Capstone Project

Machine Learning Engineer Nanodegree

Stefan Bergstein

September, 2016

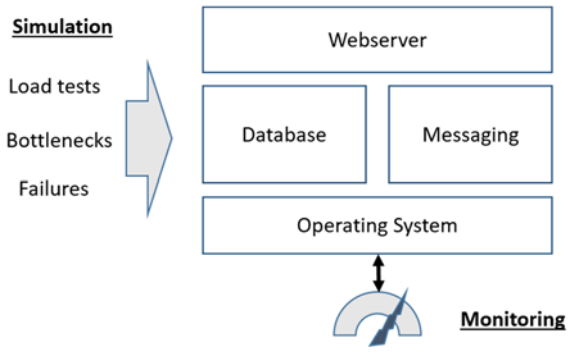
Definition

Project Overview

The continuous growth of web-based services and the massive consumption through mobile devices and various web clients create new challenges for application and server performance and availability management[1]. Private and business consumers expect a responsive, always-on experience.
Performance and resources monitoring of services, applications, middleware[2] and servers is standard practices so that the IT personnel can be proactively alerted in case of performance or availability issues. Proper instrumentations and domain-expert driven rules are required for many technology layers. This approach is costly and still leads to too many false-positive alerts.

This project aims to automate and simplify the prediction of application bottlenecks and system overloads by applying machine learning. It is a simplified approach because the solution does not need performance metrics from all application technology layers. Monitoring and analyzing basic systems metrics should be enough to predict application performance and resources problems.
The idea for this project originated from my daily work in this domain, talking to users and developers of application and server performance and availability management software. It is my own project and not from a site like Kaggle of Devpost.
A reasonable analogy is a simple health check at a medical doctor that possibly will indicate various diseases by just measuring human temperate, blood pressure and heart beat.

The picture on the right shows the basic approach for this machine learning scenario. Multiple simulations are performed on real application servers and the solution monitors only base operating systems metrics.
Supervised machine learning is used to create a model for predicting situations in production environments later on. The input data for the learning are the simulations and monitored systems metrics.



[1] http://searchenterprisedesktop.techtarget.com/definition/Application-monitoring-app-monitoring
[2] https://en.wikipedia.org/wiki/Middleware

Instead of monitoring hundreds of metrics for all technology layers and requiring deep domain expert knowledge for alerting rules for, the solution learns categories of performance and resources issues.

Generating an authentic dataset for the problem domain is part of this project. Multiple simulations are performed on real application servers and the logged systems metrics is the data source for this project. Details are described in the Data Acquisitions section below.

Problem Statement

Understanding and analyzing application and server performance problems is a complicated matter because many system metrics have interdependencies[3]. For example, high memory consumption of workloads causes swapping in/out, this causes IO load and followed by CPU load and wait-states, etc. CPU load can be caused by poor application code or just heavy user demand. Additionally, modern application architectures pull in more and more new technology components that expose more and more performance metrics (features in ML terminology).

What we need is a method for classifying application and server performance bottlenecks with a minimal amount of operating system metric and without deep domain expert knowledge of the application technology layers. The solution must provide better predictions than "hard-coded" expert rules. Not requiring expert knowledge and metrics for all application technology layers is going to save cost for application and server performance and availability management.

In this project, we are going to predict if an application server is affected by a CPU, memory or disk bottleneck.

The strategy to achieve the desired solution includes following high-level task:

1. Develop representative workloads and simulations.
2. Run simulations on real servers and capture the monitoring data.
3. Explore, analyze and reduce the amount of operating system metric (feature reduction in ML terms)
4. Apply supervised learning techniques to find and optimize a model that can predict application and server performance issues.
5. Compare the predictions with expert rules (benchmark) to proof the concepts and correctness of the solution.

Metrics to measure the performance of the project results

1. The predictions of the model must provide better results as the expert rules (see benchmark below). Any new approach in this domain must be better that the legacy approach to justify the adoption.
2. The accuracy should be above 90% (F1 score > 0.9). Anything below would create either too many false-positives or missed alerts that lead to poor acceptance of the solutions.

[3] Hunting the Performance Wumpus

3. The model should perform the predictions using a small number of features (operating system metrics). Less than 20, preferably only 10. Fewer features result in lower costs for the instrumentation and monitoring of the workload.

Analysis

Data Acquisitions

Generating a realistic dataset for the problem is part of this project. The script mkload.sh [4] creates overloads on Linux systems for simulating CPU, memory and disk bottlenecks. The system overloads are the base scenario for this projects for learning a model to predict bottlenecks. Once the overall concept is validated, further models for application and middleware performance issues can be created for building up a library of models for prediction many issues.
This script mkload.sh uses the open source monitoring tool collectl[5] to log the needed system performance metrics. The execution of the load tests are logged in a separate file. The python script prepare_dataset.py merges the data into a single time-series dataset data.csv that is discussed below.

Data Exploration

The input data for the project is a time-series dataset with 65 features (system metrics) that are collected during the load tests. The load tests run for 1 hours and generates 522 samples at 5 second intervals. All features are present and there aren't any missing values or outliers. The features don't contain any categorical variables that would require any preprocessing with an OneHotEncoder. The last two columns contain information categorical variables about the target labels and benchmark (vm, cpu, hdd, idle).
The raw data (data.csv) can be accessed and viewed on GitHub. A visualization of the raw feature data is available as well (metric_overview.png).

Why 65 features? The data acquisition uses the verbose mode of collectl to log CPU, Interrupt, Disks, Memory and Network data. The metrics are described on the collect web page[6]. At the time of data collection, it is not known which features are relevant for the desired predictions. Unwanted or unneeded feature are removed as follows.
The count, 25, 50, 75 percentile, Max, Mean, Min and Stdev for all feature are calculated in the prepare_dataset.py script. The output is documented in the table 'Basic statistics and information of the dataset' in the appendix of this document.

The first finding are features with constant values that cap be dropped. Additionally, we can drop '[CPU]L-Avg1', '[CPU]L-Avg5', '[CPU]L-Avg15' because these features show the load average over the last 1,5 and 15 minutes, but we want to focus on near real-time 5 second intervals.

The statistics also show that the features vary a lot in their scale and this leads to the conclusion to standardize the data. "Standardization of datasets is a common requirement for many machine learning estimators implemented in the scikit."[7]

[4] All sources and datasets are available on Github: https://github.com/stefan-bergstein/ml-capstone
[5] http://collectl.sourceforge.net/
[6] http://collectl.sourceforge.net/Data-verbose.html
[7] http://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler

Exploratory Visualization

It is not possible to spot any good correlation between the features or between the features and target labels by just looking at the input data (data.csv, metric_overview.png).

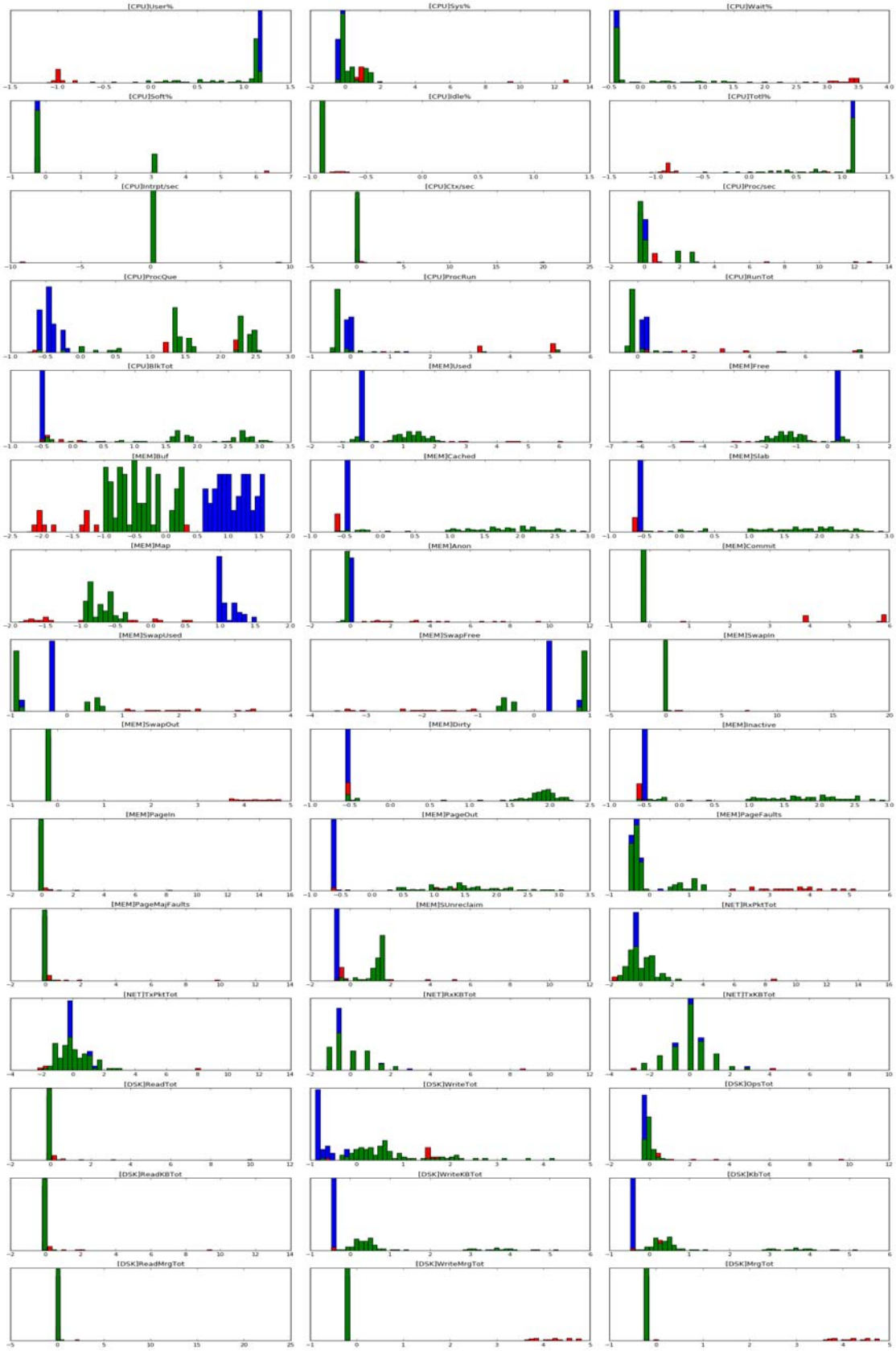
Using scatter plots of data that has more than two features is not very useful. If we want to look at the dataset, even using a pair-plot is tricky. The project's dataset has 45 features, which would result in 45 * 22 = 990 scatter plots. It is not possible to look at all these plots in detail, let alone try to understand them. We could go for an even simpler visualization, showing histograms of each of the features for the three main labels cpu, vm, hdd. [8]

The histograms on the next page are standardized with a scaler. Each plot overlays three histograms, one for all of the points of the cpu label (blue), one for the vm label (orange) and one for all the points for the hdd label (green). This gives us some idea of how each feature is distributed across the three labels, and allows us to venture a guess as to which features are better at distinguishing cpu, memory and disk overloads. For example, the features such as '[CPU] Ctx/sec' seems quite uninformative, while the feature 'MEM'Buf' seems quite informative, because the histograms are quite disjoint.[8]

Since one of the project goals is to use only few features, feature selection/dimensionality reduction will be applied to the data set to improve accuracy scores and to increase the performance on this high-dimensional dataset. See 'Data Preprocessing' below.

[8] Introduction to Machine Learning with Python by Sarah Guido, Andreas C. Mueller

Figure: Histograms of all features: cpu (blue), vm (orange), hdd (green)



Algorithms and Techniques

The problem is going to be solved with a supervised learning approach with the following steps:

1. Dimensionality reduction / feature reduction applying recursive feature elimination (RFE)
   RFE recursively removes features and builds models on the remaining features. RFE evaluates the model accuracy to identify which features contribute most for predicting the target labels. Due to the recursive approach, RFE is time and resource intensive, but the dataset is not very big and it is important for this project to automate the dimensionality reduction.
2. Splitting the data into training and test sets:
   "Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called overfitting. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a test set."[9]
3. Selecting best performing multi-class classifier:
   The target label is a multi-class attribute. Four different multi-class classifiers will use evaluated and scored: DecisionTree, SVC, LinearSVC, GaussianNB.
   Decision trees are simple to understand and interpret. If required, trees can be visualized. Another advantage is that decision trees don't require much computation.
   Decision-tree can create over-complex trees that do not generalize the data well. Additionally, decision tree learners create biased trees if some classes dominate.
   Support vector machines usually offer best classification performance on the training data. SVM renders more efficiency for correct classification of the future data. A good thing about SVM is that it does not make any strong assumptions on data.[10] On the other hand, SVMs have disadvantages[11]: If the number of features is much greater than the number of samples, the SVMs are likely to give poor performances. However, this is not the case for this project. SVMs do not directly provide probability estimates. These are calculated using an expensive five-fold cross-validation.
   LinearSVC are 'Similar to SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.'[12]
   As described on scikit-learn.org, Naive Bayes can be extremely fast compared to more sophisticated methods. NB classifier converges faster, requiring relatively little training data than other models. Naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require a small amount of training data to estimate the necessary parameters.
4. Perform cross validation and tune parameters with GridsearchCV.
   The classifiers have parameters that influence the learning of the model. These parameters are called hyper-parameters. For example, for the SVC it is important to tune the parameters C and

9 http://scikit-learn.org/stable/modules/cross_validation.html
10 https://www.dezyre.com/article/top-10-machine-learning-algorithms/202
11 http://scikit-learn.org/stable/modules/svm.html
12 http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

gamma. C controls the smoothness of the decision boundary and gamma controls the inverse radius of the samples around the data points that are considered to be relevant.[13]

5. Compare results against the benchmark.
   For details see next section.

Benchmark

Domain expert rules are going to be used to define a benchmark for comparing the results of the machine learning. Such expert rules can be leveraged from open source or commercial systems and application performance management tools. The primary input for the applied expert rules are from the alarm definition[14] of the HP Performance agent[15].

```
Disk bottleneck present if
        actual [DSK]KbTot >= 80 percentile of [DSK]KbTot OR
        actual [CPU]Wait% >= 90 percentile of [CPU]Wait%

Memory bottleneck present if
        actual [MEM]Free < 10% of MEM]Tot' OR
        actual [MEM]SwapOut > 95 percentile of [MEM]SwapOut

CPU bottleneck present if
        actual [CPU]Totl% >= 90 percentile of [CPU]Totl% OR
        actual [CPU]ProcRun' >= 90 percentile of [CPU]ProcRun
```

Assessing the performance of the expert rules compared to the target labels shows[16] following results:

	'cpu'	'vm'	'hdd'	'idle'
Induvial F1 scores	0.61	0.72	0.64	0.74
Precision based on total true positives, false negatives and false positives	0.68			

[13] Advanced Machine Learning with scikit-learn Andreas C. Mueller
[14] https://gist.github.com/ramkumardevanathan
[15] HP Performance Agent and HP Performance Manager software
[16] https://github.com/stefan-bergstein/ml-capstone/blob/master/prepare_dataset.py

Methodology

Data Preprocessing

After the data acquisition, feature reduction and scaling were applied. There aren't any missing values or outliers in the data set that requires special handling.

Feature reduction is performed in three steps:

1. Removed columns that contain only constant data. The table 'Basic statistics and information of the dataset' in the appendix showed the constant data. Therefore it is possible to drop the following features instantly:
   '[CPU]Nice%','[CPU]Irq%','[CPU]Steal%','[MEM]Tot', '[MEM]Shared', '[MEM]Locked', '[MEM]SwapTot', '[MEM]Clean', '[MEM]Laundry', '[MEM]HugeTotal', '[MEM]HugeFree', '[MEM]HugeRsvd', '[NET]RxCmpTot', '[NET]RxMltTot', '[NET]TxCmpTot', '[NET]RxErrsTot', '[NET]TxErrsTot'.

2. Additionally, dropped '[CPU]L-Avg1', '[CPU]L-Avg5', '[CPU]L-Avg15' because these features show the load average over the last 1,5 and 15 minutes and the focus on near real-time 5 second intervals.

3. Dimensionality is applied with recursive feature elimination (RFE). In the script reduce_learn_tune.py, after scaling all features with the StandardScaler, RFE is used to select 10 features. The algorithm picks [CPU]User%, [CPU]Idle%, [CPU]Totl%, [CPU]ProcQue, [CPU]RunTot, [CPU]BlkTot, [MEM]Commit, [MEM]Dirty, [MEM]Inactive, [DSK]WriteMrgTot. All other features are removed from the dataset.

Implementation

The implementation in reduce_learn_tune.py covers the following flow:

Prepare data pass 1	• Remove constant or unneeded data • Shuffle and split the dataset

1st basic training on the original data	• Get an understanding of the performance of the selected classifiers with the original (not standardized, unreduced) data. Train and predict using,
	  o DecisionTreeClassifier()
	  o SVC()
	  o LinearSVC()
	  o GaussianNB()
	• Assess the F1 score for the training and test set
	• Perform cross validation

Prepare data pass 2	• Scale data with the StandardScaler • Shuffle and split the scaled dataset
2nd basic training on scaled data	• Loop thru models like with the 1st basic training, but with the scaled data • Assess the F1 score for the training and test set

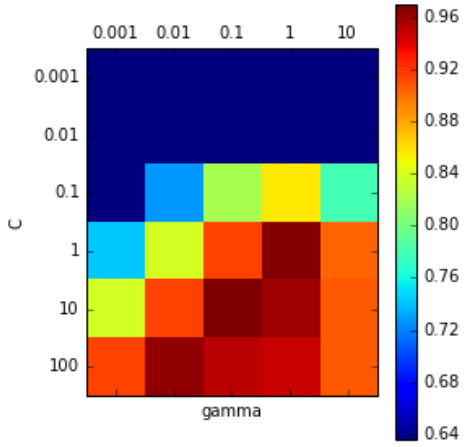
	• Perform cross validation
Feature selection	• Apply RFE to select the 10 most relevant features • Drop all other features from the data set.
Training on scaled and reduced features	• Loop thru models like with the 1st and 2nd basic training, but with the scaled, reduced data • Assess the F1 score for the training and test set • Perform cross validation • Since LinearSVC and SVC provide best performance, move on with these two.
Tune LinearSVC	• Perform GridSearchCV on LinearSVC with Cs = [0.001, 0.01, 0.1, 1, 10, 100]
Tune SVC	• Perform GridSearchCV on SVC with Cs = [0.001, 0.01, 0.1, 1, 10, 100], gammas = [0.001, 0.01, 0.1, 1, 10]
GridSearchCV only on training data	• Move on with SCV. • To avoid overfitting, do GridSearchCV only on training data • GridSearchCV(SVC(), param_grid, cv=5)
Final cross-validation	• Evaluate a final scores by cross-validation

Refinement

The training, refinement and tuning created following scores. It led into applying Support Vector Classification with C =10 and gamma = 0.1. The very high training scores (1.0) should be interpreted as overfitting.

	DecisionTree	**SVC**	**LinearSVC**	**GaussianNB**
1) Original features (not standardized, unreduced)				
Training	1.00	1.00	0.74	0.63
Test	0.95	0.64	0.75	0.63
Cross-validation	0.93	0.64	0.67	0.60
2) Scaled features				
Training	1.00	0.92	0.93	0.84
Test	0.95	0.91	0.93	0.84
Cross-validation	0.93	0.86	0.89	0.81
3) Scaled and reduced features				
Training	1.00	0.92	0.92	0.85
Test	0.96	0.90	0.90	0.85
Cross-validation	0.90	0.92	0.92	0.84
GridSearchCV best scores				
		0.969	0.925	
4) Scaled and reduced features with tuned hyper-parameters				
Training		0.987		
Test		0.947		
Cross-validation		0.933		

We tune the parameters C and gamma. C should be tuned for this model because C controls the smoothness of the decision boundary[17]. Gamma should be tuned as well because, gamma defines how far the influence of a single training data reaches[18]. A low gamma value also considers the data points that are far away from the decision boundary. A high gamma value takes only data point close into consideration. It is good practice to use ranges in powers of 10 for gamma and C[19]. I used wide ranges: Cs = [0.001, 0.01, 0.1, 1, 10, 100], gammas = [0.001, 0.01, 0.1, 1, 10]. Additionally, validating visually if good ranges are used is important:



The chart the on left visualizes the scores for hyper-parameters C and gamma.
It shows a best score with C =10 and gamma = 0.1.
The value range for the parameters is correct since gamma and C stabilize around C =10 and gamma = 0.1.

Results

Model Evaluation and Validation

The implementation and resulting data above illustrates that the chosen approach worked very well. The amount of features is reduced to 10 with recursive feature elimination method. The classifiers showed a better performance with less overfitting. Evaluating several classifiers has been proven as a good approach to gain confidence for selecting the right classifier.

As expected, tuning the hyper-parameters created even better results and the chart with the hyper-parameters and score creates confidence that the selected hyper-parameters are in a good range.

The over-time graph in the 'Free-Form Visualization section' below creates further confidence that the model is correct and precise. Only a very few labels aren't predicted correctly (hit=false).

The last task in reduce_learn_tune.py validate the robustness of this model by manipulating the input data to see how the model's solution is affected. This sensitivity analysis shows that the model is very robust even the input data is manipulated randomly up to 20%:

[17] SVM C Parameter
[18] SVM Gamma Parameter
[19] Advanced Machine Learning with scikit-learn Andreas C. Mueller

Table: Sensitivity analysis - Scores in relation to percent of manipulated input data

0.0 %	5.0 %	10.0 %	15.0 %	20.0 %	30.0 %	40.0
0.977	0.977	0.977	0.977	0.977	0.908	0.908

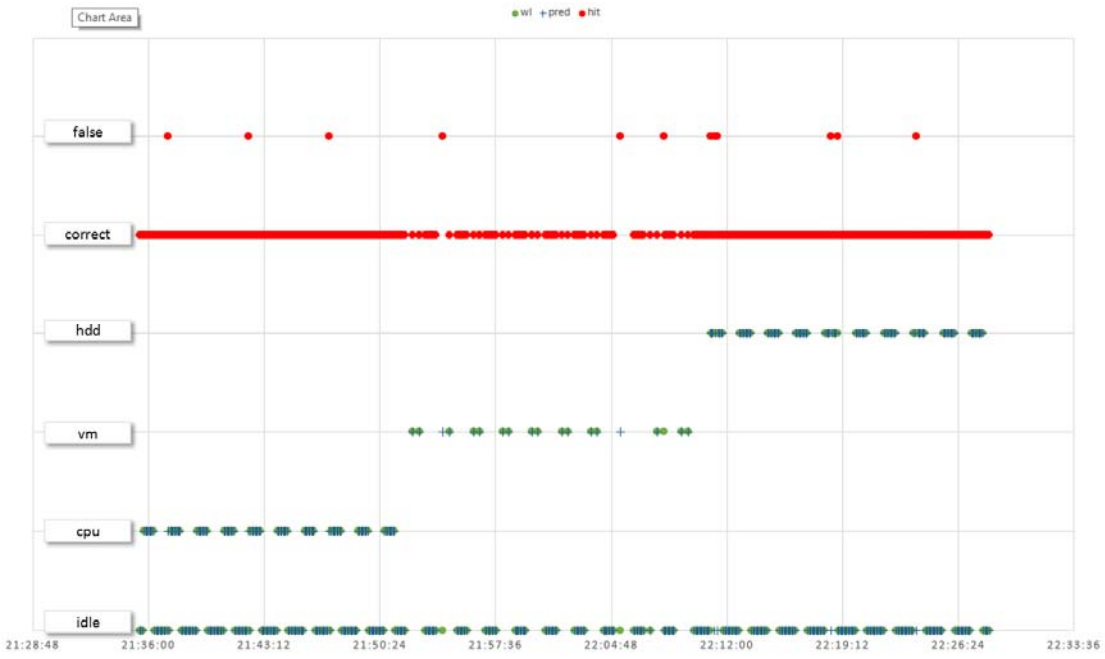
Justification

The model of the final solutions creates far better results that the rules of the benchmark. The precision of the benchmark is 0.68 and the final model has a score above 0.9. It should be possible to create better scopes for the hard coded rules by doing further analyses and applying domain knowledge. However, the goal of this project was to a method for classifying application and server performance bottlenecks with a minimal amount of operating system metric and without deep domain expert knowledge of the application technology layers. Not requiring expert knowledge and metrics for all application technology layers is going to save cost for application and server performance and availability management.

Conclusion

Free-Form Visualization

The over-time graph below illustrates that the model is correct and precise. It shows the labels of the workload (wl, green), the predictions (pred, +) and the correctness (hit, red) for the whole dataset (500 data points). The workload and predicted labels show the same values and therefore the hits are most of the time correct. This is aligned with the computed model score of more than 0.9.

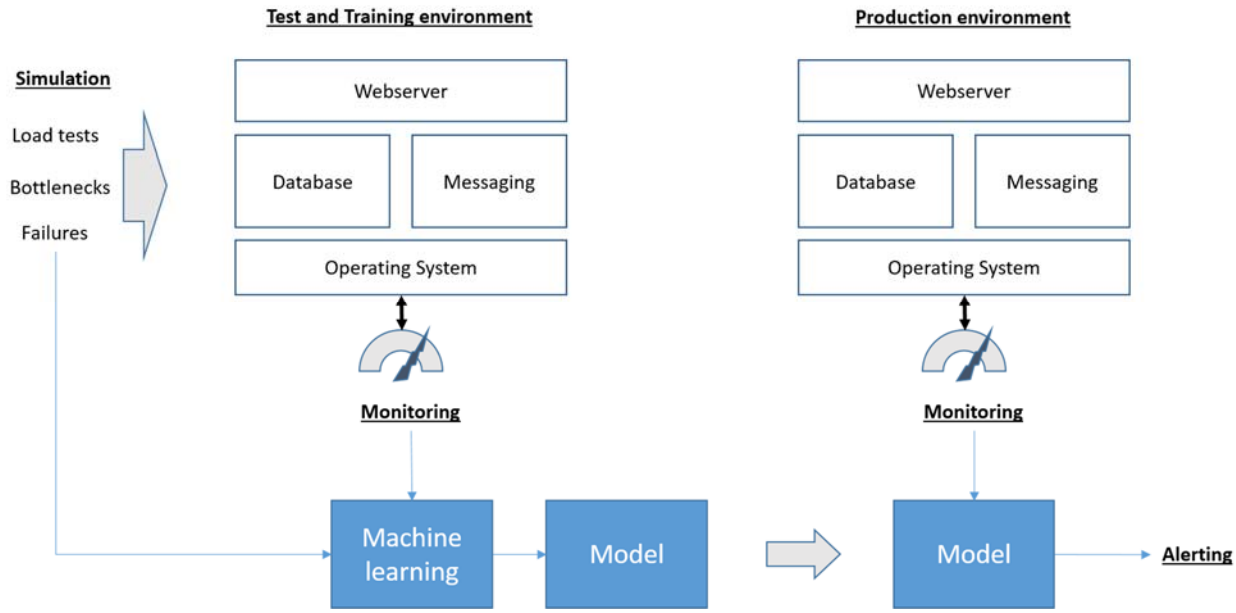


Reflection

This project automates and simplifies the prediction of application bottlenecks and system overloads by applying machine learning. It is a simplified approach because the solution does not need performance metrics from all application technology layers. Monitoring and analyzing basic systems metrics is simple and sufficient even to predict application performance and resources problems.

The picture below shows the end to end approach for this solution. Multiple simulations are performed on real application servers and the solution monitors only base operating systems metrics. Supervised machine learning is used to create a model for predicting situations in production environments later on. The input data for the learning are the simulations and monitored systems metrics.

Instead of monitoring hundreds of metrics for all technology layers and requiring deep domain expert knowledge for alerting rules for, the solution learns categories of performance and resources issues.



The whole process from simulation, data acquisition, feature reduction, selecting and tuning the right classifier worked out very well. The creation of a realistic simulation environment was not too difficult, but took some time because a full simulation cycle take one hour and I had to spend several cycles to get it right.
Most interesting aspects of the project was that the learned methodologies from the previous ML projects could be practically applied without any major hiccups. The outcome exceeds my expectation.

Improvement

As next steps, a few improvements can be to make the solution better:

1. The available input data (systems metrics) needs to be analyzed to specific a superset that is adequate for prediction many different kinds of application and server performance and availability management. The current project uses ‘only’ the first 65 common metrics as first features sets. Tools such a collectl offer more metrics that are eventually more meaningful and lead into better results.
2. The input data needs to be normalized so that the metrics are independent of the computer system size and capacity. For example, metrics such a CPU Utilization are in % and don’t need to be further normalized, but CTX/sec or SwapUsed require normalization that should not be done with a standard scaler.
3. The current solution does the prediction on every sample. The time-series input data contains 5 second samples. It would be interesting to perform machine learning on slices of data. For example, 1 minute, 12 samples on 10 features and try to leverage image recognition techniques such as PCA to identified similar slices.

Appendix

Table: Basic statistics and information of the dataset

	25%	50%	75%	Count	Max	Mean	Min	Stdev
[CPU]User%	7	34	96	522	98	47.7	1.0	42.1
[CPU]Nice%	0	0	0	522	0	0.0	0.0	0.0
[CPU]Sys%	2	3	4	522	84	4.4	1.0	6.2
[CPU]Wait%	0	1	2	522	90	9.3	0.0	22.0
[CPU]Irq%	0	0	0	522	0	0.0	0.0	0.0
[CPU]Soft%	0	0	0	522	2	0.1	0.0	0.3
[CPU]Steal%	0	0	0	522	0	0.0	0.0	0.0
[CPU]Idle%	0	1	89	522	92	38.4	0.0	42.2
[CPU]Totl%	10	40	100	522	100	52.3	7.0	42.3
[CPU]Intrpt/sec	18	18	18	522	19	18.0	17.0	0.1
[CPU]Ctx/sec	1226.25	1291	1436	522	85382	1673.6	1115.0	4173.0
[CPU]Proc/sec	0	0	1	522	31	0.8	0.0	2.3
[CPU]ProcQue	356	358	367	522	390	362.9	355.0	10.5
[CPU]ProcRun	1	1	2	522	31	2.7	0.0	5.4
[CPU]L-Avg1	1.55	12.85	17.925	522	34.69	11.1	0.5	8.4
[CPU]L-Avg5	1.96	14.965	16.8975	522	21.88	10.9	1.4	7.4
[CPU]L-Avg15	2.06	11.27	14.99	522	15.59	9.1	1.7	6.1
[CPU]RunTot	0	1	2	522	31	1.6	0.0	3.7
[CPU]BlkTot	0	0	4	522	37	5.2	0.0	9.9
[MEM]Tot	3523128	3523128	3523128	522	3523128	3523128.0	3523128.0	0.0
[MEM]Used	629465	747104	1087581	522	3334348	875203.7	493700.0	374754.0
[MEM]Free	2435547	2776024	2893663	522	3029428	2647924.3	188780.0	374754.0
[MEM]Shared	0	0	0	522	0	0.0	0.0	0.0
[MEM]Buf	5542	7090	9968	522	11596	7562.4	1872.0	2551.9
[MEM]Cached	58000	85240	97807	522	1086908	226433.5	37700.0	293680.3
[MEM]Slab	54276	54340	58971	522	81860	58752.1	53432.0	7948.0
[MEM]Map	33572	36046	47172	522	52044	39718.4	25668.0	7455.0
[MEM]Anon	330416	357270	374683	522	2967488	386818.0	207680.0	254470.5
[MEM]Commit	2296951	2297440	2299392	522	55159284	4045587.4	2271228.0	8643020.4
[MEM]Locked	0	0	0	522	0	0.0	0.0	0.0
[MEM]SwapTot	4194300	4194300	4194300	522	4194300	4194300.0	4194300.0	0.0
[MEM]SwapUsed	1031580	1047224	1069591	522	1151100	1054809.5	1029312.0	27098.2
[MEM]SwapFree	3124709	3147076	3162720	522	3164988	3139490.5	3043200.0	27098.2
[MEM]SwapIn	0	0	2	522	4024	36.8	0.0	243.8
[MEM]SwapOut	0	0	0	522	5920	284.2	0.0	1178.2
[MEM]Dirty	1797	2220	4117	522	484396	94385.1	4.0	170570.9
[MEM]Clean	0	0	0	522	0	0.0	0.0	0.0
[MEM]Laundry	0	0	0	522	0	0.0	0.0	0.0
[MEM]Inactive	21596	39004	62317	522	1048300	186309.0	7672.0	294364.1
[MEM]PageIn	1	2	18	522	16658	212.3	0.0	1169.1
[MEM]PageOut	27	145.5	18655.25	522	42340	7434.5	0.0	11355.0
[MEM]PageFaults	155	231	347.75	522	7224	621.2	9.0	1199.8
[MEM]PageMajFaults	0	0	2	522	1944	27.3	0.0	158.1
[MEM]HugeTotal	0	0	0	522	0	0.0	0.0	0.0
[MEM]HugeFree	0	0	0	522	0	0.0	0.0	0.0
[MEM]HugeRsvd	0	0	0	522	0	0.0	0.0	0.0
[MEM]SUnreclaim	26316	26348	27078	522	33256	26665.5	26184.0	626.3
[NET]RxPktTot	7	8	10	522	82	8.8	0.0	4.8

[NET]TxPktTot	11	13	16	522	95	13.5	0.0	6.0
[NET]RxKBTot	1	1	2	522	17	1.7	0.0	1.4
[NET]TxKBTot	3	4	5	522	18	4.1	0.0	1.4
[NET]RxCmpTot	0	0	0	522	0	0.0	0.0	0.0
[NET]RxMltTot	0	0	0	522	0	0.0	0.0	0.0
[NET]TxCmpTot	0	0	0	522	0	0.0	0.0	0.0
[NET]RxErrsTot	0	0	0	522	0	0.0	0.0	0.0
[NET]TxErrsTot	0	0	0	522	0	0.0	0.0	0.0
[DSK]ReadTot	0	1	3	522	2183	34.2	0.0	192.0
[DSK]WriteTot	2	24	63	522	219	37.7	0.0	42.9
[DSK]OpsTot	3	33.5	73.75	522	2187	72.1	1.0	196.0
[DSK]ReadKBTot	1	2	17.75	522	17494	233.0	0.0	1304.1
[DSK]WriteKBTot	29	150.5	19586.25	522	167164	13596.1	0.0	29591.5
[DSK]KbTot	38	240.5	19828	522	167183	13829.0	12.0	29566.4
[DSK]ReadMrgTot	0	0	0	522	2136	11.2	0.0	104.9
[DSK]WriteMrgTot	4	6	9	522	5856	286.5	0.0	1159.5
[DSK]MrgTot	4	6	10	522	5861	297.6	0.0	1164.7