HotspotRank: Hotspot Detection in Large-Scale Microservice Architecture

Yi Zhen[*], Yung-Yu Chung[*], Yang Yang, Lei Zhang, Ruoying Wang, Bo Long, Tie Wang, Pranay Kanwar, Dong Wang, Mike Snow, Sanket Patel, Stephen Bisordi, Viji Nair
LinkedIn Corporation, Mountain View, California, USA
{yzhen, ychung, yyang, lzhang1, ruowang, blong, tiewang, pkanwar, dowang, msnow, spatel1, sbisordi, vijinair}@linkedin.com

Abstract

To serve hundreds of millions of users, it is crucial to ensure performance and availability of a large-scale microservice architecture such as LinkedIn. Although engineers could check aggregated operational metrics such as latency and error count at various granularity via monitoring tools and set threshold alerts, it is still challenging to detect and fix the service bottlenecks in a timely manner with as few false alarms as possible. In this paper, we propose a machine learning based framework to detect the service bottlenecks which are called performance hotspots hereafter. The framework consists of four components: a severity scoring component, a criticality weighting component, a filtering component, and a ranking component. Experimental evaluation on real LinkedIn data shows that the framework is significantly more effective than baseline methods.

1 Introduction

Microservices, also known as (a.k.a.) the microservice architecture, is an architectural style that structures an application as a collection of services that are highly maintainable and testable, loosely coupled and independently deployable. The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. Note that a service oftentimes expose various RESTful APIs (Application Programming Interface), a.k.a. endpoints, for communication.

Microservice architecture has been widely adopted in industry and it usually consists of hundreds of services spreading across hundreds and thousands of machines located in different data centers, to serve users' requests. For example, a user request is routed via a load balancer to a frontend service, which fans out requests to backend services to collect and process the data to finally respond to the incoming request. The callee services of this request can also call other services, creating a call graph of requests. For example, LinkedIn has a recommendation feature called "People You May Know" that attempts to find other members to connect with on the site for one member. To show this feature, several services are called: a web server wrapped as a service to receive and parse the member's request, a recommendation service that receives the member id from the web server and retrieves recommendations, and finally a profile service to collect metadata about the recommended members for decorating the web page shown to users.

It is critical to keep the services running reliably as site availability is of paramount concern due to its revenue-impacting nature. Social network companies such as LinkedIn have dedicated monitoring teams to inspect the overall health of these services and immediately respond to any issue (Cherkasova et al. 2009). To that end, these services are heavily instrumented and alert thresholds are set and aggressively monitored. However, the alert thresholds may not always be correct, either too high or too low, because engineer may have inaccurate understanding of the service performance. It becomes even more challenging when the site traffic is growing and the alert threshold could be outdated. Furthermore, even if an alert is triggered or an anomalous service is detected, it is difficult and time-consuming to find the actual root cause of a service bottleneck.

We briefly summarize the challenges of detecting the service bottlenecks as follows:

- services are continuously developed and deployed
- each service maintains various APIs, making analyzing bottlenecks demands considerable human effort
- the reliability requirement could vary as the site grows

To cope with these challenges, this paper introduces HotspotRank, a novel machine learning framework that detect service bottlenecks, a.k.a. hotspots, in a microservice architecture. It combines historical and latest operational metric data to rank, in near real time, potential hotspots. This ranked list of hotspots decreases the overall search space for the monitoring team. As part of our evaluation, we analyzed hotspots over the course of three months at LinkedIn. HotspotRank consistently outperformed the basic heuristics that were adopted by LinkedIn's monitoring team. In terms of mean average precision (MAP) and recall (Recall), which quantifies the goodness of hotspot ranking, HotspotRank yields significantly more predictive power than any other

[*]equal contribution

technique we have tried. HotspotRank has been deployed in Capacity Analyzer by LinkedIn's Capacity Engineering team for months and has become a popular tool ever since.

The rest of the paper is organized as follows. Our proposed framework, namely, HotspotRank, is presented in Section 2 in details. The experiment results on real data are reported and discussed in Section 3, followed by related works in 4. Section 5 concludes the paper.

2 HotspotRank

2.1 Problem Statement

To detect bottlenecks of a microservice architecture, one common practice is to perform pressure tests, a.k.a., *Load Test*(Zhang et al. 2016), on each data center. Specifically, during one load test on one target data center, the majority of the traffic from other data centers will be routed to the target data center. Operational metrics, such as latency, error count, CPU and thread pool usage, and etc., of the services and endpoints located at the target data center are monitored. The load test will fail if there are services or endpoints being alerted due to failure (defined by hand crafted rules on various operational metrics). Service owners will then be notified to investigate and fix the problems. We call the alerted services bottleneck *hotspots*. One common rule based approach is to hand craft the alert thresholds for different operational metrics for each service and endpoint, however, these rules can become outdated and/or inaccurate as overall site traffic grows, and it's challenging for service owners to find appropriate thresholds at higher traffic levels.

In this work, we present a machine learning approach to the problem above. First of all, we calculate the severity score based on the deviation between observed latency and expected latency at normal traffic. Then we calculate criticality weight for each endpoint by exploiting the call graph structure. To further find out hotspots that impact others, we propose a filtering technique called call path grouping. At last, with ranking lists generated by above methods, we collect labels from users and build a gradient boosted tree model to predict the root causes and use the probability to generate the final ranking list. Each step is described in details in following subsections.

2.2 Hotspot Severity Score

In practice, monitoring teams always use hand crafted rules to measure performance degradation. The rules are always based on domain knowledge and system requirements. However, it has always been argued that the rules are not always available for many services and they are not reliable as the services are evolving all the time.

In this subsection, we introduce the the concept of *severity score* for each endpoint[1] which describes how severe an endpoint is in terms of performance degradation from expectation. The severity score is calculated based on historical operational metric data which could be used to measure performance degradation. For example, significantly increased latency and error count are known to be directly related to performance degradation. As a result, the proposed severity score should always reflects the performance degradation more accurately than hand crafted rules.

In the following context, we define the severity score as the difference between the observed latency and the expected normal latency, using 95% quantile call time as the delegation of latency here. Note that we could also use other performance metrics such as error count, CPU and memory utilization rate, individually or collectively, to measure the performance degradation, and we leave this for future work. Specifically, we define the severity score s of endpoint e of a set of consecutive timestamps $[t_1, t_2, \cdots, t_N]$ (e.g., a load test period) as follows:

$$s = \sum_{t \in [t_1, t_2, \cdots, t_N]} I[l_t > \bar{l}_t^{+}] \times s_t, \qquad (1)$$

where

- t is a timestamp between t_1 and t_N,
- l_t is the observed latency at timestamp t,
- $\bar{l}_t^{+}$is the upper confidence bound of expected normal latency at timestamp t,
- $I[X]$ is an indicator function which equals to 1 if the statement X is true and 0 otherwise,
- $s_t = \frac{l_t - \bar{l}_t}{\sigma_t}$ is the severity score where $\bar{l}_t$ and σ_t are the expectation and standard deviation of normal latency at timestamp t, respectively

The key of calculating severity score is an accurate estimation of expected normal latency (e.g., 95% call time when traffic is normal) and its confidence. We can simply estimate $\bar{l}_t$, σ_t, and $\bar{l}_t^{+}$ using the mean and standard deviation of the latency of the endpoint based its historical observed latency data. For example, we can use call time during past two weeks to calculate those values. However, this naive approach does not consider seasonality of call time and ignores the fact that there are chances that the observed values during the past two weeks are abnormal or unexpected. For example, the planned load tests and the unplanned traffic shifts may happen during past weeks and the latency during that period should not be considered as *normal*. Because these latency values may severely impact the normal latency estimation, which in return contaminates the severity score estimation, especially when these situations are frequent.

To get an accurate estimate of expected normal latency and its upper bound, we propose a two-step approach which first estimates the fabric traffic and then estimates endpoint latency. We estimate traffic of fabric, a.k.a., data center, as an intermediate variable because (1) it has strong seasonality behavior and it fans out to different services in a relatively stable pattern, (2) it is closely related to endpoint latency.

Step 1: Normal Traffic Estimation The goal of this step is to estimate the normal fabric traffic at time t[2], i.e., $q_t = g(t)$, assuming g a deterministic function without prediction

[1]Calculating severity score at endpoint level provides an easier way for engineers to investigate the failures. The approach, however, can be applied to service level or host level as well.

[2]'normal' means the traffic is not during any load test or traffic shift period

variance. We build a regression model based on ordinary least square loss for each fabric and call it *traffic model*. The input is the total traffic and time features such as day of the week and hour of day and the output is normal traffic of the fabric. Please note that the observed traffic data of each fabric could be very noisy and we removed the observations outside of the interquartile range (IQR).

Step 2:Latency Estimation from Traffic Our next step is to estimate the endpoint latency $\bar{l}_t$ given expected normal fabric traffic q_t, i.e., $\bar{l}_t = f(q_t)$, assuming f a function with prediction variance, where $\bar{l}_t$ and q_t are endpoint latency and fabric traffic at time t, respectively. Since we have a lot of observations, i.e., minute level aggregation, of fabric traffic (QPS) and endpoint latency (call time), training a regression model f that best fits the observation should be reliable even though there are noises in observations. Note that we learn f for each endpoint. In Figure 1, we plot the trained model for one endpoint. It shows that a polynomial regression model can be used to capture the super-linear relationship between fabric traffic and latency.

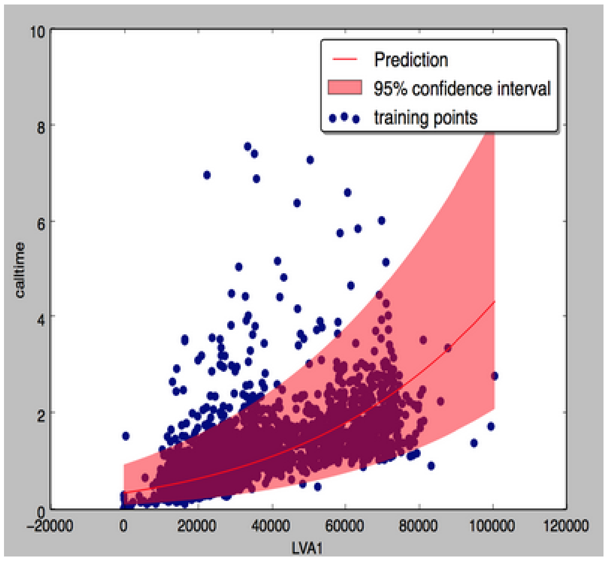

Figure 1: latency vs fabric traffic. The X-axis is traffic and the Y-axis is the latency (call time).

Given learned latency model f and fabric model g, we can easily obtain $\bar{l}_t = f(q_t)$, and $\sigma_t = sd(f(g(t)))$ for any time t, and then calculate severity score of s. In practice, we may need to filter out non-severe hotspots by applying a threshold cutoff, i.e., filtering scores $s < \epsilon_s$, where ϵ_s is a threshold which could be provided by service owners or learned from user feedback or by applying the time period threshold, e.g., removing endpoints with duration (number of timestamps) less than ϵ_t.

2.3 Hotspot Weighting via Traffic Pattern

Severity score measures severity in terms of performance degradation. On the other hand, we observe that the services and endpoints are not equally important, in other words, different services or endpoints may be of different levels of interest (Zhou et al. 2013). For example, one service may impact many other services and hence its failure may lead to much severe availability issue compared to one service which impact only a few services. As a result, we define the criticality of endpoints to measures how critical the endpoint is.

The criticality weight c is defined based on the call graph of the service network. Different from existing approaches (Kim, Sumbaly, and Shah 2013) that solely use call graph structure, we propose to use the traffic information along with the call graph.

Assume that each endpoint is a node, and the nodes are connected by caller-callee relationships and the edges are weighted by traffic such as QPS (query per second). The heavier the traffic is, the more weight the edge incorporates. Note that the graph structure and edge weights are dynamic, for example, endpoints could be added or deleted, and the traffic among endpoints could change over time. In this work, we assume a static graph structure and static distribution of traffic and leave the exploration of dynamic call graph to future work.

Inspired by the idea of weighted PageRank (Xing and Ghorbani 2004), we construct an undirected call graph[3] where the nodes are endpoints and the edges are traffic among nodes. The criticality weight of an endpoint e can be defined as

$$c(e) = \frac{1-d}{N} + d \sum_{e_j \in \mathcal{N}(e)} \frac{c(e_j)}{d(e_j)}, \qquad (2)$$

where N is the total number of nodes, d is a user provided damping factor, $\mathcal{N}(e)$ is the set of neighbors of node e, and $d(e)$ is the sum of weights of edges connected to e. The criticality weights are the entries of the dominant right eigenvector of the modified adjacency matrix which is rescaled so that each column adds up to one.

2.4 Hotspot Filtering via Call Path Grouping

With severity score s and criticality weight c, it is easy to rank the endpoints by the weighted severity score $c \times s$. However, this weighted severity score does not really tell the causal relationship between the endpoints. In practice, it is commonly observed that the performance degradation of an affected endpoint is caused by bottleneck endpoints of which the performance degradation is caused by themselves. From the monitoring teams' perspective, the bottlenecks are way more important than the affected endpoints, because the only way to resolve the system performance degradation is to fix the bottleneck endpoints. Besides, reporting affected endpoints lead to noisy alerts which could often be regarded as "false alarms". As a result, we propose a filtering method to rule out the affected endpoints and keep the candidates which are more likely to be bottlenecks.

By checking the call graph of the service network, we observe that bottleneck endpoints are always among the following cases:

[3]We use undirected graph here because the criticality is measured by the traffic volume instead of directions.

- The endpoint with the highest severity score in one call path
- The leaf endpoint [4] in one call path
- The isolated endpoints

Based on above observations, we design a filter to pick up the bottleneck endpoints and rule out the affected endpoints.

Given a microservice architecture and its call graph structure, we construct a Directed Acyclic Graph (DAG) $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of endpoints and $\mathcal{E}$ is the set of edges. For example, $e_j = (v_k, v_l)$ is a directed edge from endpoint v_k and endpoint v_l, meaning that v_k calls v_l.

Let $\mathcal{H} \subset \mathcal{V}$ be a set of candidate hotspots which is a subset of all endpoints. Let $w(h_\eta)$ be the weight of a candidate endpoint h_η.[5] The weight can either be the severity score or criticality weighted severity score. We construct a subgraph $\mathcal{G}_\mathcal{H}$ with all endpoints in $\mathcal{H}$ and edges with both nodes in $\mathcal{H}$.

Traversing the subgraph $\mathcal{G}_\mathcal{H}$, it is easy to obtain a set of call paths. Let $\mathcal{P} = \{p_\zeta\}$ be the set of the longest paths. Please note that the longest paths are the paths which are not part of by any other paths. In other wards, paths p_ζ is a path from *ROOT* to *LEAF*. *ROOT* is an endpoint without upstream endpoints and *LEAF* is an endpoint without downstream endpoints.

Lemma 1. *A path is a longest path if and only if the path starts from root and ends at leaf.*

Proof. Let p be a path starts from *ROOT* and ends at *LEAF*, and $|p|$ is the length of the path p. If p' is longer than p ($|p'| > |p|$) and $p \subset p'$, there must be an endpoint which is upstream of *ROOT* or the downstream of *LEAF*, which contradicts the definition of root and leaf. Hence p is a longest path.

Let p be a longest path, but it doesn't start from *ROOT* or ends at *LEAF*. We can either find an upstream of the starting endpoint or downstream of the ending endpoint, and hence get a longer path p' with $|p'| > |p|$ which contradicts the definition of longest path. $\square$

With longest paths $\mathcal{P}$, we apply an aggregation function, $r(p_\zeta)$, to calculate the score of each path p_ζ in $\mathcal{P}$. $r(p_\zeta)$ can be the summation of node weights of nodes in p_ζ, e.g., $r(p_\zeta) = \sum_{v \in p_\zeta} w(v)$, or the average of node weights, e.g., $rp_\zeta) = \sum_{v \in p_\zeta} w(v)/|p_\zeta|$[6]. The score $r(p_\zeta)$ represents the importance of the path. We reorder the paths p_ζs in P by the score $r(p_\zeta)$. After the paths are ordered by their importance, we select the representative endpoints of each path based on the following criteria: the endpoint with the highest weight, the leaf endpoint and the isolated endpoint which cannot be affected by the other nodes. With all representatives selected from all paths, duplicates are removed and a candidate set of bottlenecks is formed and ranked by the aggregated score of the path.

For example, from a subgraph in Figure 2, we can extract five longest paths in Figure 3.

[4]A leaf node is a node without downstream nodes

[5]The weight introduced here is different from the criticality weight introduced in Section 2.3

[6]We leave the exploration of other aggregation function in future work.

2.5 HotspotRank

So far we have proposed the concepts of severity score and criticality weight, and a filtering technique called call path grouping. To find out the service bottlenecks or hotspots, one can simply rank the endpoints using weighted severity scores and we call this approach Weighted Severity Ranking (WSR). One can also filter the bottlenecks using call path grouping on top of the candidates found by WSR and we call this approach Call Path Grouping (CPG). We observe that these two approaches reflect complimentary information. Specifically, WSR ranks higher the services with more severe performance degradation and higher traffic, whereas CPG ranks higher more significant services in a call path. Moreover, the two ranking approaches are unsupervised and hence ignore the user labels even when they are available. However, in practice, we can obtain a few such labels which reflect the nature the hotspots. In this section, we propose a semi-supervised learning approach to find out hotspots that the monitoring team is really interested in.

For each load test, we may have a few endpoints labeled as bottlenecks. Although we do not have negative labels such as which endpoints are *not* the bottlenecks, we can infer those negative labels from the above proposed approaches. Specifically, assuming we have N ranking lists already calculated, we collect the top K endpoints from each ranking list and merge them into a candidate set. Each candidate should have an order number from each ranking list (if it does not appear in a list, the order number is $N \times K + 1$). The candidate will be labeled as $+1$ if it is a hotspot or bottleneck (label is given by users for the load test) and -1 otherwise. Then we train a gradient boosted decision trees classifier implemented in XGBoost (Chen and Guestrin 2016) for the binary classification problem. Please note that the input of the classifier is a candidate hotspot with N rank order numbers and the output is the probability of being hotspots. Then we ranking the candidates using the output of XGBoost classifier. We call this approach HotspotRank.

3 Experiments

3.1 Evaluation Metrics

To verify the proposed approach, we use Mean Average Precision (MAP) and Recall as evaluation metrics. The notations are defined in Table 1.

Notation	Meaning
J	number of loat test
P_{jk}	precision of top k hotspots for load test j
R_{jk}	relevance of the kth hotspot for load test j. 1 for true and 0 otherwise
M_j	total number of true hotspots of load test j
H_{jk}	number of true hotspots in top K hotspots

Table 1: Notations for Evaluation Metric

MAP@K measures accuracy of the top K hotspot of a

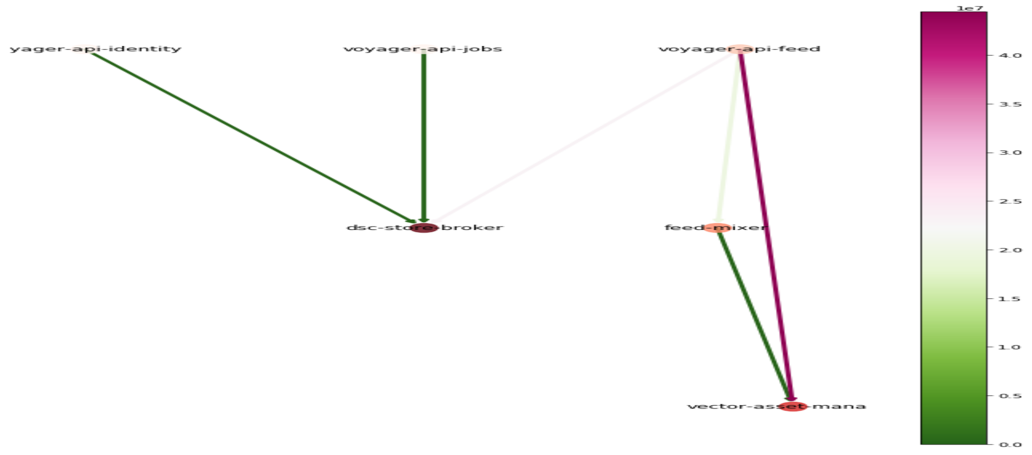

Figure 2: Sub call graph with six nodes. The color of edges indicates the traffic.

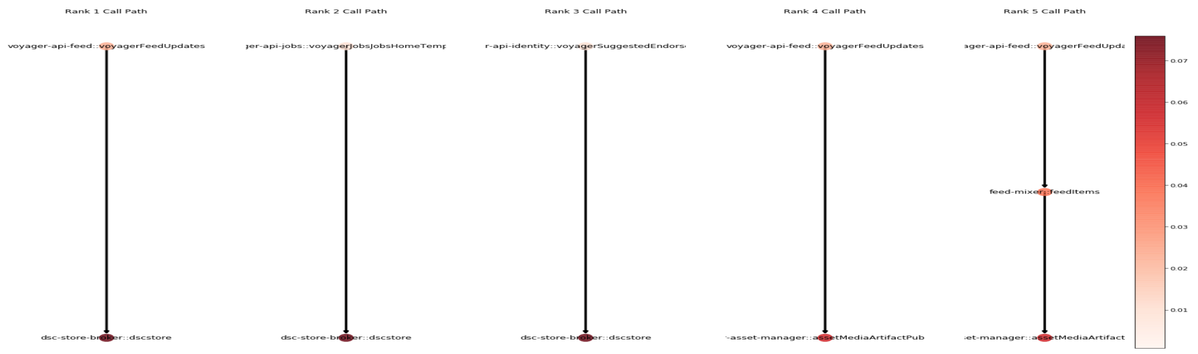
Rank 1 Call Path
voyager-api-feed::voyagerFeedUpdates
dsc-store-broker::dscstore
Rank 2 Call Path
ger-api-jobs::voyagerJobsJobsHomeTemp
dsc-store-broker::dscstore
Rank 3 Call Path
r-api-identity::voyagerSuggestedEndors
dsc-store-broker::dscstore
Rank 4 Call Path
voyager-api-feed::voyagerFeedUpdates
-asset-manager::assetMediaArtifactPub
Rank 5 Call Path
ager-api-feed::voyagerFeedUpda
feed-mixer::feedItems
et-manager::assetMediaArtifact
0.07
0.06
0.05
0.04
0.03
0.02
0.01

Figure 3: Five longest paths extracted from the call graph in Figure 2. The node color shows the weight of nodes.

ranking list and can be defined as follows:

$$\text{MAP@}K = \frac{1}{J}\sum_{j=1}^{J}\frac{1}{\min(M_j, K)}\sum_{k=1}^{K} P_{jk} \times R_{jk}. \quad (3)$$

Higher MAP@K value indicates more true hotspots are ranked higher. For example, $MAP@10 = 0.2$ means 2 hotpots among the top 10 hotspots are true hotspots.

Recall@K measures the percentage of true hotspots are reported in the top K hotspots and can be defined as follows:

$$\text{Recall@}K = H_{jk}/M_j. \quad (4)$$

Higher Recall@K value indicates more true hotspots are reported. For example, Recall@$10 = 0.4$ means if we look at the top 10 hotspots in Dashboard, these 10 hotspots will cover 40% true hotspots during a Load Test.

3.2 Dataset

We collected the operational metric data such as inbound traffic and call time (95% quantile) of all LinkedIn applications and their endpoints at one minute granularity. Note that the inbound call time of an endpoint is the aggregation among all hosts which are serving the endpoint. To verify the detected hotspots, three experienced engineers from the monitoring team reviewed the load tests during 1st February, 2019 to 30th June, 2019 and verified the detected hotspots based on consensus.

We collected call graph data over all services and endpoints and aggregated over a short period of time, such as one week[7]. For example, the value of and edge indicates the traffic (QPS) between the endpoints.

3.3 HotspotRank

We compare HotspotRank with its components in this section. Three ranking lists are compared:

- **WSR** uses criticality weighted severity score
- **CPG** uses call path weights
- **HotspotRank**: uses probability of being root cause

.

In Table 2 and 3, we observe that HotspotRank significantly improves WSR and CPG in terms of MAP and Recall,

[7]The reason why aggregated data instead of snapshot data is used is that it will remove the noise and avoid the data unavailability issue

K	5	10	20
WSR	12.27%	15.73%	16.75%
CPG	19.71%	21.60%	23.19%
HotspotRank	27.24%	30.28%	30.79%

Table 2: MAP@K

K	5	10	20
WSR	25.56%	48.41%	59.52%
CPG	22.70%	30.63%	49.21%
HotspotRank	59.05%	75.24%	79.37%

Table 3: Recall@K

respectively. It validates our assumption that the user provided labels, although limited, are very important for hotspot ranking.

4 Related Work

Detecting root causes of service failures among a service network has been studied in MonitorRank (Kim, Sumbaly, and Shah 2013). However, MonitorRank is different from HotspotRank in the following aspects: (1) MonitorRank is to find root causes on normal traffic whereas HotspotRank is to detect service bottleneck during the high traffic pressure. (2) MonitorRank is unsupervised without considering the user labels even if they exist. However, HotspotRank utilizes the user labels which are apparently important and effective to find hotspots. Specifically, HotspotRank adopts a gradient boosted tree model trained on multiple ranking lists and is also novel in terms of ranking list aggregation (Lee et al. 2015).

Generally speaking, HotspotRank is related to anomaly or outlier detection which has been a hot topic during the past few years. (Hodge and Austin 2004) did a survey on the outlier detection methodologies. (Laptev, Amizadeh, and Flint 2015) proposed a generic framework for automatic anomaly detection on time series data. They provided methods for outlier detection and alert filters to avoid false alarms. (Kieu, Yang, and Jensen 2018) proposed outlier detection methods using deep neural network for multidimensional time series. Besides, anomaly detection techniques have also been applied in data streams (Bose et al. 2017), videos (Xu et al. 2017) and networks (Jabez and Muthukumar 2015).

5 Conclusion

Detecting service bottleneck, a.k.a. hotspots, in large-scale microservice architecture under pressure tests is a challenging task. In this paper, we propose a machine learning based framework named HotspotRank to detect the hotspots. HotspotRank can calculate the severity scores based on estimation of normal operational metrics, reweight the endpoints via traffic pattern, filter the bottlenecks by exploiting the call graph structure, and rank the hotspots using a tree based model trained with user provided hotspot labels. Experimental evaluation on real data shows the MAP and Recall have been significantly improved. HotspotRank has been deployed in production for months and helped reduce the number of failed load tests significantly and speed up the debugging process of the monitoring team.

In future, we plan to provide further insights about failure reasons, such as hardware and/or software issues, to help users diagnose and fix the service bottlenecks promptly.

References

Bose, B.; Avasarala, B.; Tirthapura, S.; Chung, Y.-Y.; and Steiner, D. 2017. Detecting Insider Threats Using RADISH: A System for Real-Time Anomaly Detection in Heterogeneous Data Streams. *IEEE Systems Journal* 11(2):471–482.

Chen, T., and Guestrin, C. 2016. Xgboost: A scalable tree boosting system. *CoRR* abs/1603.02754.

Cherkasova, L.; Ozonat, K.; Mi, N.; Symons, J.; and Smirni, E. 2009. Automated anomaly detection and performance modeling of enterprise applications. *ACM Transactions on Computer Systems (TOCS)* 27(3):6.

Hodge, V., and Austin, J. 2004. A Survey of Outlier Detection Methodologies. *Artificial Intelligence Review* 22(2):85–126.

Jabez, J., and Muthukumar, B. 2015. Intrusion Detection System (IDS): Anomaly Detection Using Outlier Detection Approach. *Procedia Computer Science* 48:338–346.

Kieu, T.; Yang, B.; and Jensen, C. S. 2018. Outlier Detection for Multidimensional Time Series Using Deep Neural Networks. In *2018 19th IEEE International Conference on Mobile Data Management (MDM)*, 125–134. IEEE.

Kim, M.; Sumbaly, R.; and Shah, S. 2013. Root cause detection in a service-oriented architecture. *ACM SIGMETRICS Performance Evaluation Review* 41(1):93–104.

Laptev, N.; Amizadeh, S.; and Flint, I. 2015. Generic and Scalable Framework for Automated Time-series Anomaly Detection. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1939–1947. ACM Press.

Lee, C.-J.; Ai, Q.; Croft, W. B.; and Sheldon, D. 2015. An optimization framework for merging multiple result lists. In *Proceedings of the 24th ACM international on conference on information and knowledge management*, 303–312. ACM.

Xing, W., and Ghorbani, A. 2004. Weighted pagerank algorithm. In *Proceedings. Second Annual Conference on Communication Networks and Services Research, 2004.*, 305–314. IEEE.

Xu, D.; Yan, Y.; Ricci, E.; and Sebe, N. 2017. Detecting anomalous events in videos by learning deep representations of appearance and motion. *Computer Vision and Image Understanding* 156:117–127.

Zhang, Y.; Meisner, D.; Mars, J.; and Tang, L. 2016. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. *Acm Sigarch Computer Architecture News* 44(3):456–468.

Zhou, Y.; Liu, L.; Perng, C.-S.; Sailer, A.; Silva-Lepe, I.; and Su, Z. 2013. Ranking services by service network structure and service attributes. In *2013 IEEE 20th International Conference on Web Services*, 26–33. IEEE.