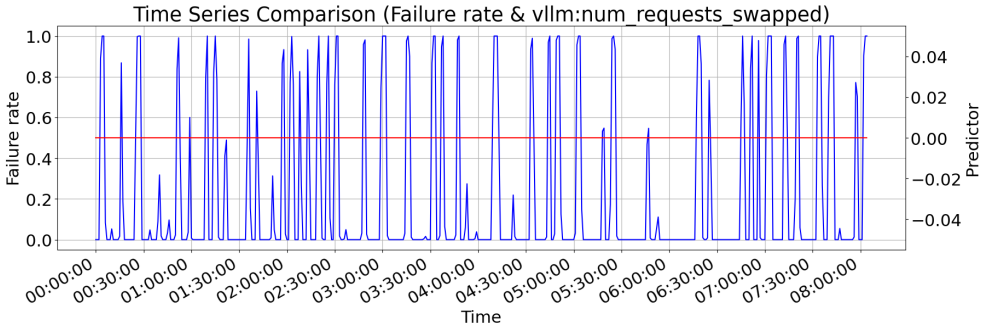

Figure A.11: Time series of 'vllm:num_requests_swapped' with the target variable 'fail'.

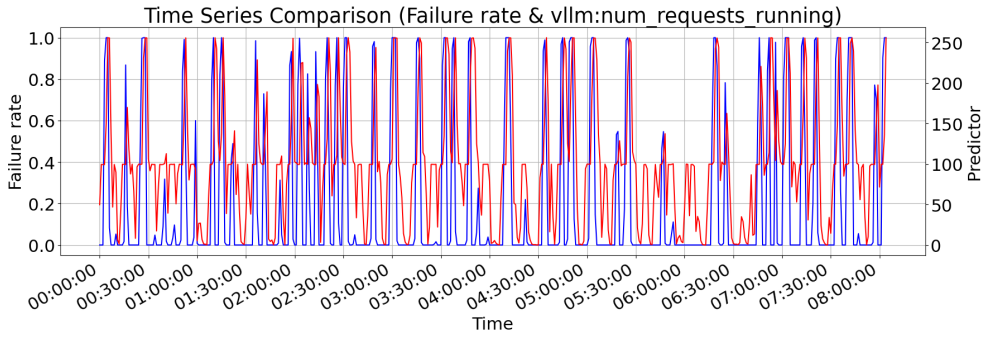

Figure A.12: Time series of 'vllm:num_requests_running' with the target variable 'fail'.

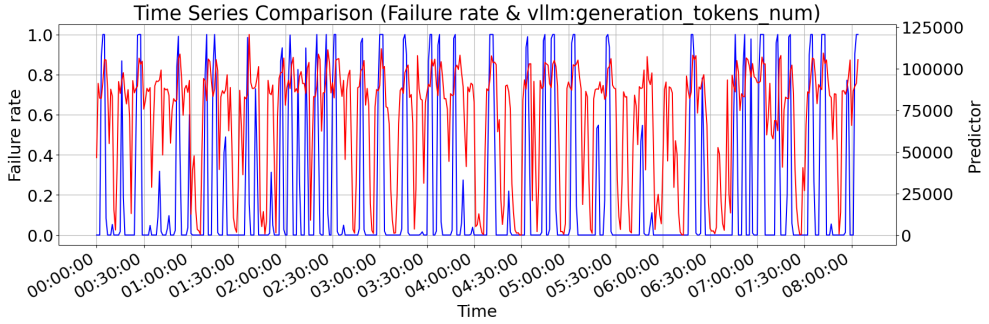

Figure A.13: Time series of 'vllm:generation_tokens_num' with the target variable 'fail'.

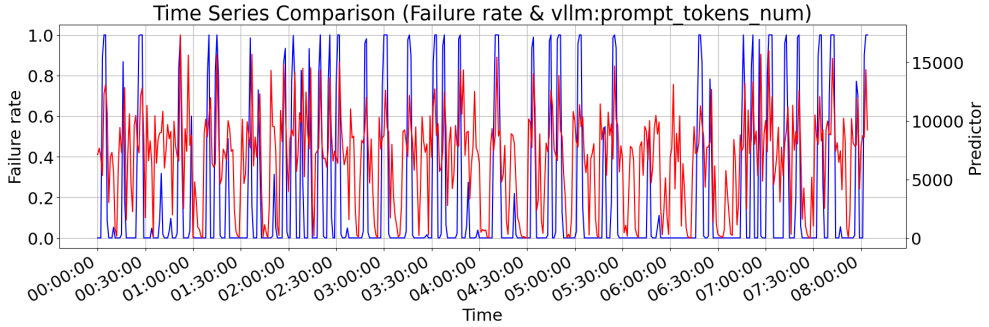

Figure A.14: Time series of 'vllm:prompt_tokens_num' with the target variable 'fail'.

B

FORECAST VISUALIZATION

The following plots visualize the prediction and true value of the test data using the rolling and expanding window evaluation setup.

B.1 ROLLING WINDOW

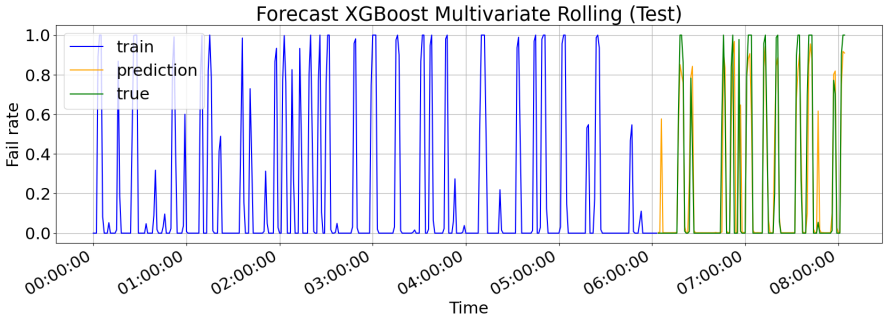

Figure B.1: Forecast of the multivariate XGBoost model with rolling window evaluation.

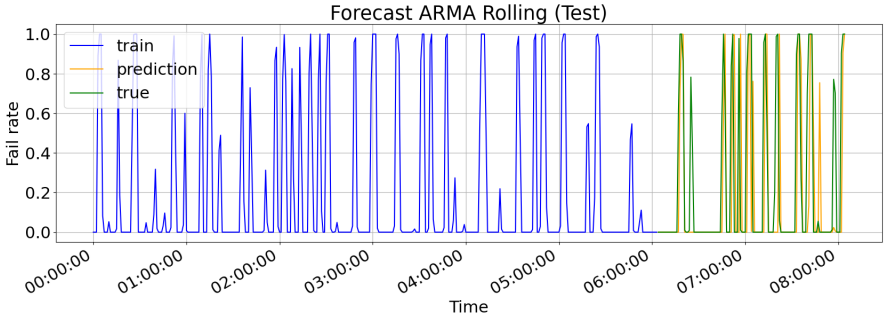

Figure B.2: Forecast of the ARMA model with rolling window evaluation.

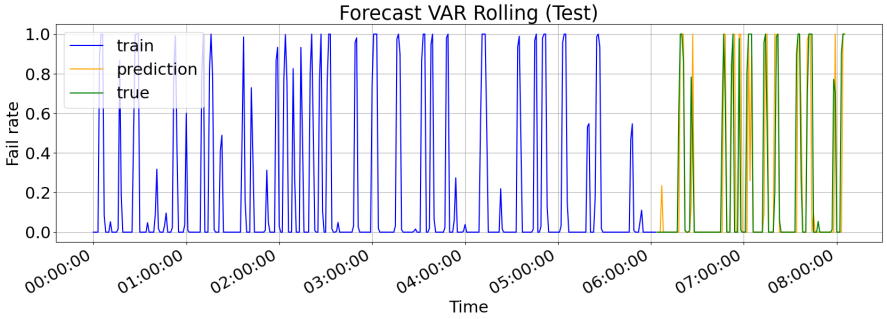

Figure B.3: Forecast of the VAR model with rolling window evaluation.

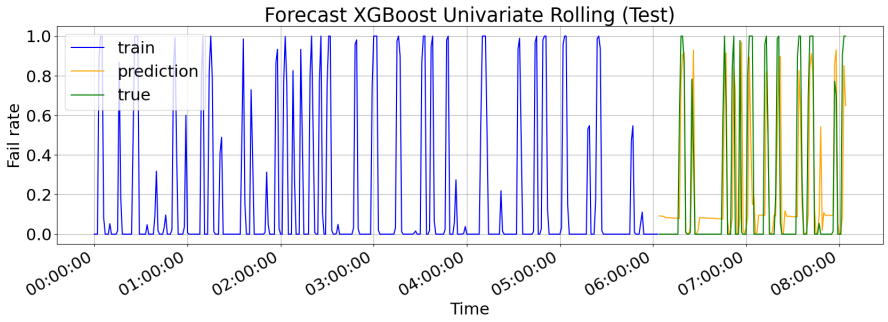

Figure B.4: Forecast of the univariate XGBoost model with rolling window evaluation.

B.2 EXPANDING WINDOW

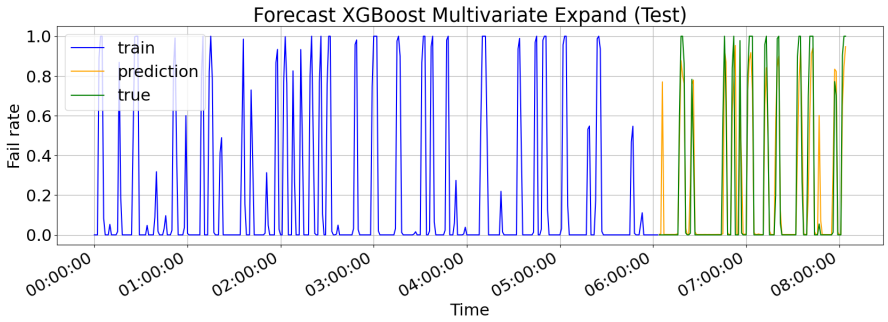

Figure B.5: Forecast of the multivariate XGBoost model with expanding window evaluation.

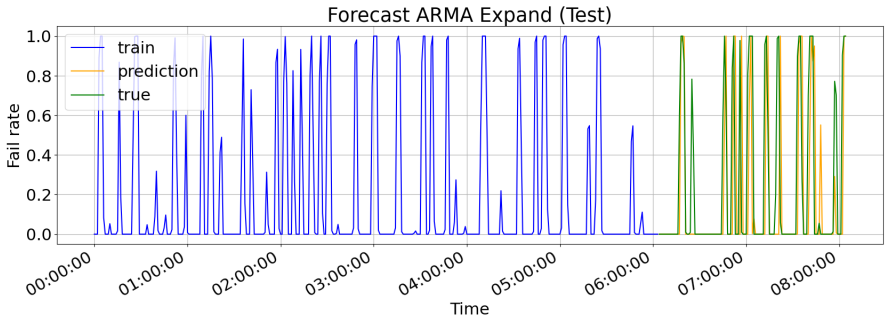

Figure B.6: Forecast of the ARMA model with expanding window evaluation.

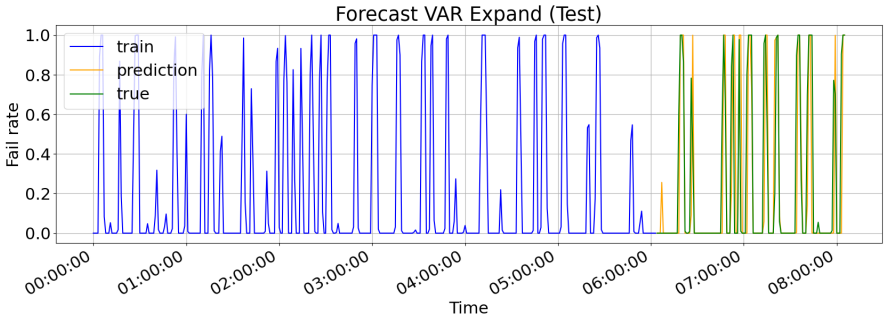

Figure B.7: Forecast of the VAR model with expanding window evaluation.

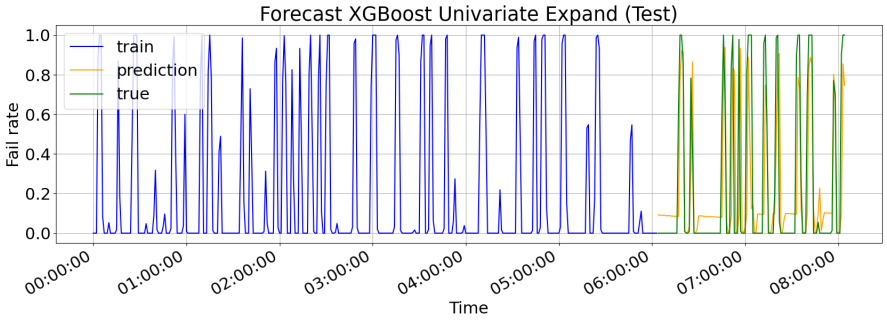

Figure B.8: Forecast of the univariate XGBoost model with expanding window evaluation.

C

INTERPRETATION VISUALIZATION

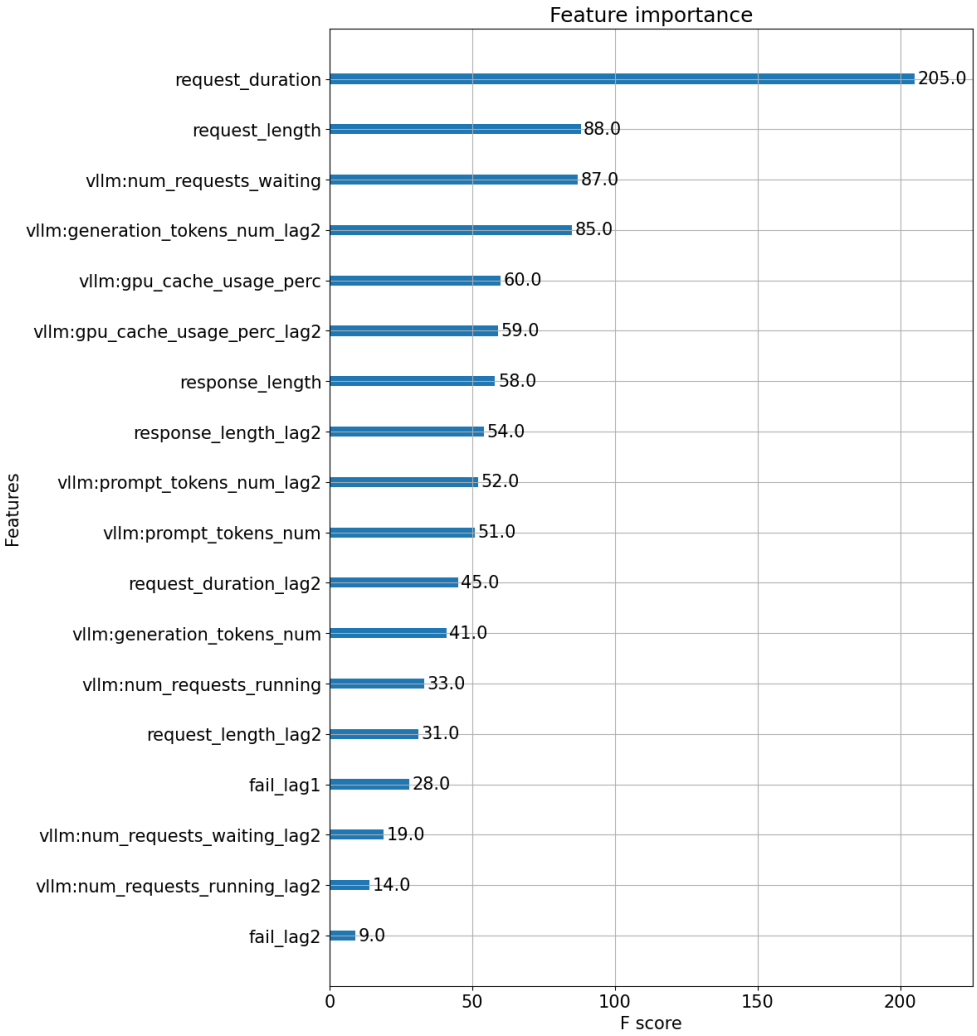

Figure C.1: XGBoost feature importance plot based on weight.

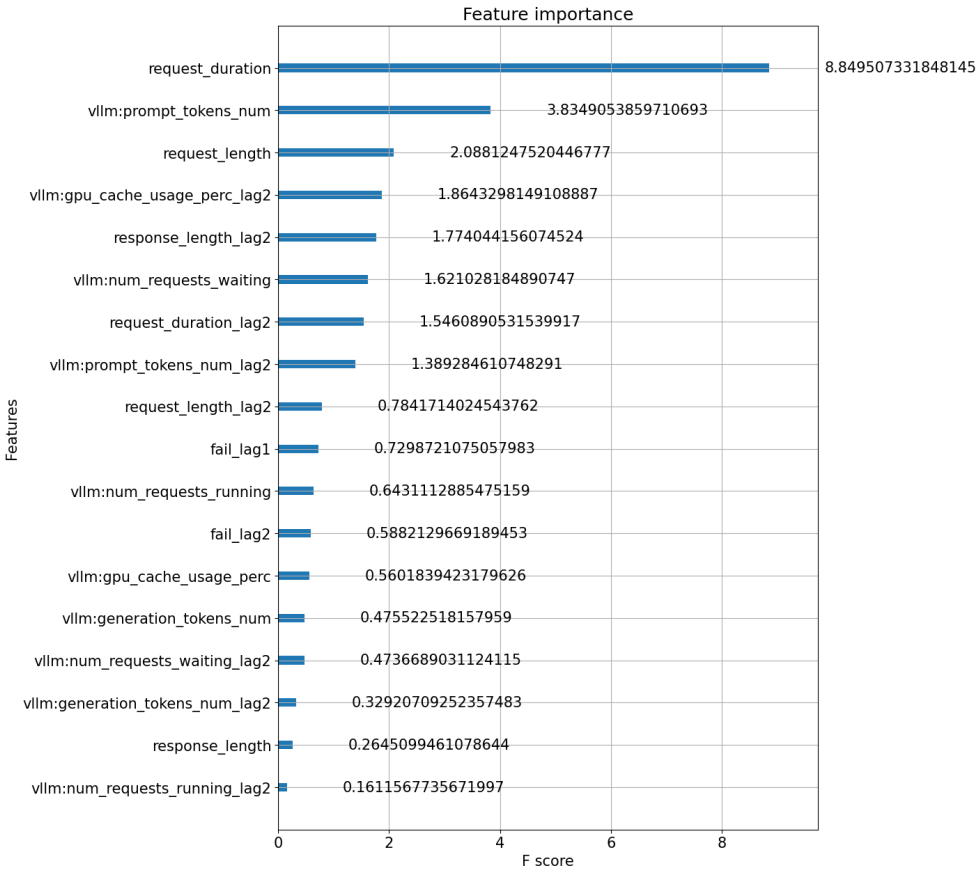

Figure C.2: XGBoost feature importance plot based on average gain.

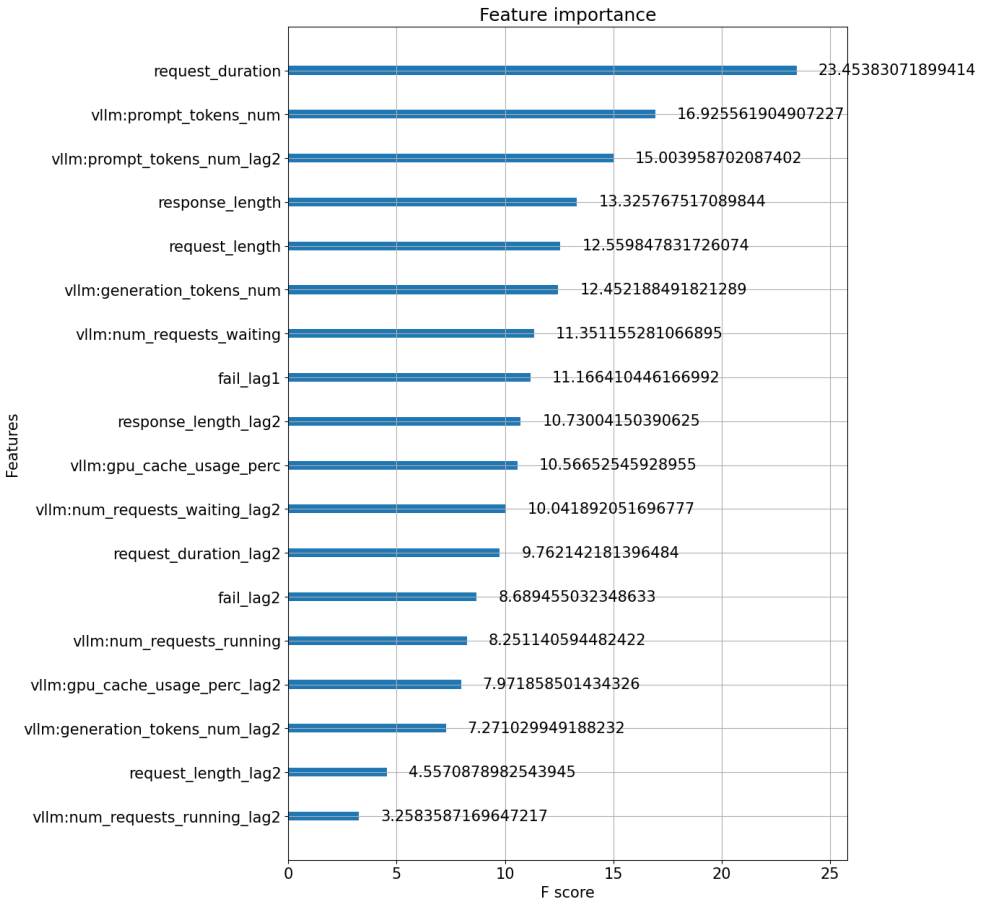

Figure C.3: XGBoost feature importance plot based on average coverage.

D

DISTRIBUTION COMPARISON VISUALIZATION OF PROMPT SAMPLING

The following plots compare the Zipf and empirical distribution for prompt sampling using KDE and ECDF. These plots visualize the distribution of the values after data preprocessing using steps in Section 4.2.1. The dashed lines on the KDE plot indicate the mean values of the Zipf and empirical distribution. Meanwhile, the dashed line on the ECDF plot indicates the location on the ECDFs where the maximum difference was calculated for the Kolmogorov-Smirnov test.

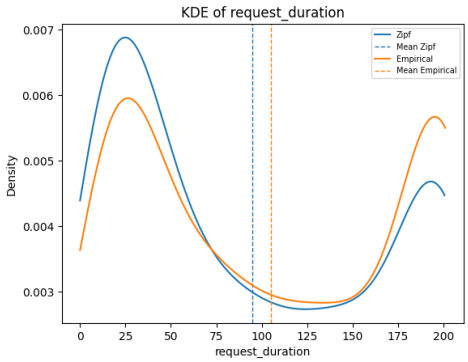

Figure D.1: KDE comparison of 'request_duration'.

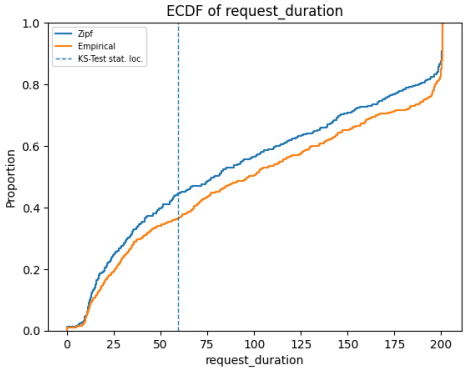

Figure D.2: ECDF comparison of 'request_duration'.

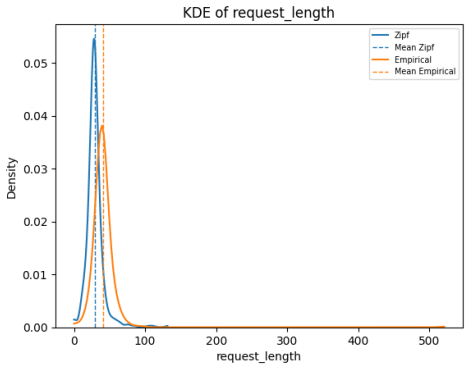

Figure D.3: KDE comparison of 'request_length'.

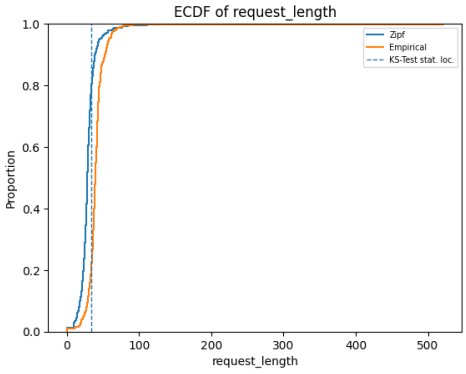

Figure D.4: ECDF comparison of 'request_length'.

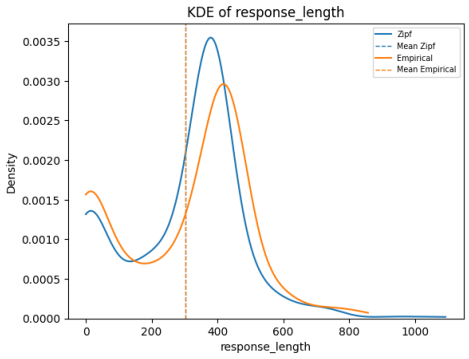

Figure D.5: KDE comparison of 'response_length'.

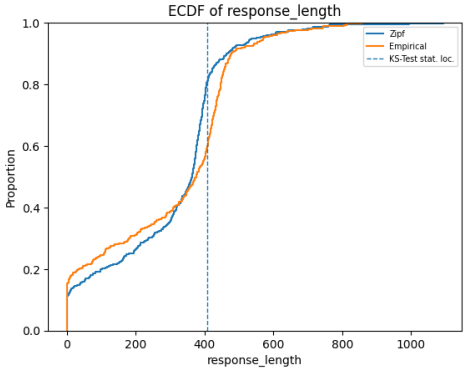

Figure D.6: ECDF comparison of 'response_length'.

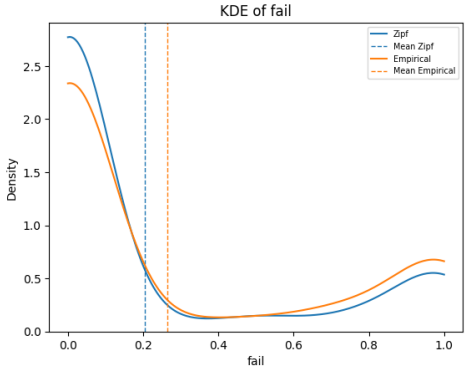

Figure D.7: KDE comparison of 'fail'.

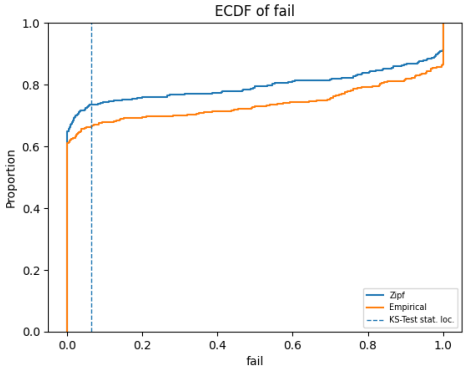

Figure D.8: ECDF comparison of 'fail'.

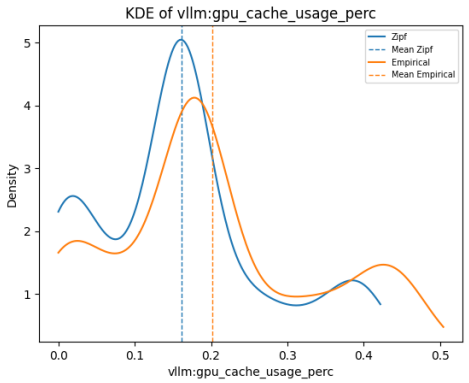

Figure D.9: KDE comparison of 'vllm:gpu_cache_usage_perc'.

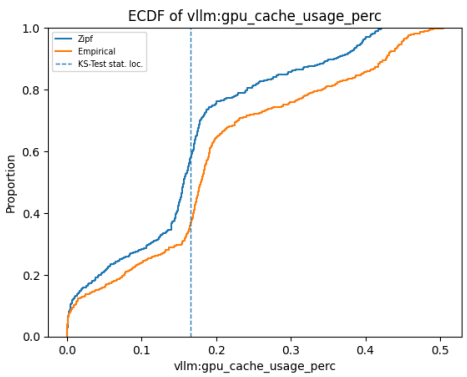

Figure D.10: ECDF comparison of 'vllm:gpu_cache_usage_perc'.

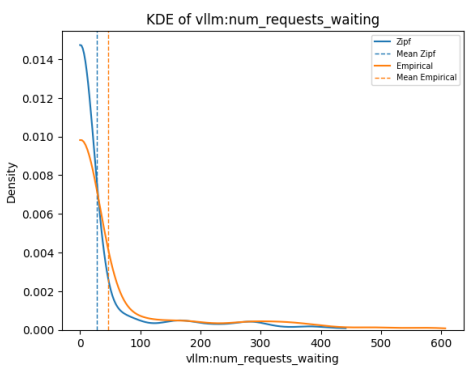

Figure D.11: KDE comparison of 'vllm:num_requests_waiting'.

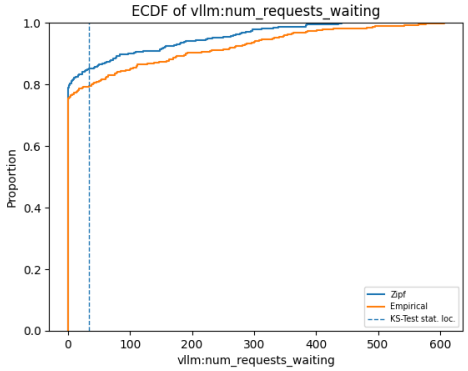

Figure D.12: ECDF comparison of 'vllm:num_requests_waiting'.

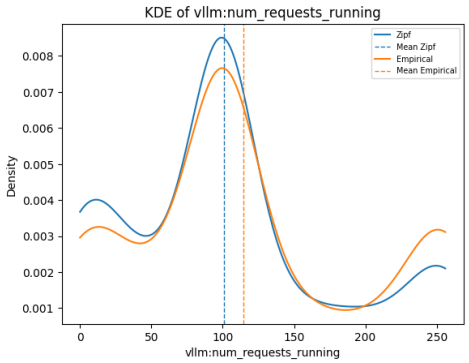

Figure D.13: KDE comparison of 'vllm:num_requests_running'.

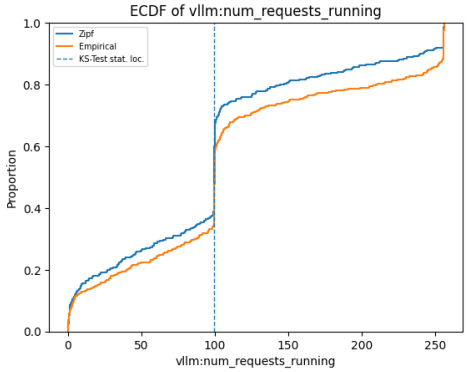

Figure D.14: ECDF comparison of 'vllm:num_requests_running'.

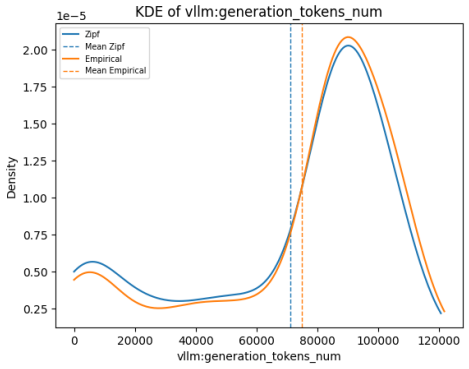

Figure D.15: KDE comparison of 'vllm:generation_tokens_num'.

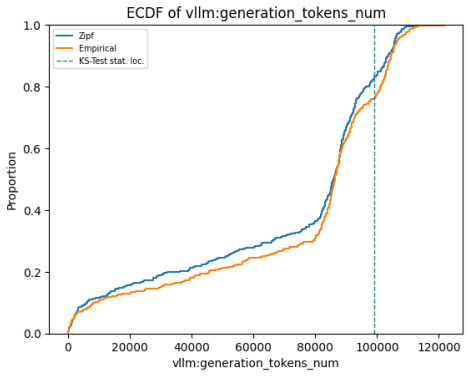

Figure D.16: ECDF comparison of 'vllm:generation_tokens_num'.

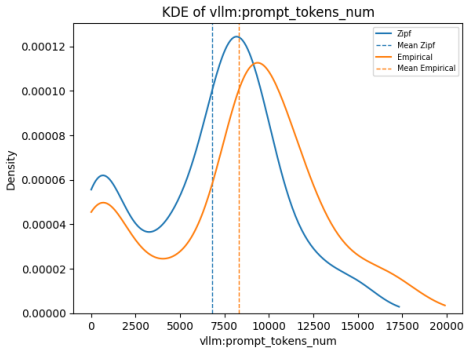

Figure D.17: KDE comparison of 'vllm:prompt_tokens_num'.

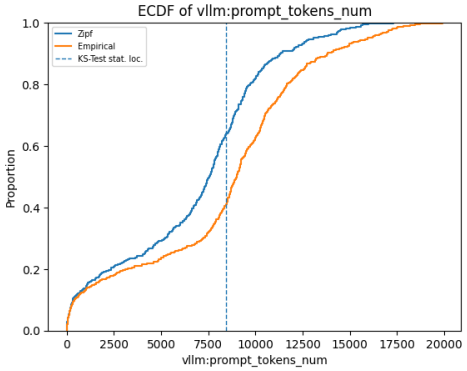

Figure D.18: ECDF comparison of 'vllm:prompt_tokens_num'.

BIBLIOGRAPHY

[AE+14] Ahmed Ali-Eldin, Oleg Seleznjev, Sara Sjöstedt-de Luna, Johan Tordsson, and Erik Elmroth. "Measuring Cloud Workload Burstiness." In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. 2014, pp. 566–572. DOI: 10.1109/UCC.2014.87.

[BCMM21] Candice Bentéjac, Anna Csörgő, and Gonzalo Martínez-Muñoz. "A comparative analysis of gradient boosting algorithms." In: *Artificial Intelligence Review* 54.3 (2021), pp. 1937–1967. ISSN: 1573-7462. DOI: 10.1007/s10462-020-09896-5. URL: https://doi.org/10.1007/s10462-020-09896-5.

[BD16] Peter J. Brockwell and Richard A. Davis. "Introduction." In: *Introduction to Time Series and Forecasting*. Cham: Springer International Publishing, 2016, pp. 1–37. ISBN: 978-3-319-29854-2. DOI: 10.1007/978-3-319-29854-2_1. URL: https://doi.org/10.1007/978-3-319-29854-2_1.

[CG16] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System." In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, 785–794. ISBN: 9781450342322. DOI: 10.1145/2939672.2939785. URL: https://doi.org/10.1145/2939672.2939785.

[Dwi+23] Rudresh Dwivedi et al. "Explainable AI (XAI): Core Ideas, Techniques, and Solutions." In: *ACM Comput. Surv.* 55.9 (Jan. 2023). ISSN: 0360-0300. DOI: 10.1145/3561048. URL: https://doi.org/10.1145/3561048.

[Fan+22] Zheng-gang Fang, Shu-qin Yang, Cai-xia Lv, Shu-yi An, and Wei Wu. "Application of a data-driven XGBoost model for the prediction of COVID-19 in the USA: a time-series study." In: *BMJ Open* 12.7 (2022). ISSN: 2044-6055. DOI: 10.1136/bmjopen-2021-056685. eprint: https://bmjopen.bmj.com/content/12/7/e056685.full.pdf. URL: https://bmjopen.bmj.com/content/12/7/e056685.

[Fen+24] Duanyu Feng, Yongfu Dai, Jimin Huang, Yifang Zhang, Qianqian Xie, Weiguang Han, Zhengyu Chen, Alejandro Lopez-Lira, and Hao Wang. *Empowering Many, Biasing a Few: Generalist Credit Scoring through Large Language Models*. Preprint. 2024. arXiv: 2310.00566 [cs.LG]. URL: https://arxiv.org/abs/2310.00566.

[HAB23] Hansika Hewamalage, Klaus Ackermann, and Christoph Bergmeir. "Forecast evaluation for data scientists: common pitfalls and best practices." In: *Data Mining and Knowledge Discovery* 37.2 (2023), pp. 788–832. ISSN: 1573-756X. DOI: 10.1007/s10618-022-00894-5. URL: https://doi.org/10.1007/s10618-022-00894-5.

[JMWV24] Sami Ben Jabeur, Salma Mefteh-Wali, and Jean-Laurent Viviani. "Forecasting gold price with the XGBoost algorithm and SHAP interaction values." In: *Annals of Operations Research* 334.1 (2024), pp. 679–699. ISSN: 1572-9338. DOI: 10.1007/s10479-021-04187-w. URL: https://doi.org/10.1007/s10479-021-04187-w.

[Kwo+23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. "Efficient Memory Management for Large Language Model Serving with PagedAttention." In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, 611–626. ISBN: 9798400702297. DOI: 10.1145/3600006.3613165. URL: https://doi.org/10.1145/3600006.3613165.

[Liu+23] Xiao-Yang Liu, Guoxuan Wang, Hongyang Yang, and Daochen Zha. "Data-centric FinGPT: Democratizing Internet-scale Data for Financial Large Language Models." In: *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*. 2023. URL: https://openreview.net/forum?id=5BqWC1Fz8F.

[Lou+23a] Lefteris Loukas, Ilias Stogiannidis, Odysseas Diamantopoulos, Prodromos Malakasiotis, and Stavros Vassos. "Making LLMs Worth Every Penny: Resource-Limited Text Classification in Banking." In: *Proceedings of the Fourth ACM International Conference on AI in Finance*. ICAIF '23. Brooklyn, NY, USA: Association for Computing Machinery, 2023, 392–400. ISBN: 9798400702402. DOI: 10.1145/3604237.3626891. URL: https://doi.org/10.1145/3604237.3626891.

[Lou+23b] Lefteris Loukas, Ilias Stogiannidis, Prodromos Malakasiotis, and Stavros Vassos. "Breaking the Bank with ChatGPT: Few-Shot Text Classification for Finance." In: *Proceedings of the Fifth Workshop on Financial Technology and Natural Language Processing and the Second Multimodal AI For Financial Forecasting*. Ed. by Chung-Chi Chen, Hiroya Takamura, Puneet Mathur, Remit Sawhney, Hen-Hsen Huang, and Hsin-Hsi Chen. Macao: -, 2023, pp. 74–80. URL: https://aclanthology.org/2023.finnlp-1.7.

[Lun+20] Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. "From local explanations to global understanding with explainable AI for trees." In: *Nature Machine Intelligence* 2.1 (2020), pp. 56–67. ISSN: 2522-5839. DOI:

10.1038/s42256-019-0138-9. URL: https://doi.org/10.1038/s42256-019-0138-9.

[LL17] Scott M Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions." In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/8a20a8621978632d76c43dfd28b67767-Paper.pdf.

[MCB20] Christoph Molnar, Giuseppe Casalicchio, and Bernd Bischl. "Interpretable Machine Learning – A Brief History, State-of-the-Art and Challenges." In: *ECML PKDD 2020 Workshops*. Ed. by Irena Koprinska et al. Cham: Springer International Publishing, 2020, pp. 417–431. ISBN: 978-3-030-65965-3.

[Niu20] Yiyang Niu. "Walmart Sales Forecasting using XGBoost algorithm and Feature engineering." In: *2020 International Conference on Big Data Artificial Intelligence Software Engineering (ICBASE)*. 2020, pp. 458–461. DOI: 10.1109/ICBASE51474.2020.00103.

[Qiu+17] Junping Qiu, Rongying Zhao, Siluo Yang, and Ke Dong. "Word Frequency Distribution of Literature Information: Zipf's Law." In: *Informetrics: Theory, Methods and Applications*. Singapore: Springer Singapore, 2017, pp. 121–143. ISBN: 978-981-10-4032-0. DOI: 10.1007/978-981-10-4032-0_5. URL: https://doi.org/10.1007/978-981-10-4032-0_5.

[Shi22] Hangsik Shin. "XGBoost Regression of the Most Significant Photoplethysmogram Features for Assessing Vascular Aging." In: *IEEE Journal of Biomedical and Health Informatics* 26.7 (2022), pp. 3354–3361. DOI: 10.1109/JBHI.2022.3151091.

[SS17a] Robert H. Shumway and David S. Stoffer. "ARIMA Models." In: *Time Series Analysis and Its Applications: With R Examples*. Cham: Springer International Publishing, 2017, pp. 75–163. ISBN: 978-3-319-52452-8. DOI: 10.1007/978-3-319-52452-8_3. URL: https://doi.org/10.1007/978-3-319-52452-8_3.

[SS17b] Robert H. Shumway and David S. Stoffer. "Additional Time Domain Topics." In: *Time Series Analysis and Its Applications: With R Examples*. Cham: Springer International Publishing, 2017, pp. 241–287. ISBN: 978-3-319-52452-8. DOI: 10.1007/978-3-319-52452-8_5. URL: https://doi.org/10.1007/978-3-319-52452-8_5.

[SS17c] Robert H. Shumway and David S. Stoffer. "Characteristics of Time Series." In: *Time Series Analysis and Its Applications: With R Examples*. Cham: Springer International Publishing, 2017, pp. 1–

44. ISBN: 978-3-319-52452-8. DOI: 10.1007/978-3-319-52452-8_1. URL: https://doi.org/10.1007/978-3-319-52452-8_1.

[Sun+24] Wenbo Sun, Jiaqi Wang, Qiming Guo, Ziyu Li, Wenlu Wang, and Rihan Hai. *CEBench: A Benchmarking Toolkit for the Cost-Effectiveness of LLM Pipelines*. Preprint. 2024. arXiv: 2407.12797 [cs.PF]. URL: https://arxiv.org/abs/2407.12797.

[Tug+24] Lukas Tuggener, Pascal Sager, Yassine Taoudi-Benchekroun, Benjamin F. Grewe, and Thilo Stadelmann. "So you want your private LLM at home? A survey and benchmark of methods for efficient GPTs." In: *2024 11th IEEE Swiss Conference on Data Science (SDS)*. 2024, pp. 205–212. DOI: 10.1109/SDS60720.2024.00036.

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. "Attention is All you Need." In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[Wan+24] Yuxin Wang, Yuhan Chen, Zeyu Li, Zhenheng Tang, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. *Towards Efficient and Reliable LLM Serving: A Real-World Workload Study*. Preprint. 2024. arXiv: 2401.17644v2 [cs.DC]. URL: https://arxiv.org/abs/2401.17644v2.

[WKP98] A. Witt, J. Kurths, and A. Pikovsky. "Testing stationarity in time series." In: *Phys. Rev. E* 58 (2 1998), pp. 1800–1810. DOI: 10.1103/PhysRevE.58.1800. URL: https://link.aps.org/doi/10.1103/PhysRevE.58.1800.

[Zha+21] Lingyu Zhang, Wenjie Bian, Wenyi Qu, Liheng Tuo, and Yunhai Wang. "Time series forecast of sales volume based on XGBoost." In: *Journal of Physics: Conference Series* 1873.1 (2021), p. 012067. DOI: 10.1088/1742-6596/1873/1/012067. URL: https://dx.doi.org/10.1088/1742-6596/1873/1/012067.

[Zhe+24] Lianmin Zheng et al. "Judging LLM-as-a-judge with MT-bench and Chatbot Arena." In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. NIPS '23. New Orleans, LA, USA: Curran Associates Inc., 2024.