Automated Detection of Individual Anomalies and General Behavioral Changes in Time Series Metrics

Tifany Yung, Renato Martins
Groupon, Inc.
tiyung@groupon.com, rmartins@groupon.com

Abstract

Metrics provide insight into system health, performance, and stability and must be monitored throughout the software release process to catch issues before they reach production. Therefore, a fast, effective method of identifying deviant metrics is necessary to more quickly identify and stop regressions. However, manual analysis of hundreds of metrics is time-consuming, subjective, and error-prone. Some automation can be achieved by alerts that trigger when metrics cross a hardcoded threshold, but smaller changes with a noticeable but less severe impact would not be picked up.

In this paper, we introduce a technique we developed to automate building a model of expected behavior for each metric based on its previous data and using it to compute variable thresholds for the current metric data. We then describe our experience using it in our own deployment process.

We found that all metrics flagged as deviant were confirmed by human analysts to be so, including subtle cases where the deviation was not immediately obvious. Our technique also streamlined our deployment process by reducing the number of metrics our analysts had to review by 92-98%, and manual review of the remaining 2-8% was only needed to use the flagged metrics to diagnose the problem.

Biography

Tifany Yung is a biomedical engineer and computer scientist with a minor in Applied Math and Statistics. She has a Master's in Computer Science from the University of Illinois at Urbana-Champaign, where she did research on software quality and test automation. She has now been working at Groupon for 2.5 years developing monitoring tools to improve early detection of regressions and new issues.

Renato Martins is a EE who has never developed hardware for a living. Renato also has a Master’s Degree in Computer Science and an MBA. Renato started his career developing firmware for phones and accessories at Motorola. After that Renato spent 11 years at Microsoft on many quality-oriented roles where Renato worked on projects involving automotive systems, embedded OSs, Windows Phone, and then made a transition from the world of small computers to custom online tools used by Microsoft Support. Renato fulfilled his entrepreneurial aspirations by becoming the co-founder and CTO of Stringr.com where he built the platform from scratch. Renato now leads an engineering team at Groupon.com.

1 Introduction

Metrics provide insight into system health, performance, and stability and must be monitored throughout the software release process as new software is deployed first to the testing, staging, and then production environments. They are needed to catch issues before they reach the latter and affect end users, as well as to detect issues that do slip into production. However, manual analysis of hundreds of these metrics, usually by visual inspection of their graphs, is time-consuming, subjective, and error-prone.

Some automation may be achieved by setting alerts that automatically trigger when a particular metric crosses some critical threshold, but it may take time for the metric to reach them, or the metric may just never reach a level that triggers it, thereby allowing moderately severe issues to slip through. Therefore, it is still necessary to inspect metrics for anomalous behavior that occurs below the critical level.

One way this can be done, whether through manual inspection or an automated detection algorithm, is by comparing metric data from after a deployment, which we call test data against metric data from before the deployment, which is selected to exhibit the expected behavior of the metric and which we call model data. Another method, used by Elasticsearch's X-Pack extension [1], is to look for individual anomalous points within a dataset.

Many algorithms have been proposed to do one or the other. Some can find individual anomalous points, such as one algorithm that compares a data point to those around it and flags the point if it differs from its neighboring points by more than some threshold [2]. Others instead detect general changes in behavior between the model and test data, including algorithms that encode time series data as strings and generate detector strings that do not match the string for the model data [3], or use the edit distance between model and test data strings [5].

However, while solutions that detect individual anomalous points can help identify exactly when a metric deviated, they do not indicate whether the deviation was temporary noise or a change in the metric’s behavior. On the other hand, algorithms that detect overall changes based on how different the encoded datasets are do not identify individual anomalies that could help diagnose whether a deviation is ongoing or was resolved.

In this paper, we describe an algorithm for automated detection of both individual anomalies and overall changes in metric behavior, which allows us to determine not only whether a metric's behavior has changed after a deployment, but also which data points in the time series contributed to the change. A model of expected behavior is built and used to predict the test data values, and anomalous points are identified based on how far off the predictions are from the actual observed values. Then, the number of anomalous points and how recent they are used to determine whether the test data exhibits different behavior from the model data. Human involvement is necessary only when alerts have been generated for the flagged metrics, and only to review them and make a call on the deployment based on them.

2 Methodology

The detection algorithm we describe below is entirely automated except for the one-time, initial configuration setup and runs as a cron job that sends notification emails listing the flagged metrics when alerts are triggered.

For the purposes of this paper, we will focus on the detection algorithm and minimize implementation details, such as configuration file formatting and the underlying statistical packages used, to what is necessary to understand the algorithm.

2.1 Initial setup

The initial setup step involves the creation of configuration files that describe the metrics to be monitored and is the only manual step in the otherwise automated detection algorithm.

Configuration files contain a list of metrics and their metadata. Each job has its own configuration file, and we define a *job* as a set of related metrics to be monitored, e.g. metrics for the staging environment.

Metadata for each metric will be explained in the algorithm below as they turn up but include the source to retrieve the metric data from, time ranges to use for the model and test data, season size to use when modeling a metric that exhibits seasonal behavior, etc. These are fed as parameters into the algorithm and used to customize modeling and detection for each metric.

2.2 Modeling the expected behavior

For each metric to be monitored, the following steps are performed when building a model of the metric's expected behavior to compare against its behavior in the test data:

- Get the model data based on the data source and model data time range metadata given in the configuration file. The data source is queried for the metric values within the time range and returned as a *time series*, where the metric values are matched with their corresponding timestamps.

  For our own use case, we selected the metric's data from midnight to noon of the second most recent day of data available. This was done since our deployments almost never occur during that time range, so the metric should be most stable and best demonstrate the ideal behavior for that metric.

- Similarly, query for the test data that needs to be checked for anomalies and behavioral changes, also using the data source, as well as the test data time range.

  For our own use case, we select the most recent hour of data available for a metric as its test data. This time range was selected to give our deployed application just enough time to warm up and receive enough traffic to stabilize the metric, and to allow us to evaluate the deployment without losing several hours or an entire day while accumulating data.

- Remove outliers from the model data so that they do not affect the model. Here, we use the interquartile range (IQR) method to find outliers rather than the mean and standard deviation method, since the latter depends on the data being normal. However, since this is often not the case with time series data, as can be seen by visual inspection of the graphs of ours and other time series, we use the more generic IQR method.

Using this method, we define any data point below *p25 - n*IQR* or above *p75 + n*IQR* as an outlier, where *p25* and *p75* are the 25th and 75th percentiles of the model data, respectively.

For normally-distributed data, the commonly used value in statistics for n is 1.5, but since time series data often does not appear normal, we make n a parameter in the configuration file so that it can be customized depending on how sensitive we want the alert to be. Larger values of n generally result in less sensitive alerts that tolerate more deviation before flagging the metric.

- Build the time series model for model data.

  We use the autoregressive integrated moving average (ARIMA) class of models to model the data. Since ARIMA models are well-known and common models in time series analysis and statistics in general, we will not explain them in detail in this paper.

  The algorithm may select either an ARIMA model or a seasonal variant of it as the model to fit the model data to. Which one the algorithm chooses to use is determined as follows:
    - If a metric is marked as seasonal in the configuration file, the seasonal variant is selected. To use this model, the seasonal difference of the data is computed, and the resulting time series is used to build an ARIMA model.
    - Otherwise, the algorithm checks that the conditions for using ARIMA, such as stationarity of the data, are satisfied. If not, the data is processed so that the resulting time series satisfies the condition; else, the original time series is used. The model is then built using the resulting time series.

  Note that, though the model orders are usually chosen by inspecting a graph of the data, we use an information criterion to estimate them so that we could automate the process of building the model. Maximum likelihood estimation is then used to estimate the best-fitting model coefficients. Since both are done via the Python statsmodels package, we do not go into more detail here on how model for the model data is developed.

- Compute the prediction errors between the actual observed model data points and the values predicted by the model. These are the *model prediction errors*, and we call the distribution of these errors the *model error distribution*.

  Models are expected to exhibit some prediction error, as the model may otherwise be overfitted and unsuitable for forecasting on new data. We use the model error distribution to determine approximately how much error is expected for a given model.

  This distribution will then be used to determine whether the errors for the model's prediction for the test data points are within the expected errors for the model, or if the test point value deviates significantly from the expected value and is likely an anomaly.

2.3 Finding anomalous data points in the test data

Once the model for a given metric has been generated, we compare its predictions with the actual test data. If the behavior of the metric hasn't changed between the model and test data, the test data predictions should have approximately the same distribution of errors as the model data predictions. Any test point that is too far from the prediction would be flagged as an anomaly.

The following is done to determine whether the error between the actual value of a test point and its prediction is too large:

- Forecast the predicted values for the test data point values using the ARIMA (or seasonal ARIMA) model generated for the model data.

- Similar to what was done to get the model prediction errors, compute the prediction errors between the actual test data points and the predicted values from the model.

- Check whether each test point's prediction error is an outlier with respect to the model error distribution, using the same IQR method and parameter *n* to identify outliers as was used to remove outliers from the model data. Any test point whose prediction error is an outlier with respect to the model errors is flagged as an anomaly.

  We do this because the model error distribution gives the approximate amount of error to expect from predicting with the model, since the model points are known to exhibit expected behavior. So if a test point's prediction error is too large, whether in the positive or negative direction, that point’s actual value exhibits a greater deviation from the forecasted value than expected, and are thus anomalous.

2.4 Identifying overall changes in metric behavior

After identifying individual anomalies, we use the number of anomalies and their times of occurrence to determine whether a given metric is currently exhibiting different behavior from the model data.

Essentially, if most of the points in the test data are anomalous and the anomalies are relatively recent in the test data, then the metric behavior is flagged as anomalous. We consider how recent an anomaly is so that a metric isn't flagged for possibly unrelated behavioral changes if it was behaving differently earlier in the test data but has returned to expected behavior. Such anomalies may be caused by factors like unusually high spikes in traffic or load to the deployed application that are independent of the deployment itself.

- If the majority of the test data points are flagged as anomalies and the majority of those anomalies are in the more recent half of the test data, then that metric's behavior is flagged as changed.

  In our own default use case, we defined *majority* as at least half of the test data points, so a metric is flagged only if it has been consistently deviating. However, we also did not want the alert to trigger only when all of the test data points deviated since it is possible that there are still occasional test points that occur within expected levels. Requiring that multiple, but not all, test points deviate is done so that a temporary anomaly will not generate noisy alerts, but also so that the metric need only be deviating consistently and repeatedly, without errors having to be permanently elevated.

  For example, consider a latency metric that measures how long an application took to handle a request. A spike in the number of requests may cause the metric to deviate temporarily, even if it was unrelated to the deployment being tested and was resolved by itself. Or an actual performance issue in the deployment could cause the latency to increase most of the time but sometimes fall within the error range due to multiple requests that were not affected by the issue.

In the case where a metric's trigger should be more or less sensitive, our algorithm does allow a value to be passed in in each metric's configuration. With this, a lower threshold for majority can be used to create more sensitive alerts that are triggered by fewer anomalies and vice versa.

- Send an email alert detailing which metrics were flagged for changes in behavior and which points in each of the metric's test data were anomalies (i.e., contributed to that metric's behavior being flagged as changed).

2.5 Making a call on a release

Once all metrics in a job have been analyzed, a single alert email is sent to the interested developers for all metrics flagged in that job.

Since the algorithm only automates the detection of anomalies and behavioral changes, human involvement is still required after it is run to make a call on whether to proceed with deploying a release to the next environment in the release process, or whether the build has an issue and needs to be blocked and reworked.

This involves reviewing the alert to determine what metrics were flagged and whether the anomalies and changes in those metrics were expected and/or acceptable since depending on the metric, some increases or decreases are considered improvements, or a new feature could cause a metric's behavior to change.

For example, if a computationally-heavy new feature was added, it may be expected that latency metrics would increase. Or if latency decreased, the release would be considered an improvement and allowed to proceed. However, if error rates also increased, the build may be blocked pending an investigation since that would suggest that the system is encountering problems that cause it to terminate early, thus decreasing latency.

In short, our algorithm analyzes the metrics and provides the results, but not the final call on a release, so the final decision on whether a release moves forward depends on the human analyst and whether they are able to use the information in the alerts to make a judgement.

2.6 Related works

A similar approach to ours was proposed in an earlier work by Laptev et al. [4], where expected behavior is modeled by an ARIMA time series model and anomalous points are identified by the prediction error of a test point.

The primary difference is that Laptev et al. compared a given test point's prediction error with that of other test points and flagged a given point when its error was an outlier with respect to other test point errors. In contrast, we compare with the prediction errors of the model points and flag a given test point whose error is an outlier with respect to the model errors.

This is significant, as it allows our algorithm to flag an entire test data set whose overall behavior differs from the expected. It also allows us to detect when all test points are anomalous. Additionally, if all test data points deviate from the model by approximately the same amount, even if the test point values are visibly behaving differently from the model, the test points would not be flagged due to all of them having approximately the same amount of prediction error. Therefore, none of the test points would appear to be an outlier when compared to each other.

Other differences also include our support for seasonal ARIMA models and our use of the IQR method of identifying outliers rather than the standard deviation method, so that the algorithm is viable even in the case where the prediction error distribution does not fit a normal distribution and thus cannot use the three-sigma rule.

3 Results

Using our own deployment process and its associated set of metrics, we ran this algorithm and also had our human analysts perform manual metrics analysis to compare the algorithm's ability to detect anomalies and overall changes against that of the analysts.

We found that the algorithm was at least as capable of identifying changes in metric behavior as manual monitoring by a human. In some cases, the algorithm was also able to pick up changes that were missed during a manual inspection of a metric's graph (e.g., a slow but persistent increase in a latency metric that suggested a growing problem but that was not steep enough to be flagged during a manual analysis). Some changes were too subtle for the human to notice, depending on the resolutions of the graphs they used for manual inspection.

We also greatly reduced the amount of time spent on inspecting metrics since we now need to manually review only the flagged metrics. The need to manually review metrics was not eliminated as it is still necessary to figure out, when diagnosing a problem with a deployment, what metrics were flagged and in which direction they deviated. There is no need, however, to analyze the data and make a decision on whether the metric is deviant.

This resulted in a reduction of about 92-98% in the number of metrics that our human analysts needed to review, where the type and magnitude of the issue in a deployment affected the number of deviant metrics that were flagged. For the purpose of the comparison, metrics that were not flagged were also reviewed to ensure that they did not exhibit deviant behavior that was missed by the algorithm. But when using this algorithm in practice, the non-flagged metrics, which made up 92-98% of the total number of metrics, need not be manually inspected.

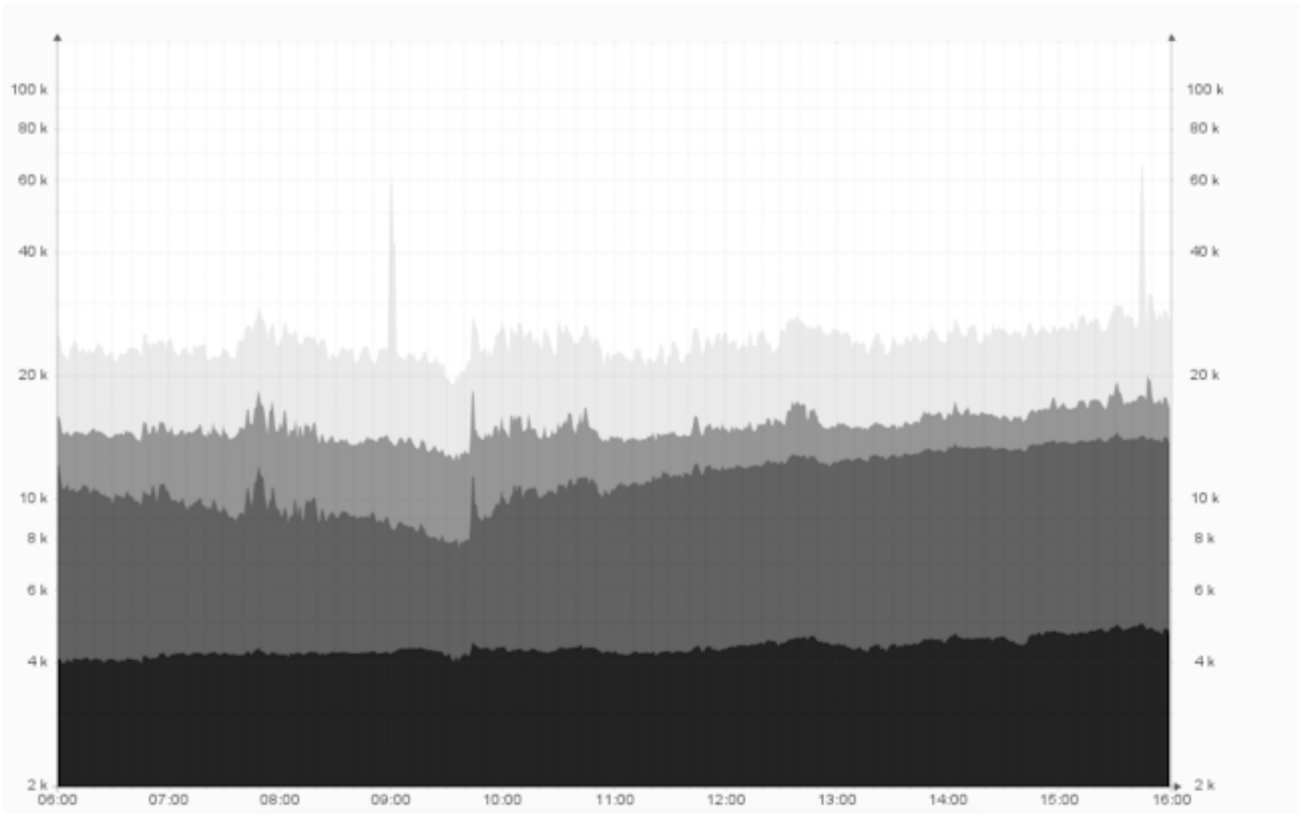
Additionally, no further analysis of the metric's behavior needed to be done during the review; only a go/no-go decision on the deployment needed to be made based on which metrics were found to exhibit changed behavior.

Overall, use of this algorithm has allowed us to greatly reduce the time spent on metrics monitoring while increasing the rate of detection of both individual anomalies and overall changes in metric behavior. Use of the algorithm also made the metrics analysis results less subjective than “eyeballing” the metrics graphs since the data models and error distributions could be used to justify the decision on whether a metric’s behavior changed.

3.1 Example analysis

The figure below shows a graph of several of the metrics we monitor during the deployment process. The four metrics are the maximum (lightest grey), 99th percentile (lighter grey), 95th percentile (darker grey), and median (black) values at a given time of a metric being collected by our application.

The model data was taken from 6:00 to 9:00, and the test data was taken from 15:00 to 16:00.



Based on a visual manual inspection, the 95th percentile appears to exhibit a visible increase from the model to the test data, while the 99th percentile and maximum metrics' behaviors may have changed, though it is hard to tell whether the increase is due to a few spikes from noise or if the increase is persistent. And without much closer inspection, for which there is not often time when analyzing hundreds of metrics for a deployment, it is not noticeable that the median metric also increased.

Meanwhile, our algorithm was able to correctly flag all four of the metrics for an overall increase in the test data when compared to the model data. Not only do the model parameters and other data provided by the algorithm reduce the subjectivity of the judgement, but the algorithm also runs in less time than it takes for a human analyst to retrieve and inspect the metric graphs; approximately one to two minutes compared to ten to fifteen seconds. It is also worth noting that since dozens of analyses can be run in parallel, the percentage reduction in time spent when using the algorithm is even greater for larger numbers of metrics.

4 Conclusion

In this paper, we have described a new algorithm to automate the detection of both individual anomalies and overall changes in metrics behavior. The algorithm offers an improvement on methods that detect only individual anomalous points by analyzing overall behavior within the data points being tested. It also improves upon previous methods that measured overall changes by using the individual anomalies to identify what times in the test data exhibited the most deviant behavior and contributed most to the change in metric behavior.

We found that the algorithm could identify changes in our deployment metrics at least as well as manual monitoring of metrics graphs by human analysts. In some cases, the algorithm was also able to identify more subtle changes in metric behavior that were missed during manual inspection, and reduced the time spent on analyzing metrics, in addition to reducing the subjectivity of the analysis.

4.1 Future work

Future work would include expanding the types of models available beyond ARIMA and experimenting with other time series-specific models, and even non-time series models to better support the rare metrics that do not behave like typical time series.

We also plan to gather results and feedback from more teams, including those with fewer metrics but who still prefer to automate the monitoring process. While we have made most of the parameters involved customizable via the job configuration file, we would like to receive feedback on what other options other teams would like to have control over.

References

[1] X-pack: Extend elasticsearch, kibana & logstash — elastic. https://www.elastic.co/products/ x-pack. Accessed: 2018-03-02.

[2] Sabyasachi Basu and Martin Meckesheimer. Automatic outlier detection for time series: an application to sensor data. Knowledge and Information Systems, 11(2):137–154, 2007.

[3] Dipankar Dasgupta and Stephanie Forrest. Novelty detection in time series data using ideas from immunology. In Proceedings of the international conference on intelligent systems, pages 82–87, 1996.

[4] Nikolay Laptev, Saeed Amizadeh, and Ian Flint. Generic and scalable framework for automated timeseries anomaly detection. In Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 1939–1947. ACM, 2015.

[5] Li Wei, Nitin Kumar, Venkata Nishanth Lolla, Eamonn J Keogh, Stefano Lonardi, and Chotirat (Ann) Ratanamahatana. Assumption-free anomaly detection in time series. In SSDBM, volume 5, pages 237–242, 2005.