Why Are We Waiting? Discovering Interpretable Models for Predicting Waiting and Sojourn Times

Boris Wiegand° • Dietrich Klakow• Jilles Vreeken*

Abstract
Queueing models explain waiting times, predict sojourn times and help to identify and avoid bottlenecks. Domain experts usually create these models by intensive handcrafting, often resulting in idealized models not fitting the actual process behavior well. Discovering queueing models from data can alleviate this effort, but existing methods do not suffice as they are unable to model complex queueing behaviors.

We propose a novel approach to discover queueing models for interpretable waiting time prediction using a rich modeling language to fit complex processes. We formalize the problem in terms of the Minimum Description Length (MDL) principle, by which the best model gives the best lossless compression. The resulting optimization problem is computationally hard, and hence we propose the greedy CueMin algorithm to efficiently find good queueing models from data. Through an extensive set of experiments including a case study on call center data, we show it discovers inherently interpretable models, which explain and predict behavior of waiting lines better than the state of the art.

1 Introduction

We have all stood in a waiting line, wondering why is it taking so long and how much longer we have to wait. Explaining and predicting waiting times is a highly relevant topic in service-oriented and manufacturing processes. Process time prediction methods [18, 25] usually assume independence between jobs and neglect varying waiting times due to queueing. On the contrary, waiting time is the core concept of queueing models [21], in which servers process incoming jobs. If all servers are busy, arriving jobs must wait until a server becomes available. Although queueing models have been used in many domains such as customer service, traffic control, manufacturing and healthcare [8], modeling processes typically involves intensive handcrafting by domain experts, which often results in idealized models that do not fit the actual process behavior well [26].

Existing approaches to discover queueing models from observational data [23, 24] are restricted to first-come first-serve order, which results in poor fitness on processes with different behavior. They generally discover only one specific part of a queueing model, such as number of servers [11] or batch sizes [14], and require expert knowledge for the remaining parts. Neural networks can predict service times with high accuracy [17], however, they require large training datasets and extensive hyperparameter tuning. Their black box nature impedes what-if analysis like what happens if we increase the number of available servers.

In practice, we frequently face datasets with arrival and departure times of jobs, but without any knowledge about the underlying waiting and service times [23]. We propose a novel approach to discover interpretable queueing models with rich modeling language from such data. To this end, we formalize the problem in terms of the Minimum Description Length (MDL) principle, by which we identify the best model as the one giving the shortest lossless description of the data. Since the resulting optimization problem is computationally hard, we propose our greedy algorithm CueMin (a pun of the spice cumin and cue miner) to find good queueing models in practice. CueMin discovers the key parts of a queueing model, i.e. service order, number of servers, batch sizes and service time. Furthermore, CueMin considers additional features in the data, such as the type of product in manufacturing, to explain service order and to predict service time.

Through extensive experiments on synthetic and real-world data including a case study on call center data, we show that CueMin in contrast to the state of the art discovers inherently interpretable models, which explain and predict behavior of waiting line processes. Our main contributions are as follows. We

(a) formulate the problem of discovering queueing models in terms of the MDL principle,
(b) propose an efficient heuristic to find interpretable yet accurate models to predict waiting and sojourn time from data,
(c) perform an extensive empirical evaluation.
(d) make code, data and additional details of our empirical evaluation available in the supplementary.[1]

°Stahl-Holding-Saar, Dillingen, Germany. boris.wiegand@stahl-holding-saar.de

•Saarland University, Saarbrücken, Germany. dietrich.klakow@lsv.uni-saarland.de

*CISPA Helmholtz Center for Information Security, Germany. jv@cispa.de

[1] https://eda.rg.cispa.io/prj/cuemin

2 Preliminaries

Before we formalize the problem, we introduce necessary notation and basic concepts we use in the paper.

2.1 Univariate Discrete Distributions We model time intervals and batches of jobs using univariate discrete distributions [10]. Favoring a concise notation, we write, whenever clear from context, $\Pr(x)$ instead of $\Pr(X = x)$ when we mean the probability mass function (pmf). Below, we consider four distributions which are particularly commonly used in queueing theory. We note, however, that our theory accepts any distribution with a pmf of a finite set of parameters. The simplest distribution we consider is a degenerate, or *fixed* distribution $\mathrm{F}(k)$. We use it to model constant values, because it has only support for a single value $k \in \mathbb{N}$, i.e. $\Pr(k) = 1 \wedge \forall x \neq k : \Pr(x) = 0$. As a more flexible distribution, we denote the Poisson distribution with expected value k as $\mathrm{P}(k)$ with pmf $\Pr(x) = \frac{e^{-k}k^x}{x!}$. For a geometric distribution with success probability p and pmf $\Pr(x) = (1-p)^{x-1}p$, we write $\mathrm{G}(p)$. The negative binomial distribution $\mathrm{NB}(k, p)$ with number of successes k and success probability p has pmf $\Pr(x) = \binom{x+k-1}{k-1}p^k(1-p)^x$.

2.2 Queueing Models In queueing theory [8, 21], a queueing model M consists of c servers that process incoming jobs. We denote the arrival time of the i-th job as $a_i \in \mathbb{N}$. If all servers are busy, jobs must wait until a server is available. We use $w_i \in \mathbb{N}$ to refer to the waiting time of job i. Servers can process jobs in batches. We refer to B as the univariate discrete batch size distribution. If the current batch size is k, a server waits until k jobs are available to get served. Servers process waiting jobs in service order R. We consider first-come, first-served (FCFS), last-come, first-served (LCFS) and priority queueing (PQ), where jobs own priority classes and jobs with same priority are either served FCFS (PQ + FCFS) or LCFS (PQ + LCFS).

The service time of a job is drawn from the service time model S. In its simplest form, S is a univariate discrete distribution. If service time depends on additional features like heavier products in manufacturing need more time, S can be a regression function $f_S : \mathbb{R}^m \to \mathbb{R}$, plus an additive error distribution E_S. Load on the system may lead to different service times. Therefore, S can consist of k submodels $S'_1, \cdots, S'_k$, where $k-1$ load thresholds $\tau_1, \cdots, \tau_{k-1} \in \mathbb{N}$ define when to use which submodel. If the number of jobs in the queue is between τ_{j-1} and τ_j, service time is predicted by S'_j. We denote the service time of the i-th job as s_i.

When a job has been served, it leaves the system. We denote the departure time of the i-th job as $d_i = a_i + w_i + s_i$. The sojourn time v is the time span between arrival and departure, i.e. $v_i = d_i - a_i = w_i + s_i$.

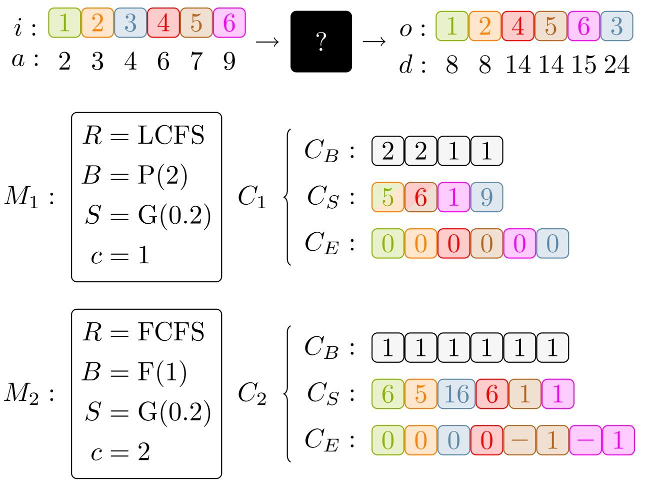

Figure 1: **[Data Encoding]** Toy example of jobs (top left) with arrival times a, departure times d, and two possible processing models M_1 and M_2 and their corresponding encodings C_1 and C_2 of the data.

2.3 MDL The Minimum Description Length (MDL) principle [7, 19] is an information theoretic approach for model selection. MDL identifies the best model as the one that provides the shortest lossless description of the given data. Formally, given a set of models $\mathcal{M}$, the best model is defined by $\arg\min_{M \in \mathcal{M}} L(M) + L(D \mid M)$, with $L(M)$ being the number of bits required to describe M, and $L(D \mid M)$ being the length of the data encoded with the model. This form of MDL is known as two-part or crude MDL. One-part or refined MDL provides stronger theoretical guarantees, however, it is only computable in specific cases [7]. Therefore and because we are especially interested in the model, we use two-part MDL. In MDL, we only compute code length, but are not concerned with actual code words.

3 MDL for Queueing Models

We favor queueing models that are simple and interpretable yet at the same time are sufficiently rich to fit real-world process behaviors. Therefore, we formalize the problem of discovering a queueing model in terms of the MDL principle. We encode the data given a model with codes in a code stream or *cover* C, where we specify how the model serves arriving jobs. Conceptually, we split C into three streams: C_B encodes the batch sizes in which jobs are served, C_S encodes service times, and C_E encodes errors to ensure a lossless encoding.

We show a toy example of data, model and cover in Figure 1. Model M_1 consists of a single server that processes jobs in LCFS order, batch sizes are Poisson and service times are geometrically distributed. Now, we decode the departure times d from the arrival times a using cover C_1. We start by reading the size of the first batch from C_B, which tells us the next two jobs are served in a batch. Then, we read the service time of this batch from C_S. Now, we know the server waits until the first two jobs arrive and needs five time steps to serve this batch, which results in a departure time of eight for both jobs. For each of the jobs, we read a code from C_E to correct the departure time if necessary. In this example, we read 0, i.e. the observed departure time equals the departure time given by the model.

We continue by reading the next batch size, two, from C_B. When the server becomes free at time step eight, job 3, 4 and 5 are waiting. Due to LCFS order, job 4 and 5 are served next. We read the code for service time six from C_S, which results in departure time 14. The next two codes in C_E tell us that 14 is correct. We decode the remaining two jobs as before and are done.

In model M_2, we have two servers processing jobs in FCFS order with batch size one. Now, we use cover C_2 to decode departure times of the arriving jobs. Job 1 and job 2 are served in batches of size one using separate servers. After the first two jobs leave, job 3 blocks one server until the end of our example, and when job 4 leaves at time step 14, job 5 requires zero service time. However, the geometric service time distribution of M_2 does not allow zeros. Therefore, C_S contains a code for service time one, which is corrected by two codes in C_E, giving us sign and magnitude of the error. We decode job 6 analogously by which we decoded all jobs.

3.1 Data Encoding We define length of the data encoding as the sum of the code lengths in the code stream C. Formally, we have

$$L(D \mid M) = -\sum_{b \in C_B} \log \Pr_B(b) - \sum_{s \in C_S} \log \Pr_S(s) + \sum_{e \in C_E} L(e),$$

where we first encode the batch sizes with optimal prefix-free codes using the model's batch size distribution, then we encode the service times also with optimal prefix-free codes using the service time model, and we encode the errors of the modeled departure times, which ensures a lossless encoding as required by MDL.

We encode the error e by first encoding its sign $\operatorname{sgn} e \in \{-1, 0, 1\}$ and then its magnitude. If we knew the distribution of the signs beforehand, we could compute lengths of optimal prefix-free codes with Shannon entropy. To avoid any arbitrary choices in the model encoding, we use prequential codes [7], which are asymptotically optimal without requiring initial knowledge of the code distribution. We start encoding with a uniform distribution and update the counts after every received message, such that at any point of time we have a valid probability distribution for optimal prefix-free codes [4]. Formally, we define the encoded length of the error by

$$L(e) = -\log \frac{\operatorname{usg}(\operatorname{sgn} e) + \epsilon}{\sum \operatorname{usg}(\cdot) + \epsilon} + \begin{cases} 0, & \text{if } e = 0 \\ L_{\mathbb{N}}(|e|), & \text{otherwise,} \end{cases}$$

where $\operatorname{usg}(\operatorname{sgn} e)$ denotes how often the code for $\operatorname{sgn} e$ has been used before, ϵ with standard choice 0.5 is for smoothing, and $L_{\mathbb{N}}$ is the MDL-optimal encoding for integers $z \geq 1$ [20], defined as $L_{\mathbb{N}}(z) = \log^* z + \log c_0$, with $\log^* z = \log z + \log\log z + \ldots$ and we sum only the positive terms, and $c_0 = 2.865064$ is set such that we satisfy the Kraft-inequality – i.e. ensure it is a lossless code. This gives us a lossless encoding of the data.

3.2 Model Encoding We encode all parts of the model. For the service order R, we use $\log 3$ bits to encode whether we have FCFS, LCFS or PQ. In case of PQ, we additionally encode, which of the categorical features in our dataset contains the priority classes, and encode the order of the categories by an index over all possible orders. Since multiple waiting jobs can have the same priority class, we use one bit to encode whether the default order is FCFS or LCFS. This results in

$$L(R) = \log 3 + \begin{cases} \log m_{cat} + \log k! + 1, & \text{if } R = \text{PQ} \\ 0, & \text{otherwise,} \end{cases}$$

where m_{cat} denotes the number of categorical features and k the number of categories of the chosen feature.

For the batch size distribution B, we specify the type of the distribution and encode its parameters. Distinguishing between four types of univariate discrete distributions costs two bits. We encode integer parameters with $L_{\mathbb{N}}$ and real parameters with $L_{\mathbb{R}}$ [16]. The idea is to encode a real number z up to a user-specified precision p by the smallest integer shift s such that $z \cdot 10^s \geq 10^p$. We then encode shift, shifted digit and sign, i.e. $L_{\mathbb{R}}(z) = L_{\mathbb{N}}(s) + L_{\mathbb{N}}(\lceil z \cdot 10^s \rceil) + 1$.

We defined three different types of service time models S, hence, encoding the type costs $\log 3$ bits. If S is a distribution, we encode it like the batch size distribution. In case of a regression function, we encode the parameters using $L_{\mathbb{R}}$ and encode the error distribution as before. If service time is load-dependent, we encode the number and values of the load thresholds using $L_{\mathbb{N}}$, and encode the submodels accordingly. We define the encoded length of the model by $L(M) = L(R) + L(B) + L(S) + L_{\mathbb{N}}(c)$, which gives us a lossless encoding of the model.

3.3 Formal Problem Definition

We now have all necessary parts to formally state the problem.

Minimal Queueing Model Problem *Given a dataset D of arrival and departure times, find the minimal queueing model M and cover C, such that the total encoded cost $L(D, M) = L(D \mid M) + L(M)$ is minimal.*

Finding the optimal cover is computationally hard: There is no product-form solution to compute the future state of our queueing models [12], i.e. whenever we choose batch size and service time at one point of the cover, we have to compute the impact on all later time steps. Due to many valid choices of batch size and service time at each step, this results in an intractable, exponentially growing search space.

Finding the optimal model is not easier. Without a product-form solution for queueing states, every change in the model requires re-computation of the cover. We cannot search for different parts of a model independently: Different service orders lead to completely different service times, and a change of batch sizes requires adapting service times or the number of servers. Hence, we resort to heuristics.

4 The CueMin Algorithm

Since solving the minimal queueing problem is difficult, we divide it into two and propose greedy solutions for finding a cover and discovering a model separately.

4.1 Finding a Cover

To find a good cover, we first need to know which jobs are served as one batch, such that we can estimate corresponding service times. Similar to the existing BatchMiner [14], we discover batches by jobs with the same departure time, however, we restrict batch sizes to values that we can explain by the batch size distribution B of the model.

We give the pseudocode of FindBatches as Algorithm 1. Initially, all servers j have empty batches at any time t (line 1). To consider all changes in the model state, we iterate over all arrival and departure times (line 2). In each iteration, we try to find a suitable batch for each job in the waiting line. We first check if we should add the job to an already existing batch (line 4): We add the job to a batch if the model demands a higher batch size ($\Pr_B(|b_j^t|) = 0$), or if the job has the same departure time than the jobs in the batch ($b_j^{d_i} \neq \emptyset$) and the model supports a larger batch ($\Pr_B(|b_j^t| + 1) > 0$). In this case, we mark the job to be part of batch b_j blocking server j until departure time d_i (line 5). If we could not add the job to an existing batch, we create a new batch if there is a free server (line 6-7). Whenever a batch has been processed, we add it to our list of detected batches (line 10).

Algorithm 1: FindBatches

```
   input  : dataset $D$, queueing model $M$
   output: list of detected batches $Z$
1  $b_j^t \leftarrow \emptyset\ \forall j, t$;
2  foreach $t \in \{a_1, \ldots, d_n\}$ do
3      foreach job $i$ waiting for $M$ at time $t$ do
4          if $\exists j : b_j^t \neq \emptyset$ and $i$ should be in $b_j^t$ then
5              $b_j^k \leftarrow b_j^k \cup \{i\}\ \forall k = t, \ldots, d_i$;
6          else if $\exists j : b_j^t = 0$ then
7              $b_j^k \leftarrow \{i\}\ \forall k = t, \ldots, d_i$;
8      foreach $j \in \{1, \ldots, c\}$ do
9          if $b_j^{t+1} = \emptyset$ then
10             add $b_j^t$ to $Z$;
11 return $Z$
```

Algorithm 2: FindCover

```
   input  : dataset $D$, queueing model $M$
   output: cover $(C_B, C_S, C_E)$
1  $C_B \leftarrow \emptyset, C_S \leftarrow \emptyset, C_E \leftarrow \emptyset, t \leftarrow 0$;
2  foreach $b \in$ FindBatches$(D, M)$ do
3      append $\boxed{|b|}$ to $C_B$;
4      $s \leftarrow d_{b_1} - t$;
5      if $\Pr_S(s) = 0$ then
6          $s \leftarrow \arg\max \Pr_S(\cdot)$;
7      append $\boxed{s}$ to $C_S$;
8      foreach $i \in b$ do
9          $e_i \leftarrow s + t - d_i$;
10         append $\boxed{\operatorname{sgn} e_i}$ to $C_E$;
11         if $e_i \neq 0$ then
12             append $\boxed{|e_i|}$ to $C_E$;
13     $t \leftarrow$ service start time of next batch;
14 return $(C_B, C_S, C_E)$
```

With the help of FindBatches, computing the cover is fairly easy. We provide the pseudocode of FindCover as Algorithm 2. Starting with an empty cover, we iterate over all batches. We add the batch size to C_B (line 3) and compute the required service time s (line 4), where t denotes the current time.

If the model cannot produce the required service time s, we replace s with the most likely service time of the model (line 5-6). Next, we add the code of s to C_S (line 7). Then, for each job in the batch, we compute the error on the departure time and add its code to C_E (line 8-12). At the end of each iteration, we set the current time t to the service start of the next batch (line 13).

Algorithm 3: BruteForce

```
input  : dataset $D$, service order $R$, upper bound on number of servers $c_{\max}$
output: queueing model $M$
1 $M \leftarrow \emptyset$;
2 foreach $c \in \{1, \ldots, c_{\max}\}$ do
3     $\hat{M} \leftarrow (R, B, S, c)$;
4     foreach $S, B \in$ FitService($D$,$R$,$c$) do
5         if $L(D, \hat{M}) < L(D, M)$ then
6             $M \leftarrow \hat{M}$;
7 return $M$
```

Algorithm 4: CueMin

```
input  : dataset $D$, service order $R$, upper bound on number of servers $c_{\max}$
output: queueing model $M$
1 $M \leftarrow \emptyset$;
2 $c \leftarrow 1, \delta \leftarrow 1$;
3 repeat
4     foreach $S, B \in$ FitService($D$,$R$,$c$) do
5         $\hat{M} \leftarrow (R, B, S, c)$;
6         if $L(D, \hat{M}) < L(D, M)$ then
7             $M \leftarrow \hat{M}$;
8     if $L(D, M)$ improved then
9         $\delta \leftarrow \delta \cdot 2$;
10    else if $\delta = 1$ then
11        $\delta \leftarrow -1$;
12    else
13        $\delta \leftarrow \lceil \frac{\delta}{2} \rceil$;
14    $c \leftarrow \max\{1, \min\{c_{\max}, c + \delta\}\}$;
15 until $\delta = 0$;
16 return $M$
```

4.2 Discovering a Good Queueing Model With FindCover, we are now able to compute our MDL score, which we now use for model selection. The overall idea is to discover a model for each possible service order and take the one with the lowest total encoding cost. While FCFS and LCFS are non-parametric, we select the most promising categorical attribute for priority queueing. To this end, we choose the attribute and the permutation of its categories π that interpreted as priority classes minimize the conditional entropy on the departure order of jobs. Formally, if Y is a random variable of the category of the next leaving job and X is a random variable of the predicted category by the order of π, we minimize $H(Y \mid X) = -\sum_{x \in X, y \in Y} \Pr(x, y) \log \frac{\Pr(x,y)}{\Pr(x)}$.

We restrict the search space by a simple yet effective upper bound on the number of servers, which is the smallest number of servers such that all jobs can be served without waiting time. Any greater number leads to unused servers and cannot be inferred from data. For reference, we give the pseudocode of our basic search algorithm in the supplementary.

BruteForce We propose two different strategies to discover a model with a given service order. The first one is a naive brute-force search, for which we give the pseudocode as Algorithm 3. We generate service time and batch size distribution candidates for each possible number of servers, and select the model with the best score. We always generate the batch size distribution candidate F(1), i.e. batch size is constantly one. In addition, we use maximum likelihood estimation to fit a candidate for each type of distribution we introduced in Section 2 on the number of jobs with the same departure time. We then use the batch size distribution candidates to compute required service times for each job as we did in line 4 of FindCover in Algorithm 2.

Next, we generate service time candidate models. We fit distributions by maximum likelihood estimation. For regression models with an error distribution, we first fit the regression model and then fit a distribution on the residuals. We generate load dependent service time models by finding $\tau_1, \ldots, \tau_{k-1}$ for multiple values of k, and then fit k submodels, i.e. distributions or regression models. For given k, we choose $\tau_1, \ldots, \tau_{k-1}$ such that we minimize $\sum_{i=1}^{k} \sum_{s \in K_i} (s - \bar{K}_i)^2$, with K_i being the i-th cluster of service times implied by $\tau_1, \ldots, \tau_{k-1}$ and $\bar{K}_i$ the average service time of the cluster.

CueMin We propose CueMin as an efficient alternative to BruteForce. Although our score is not strictly convex, it does exhibit convex-like behavior that we can exploit towards discovering good models. In particular, whenever a model contains *too few* servers, it will incur a heavy penalty because it has trouble serving jobs on time; adding more servers will reduce this penalty. In contrast, when a model has *too many* servers, it also incurs a heavy penalty because it serves jobs too early; reducing servers reduces this penalty.

We give the pseudocode of CueMin as Algorithm 4. We start by finding the best model for one server. Whenever the current number of servers leads to an improvement, we increase the step size δ (line 7-8), which determines the next candidate number of servers (line 13). This way, if the number of optimal servers is large, we quickly jump over values, which are much too low. At some point, large steps do not lead to better models. We then start to decrease the step size (line

12). If increasing the number of servers does not have an effect anymore, we repeat the search in the opposite direction in case we jumped over the optimum (line 10).

As we show in Section 6, CueMin works well. It finds models as good as those by BruteForce, while being significantly faster. Domain experts can, if wanted, easily include knowledge into the search: They can restrict the number of servers to speed up the search or adapt any part of the model to their expectation. For instance, they can create a domain-specific service order or service time distribution.

4.3 Runtime complexity FindBatches has runtime $O(nc)$, i.e. it scales linearly with the number of observed jobs n and the number of servers c. Hence, it is fast enough to compute sufficiently many covers during model search. The runtime of FindCover is mainly driven by FindBatches, i.e. $O(nc)$.

The runtime of BruteForce is strongly dependent on the upper bound on the number of servers $c_{\max}$. In each of the $c_{\max}$ iterations, we compute our score with runtime $O(nc)$, which results in a total runtime of $O(nc^2_{\max})$. Depending on the dataset and the underlying data generating process, $c_{\max}$ is large enough, such that this runtime becomes unacceptable. In practice, we can either restrict the number of servers with domain knowledge or use heuristics to speed up the search.

Instead of exhaustively searching over all possible number of servers, CueMin uses an adaptive step size δ to skip unpromising candidates. Since our score is not convex, in the worst case, every second search candidate improves the score. This means, δ alternates between one and two, and we test all $c_{\max}$ possible values of c. In practice, however, CueMin is significantly faster than BruteForce as we show in Section 6.

5 Related Work

Before we show our evaluation, we first give an overview of related work. Predicting event duration in business processes is closely related to our problem. Polato et al. [18] predict future events of running process instances based on a Naïve Bayes classifier and use support vector regression to estimate event duration. Deep learning leads to more accurate predictions, however, existing approaches [3,25] equally neglect dependencies between jobs and thus cannot consider increased waiting times due to queueing effects in the process.

Surprisingly little work deals with the discovery of queueing models from data. Senderovich et al. [24] coined the term *Queue Mining* and pioneered in synthesizing data mining and queueing theory to predict delays in service processes. In addition to arrival times and departure times, they assumed availability of observations on the waiting times as well.

Later, Senderovich proposed K-PHF for waiting time prediction from arrival and departure times only [23]. Under the assumption of FCFS order, K-PHF clusters sojourn times by k-means and infers a phase-type distribution for each cluster. The clusters correspond to different load states (e.g. low, moderate, high).

Unfortunately, K-PHF does not provide any information about batch service or the number of servers. Keith et al. [11] developed COrder to estimate the number of servers in a FCFS queue. They also proposed a LCFS version of COrder, however, one needs to know the service order to choose the right version.

Klijn and Fahland [14] detected service batch sizes through jobs with close departure time, but did not consider other queue modeling aspects. Ojeda et al. [17] combined queueing theory and adversarial deep learning with Wasserstein loss [1] to predict sojourn times and their distribution from an embedding of arrival times [6] and covariates of jobs in a queue.

In contrast to the above, CueMin finds models for the prediction of process behavior, where all parts of the model are based on interpretable building blocks from queueing theory. To the best of our knowledge CueMin is the first method that jointly discovers batch service, service order, number of servers and service time.

6 Experiments

Now, we evaluate CueMin on both synthetic and real-world datasets. We conducted all our experiments on a PC with an Intel i7-6700 CPU and 32 GB of memory, running Windows 10. We report wall-clock running times for single-threaded execution, except for RAS, which used our GeForce RTX 2080 Ti during training.

6.1 Synthetic Data We start with experiments on synthetic data. We sample 1000 ground-truth models with $R \in \{\text{FCFS}, \text{LCFS}\}$, $c \in [1, 30]$, $B = \text{F}(1)$ and a large set of different service time distributions. We generate data by sampling job arrivals from several interarrival distributions from which we sample departure times with the ground-truth models.

First we evaluate CueMin's ability to find the ground-truth number of servers compared to BruteForce, COrder [11] and a naive baseline DeltaMax [11]. Based on the intuition that more servers are required to change departure order of a deterministic service order, DeltaMax enumerates arriving jobs from 1 to n, and estimates the number of servers by the maximal difference of leaving jobs. For a fair comparison, we feed our knowledge of the ground-truth service order into COrder, whereas CueMin and BruteForce additionally have to discover the service order.

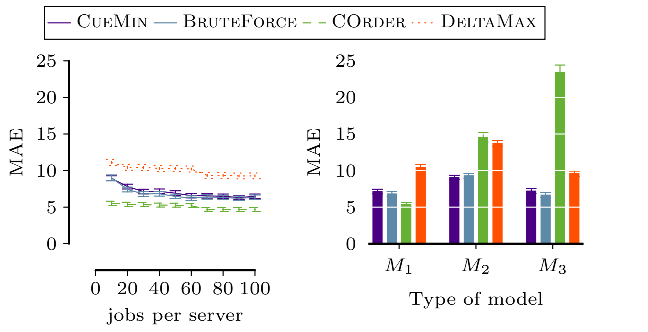

Figure 2: [**Number of servers estimation**] Mean absolute error (MAE) on estimating the number of servers $c \in [1, 30]$ of a $R \in \{\text{FCFS}, \text{LCFS}\}$ queue with $B = \text{F}(1)$ dependent on the number of observed jobs per server $\frac{n}{c}$ (left) and MAE dependent on the type of model (right). M_1 is the model of the left plot, M_2 extends M_1 by Poisson batch sizes, and M_3 serves jobs with $R = \text{PQ}$. We show standard error in both plots.

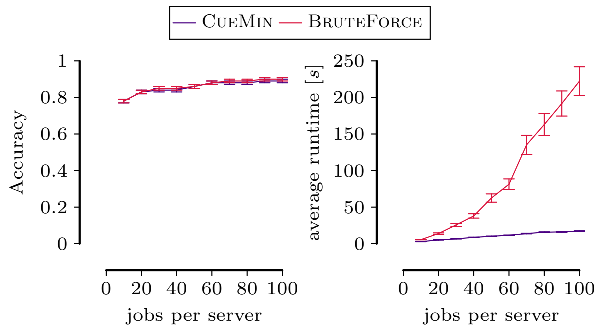

Figure 3: [**Service order discovery**] Accuracy on discovering service order R in synthetic data dependent on the number of observed jobs per server $\frac{n}{c}$ (left) and average runtime in seconds (right) for CueMin and BruteForce. We show standard error in both plots.

We report the mean absolute error (MAE) on estimating the number of servers on the left of Figure 2. Evaluating the MAE dependent on the number of observed jobs only would overpenalize complex models with more servers, which need more jobs to utilize all servers. We therefore normalize the number of jobs by the number of ground-truth servers. As expected, all methods improve with more data. Although we gave COrder the advantage of knowing the ground-truth service order, we see CueMin and BruteForce have a competitive MAE and significantly beat the naive DeltaMax baseline. CueMin produces almost equivalent results to the exhaustive search by BruteForce.

Next, we show the MAE dependent on the type of ground-truth queueing model on the right of Figure 2. The left group of bars (M_1) refers to the left line plot, i.e. the assumptions of COrder still hold. If we add Poisson batch size distributions to the ground-truth models, both COrder and DeltaMax significantly lose accuracy as we show in the middle bar group (M_2), whereas CueMin successfully detects batch service. Service order based on priority classes, i.e. $R = \text{PQ}$, heavily violates the assumptions of COrder. We add three uniformly distributed categorical features with three categories each to the jobs and randomly select one of the features as the priority class. In this scenario (M_3), we see COrder has an even higher increase of MAE. CueMin considers the service order in its search for the number of servers, and thus shows stable performance under varying service order and batch size.

Finally, we report accuracy and runtime on discovering the ground-truth service order for CueMin and BruteForce in Figure 3. We see that CueMin achieves accuracy equivalent to BruteForce, while being magnitudes faster.

6.2 Real-World Data Next, we show practical applicability of CueMin by evaluating on six real-world datasets of different domains. The Callcenter dataset [2] contains service calls along with customer priority, weekday and daytime of an Israeli bank. Laser and Lapping consist of arrival and departures times together with product category and work order quantity at two stations of a production process [15]. With relatively few jobs, they test the ability to learn from little data. Finally, Steel A, Steel B and Steel C correspond to three stations in the rolling mill of a German steel producer.

We split all datasets into a train timespan followed by a test timespan with roughly 20% of all jobs. Then, we use CueMin to discover queueing models on the training data and run 1000 simulations on all arrivals to predict a distribution of sojourn times for each job. We do the same for a re-implementation of the ideas from K-PHF [23] and set hyperparameters as suggested by the authors. Furthermore, we compare to RAS [17] for which we conducted an extensive hyperparameter search and selected the best performing. As an additional baseline, we train a random forest (RF) to predict sojourn times just by the features of jobs and thus ignoring any dependence between jobs and system load. We also train a random forest (AW+RF) on the interarrival times and features in a window of k jobs to consider system load. We select hyperparameters of the random forests by grid search and cross-validation.

To compare across datasets with different time scales, we evaluate predicting sojourn time of individual jobs using the mean absolute scaled error (MASE).

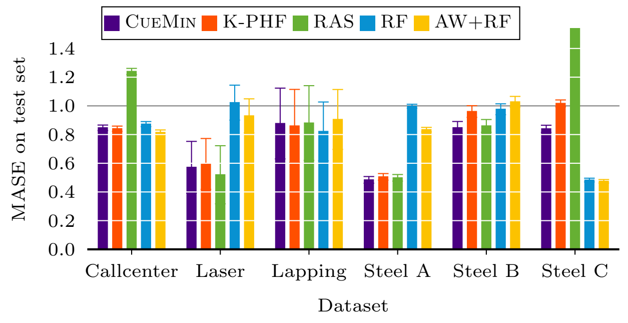

Figure 4: [**Sojourn time predictions**] Mean absolute scaled error (MASE, lower is better) with standard error on predicted sojourn times of six real-world datasets for CueMin, RAS, K-PHF, RF and AW+RF.

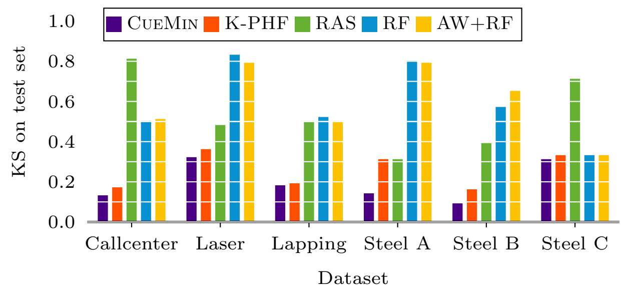

Figure 5: [**Distribution fitting**] KS distance (lower is better) between predicted and actual sojourn time distribution of six real-world datasets for CueMin, RAS, K-PHF, RF and AW+RF.

MASE is defined by the mean absolute error (MAE) of the individual predictor divided by the MAE of the naive predictor that predicts the mean of the training set. We show the MASE on the test set for all methods in Figure 4. We see CueMin always beats the naive baseline. On all datasets, it performs on par or better than K-PHF, and with exception to the Steel C dataset, performs on par or better than RF and AW+RF.

During process performance analysis, domain experts are especially interested in the distribution of sojourn times and not on individual jobs. We report the Kolmogorov-Smirnov (KS) statistic between predicted and actual sojourn time distribution in Figure 5. We clearly see that the random forests suffer from regression to the mean. K-PHF and CueMin provide a much better approximation of the distribution than RF and AW+RF. Between the two, CueMin outperforms K-PHF especially on the Steel A and Steel B dataset.

6.3 Case Study: Call Center We finish evaluation with a case study on the Callcenter dataset, in which we highlight the insight we can gain from the model discovered by CueMin. CueMin finds a FCFS queue with nine servers, batch size one and a load-dependent service time. Although the dataset contains an attribute for customer priority, we see CueMin favors FCFS over PQ. According to the dataset description, customers are served by their waiting time. Prioritized customers are assigned 1.5 minutes of initial waiting time. CueMin discovers that this is a neglible advantage and FCFS captures the actual behavior of the process.

We see batch size one is the correct description for calls being served one after another. Nine servers almost perfectly match the eight agents in the call center. The service time distribution is $\mathrm{NB}(1.4, 0.007)$ if six or fewer customers are in the line, and changes to $\mathrm{NB}(2.2, 0.007)$ if there are more calls. A former study on this dataset [2] confirms increased service time due to system load.

Finally, we evaluate how well the models discovered by CueMin, K-PHF, RF and AW+RF extrapolate. We down- and up-sample the number of arriving jobs in the test set to simulate decreased and increased system load. We show the mean predicted sojourn time $\bar{v}$ and the departure rate μ, i.e. the number of leaving jobs per second, in Figure 6. RF and AW+RF do not capture the expected increase of waiting time for a higher arrival rate: They predict a constant sojourn time and a linearly increasing departure rate. We see that K-PHF does slightly better: Its predicted sojourn time increases with growing arrival rate. However, since it does not model the number of servers, it misses that exceeding a certain load threshold leads to exponentially increasing waiting and thus sojourn times [9].

In contrast to all other methods, CueMin models the dependency between system load and performance

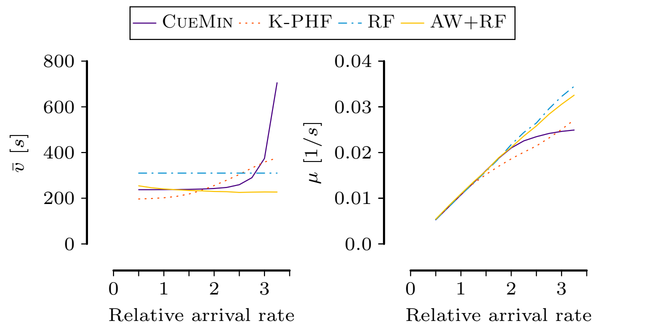

Figure 6: [**CueMin ably extrapolates**] Mean predicted sojourn time $\bar{v}$ (left) and predicted departure rate μ (right) dependent on down- or up-sampled arrival rate in the Callcenter dataset. We expect $\bar{v}$ to grow exponentially with system load, whereas μ should flatten when the maximal capacity of the call center is reached.

that we by human intuition would expect. It predicts the expected explosion of waiting times and a limit on the departure rate, if servers are constantly overloaded. This makes CueMin a valuable tool for analyzing different scenarios in waiting line processes.

7 Conclusion

We studied discovering queueing models for interpretable waiting and sojourn time prediction from data. We formalized the problem in terms of the MDL principle, by which the best model gives the best lossless compression of the data. Due to hardness of the resulting optimization problem, we proposed the greedy CueMin algorithm to find good models in practice. Through an extensive set of experiments and a case study on call center data, we showed it ably discovers inherently interpretable models of queueing processes.

While we achieved reasonable prediction accuracy on real-world processes by using general building blocks from queueing theory, we expect that domain-specific extensions lead to an even better performance. For instance, our service order model could be extended by earliest-deadline-first [5], or by priority auctions [13], where customers bid for priority. Discovery of impatience, i.e. customers leave before served, would increase insights on service-oriented processes. Another enhancement would consider varying availability of servers over time due to vacation or machine breakdown.

Last but not least, we see the opportunity to apply our approach in a network of process stations [22]. In contrast to regression-based remaining time prediction [18, 25], our queueing model is expected to identify congestion effects in these networks.

References

[1] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein generative adversarial networks. In *ICML*, pages 214–223, 2017.

[2] L. Brown, N. Gans, A. Mandelbaum, A. Sakov, H. Shen, S. Zeltyn, and L. Zhao. Statistical analysis of a telephone call center: A queueing-science perspective. *JASA*, 100(469):36–50, 2005.

[3] M. Camargo, M. Dumas, and O. González-Rojas. Learning accurate LSTM models of business processes. In *BPM*, pages 286–302. Springer, 2019.

[4] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* Wiley-Interscience New York, 2006.

[5] B. Doytchinov, J. Lehoczky, and S. Shreve. Real-time queues in heavy traffic with earliest-deadline-first queue discipline. *Ann. Appl. Probab.*, pages 332–378, 2001.

[6] N. Du, H. Dai, R. Trivedi, U. Upadhyay, M. Gomez-Rodriguez, and L. Song. Recurrent marked temporal point processes: Embedding event history to vector. In *KDD*, pages 1555–1564, 2016.

[7] P. Grünwald. *The Minimum Description Length Principle.* MIT Press, 2007.

[8] R. W. Hall. *Queueing Methods: For Services and Manufacturing.* Prentice-Hall, 1990.

[9] O. Handel and A. Borrmann. The relationship between the waiting crowd and the average service time. In *Traffic and Granular Flow '15*, pages 209–216. 2016.

[10] N. L. Johnson, S. Kotz, and A. W. Kemp. *Univariate discrete distributions.* John Wiley & Sons, 2005.

[11] A. Keith, D. Ahner, and R. Hill. An order-based method for robust queue inference with stochastic arrival and departure times. *Computers & Industrial Engineering*, 128:711–726, 2019.

[12] J. F. C. Kingman. The first erlang century – and the next. *Queueing Systems*, 63(1):–12, 2009.

[13] T. Kittsteiner and B. Moldovanu. Priority auctions and queue disciplines that depend on processing time. *Management Science*, 51(2):236–248, 2005.

[14] E. L. Klijn and D. Fahland. Performance mining for batch processing using the performance spectrum. In *BPM*, pages 172–185, 2019.

[15] D. Levy. Production analysis with process mining technology. 2014. 10.4121/uuid:68726926-5ac5-4fab-b873-ee76ea412399.

[16] A. Marx and J. Vreeken. Telling cause from effect by local and global regression. *Knowl. Inf. Sys.*, 60(3):1277–1305, Sep 2019.

[17] C. Ojeda, K. Cvejoski, B. Georgiev, C. Bauckhage, J. Schuecker, and R. J. Sanchez. Learning deep generative models for queuing systems. In *AAAI*, pages 9214–9222, 2021.

[18] M. Polato, A. Sperduti, A. Burattin, and M. de Leoni. Time and activity sequence prediction of business process instances. *Computing*, 100(9):1005–1031, 2018.

[19] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(1):465–471, 1978.

[20] J. Rissanen. A universal prior for integers and estimation by minimum description length. *Annals Stat.*, 11(2):416–431, 1983.

[21] T. L. Saaty. *Elements of Queueing Theory: With Applications.* McGraw-Hill Book Company, 1961.

[22] A. Senderovich, J. C. Beck, A. Gal, and M. Weidlich. Congestion graphs for automated time predictions. In *AAAI*, pages 4854–4861, 2019.

[23] A. Senderovich, S. J. Leemans, S. Harel, A. Gal, A. Mandelbaum, and W. M. P. van der Aalst. Discovering queues from event logs with varying levels of information. In *BPM*, pages 154–166, 2016.

[24] A. Senderovich, M. Weidlich, A. Gal, and A. Mandelbaum. Queue mining – predicting delays in service processes. In *CAiSE*, pages 42–57, 2014.

[25] F. Taymouri, M. La Rosa, and S. Erfani. A deep adversarial model for suffix and remaining time prediction of event sequences. In *SDM*, pages 522–530, 2021.

[26] W. M. P. van der Aalst. *Process Mining – Data Science in Action.* Springer, second edition, 2016.

Algorithm 5: DiscoverQueueingModel

```
input  : dataset $D$, search strategy
         $\phi \in \{\textsc{CueMin}, \textsc{BruteForce}\}$
output: queueing model $M$
1 $M \leftarrow \emptyset$;
2 foreach $R \in \{\text{FCFS}, \text{LCFS}, \textsc{FitPQ}(D)\}$ do
3     $c_{\max} \leftarrow \textsc{UpperBoundC}(D, R)$;
4     $\hat{M} \leftarrow \phi(D, R, c_{\max})$;
5     if $L(D, \hat{M}) < L(D, M)$ then
6         $M \leftarrow \hat{M}$;
7 return $M$
```

A Appendix

Here, we include supplementary material which could not be part of our main paper.

A.1 The CueMin Algorithm We give the pseudocode of the base algorithm for queueing model discovery as Algorithm 5. The overall idea is to discover a model for each possible service order and take the one with the lowest total encoding cost. We restrict the search space by a simple yet effective upper bound on the number of servers, which is the smallest number of servers such that all jobs can be served without waiting time. Any greater number leads to unused servers and cannot be inferred from data.

For a given service order, we propose two different search strategies. BruteForce exhaustively searches over all possible number of servers. CueMin is an efficient alternative to BruteForce and skips unpromising candidates using an adaptive step size.

A.2 Source Code and Data To ensure and facilitate reproducibility of our results, we make our source code and all the data we use in our experiments publically available.[2] We give instructions in the README file how to run the code and reproduce our results.

The real-world datasets in our evaluation have different characteristics. For each dataset, we report the number of jobs n, the number of all features m, the number of numerical features m_{num} and mean sojourn time of train $\bar{v}_{\text{train}}$ and test timespan $\bar{v}_{\text{test}}$ in Table 1. We see from the difference between $\bar{v}_{\text{train}}$ and $\bar{v}_{\text{test}}$ that any method learning from the training data must generalize well to predict behavior of the test data.

A.3 Case Study: Call Center In this section, we extend our call center case study. We predict total

[2]https://eda.rg.cispa.io/prj/cuemin

Data	n	m	m_{num}	$\bar{v}_{train}$	$\bar{v}_{test}$
Callcenter	21703	3	2	246s	191s
Laser	196	2	1	51h	28h
Lapping	224	2	1	79h	129h
Steel A	6969	4	3	82167	44739
Steel B	6961	4	3	1966	2232
Steel C	16458	4	3	440	635

Table 1: [**Real-world datasets**] Number of jobs n, number of all features m, number of numerical features m_{num} and mean sojourn time of train $\bar{v}_{train}$ and test timespan $\bar{v}_{test}$ for six different real-world datasets.

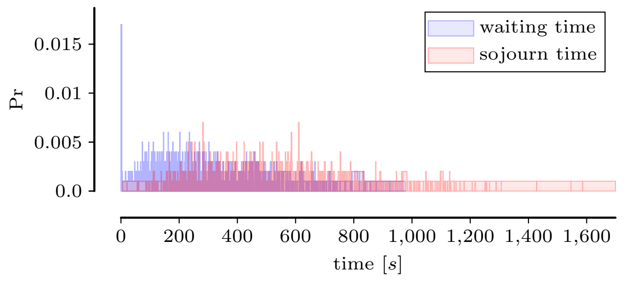

Figure 7: [**CueMin provides detailed predictions**] Empirical predicted waiting time w and sojourn time distribution of a single job in the Callcenter dataset for 1000 simulation runs of a model discovered by CueMin.

call duration, i.e. sojourn time, and waiting time of customers in this dataset by conducting 1000 simulation runs of the model discovered by CueMin. We show the predicted waiting and sojourn time distribution of a single customer in Figure 7. We see CueMin reveals the whole bandwidth of stochastic behavior of a service call. This customer has a high chance to have no waiting time at all, however, if the preceding calls take more time, the waiting time increases. We see a small probability for a call duration of a few seconds due to technical issues, and we see the chance of a very long service call.

CueMin discovers queueing models that are inherently interpretable. On top of that, domain experts can modify the model to simulate different scenarios and to find potential process optimizations. We vary the number of servers in the queueing model found by CueMin and report the maximal predicted waiting time of customers on the left of Figure 8. As we expected, reducing the number of servers results in an exponential growth of waiting time. If we assign costs to the usage of servers and weight them against the risk of losing customers due to high waiting times, we can run such an experiment to find the optimal number of servers.

Finally, we show the impact of different service

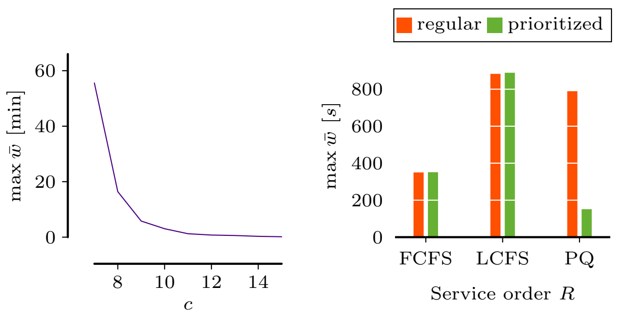

Figure 8: [**How to influence waiting time**] Maximal predicted waiting time $\max \bar{w}$ with varying number of servers c of the queueing model discovered by CueMin (left) and $\max \bar{w}$ with varying service order R for different classes of customers (right).

orders on the waiting time of regular and prioritized customers in the dataset on the right of Figure 8. We see that the model with FCFS service order results in equal and relatively low waiting times for both types of customers. LCFS order increases the maximal waiting time and does not make sense in a call center. If the call center always served prioritized before regular customers, regular customers would face a significant increase of waiting time. This explains as we discussed in Section 6, why in the actual process prioritized customers gain a rather small advantage in waiting time. Therefore, FCFS is a reasonable model.

Acknowledgment

We thank Stahl-Holding-Saar for providing the Steel datasets. Furthermore, we thank Alexander Herzog from TU Clausthal for sharing his valuable knowledge on queueing theory and how domain experts describe real-world process behavior with queueing models.