Online Automated Reliability Classification of Queueing Models for Streaming Processing Using Support Vector Machines

Jonathan C. Beard
Cooper Epstein
Roger D. Chamberlain

Jonathan C. Beard, Cooper Epstein, and Roger D. Chamberlain. "Online Automated Reliability Classification of Queueing Models for Streaming Processing Using Support Vector Machines," in *Proc. of 21st International Conference on Parallel and Distributed Computing* (Euro-Par)*, Lecture Notes in Computer Science,* Vol. 9233, August 2015, pp. 82-93.

Dept. of Computer Science and Engineering
Washington University in St. Louis

Online Automated Reliability Classification of Queueing Models for Streaming Processing Using Support Vector Machines

Jonathan C. Beard(✉), Cooper Epstein, and Roger D. Chamberlain

Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis, Missouri
{jbeard,epsteinc,roger}@wustl.edu

Abstract. When do you trust a performance model? More specifically, when can a particular model be used for a specific application? Once a stochastic model is selected, its parameters must be determined. This involves instrumentation, data collection, and finally interpretation; which are very time consuming. Even when done correctly, the results hold for only the conditions under which the system was characterized. For modern, dynamic stream processing systems, this is far too slow if a model-based approach to performance tuning is to be considered. This work demonstrates the use of a Support Vector Machine (SVM) to determine if a stochastic queueing model is usable or not for a particular queueing station within a streaming application. When combined with methods for online service rate approximation, our SVM approach can select models while the application is executing (online). The method is tested on a variety of hardware and software platforms. The technique is shown to be highly effective for determining the applicability of $M/M/1$ and $M/D/1$ queueing models to stream processing applications.

1 Introduction

Stochastic modeling is essential to the optimization of performant stream processing systems. Successful application of a stochastic queueing model often requires knowledge of many factors that are unknowable without extensive application and hardware characterization. Extensive characterization, is however quite expensive (both in time and effort) when considering streaming applications of any appreciable size. Complicating matters further is that each streaming application could require that multiple models be selected in order to fully model its performance; each with its own assumptions and parameters that must be quantified before use. Even when modeling assumptions are verified offline, often they are broken by unpredictable behavior that can occur during execution.

R.D. Chamberlain—This work was supported by Exegy, Inc. Washington Univ. and R. Chamberlain receive income based on the license of technology by the university to Exegy, Inc.

J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 82–93, 2015.
DOI: 10.1007/978-3-662-48096-0_7

This paper proposes a machine learning method for classifying the reliability of stochastic queueing models for stream processing systems.

Stream processing is a compute paradigm that views an application as a set of compute kernels connected via communications links or "streams" (example shown in Fig. 1). Stream processing is increasingly used by multi-disciplinary fields with names such as computational-x and x-informatics (e.g., biology, astrophysics) where the focus is on safe and fast parallelism of a specific application. Many of these applications involve real-time or latency sensitive big data processing necessitating usage of many parallel kernels on several compute cores. Intrinsically stream processing comes with a high communications cost and infrastructure overhead. Optimizing or reducing the communication within a streaming application is often a non-trivial task, however it is central to the widespread adoption of stream processing techniques.

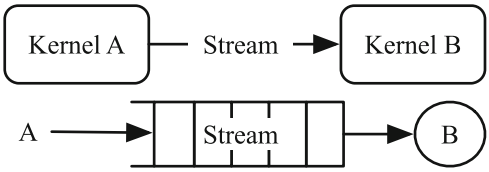

Fig. 1. The top image is an example of a simple streaming system with two compute kernels (labeled A & B). Each kernel could be assigned to any number of compute resources depending on the platform (e.g., processor core, graphics engine). The bottom image is the resulting queue with arrival process A (emanating from compute kernel A) and server B. For more complex systems this becomes a queueing network.

Streams allocated within a streaming application can be modeled as a stochastic queueing network for which there are well understood relationships between input arrival rates, computational service rates, queue occupancies, etc., in the steady state. Understanding the streaming application's queueing network is essential to its optimization. Streaming systems such as RaftLib [4], can spawn tens to hundreds of queues; each potentially with a unique environment and characteristics to model. Hand selection of performance models for these applications is clearly impractical. Offline modeling attempts are often thwarted by dynamic characteristics present within the system that were not included in the model. This paper outlines what is perhaps an easier route. Utilizing 76 features easily extracted from a system along with a streaming approximation of non-blocking service rate, we show that a Support Vector Machine (SVM) can identify where a model can and cannot be used. Results are shown that demonstrate that this model is generalizable in trained form to multiple operating systems and hardware types. In addition to testing the trained model on micro-benchmark data, two full streaming applications are utilized: matrix multiplication and Rabin-Karp string search.

2 Methodology

For most stream processing systems (including RaftLib) the queues between compute kernels are directly implied as a result of application construction. In order to use models for optimizing the queues within a stream processing application the service rate must be determined. Work from Beard and Chamberlain [3] enables the online determination of mean service rates for kernels. Working under the assumption that accurate determination of service distributions will be too expensive to be practical, we instead learn the applicability of two distinct queueing models based on features (shown in Fig. 2) that are knowable during execution with low overhead. Once trained, the parameters are supplied to a SVM which will label each parameter combination as being "usable" or "not" for the stochastic queuing model (in our case the $M/D/1$ and $M/M/1$ models) for which the SVM is trained.

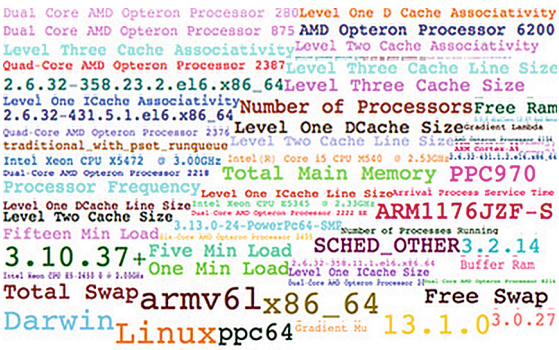

Fig. 2. Word cloud depicting features used for machine learning process with the font size representing the significance of the feature as determined by [8].

A SVM is a method to separate a multi-dimensional set of data into two classes by a separating hyperplane. It works by maximizing the margin between the hyperplane and the support vectors closest to the plane. The theory behind these are covered by relevant texts on the subject [10,15]. An SVM labels an observation with a learned class label based on the solution to Eq. (1) [5,9] (the dual form is given, $\mathbf{e}$ is a vector of ones of length l, Q is an $l \times l$ matrix defined by $Q_{i,j} \leftarrow y_i y_j K(x_i, x_j)$ K is a kernel function, specific symbolic names match those of [6]). A common parameter selected to optimize the performance of the SVM is the penalty parameter, C, discussed further in Sect. 2.2.

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \tfrac{1}{2}\boldsymbol{\alpha}^T \mathbf{Q}\boldsymbol{\alpha} - \mathbf{e}^T\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \leq \boldsymbol{\alpha}_i \leq C, i = 1, \ldots, l, \end{aligned} \tag{1}$$

$$K(x, y) = e^{-\gamma||x-y||^2}, y > 0. \tag{2}$$

A Radial Basis Function (RBF, [13], Eq. (2)) is used to map attributes to features. The parameter γ is optimized separately in order to maximize the performance of the SVM/Kernel combination. This work does not seek to add new techniques to SVM or machine learning theory, rather it is focused on expanding the application of SVMs to judging the reliability of stochastic performance models for streaming systems.

The stochastic mean queue occupancy models under consideration by our SVM are intended for systems at steady state. We apply these models to non-steady state applications with steady state behavior over small periods of time (i.e., they have multi-phase distributions). Applications whose behavior is too erratic to have any steady state behavior over any period of time are not good candidates for these models or our method. When testing the SVM we will test with applications that do exhibit steady state behavior and we discuss how this changes for applications whose distributions are more variable. Architectural features of a specific platform such as cache size are used as features for this work. As such we assume that we can find them either via literature search or directly by querying the hardware. Platforms where this information is unknown are avoided, however a surfeit of such platforms exists (see Table 1).

Implicit within most stochastic queueing models (save for the circumstance of a deterministic queue) is that $\rho < 1$ to obtain a finite queue. In this work, it is expected that the SVM should be able to find this relationship based upon the training process. It is shown in Sect. 3 that this is indeed the case. If deciding on a queueing model were as simple as selecting one class for $\rho \geq 1$ and another for $\rho < 1$ then the method described in this paper would be relatively inconsequential. However we also assume that the SVM is not explicitly told what the actual service process distributions are of the compute kernels modulating data arrival and service so this relationship is not quite so binary. It is also shown in the results that training the SVM with broader distributions slightly decreases the overall classification accuracy while increasing the generalizability of the trained SVM.

In order to train and test the SVM we use a set of micro-benchmark and full benchmark streaming applications (described below). All are authored using the RaftLib library in C++ and compiled using `g++` using the `-O1` optimization flag.

A micro-benchmark (with the topology shown in Fig. 1) has the advantage of having a known underlying service distribution for both compute kernels `A` and `B`. A synthetic workload for each compute kernel is composed of a simple busy-wait loop whose looping is dependent on a random number generator (either Exponential, Gaussian, Deterministic, or a mixture of multiple random distributions are produced). Simple workloads similar to those used within the real applications also constitute up to 5 % of the micro-benchmark loop workloads. Data exiting the servers are limited to one 8-byte item per firing.

Dense matrix multiply ($C = AB$) is a staple of many computational tasks. Our implementation divides the operation into a series of dot-product operations. Matrix rows and columns are streamed to n parallel dot-product kernels (see Fig. 3 for the topology). The result is streamed to a reducer kernel (at right)

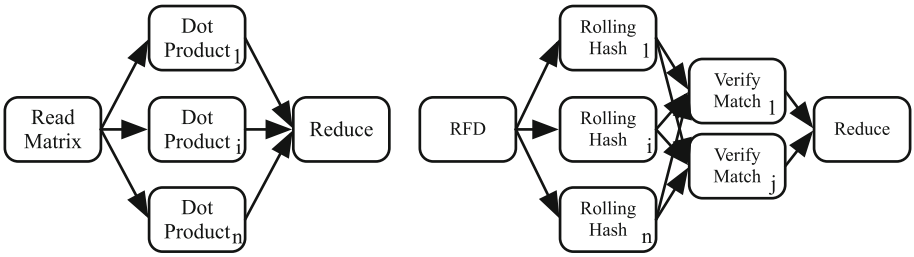

Fig. 3. Matrix multiply application (left image). The first kernel reads both matrices to be multiplied and streams the data to an arbitrary (n) number of dot product kernels. The final kernel reduces the input from the dot to a multiplied matrix. Rabin-Karp matching algorithm (right image). The first compute kernel (RFD, at left) reads the file to be searched, hashes the patterns to search and distributes the data to n "rolling-hash" kernel(s). Next are $j, j \leq n$ verification kernel(s) to guard against matches due to hash collision. The final kernel (at right) is a reducer which consolidates all the results.

which re-forms the output matrix, C. This application differs from the micro-benchmark in that it uses data read from disk and performs multiple operations on it. As with the micro-benchmark, it has the advantage of having a continuous output stream from both the matrix read and dot-product operations. The data set used is a single matrix (10000×10000) of single precision floating point numbers produced by a uniform random number generator.

The Rabin-Karp [12] algorithm is classically used to search a text for a set of patterns. The implementation divides the text amongst n parallel rolling-hash functions whose output is streamed to j parallel verification kernels. The final kernel simply reduces the output from the verification kernel(s), returning the byte position of each match (see Fig. 3). The data set for our tests is 2 GB of the phrase "foobar."

2.1 Data Collection and Hardware

Using benchmarking the applications enumerated above, we were able to collect a variety of features from each platform using a myriad of methods ranging from system calls through architecture-specific methods. Service rate is also used, which is approximated online via methods [3]. The number of features utilized prohibit their complete enumeration, however some of the more pertinent ones include: service rate, instruction set architecture, cache hierarchy sizes, operating system (OS) and version, scheduler, and main memory available (further enumerated in Fig. 2).

To collect mean queue occupancy, a separate monitor thread is used for each queue to sample the occupancy over the course of the application. For both real and synthetic applications, the service times of compute kernels are verified via monitoring the arrival and departure rate of data from each kernel with a non-blocking infinite queue (implemented by ignoring the read and write pointers).

All timing is performed using the POSIX.1–2001 `clock_gettime()` function with a real time system clock using the setup described in [2].

Relying on measurements from only one hardware type or operating system would undoubtedly bias any classification algorithm. To reduce the chance of bias for one particular platform, empirical data are collected from platforms with the processors and operating systems listed in Table 1. For all tests either the Linux or Apple OS X versions of the completely fair scheduler are used. To unbias the results further, task parallel sections of each application are replicated varying numbers of times (up to $2x$ the number of physical processor cores available). Application kernels are run "un-pinned." That is, the compute core which each executes on is assigned by the operating system and not by the user. Presumably more stable results could be obtained by "pinning" each compute kernel to dedicated cores, however this is not a realistic environment for many platforms. Micro-benchmark data are collected from all of the platforms in Table 1, Matrix multiply and Rabin-Karp Search data are collected from platforms 2, 8, 10, and 15.

In all, approximately 45,000 observations were made for the micro-benchmark application. This data is divided using a uniform random process into two sets with a 20/80 split. The set with 20 % of the data is used for training the SVM and the 80 % is set aside as a testing set. To give an idea of the range with which the SVM is trained, the micro-benchmark training set has the following specifications: approximately 8,200 observations, server utilization ranges from close to zero to greater than one and distributions vary widely (a randomized mix of Gaussian, Deterministic and the model's expected Exponential Distribution as well as some mixture distributions). For each of the other two applications, the SVM trained exclusively on the training micro-benchmark data (same training set as above) is used, with classification results reported in Sect. 3.

2.2 SVM and Training

Before the SVM can be trained as to which set of attributes to assign to a class, a label must be provided. Our two classes are "use" and "don't use" which are encoded as a binary one and zero respectively. The SVM is trained to identify one stochastic model at a time (i.e., either "use" or "don't use" for $M/M/1$ or $M/D/1$ but not both at the same time). In order to label the dataset as to which queueing model to use, a fixed difference is used. If the actual observed queue occupancy is within $n \leftarrow 5$ items, then the model is deemed acceptable otherwise false. A percentage based function for l shows a similar trend. After sampled mean queue occupancy is used for labeling purposes, it is removed from the data set presented to the SVM.

Feature selection is a very hot topic of research [11]. There are several methods that could be used including (but not limited to) Pearson correlation coefficients, Fisher information criterion score [8], Kolmogorov-Smirnov statistic [7]. Our selected feature set has a total of 35 linearly independent variables. The rest of the features exhibit weak non-linear dependence between variables. Extensive cross-validation followed by evaluating the Fisher information criterion score

Table 1. Summary of processor types and operating systems used for both the micro-benchmark and application data collection.

Platform	Processor type	OS	Kernel version
P_1	Intel Xeon CPU E5-2650	Linux	2.6.32
P_2	Quad-Core AMD Opteron 2376	Linux	2.6.32
P_3	Intel Xeon X5472	Darwin (OS X)	13.1.0
P_4	Dual-Core AMD Opteron 2218	Linux	2.6.32
P_5	ARM1176JZF-S	Linux	3.10.37
P_6	Dual-Core AMD Opteron 2222 SE	Linux	3.0.27
P_7	IBM Power PC 970	Linux	3.13.0
P_8	Six-Core AMD Opteron 2431	Linux	3.0.27
P_9	Intel Xeon E5345	Linux	2.6.32
P_{10}	Intel Xeon CPU E3-1225	Linux	3.13.9
P_{11}	Dual Core AMD Opteron 875	Linux	2.6.32
P_{12}	AMD Opteron 6136	Linux	2.6.32
P_{13}	ARM Cortex-A9	Linux	3.3.0
P_{14}	Intel Core i5 M540	Darwin (OS X)	13.1.0
P_{15}	AMD Opteron 6272	Linux	2.6.32
P_{16}	Six-Core AMD Opteron 2435	Linux	3.0.27
P_{17}	Dual Core AMD Opteron 280	Linux	2.6.32
P_{18}	Quad-Core AMD Opteron 2387	Linux	2.6.32
P_{19}	Dual-Core AMD Opteron 2220	Linux	2.6.32
P_{20}	Dual-Core AMD Opteron 8214	Linux	2.6.32

showed that the training data relied extensively on 67 of our candidate features. Most notably the variables that indicated the type of processor, operating system kernel version and cache size ranked highest followed closely by amount of main memory and total number of processes on the system. During the training phase we noted that despite the Fisher information criteria results, the additional 9 features provided a significant increase in correct classification, therefore we decided to include all 76 as opposed to the reduced set selected via statistical feature selection.

For all data sets (and all attributes contained in each set) the values are linearly scaled in the range $[-1000, 1000]$ (see [14]). This causes a slight loss of information, however it does prevent extreme values from biasing the training process and reduces the precision necessary for the representation. Once all the data are scaled, there are a few SVM specific parameters that must be optimized in order to maximize classification performance (γ and C). We use a branch and bound search for the best parameters for both the RBF Kernel ($\gamma \leftarrow 4$) and for the penalty parameter ($C \leftarrow 32768$). The branch and bound search is performed

by training and cross-validating the SVM using various values of γ and C for the training data set discussed above. The SVM framework utilized in this work is sourced from LIBSVM [6].

3 Results

To evaluate how effective a SVM is for model reliability classification we'll compare the class label predicted by the SVM compared to that of ground truth as determined by the labeling process. If the queueing model is usable and the predicted class is "use" then we have a true positive (TP). Consequently the rest of the error types true negative (TN), false positive (FP) and false negative (FN) follow this pattern.

The micro-benchmark data ($\text{Micro}_{\text{test}}$) consists of queues whose servers have widely varying distributions and server utilizations. Utilization ranges from close to zero through greater than one (i.e., the queues are always full). As enumerated in Fig. 4, the SVM correctly predicts (TP or TN) 88.1 % of the test instances for the $M/M/1$ model and 83.4 % for the $M/D/1$ model. Overall these results are quite good compared to manual selection [1]. Not only do these results improve upon manual mean queue occupancy predictions, they are actually faster since the user doesn't have to evaluate the service time and arrival process distributions, and they can be done online while the application is executing.

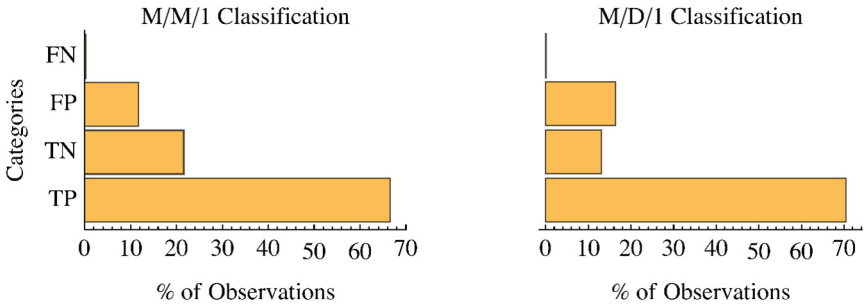

Fig. 4. Summary of overall classification rate by error category. In general the correct classification is quite high $TP + TN > 83\,\%$ in all cases.

Server utilization (ρ) is a classic and simple test to divine if a mean queue length model is suitable. At high ρ it is assumed that the $M/M/1$ and $M/D/1$ models can diverge widely from reality. It is therefore assumed that our SVM should be able to discern this intuition from its training without being given the logic via human intervention. Figure 5 shows a box and whisker plot for the error types separated by ρ. As expected the middle ρ ranges offer the most true positive results. Also expected is the correlation between high ρ and true negatives. Slightly unexpected was the relationship between ρ and false positives.

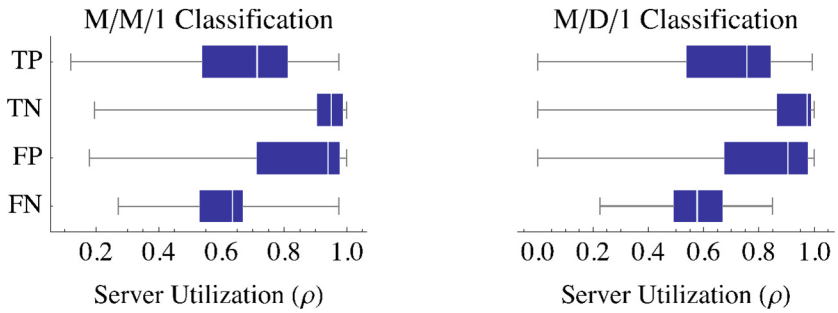

Fig. 5. Summary of true positive (TP), true negative (TN), false positive (FP), false negative (FN) classifications for the $M/M/1$ (left) and $M/D/1$ (right) queueing models for the microbenchmark's single queue by server utilization ρ demonstrating empirically that the SVM can recognize the instability of these models at high ρ.

Directly addressing the performance and confidence of the SVM is the probability of class assignment. Given the high numbers of TP and TN it would be useful to know how confident the SVM is in placing each of these feature sets into a category. Probability estimates are not directly provided by the SVM, however there are a variety of methods which can generate a probability of class assignment [16]. We use the median class assignment probability for each error category as it is a bit more robust to outliers than the mean. For the $M/M/1$ model we have the following median probabilities: TP = 99.5 %, TN = 99.9 %, FP = 62.4 % and FN = 99.8 %. The last number must be taken with caution given that there are only 79 observations in the FN category for $M/M/1$. For the $M/M/1$ FP it is good to see that these were low probability classifications on average, perhaps with more training and refinement these might be reduced. For the $M/D/1$ classification, probabilities mirror those of the $M/M/1$: TP=95.9 %, TN=95.8 %, FP=50.9 %, FN=85.3 %. The same qualification applies to the $M/D/1$ trained SVM for the FN probabilities as the FN category only contains 39 examples. Calculating probabilities is expensive relative to simply training the SVM and using it. It could however lead to a way to reduce the number of false positives. Placing a limit of $p = .65$ for positive classification reduces false positives by an additional 95 % for the micro-benchmark data. Post processing based on probability has the benefit of moving this method from slightly conservative to very conservative if high precision is required, albeit at a slight increase in computational cost.

The full application results are consistent with those of the micro-benchmark applications. Each application is run with a varying number of compute kernels with its queue occupancies sampled as described in Sect. 2.1. Table 2 breaks the application results into categories by model and application. Due to the processor configuration and high data rates with this application all examples are tested with a high server utilization. One trend that is not surprising is the lack of true positives within Table 2. The application as designed has very high

throughput, consequently all servers are highly utilized. In these cases (ρ close to 1), it is expected that neither of these models is usable. As is the case for the micro-benchmark data, the overall correct classification rates are high for both applications and models tested.

Table 2. % SVM classification rate for application data.

Application	Model	TP	TN	FP	FN	Correct classification
Matrix multiply	$M/M/1$	17.1 %	75.2 %	5.4 %	2.4 %	92.3 %
Matrix multiply	$M/D/1$	5.4 %	83.9 %	4.6 %	6.1 %	89.3 %
Rabin-Karp	$M/M/1$	0.0 %	86.0 %	14.0 %	0.0 %	86.0 %
Rabin-Karp	$M/D/1$	0.0 %	87.4 %	12.6 %	0.0 %	87.4 %

One potential pitfall of this method is the training process. What would happen if the model is trained with too few distributions and configurations. To test this a set of the training data from a single distribution (the Exponential) is labeled in order to train another SVM explicitly for the $M/M/1$ model. We then apply this to two differing test sets. The first is data drawn from an exponential distribution and the second is data drawn from many distributions (training data is excluded from all test sets). The resulting classification rates are shown in Table 3. Two trends are apparent: specifically training with a single distribution increases the accuracy when attempting to classify for only the distribution for which the model was trained, and conversely lack of training diversity increases the frequency of false positives when attempting use the SVM to classify models with distributional assumptions that it have not been trained for. Unlike the false positives seen in prior sets, these are high confidence predictions that post processing for classification probability will not significantly improve. One thing is clear, training with as many service rate distributions as possible and as many configurations tends to improve the generalizability of the SVM for our application.

Table 3. % for SVM predictions with SVM trained only with servers having an exponential distribution and tested as indicated.

Dist	# obs	Model	TP	TN	FP	FN	Correct classification
Exp.	3249	$M/M/1$	53.0 %	31.2 %	15.7 %	0.1 %	84.2 %
Many	6297	$M/M/1$	55.8 %	0.0 %	44.2 %	0.0 %	55.8 %

Our method is currently only applicable to queues with sufficient steady state behavior. To show what happens when a local steady state is not reached we will use the Rabin-Karp string searching kernel and change the packet to be extremely large proportionate to the size of the data set. This results in fewer

data packets sent from the source kernel to the "rolling-hash" kernel and no steady state. The resulting observed queue occupancies are much lower than what is calculated by either queueing model. Applying an $M/M/1$ mean queue occupancy model to this application will still result in a queue which is sized for the mean potential occupancy. Table 4 shows the result of attempting to evaluate the SVM against a queue that has not reached steady state. As a consequence of the streaming streaming service rate approximation method, it is knowable when the application has reached at least a local steady state and this condition can generally be avoided.

Table 4. % for SVM evaluated against a Rabin-Karp string search algorithm that has not reached steady state.

Model	*#obs*	TP	TN	FP	FN	Correct classification
$M/M/1$	120	21.6 %	41.5 %	20.7 %	16.2 %	63.1 %
$M/D/1$	120	11.1 %	44.4 %	44.4 %	0.0 %	55.5 %

4 Conclusions and Future Work

We have shown a proof of concept for using a SVM to classify a stochastic queuing model's reliability for a particular queue within a streaming application that is usable online. This enables fast online modeling and re-optimization of stream processing systems. Across multiple hardware types, operating systems, and applications it has been shown to produce fairly good reliability estimates for both the $M/M/1$ and $M/D/1$ stochastic queueing models.

This work chose to ignore the actual distribution of each compute kernel. What would happen if we knew the underlying distribution of the service and arrival process for each compute kernel in the system? Manually determining the distributions of each compute kernel and retraining the SVM with this knowledge for the $M/M/1$ model we arrive at a 96.6 % correct classification rate. This works just as well for the $M/D/1$ model where we observed 96.4 % of the queues being correctly classified as either "use" or "don't use." One obvious path for future work is faster and lower overhead process distribution estimation. Mathematically this can be done with the method of moments, what is left is an engineering challenge.

Empirical data could also be seen as a weakness of our approach since it is obviously finite in its coverage of the combinatorial plethora of possible configurations. We trained our SVM using as wide a variety of configurations as possible, however the permutations of possible application configurations are quite high. Other combinations of applications could provide slightly differing results. Our choices of attributes is limited to what the hardware and operating system could provide. Omniscient knowledge of the system would obviously be helpful, it is possible that future platforms will provide more robust identification and monitoring features which could improve the training and classification process.

In conclusion we have demonstrated an automated way to classify the reliability of stochastic queueing models for streaming systems. We have shown that it can be done, and that in many cases it works quite well for the applications and configurations tested. There are several avenues for future work to improve upon what is demonstrated here ranging from improved instrumentation to improved kernel functions.

References

1. Beard, J.C., Chamberlain, R.D.: Analysis of a simple approach to modeling performance for streaming data applications. In: Proceedings of IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, August 2013, pp. 345–349 (2013)
2. Beard, J.C., Chamberlain, R.D.: Use of a levy distribution for modeling best case execution time variation. In: Horváth, A., Wolter, K. (eds.) EPEW 2014. LNCS, vol. 8721, pp. 74–88. Springer, Heidelberg (2014)
3. Beard, J.C., Chamberlain, R.D.: Run time approximation of non-blocking service rates for streaming systems. arXiv preprint (2015). arXiv:1504.00591v2
4. Beard, J.C., Li, P., Chamberlain, R.D.: RaftLib: a C++ template library for high performance stream parallel processing. In: Proceedings of 6th International Workshop on Programming Models and Applications for Multicores and Manycores, February 2015, pp. 96–105 (2015)
5. Boser, B.E., Guyon, I.M., Vapnik, V.N.: A training algorithm for optimal margin classifiers. In: Proceedings of 5th Workshop on Computational Learning Theory, pp. 144–152 (1992)
6. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines. ACM Trans. Intell. Syst. Technol. **2**, 27:1–27:27 (2011)
7. Chapelle, O., Vapnik, V., Bousquet, O., Mukherjee, S.: Choosing multiple parameters for support vector machines. Mach. Learn. **46**(1–3), 131–159 (2002)
8. Chen, Y.W., Lin, C.J.: Combining SVMs with various feature selection strategies. In: Guyon, I., Nikravesh, M., Gunn, S., Zadeh, L.A. (eds.) Feature Extraction, pp. 315–324. Springer, Heidelberg (2006)
9. Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn. **20**(3), 273–297 (1995)
10. Cristianini, N., Shawe-Taylor, J.: An Introduction to Support Vector Machines and Other Kernel-based Learning Methods. Cambridge University Press, Cambridge, UK (2000)
11. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. J. Mach. Learn. Res. **3**, 1157–1182 (2003)
12. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987)
13. Schölkopf, B., Smola, A.J.: Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, Cambridge, MA (2002)
14. Tax, D.M., Duin, R.P.: Support vector data description. Mach. Learn. **54**(1), 45–66 (2004)
15. Vapnik, V.N., Vapnik, V.: Statistical Learning Theory, vol. 2. Wiley, New York (1998)
16. Wu, T.F., Lin, C.J., Weng, R.C.: Probability estimates for multi-class classification by pairwise coupling. J. Mach. Learn. Res. **5**, 975–1005 (2004)