VRIJE UNIVERSITEIT AMSTERDAM
Faculty of Science
De Boelelaan 1081
1081 HV Amsterdam

Research Paper Business Analytics

Function Discovery in Queuing Models with Genetic Programming

Author:
Dino Špirtović

Supervisor:
Dr. René Bekker

October 2017

Abstract

Many queueing models exist in practice that are not solvable analytically. While some research has been carried out to approximate functions in queuing theory, only a few studies have investigated a machine learning approach. The objective of this paper was to explore the possibility of discovering performance functions for queueing models with genetic programming, which is a machine learning technique from the field of evolutionary computing. This was done by sampling training and test data for models for which the functions are known, namely the Erlang C and Engset delay model. The discovered functions were compared to the analytical functions and their performance was tested in an optimization context. The main finding was that GP was able find useful functions for the considered models. Because the functions are simple approximations of the underlying function, it is important to expose their flaws before applying them in practice. Another important finding was that GP was able to discover these functions with relatively little training data.

Contents

Chapter 1

Introduction

1.1 Research Goal

Everyday we encounter various queues. From simple queues at the supermarket to complex queuing networks we are less aware of. To keep the customers satisfied and the service running smoothly, service providers need to adapt to the arrival stream of customers. For example, to reduce the waiting time one can increase the number of servers or improve service speed of existing servers. To do this efficiently one needs to predict the impact of these changes on the waiting time.

The field of Queueing Theory has been successful in analytically deriving performance functions, e.g the expected queue length or the expected waiting time function, for various queuing models. These functions provide us with insights and can be used for optimization purposes. However, many queueing models exist in practice that are not solvable analytically. This means that one has to rely on simulation to obtain performance measures, which is often time consuming and provides little insight.

While some research has been carried out to approximate functions for queuing models, only a few studies have investigated a machine learning approach. Genetic Programming (GP) is a machine learning technique that is also part of the Evolutionary Algorithm (EA) family and is generally applied to discover algebraic functions fitting some training data.

The aim of this study is to explore the possibility of discovering functions for queueing models with GP. More specifically, this research focuses on finding algebraic approximations of the expected waiting time and average number of customers in the system for respectively the Erlang C and the Engset delay model. Because this study is the first to attempt a GP approach to discover performance functions for queueing models, we focus on models for which these functions are already known. This way we can compare the discovered function with the analytical functions. Furthermore, we explore how various training sets and GP parameters influence the goodness of fit of the discovered functions and what the impact is on capacity decisions.

1.2 Related Work

There is little literature involving a combination of EAs and Queuing Theory. The literature that does combine these two fields of study usually focuses on finding the optimal policy in Markov Decision Processes (MDPs). MDPs are an extension of Markov chains that include decision making. A study that combines EAs and MDPs is [1]. The authors of [1] compare an EA approach to a common Dynamic Programming (DP) approach and conclude that DPs policy iteration algorithm outperforms the EA because of its quick convergence. However, policy iteration can not be applied when dealing with very large MDPs while the EA approach can.

Recent research by Onderwater et al. [5] introduces a method, that is based on GP, to discover algebraic descriptions of value functions for basic MDPs. Furthermore, to demonstrate the applicability of this method, the authors obtain near-optimal policies via one-step policy improvement using the discovered value functions. Our research is similar to [5] since we also use GP to discover functions. However, our research is focused on finding performance functions for queuing models instead of value functions for MDPs.

An introduction to EAs, including GP, can be found in the textbook "Introduction to Evolutionary Computing" [3]. More details on queuing models, Markov chains and MDPs can be found in books such as [4] and [2].

1.3 Paper outline

In section 2.1, the reader is provided with relevant knowledge of GP. The function discovery process is described in section 2.2. Finally, results are presented and discussed in sections 3 and 4 respectively.

Chapter 2

Methods

2.1 Genetic Programming

As mentioned in the introduction, genetic programming is part of the evolutionary algorithm family. While other EAs are generally used to find the best input for some known model, GP is used to find the best fitting model for some in- and output data. The search procedure is the same as any other EA. The procedure starts with a initial population of randomly generated individuals. These individuals each represent a candidate solution. After the initial population is generated, the algorithm starts the (evolutionary) search for the best solution by evaluating the fitness of each individual with a given fitness function. Based on the computed fitness of each individual, the parent selection mechanism selects individuals from the population to generate offspring for the next generation. The offspring is created by applying so-called variation operators to the selected parents. Finally, the survival selection mechanism decides which individuals from the parent and offspring pool survive to populate the next generation. This (evolutionary) process is repeated for multiple generations until a satisfactory solution is found. The various steps of this evolutionary search process are further elaborated on in subsection 2.1.1 through 2.1.6.

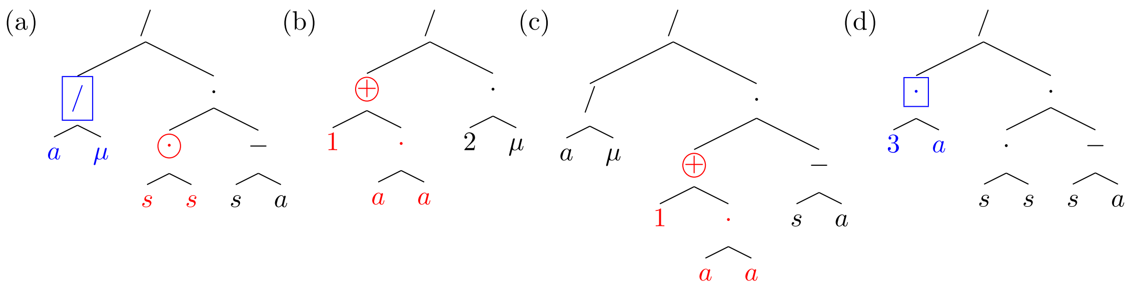


Figure 2.1: Illustration of the recombination and mutation operators. Recombination: red sub-trees from (a) and (b) are exchanged to generate (c). Mutation: tree (d) is obtained by replacing the blue sub-tree from (a) with a randomly generated sub-tree $3 \cdot a$.

2.1.1 Individual representation

In GP, the individuals are represented as parse trees with the following specification: the inner nodes of the tree may only contain elements from the user defined function set and the leaves of the tree may only contain elements from the user defined terminal set. In this paper the individuals represent arithmetic functions.

Consider figure 2.1a as an example. The function set contains simple arithmetic operators $(+, -, \cdot, /)$ while the terminal set consists of relevant variables (s, a, μ) and constants $(1, 2, 3)$. Collapsing the tree will result in a valid function for the Erlang C model. This is done by starting at the leaves. Leaves are connected to an inner node. In the example, the leaves a and μ are connected to an inner node that contains the divide operator. Collapsing this sub-tree results in the left leave being divided by the right one (a/μ). The same is done for the other leaves of the tree, $s \cdot s = s^2$ and $s - a$. The sub-trees of s^2 and $s - a$ are connected to the inner node representing the times $(\cdot)$ operator. Collapsing the three further leaves us with a/μ connected with a divide operator to $s^2(s - a)$. The final collapse of sub-trees results in $\frac{a/\mu}{s^2(s-a)}$. This is the arithmetic function the tree represents.

2.1.2 Variation operators

There are two variation operators in GP, namely recombination and mutation. Whenever new offspring is created one of the two operators is chosen to generate offspring with some probability p and the other with probability $(1 - p)$. Although various ways exist to preform recombination or mutation, only those relevant to this study will be described here.

Recombination randomly chooses an inner node from the parent trees, see figure 2.1 trees (a) and (b). The chosen sub-trees (indicated in red) are then exchanged to create offspring (c). The generated offspring contains characteristics from both parents after recombination. Mutation on the other hand generates offspring by slightly changing a single individual. Specifically, this is done by selecting an inner node of an individual and replacing the sub-tree (indicated in blue) with a small randomly generated sub-tree. Therefore, mutation adds new characteristics to the population helping the algorithm to move out of local optima.

2.1.3 Parent selection

Unlike representation and variation operators, the parent and survival selection mechanisms are not specific to GP. A commonly used method for parent selection in GP is the so called tournament selection. This method randomly draws a number of individuals from the current population and lets them compete in a ”tournament” based on their fitness. The winner of the tournament is then chosen to be a parent. Another method is called roulette selection. This method first assigns a portion of the virtual roulette wheel to all the individuals based on their relative fitness. The parents are then randomly picked by spinning this virtual roulette. If more than one parent is needed for reproduction, the individuals already chosen for recombination are excluded from the roulette or tournament selection when the next parent needs to be picked.

The main idea of EAs is related to the evolutionary process in nature, namely that fit parents are likely to generate fitter offspring over time. This is reflected in the parent selection mechanism because fit individuals have a higher likelihood of being picked as parents in both the tournament and roulette selection. To keep the population diverse, less fit individuals also have a chance of being picked. Keeping the population diverse is important because the algorithm might get stuck in a local optimum otherwise.

2.1.4 Fitness function

The fitness of an individual is determined by how well the function fits the training set (of size k). Less error means higher fitness. One can choose to use any error measure and GP will seek to minimize the chosen objective. Relevant for this study are the following error measures: Sum of Absolute Deviation (SAD), Maximum Relative Deviation (MRD) and Maximum Absolute Deviation (MAD). Define $\widetilde{y_i}$ as the approximated output and y_i as the exact output for input combination i, then SAD and MRD are calculated as follow:

$$SAD = \sum_{i=1}^{k} |\widetilde{y_i} - y_i| \qquad MRD = \max_i \frac{\widetilde{y_i} - y_i}{y_i} \qquad MAD = \max_i |\widetilde{y_i} - y_i|$$

2.1.5 Survival selection

The survival selection mechanism is generally straightforward. One can choose to replace the population entirely by a new one. Or one can choose to keep a number of the fittest individuals and replace the rest, also known as the elitist approach. Note that according to [3], the convention for GP is to use the former method.

2.1.6 Bloat

During a GP run the tree sizes of the individuals randomly change because of the variation operators. Bloat is the name of a common problem in GP, namely that the average tree size tends to grow during a run. It is still unknown why bloat occurs, but there are many ways to counter it. A simple method would be to reject any individual that exceeds a certain tree size at creation. An extension of this method is to dynamically increase the maximum tree size only if the resulting tree has a better fitness than the fittest one found so far.

2.2 Function Discovery

For this research the matrix-based programming environment MATLAB was used together with a Genetic Programming toolbox called GPLAB. The toolbox contains most of GPs core features described in the previous section. Moreover, GPLAB can be extended by the user and contains some useful features. Two important features used during this study are online adaptation of operator probabilities and dynamic three depth. The former speeds up the discovery process and the latter helps control bloat. For more details regarding GPLAB, we refer to the GPLAB user manual.

2.2.1 Queueing notation

Before we describe our research method in more detail, we introduce some (queueing) notation. The $M/M/1$, Erlang C and Engset model are considered during this study. The $M/M/1$ queue has a single server and one type of customer. The customers arrive according to a Poisson process with intensity λ. The service times are exponentially distributed with an expected service time equal to $1/\mu$. The $M/M/s$ (or Erlang C) model is an extension of the $M/M/1$ queue with s identical servers, each with service rate equal to μ. The offered load is defined by $a = \lambda/\mu$ and the load per server by $\rho = a/s$ for both models. The stability condition requires that $\rho < 1$. If this condition does not hold, the queue length (L_q) would only increase over time and the expected waiting time ($\mathbf{E}W_q$) would rise to infinity. The $M/M/s/n/n$ (or Engset delay) model is a closed network with a population of n customers. Each of the n customers arrive for service with intensity λ and can be serviced by any of the s servers with service rate μ. After receiving service they can produce new service requests. Note that it only makes sense to study this model when $s < n$. Otherwise there would always be an available server for every arriving customer and therefore no queue and zero waiting time. The analytic expressions of the functions for the models discussed in this section can be found in Appendix A.

2.2.2 Experiments

The first step in our research was to find a data set that enables the GP algorithm to discover robust functions for the considered queueing models. This was done by sampling multiple data sets (for each considered queueing model) and running GP experiments to determine which sets work best. The GP experiments were performed according to machine learning convention. Therefore, a training set was used for function discovery and a different test set was used for error analysis. This is a common practice in machine learning applied to detect and prevent ”overfitting”. Overfitting is a machine learning term that describes the model adapting too much to the training data and missing the general pattern.

The data sampling procedure is described in more detail in subsection 2.2.3. Furthermore, each GP experiment was carried out multiple times for different seeds of the random number generator to eliminate (un)lucky runs. Note that various GP settings were explored during the function discovery process. The explored GP settings are given in table 2.1.

The next step in our research was to discover how GP performs with less training data. This was done by sampling new smaller training sets and running new GP experiments to determine if its still able to discover ”good” functions. The reduction of the training set was repeated until a drastic drop in performance of the discovered function was observed.

The final step of this study consisted of applying the discovered functions in an optimization context. More precisely, we tested how the function performs when one is interested in finding the minimum number of servers necessary to maintain an average waiting time or average number of

Table 2.1: Explored GP settings

Setting	Explored values
generations	$(10, 15, 20, 25, 30)$
population size	$(500, 1000, 1500, 2000)$
operator prob	(even,variable)
terminal set	$(\rho, a, \lambda, \mu, s, n)$
function set	(plus,minus,times,divide,power,exp)
fitness	(SAD, MRD, MAD)
parent selection	(tournament, roulette)
survival selection	(replace, elitism)

customers in the system below a certain threshold.

2.2.3 Data sampling

Training sets were sampled in the following way:

1. Define value range for each N model parameter
2. For each parameter linearly sample M_i points from defined interval $(i = 1, \cdots, N)$
3. Combine parameter sample points into $L = \prod_1^N M_i$ combinations
4. Remove invalid combinations from the set
5. Get response values for the remaining combinations

At step two a choice was made to linearly sample over the parameters space. This choice is based on some knowledge of the considered queuing models. Namely, one could expect the functions (of all the considered models) to be monotonic. Step two can be generalized in the following way: Sample M_i parameter points from the interval that captures the shape of the response value. At step four parameter combinations are considered invalid if they do not satisfy some model restriction. For example, consider the $M/M/s$ queue with $a = 20$ and $s = 15$, then the stability condition $(a/s < 1)$ is not satisfied and the combination is removed from the set. At the final step (five) one can use various methods to obtain the response values, e.g simulation. The queueing functions from which the response values were obtained for this research can be found in Appendix A of this paper.

A common method to generate test data is to sample a certain percentage from the training data. Because we could obtain the response variable easily for any parameter combination from the exact functions, we sampled a new and larger set to use as a test set. To be more precise, we defined a broader value range for each parameter at step one and sampled more points at step two. Having many samples for the test set allowed us to perform better error analysis. The error analysis is described in more details in section 3.3.

Chapter 3

Experimental Results

3.1 Input & settings

As mentioned previously, various training sets and algorithm settings were used during this study. In this section we present our findings regarding GP training data and settings.

GP settings that proved to work well during the discovery for all the considered models are presented in table 3.1. We found that keeping the terminal and function set simple resulted in GP discovering more robust and less complex functions. Moreover, the setting that influenced the evolutionary search process the most was the fitness function. The MRD was used for the Engset model, while the SAD was used for the Erlang model. This choice is further elaborated on in section 3.3.

Table 3.1: "good" GP settings

Setting	Value
generations	$(15 - 25)$
population size	$(1000 - 1500)$
operator prob	(variable)
terminal set	('model parameters')
function set	(plus,minus,times,divide)
parent selection	(tournament)
survival selection	(replace)

Table 3.2 contains the training set that GP needed to discover functions presented in 3.2. One can see that more training data was needed for the Engset delay model. This is probably due to the higher dimension of the function, one needs more samples to capture the shape.

Table 3.2: Train and test set per queuing model

		Training		Test	
	Parameter	Interval	Samples	Interval	Samples
$M/M/1$	ρ	$(0, 1)$	5	x	x
	μ	$(0, 5]$	5	x	x
$M/M/s$	a	$(0, 10]$	5	$(0, 20]$	20
	μ	$[1, 5]$	2	$(0, 10]$	20
	s	$[2, 10]$	3	$[1, 20]$	20
$M/M/s/n/n$	a	$[1, 5]$	3	$[1, 8]$	10
	μ	$[0.5, 2]$	3	$[1, 4]$	10
	s	$[1, 100]$	12	$[6, 100]$	32
	n	$[2, 112]$	12	$[8, 100]$	24

3.2 Discovered functions

In this section the best function discovered during this study are presented and compared to the analytical formulas from appendix A.

3.2.1 M/M/1

$$\widetilde{w_q}(\rho, \mu) = \frac{\rho}{\mu(1-\rho)} \quad \textit{where} \quad \rho < 1$$

The $M/M/1$ queue has a simple two dimensional shape therefore GP was able to discover a function which is exactly the same as the analytically derived function A.1.

3.2.2 M/M/s

$$\begin{aligned} \widetilde{w_q}(\rho, \mu, s) &= \frac{\rho}{s^2\mu(1-\rho)} \quad \textit{where} \quad \rho = a/s < 1 \\ &= \frac{a/s^2}{s\mu - a\mu} \end{aligned}$$

The discovered waiting time function for the $M/M/s$ queue looks like an extension of the $M/M/1$ function, extended by s^2 in the denominator. After rewriting it, one can observe that it is similar to equation A.2. The main difference being the numerator, where the complex expression $C(s,a)$ is approximated by a/s^2.

3.2.3 M/M/s/n/n

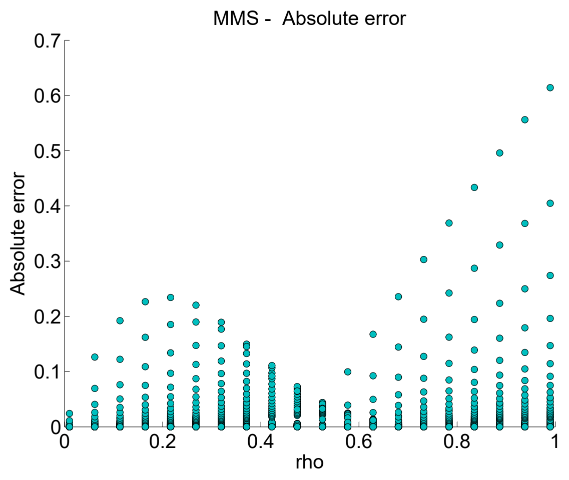
$$\widetilde{l_q}(a, \mu, s, n) = n\mu \cdot \left[\mu + \frac{2\mu}{\mu+n} + \frac{n+\mu s(s+2\mu)}{(s+n)(\mu+n)} + \frac{(s+\mu)\mu}{a^2 n}\right]^{-1}$$

No similarities can be observed between the discovered functions and A.3.

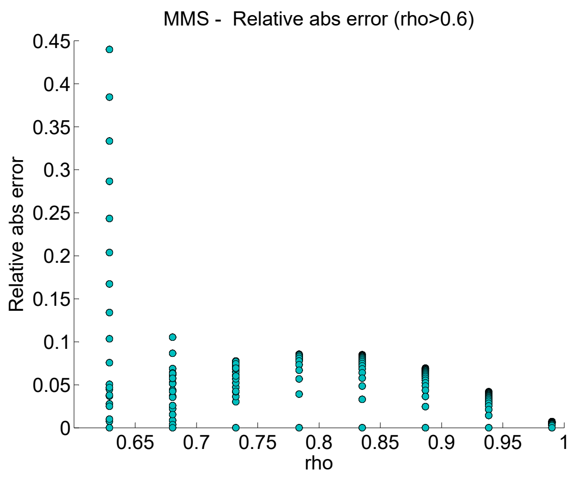
3.3 Error analysis

Since GP found the exact function for the $M/M/1$ model, it is omitted from error analysis. The test data used for error analysis, for the Erlang and Engset model, can be found in table 3.2.

From figure 3.1 one can observe that the discovered waiting time function for the $M/M/s$ queue does not perform well where ρ is smaller than 0.6. Systems with a low load have an expected waiting time of a much smaller magnitude than systems with a medium or high load. Therefore, GP was unable to find a (simple) function that fits both small and high loads well. This is why SAD was used as fitness function rather than MRD. Nevertheless, one is usually interested in analyzing systems that endure a medium or high load. One can observe from figure 3.2 that the discovered function performs well for ρ larger than 0.6. The relative error seems normally distributed around 0 and 99% of the test data has an error below 10%.

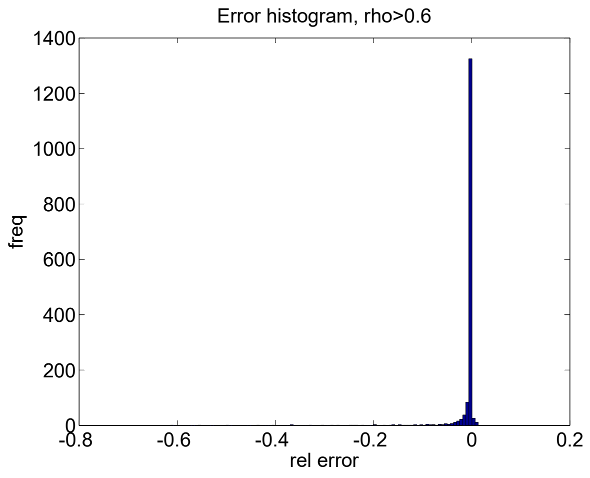


(a) Absolute error vs load per server (ρ)

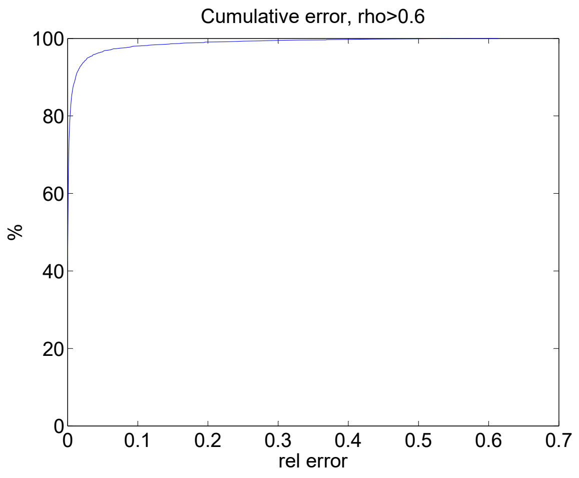


(b) Relative error vs load per server ($\rho > 0.6$)

Figure 3.1: Absolute and relative error plots of the discovered function for the $M/M/s$ queue

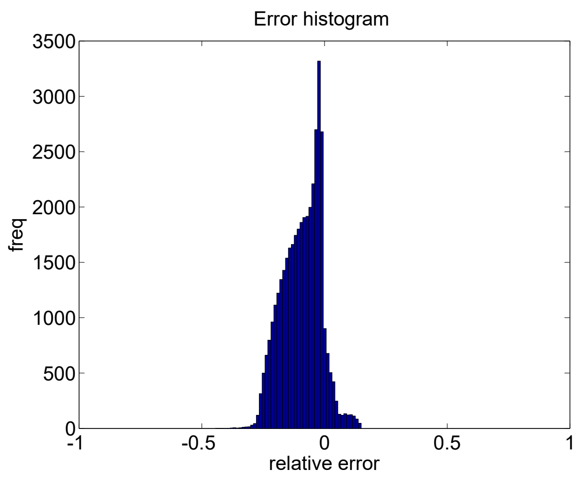


(a) Relative error histogram

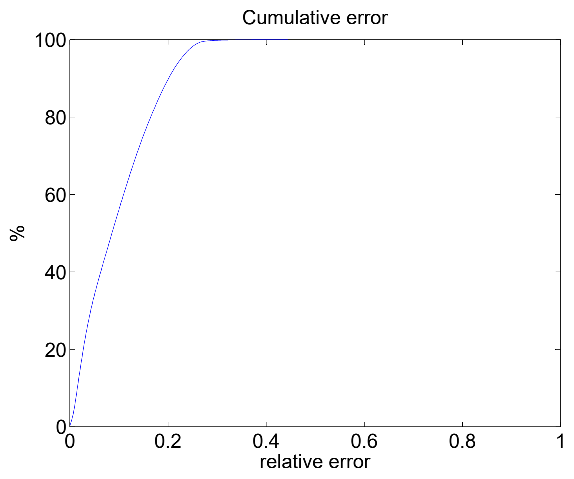


(b) Cumulative relative error distribution

Figure 3.2: Relative error distribution plots of the discovered function for the $M/M/s$ queue where $\rho > 0.6$.



(a) Relative error histogram



(b) Cumulative relative error distribution

Figure 3.3: Relative error distribution plots of the discovered function for the $M/M/s/n/n$ queue

Figure 3.3 visualizes the error of the discovered function for the $M/M/s/n/n$ queue. The histogram shows that the approximation is an under estimate of the function we are approximating. From the other plot (b) one can observe that 80% of the test data has an error smaller than 20%. Most of the errors that are higher than 20% occurs when parameters s and n are close to each other. This flaw is visualized in figure 3.4.

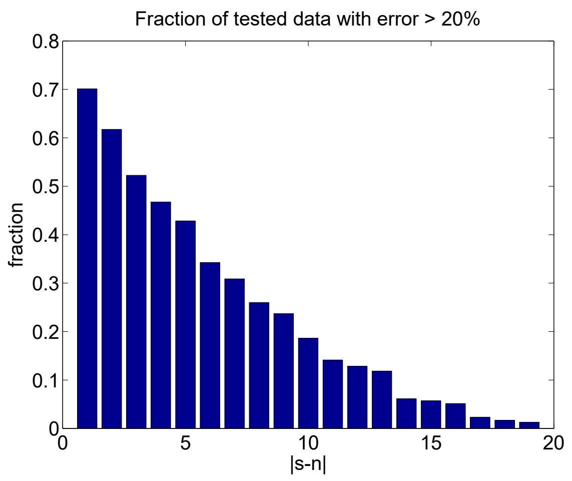


Figure 3.4: Fraction of tested data with error larger than 20% vs the absolute difference between parameters s and n

3.4 Practical application

The waiting time or queue length functions are often used to optimize the number of servers of a system. For example, a call center might want to know the number of employees they need during a shift to keep the expected caller waiting time below a certain threshold. Tables 3.3 and 3.4 illustrate the performance of the discovered functions when used in an optimization context.

The suggested number of servers s is calculated with the analytical function and $\widetilde{s}$ with the discovered function. The thresholds are given by w_q and l_q for the Erlang C and Engset delay model respectively.

Table 3.3: $M/M/s$ queue

a	μ	w_q	s	$\widetilde{s}$
40	0.1	0.1	43	43
20	0.05	0.1	27	26
3	1	0.2	5	5

Table 3.4: $M/M/s/n/n$ queue

a	μ	n	l_q	s	$\widetilde{s}$
1	1	100	70	30	32
2	0.5	150	100	102	108
1	0.1	50	30	20	19

Chapter 4

Discussion

The results of this study show that GP was able to discover useful approximation functions for the considered models, which vary in complexity. However, obtaining robust approximations becomes harder for more complex models. More model parameters require more samples to capture the shape of the underlying function. This might form an obstacle when samples are not easily obtained as in our case. More model parameters also increase the difficulty for GP to find a well fitting function because of the higher dimension of the underlying function. A solution for this could be to reduce the dimension of the model by making one or more input parameters constant. This way one would obtain a problem specific function.

Since the underlying functions of more complex queueing models are probably no simple expressions, GP is not likely to discover the exact one when using only simple arithmetic operators. Therefore, we stress the importance of error analysis. Error analysis is not only necessary to prevent overfitting, but also to expose flaws of the discovered function. Identifying such flaws allows one to use the discovered function appropriately by avoiding its weaknesses. Future work should focus on discovering approximation functions, with acceptable weaknesses, for models with unknown performance functions.

Furthermore, our secondary aim was to obtain robust functions of minimal size/complexity so that they might provide additional insights into the considered queueing models. However, extracting these insights from the discovered functions is left to the reader. Note that the discovered function for the $M/M/s$ queue contains characteristics from the related $M/M/1$ queue. As suggested by [5], a future study could include function characteristics from related models into the initial population. Initial experiments during this study support the idea that this may speed up and aid the function discovery process. If one is only interested in accuracy, then simplicity of the function can be sacrificed in favor of accuracy. For example, this can be done by populating the initial generation with larger trees or by relaxing the bloat control parameters.

Bibliography

[1] D. Barash. "a genetic search in policy space for solving Markov decision processes". Technical report, AAAI, 1999. SS-99-07. 2

[2] S. Bhulai and G. Koole. *"Stochastic Optimization"*. 2014. 2

[3] A. E. Eiben and J. E. Smith. *"Introduction to Evolutionary Computing"*. Springer, 2013. 2, 5

[4] G. Koole. *"Optimization of Business Processes"*. MG books, 2014. 2

[5] M. Onderwater, S. Bhulai, and R. D. v. d. Mei. "value function discovery in Markov decision processes with evolutionary algorithms". 2016. 2, 17

Appendix A

Queueing formulas

$$\begin{aligned} \lambda &:= \textit{arrival rate} & n &:= \textit{customer population size} \\ \mu &:= \textit{service rate} & a &:= \textit{load in Erlang } (\lambda/\mu) \\ s &:= \textit{number of servers} & \rho &:= \textit{load per server } (a/s) \end{aligned}$$

$$\begin{aligned} L &:= \textit{number of customers in the system} \\ W_Q &:= \textit{time an arbitrary customer spends waiting in a queue} \end{aligned}$$

<u>M/M/1</u>

$$\mathbf{E}W_Q = \frac{\rho}{\mu(1-\rho)} \qquad \textit{where } \rho < 1 \tag{A.1}$$

<u>M/M/s</u>

$$\begin{aligned} \mathbf{E}W_Q =& \frac{C(s,a)}{s\mu - a\mu} \qquad \textit{where } \rho < 1 \\ \textit{with} \quad C(s,a) =& \frac{a^s}{(s-1)!(s-a)} \left[\sum_{j=0}^{s-1} \frac{a^j}{j!} + \frac{a^s}{(s-1)!(s-a)} \right]^{-1} \end{aligned} \tag{A.2}$$

<u>M/M/s/n/n</u>

$$\begin{aligned} \mathbf{E}L^{n-1} =& \sum_{j=1}^{n-1} j\pi^{n-1}(j) \\ \textit{with} \quad \pi^n(j) =& \binom{n}{j} a^j \pi^n(0) \quad \textit{if} \quad 0 \le j \le s \\ \pi^n(j) =& \frac{n!}{(n-j)!s!s^{j-s}} a^j \pi^n(0) \quad \textit{if} \quad j > s \\ \pi^n(0)^{-1} =& \sum_{j=0}^{s} \binom{n}{j} a^j + \sum_{j=s+1}^{n} \frac{n!}{(n-j)!s!s^{j-s}} a^j \end{aligned} \tag{A.3}$$