Understanding Latency Variations of Black Box Services

Darja Krushevskaja*
Dept of Computer Science
Rutgers University
Piscataway, NJ
darja@cs.rutgers.edu

Mark Sandler
Google Inc
1600 Amphitheatre Parkway
Mountain View, CA
sandler@google.com

ABSTRACT

Data centers run many services that impact millions of users daily. In reality, the latency of each service varies from one request to another. Existing tools allow to monitor services for performance glitches or service disruptions, but typically they do not help understanding the variations in latency.

We propose a general framework for understanding performance of arbitrary black box services. We consider a stream of requests to a given service with their monitored attributes, as well as latencies of serving each request. We propose what we call the *multi-dimensional f-measure*, that helps for a given interval to identify the subset of monitored attributes that *explains* it. We design algorithms that use this measure not only for a fixed latency interval, but also to explain the entire range of latencies of the service by segmenting it into smaller intervals.

We perform a detailed experimental study with synthetic data, as well as real data from a large search engine. Our experiments show that our methods automatically identify significant latency intervals together with request attributes that explain them, and are robust.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: [Performance Attributes, Measurement Techniques]; D.2.8 [**Software Engineering**]: Metrics—*Complexity Measures, Performance Measures*; H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Performance Analysis, Latency Characterization

1. INTRODUCTION

Millions of people rely on cloud services in their daily activities: people search for information, work on documents, listen to music, watch videos, manage their picture libraries, chat with friends and buy goods. The quintessential requirement of these services is to provide hyper low latency. Imagine the expectations of a user who simultaneously accesses a service to search for information, publishes a blog entry, edits her video or updates a document real time while being on a video call. These users expect near-instant response in both up and down link interactions between them and services. Even slight delays frustrate users and encourage them to seek out alternatives. For example, [14] indicates that optimal wait time is 0.1 seconds, and users start abandoning web page, if it is not presented within that window. In her interview to CNET, Marissa Mayer mentioned, that *any* increase in the latency causes search queries to be abandoned. Amazon found that 100 ms increase in load time of `Amazon.com` decreases sales by 1%.[1] Engineers make it their mission to improve service running time even by a few milliseconds. As millions of people use these services each day, lower latency translates into significant savings for the society as a whole. Thus, latency is a critical factor for the success of any cloud service and the key to user experience.

Cloud services operate in warehouse-sized data centers and run on clusters of machines to handle user requests. These systems, in turn, distribute work to ever larger number of machines and sub-systems for data procession and machine learning. These systems handle failures, heavy loads, spam and attacks by dynamically managing resources, sometimes sharing resources across services. As a result the overall systems that handle cloud services tend to be fairly complex. Even for a single service one can observes significant variations of the latency. In Figure 1 we plot distribution of latencies for 2 back end services, we also refer to observed latency distribution as *latency profile*. Notice, that for the service on the left latency can be anywhere between 0 and 500 ms. For the second service observe that while majority of requests take around 50-60 ms, significant fraction of requests takes longer than 100 ms.

Cloud service providers address latency in a variety of ways. They typically have dashboards that track multiple measurements, including latency. This gives some visibility into how latency varies over time, number of failures or fatal symptoms. Our focus is however on normal operation of the cloud services. Engineers routinely develop, test, and launch new code. Often newly launched or modifed services, disrupt the performance of shared services they depend on. Given that such changes are often coordinated by different teams, engineers need fine grained visibility into latency. In particular they need to find what explains the changes in the latency profile, so they can tweak and fix if needed.

Motivated by this, we address the problem of explaining the latency profile of any service. In general, there has been

*Work was done while the author was visiting Google

WWW 2013, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2035-1/13/05.

[1] `http://www.speedawarenessmonth.com/does-latency-really-matter/`

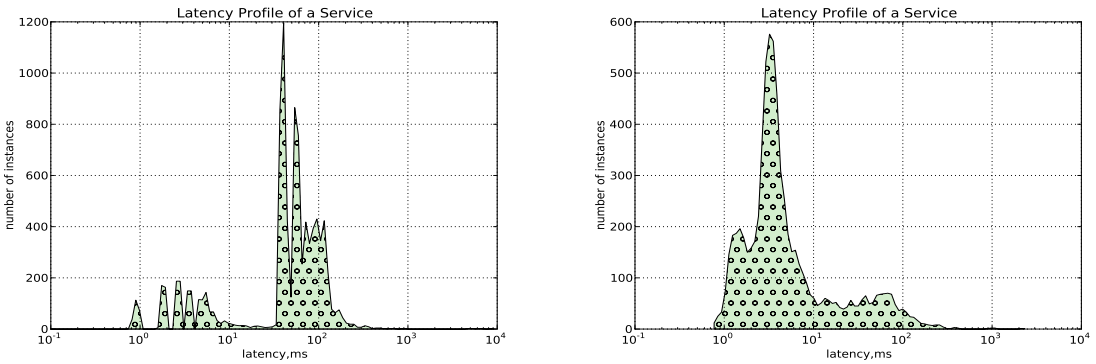

Figure 1: Latency distributions for two backend services (real data)

very little research on latency performance of cloud services. Some of the recent work focuses on modeling the call tree of complex services and trying to predict latency [18, 13, 19, 3]. We are not aware of any prior work on understanding latency profile in a way that is useful to engineers and network monitors. We study this problem and make the following contributions:

- We develop a general approach to understanding the latency profile of services. In particular, we model services as black boxes, so our framework is applicable to any service, notwithstanding its semantics or subsystems called. Also, we focus on analyzing the latency profile *directly* in terms of the measurable attributes, *e.g.,* origin of request, machine that service was running on, CPU rate. This turns out to be what engineers need, as we explain in Section 2.

- We propose what we call the *multidimensional f-measure*, and use it, given latency interval, to find a subset of monitored attributes that explains it. This measure is combinatorial and generalizes the well known (one-dimensional) f-measure used in information retrieval area to our problem. We propose algorithms to use this measure not only to explain any given interval of latency, but also to explain the entire range of latencies of the service by segmenting it into smaller intervals. Our overall algorithm uses dynamic programming to select small subset of intervals that provide insight into reasons for latency variations.

- We use real data from 2 large internet services, as well as several synthetic data sets for experimental study. We use synthetic data to study robustness and performance of our algorithms. Application of the framework to real data sets showed that analysis discovers interesting patterns.

In what follows, in Section 2 we introduce our approach to understanding latency variations for a fixed service and define multi-dimensional f-measure. In Section 3 we formally define problems. We present our algorithms for single interval analysis in Section 4.In Section 5 we present algorithm that helps to understand the whole range of latency variation for the fixed service. Section 6 contains detailed experimental study.

2. OUR APPROACH

There are many web sites, for instance Amazon, Ebay or Google, each offering multiple services each supported by umpteen number of backends. Each service receives hundreds of millions of requests a day. Conceptually, each such service invocation can be represented by a call tree [18]; in order to fulfill the request, each service typically calls other services which in turn can call others. Dependent services can be executed in parallel or sequentially, synchronously or asynchronously. The main observation is that even for a fixed service, latency varies significantly over the many times it is invoked. Figure 1 illustrates latency variation of real world services.

There are many reasons for these variations: the size of the request svaries from small to large; sometimes the answer to user query is in the cache, other times on the disk; each instance of the service may call a variable number of sub services; systems use a combination of techniques to deal with load, from moving slower operations to asynchronous mode to dynamically adjusting computing and other resources; there are outright resource failures; etc. Furthermore, there are implicit dependencies across services as they share computing, storage or network resources.

Our goal is to develop methods to understand latency variations. We start with two requirements.

- The service has to be treated as a black box. First, even for a fixed service for which we have the source code, the precise call tree is an unknown and might depend on the request. Even using detailed monitoring system, call trees are difficult to reconstruct [18]. In our case, we wish to be able to understand the latency variations even if we can not reconstruct trees. Second, we would like to avoid assumptions required, *e.g.,* latencies of dependent services are independent random variables. Finally, we wish to be agnostic of the service internals. Cloud services run hundreds of services and these evolve over time. By viewing services as black boxes and being agnostic to their internal semantics, our approach will be general.

- The reasons for the variations in latency need to be understandable to humans, for example *e.g.,* in terms of few of the monitored attributes. Imagine an engineer building services. She is trying to foresee potential bottlenecks and faults, and designs, implements and

tests carefully. Nevertheless, due to the complexity of the system, the latency of the service will vary from request to request. In order to verify or improve the latency of the service, the engineer has to verify impact of multiple factors. Verification of impact of a single factor (including the impact of her code on the latencies of other services) needs a significant amount of work and time, including connecting with other development teams. Verifying many factors or combinations of factors is even more time consuming. So, in practice, it would be of a great help to provide a small number of potential factors to verify. Further, engineers prefer these factors to be monitored attributes that they can understand and reason about directly. This in particular means that regression based models are not very useful. For example, linear regression outputs a linear combination of attributes as potential explanation of latency variation. These vectors are notoriously difficult to interpret in terms of the data and underlying patterns, even if they contain only few arguments, *e.g.,* "search requests are slow if and only if $2 \times requestSize - 0.5 \times searchersAge > 50$".

Our approach is as follows. Consider some fixed service. The service is called millions of times, and for each call, we rely on monitoring systems in place (*e.g.,* X-Trace [7, 6], Dapper [20]) to gather data about executions of the service. With each execution we associate a set of possibly related attributes, derived from the service invocation itself, or invocation of any of dependent services. We lose the structure and precedence constraints, and focus on the set of attribute values. For example, instead of reconstructing call tree as in [18], we concentrate our attention on values of attributes associated with the service itself (*e.g.,* request size, response size, service name, custom annotations) and values for attributes that are relevant to the execution (*e.g.,* CPU load, disk load), as well as the latency for that request. The output of our analysis will be a small set of attributes.

Multi-dimensional f-measure.

For now assume that each request is a set of binary attributes (we will later see how to reduce the problem to this case). Let us focus on a specific latency interval I and say set F of attributes is the reason the latency is in interval I. For example, if request originates from USA and has request size of 15 bytes it is processed in 50-75 msec.

Let $r_i = (f_1^i, \ldots, f_m^i, \lambda_i)$ be request to the service, where f_j^i is the value of feature f_j observed for the request r_i, and $\lambda_i = 1$ if latency $L_i \in I$, and 0 otherwise.

A principled approach is as follows:

- Since the attributes F need to explain the latency interval, learn the best prediction algorithm $\mathcal{A}$ on $F = (f_1, f_2, \ldots, f_k)$ that maps each request r_i to 1 or 0 depending on whether latency for request r_i is in I or not, respectively.

- Any such $\mathcal{A}$ will not predict precisely the latency interval I. That is, along with true positives tp:

$$tp = \{r_i : \mathcal{A}(f_1^i, f_2^i, \ldots, f_k^i) = 1 \ \& \ \lambda_i = 1\},$$

there will be false negatives fn:

$$fn = \{r_i : \mathcal{A}(f_1^i, f_2^i, \ldots, f_k^i) = 0 \ \& \ \lambda_i = 1\},$$

and there will be false positives fp, defined as

$$fp = \{r_i : \mathcal{A}(f_1^i, f_2^i, \ldots, f_k^i) = 1 \ \& \ \lambda_i = 0\}.$$

Frequently in machine learning and data mining, the quality of prediction is measured by the f-measure [21], $FM(\mathcal{A})$ is defined as follows:

$$FM(\mathcal{A}) = 2\frac{precision \times recall}{precision + recall}. \quad (1)$$

where $precision = \frac{|tp|}{|tp|+|fp|}$ and $recall = \frac{|tp|}{|tp|+|fn|}$.

Intuitively, $\mathcal{A}(F)$ is a binary predictor, and f-measure is a proxy of how well the binary "feature" $\mathcal{A}(F)$ is correlated with the interval. If every request in the interval I has $\mathcal{A}(F) = 1$ and no request outside of I gets $\mathcal{A}(F) = 1$, then F provides the best explanation for the interval existing in the data.

In our case, as we argued earlier, engineers wish output set F to directly determine the interval of interest I, and not via some sophisticated function $\mathcal{A}$ that may be difficult to interpret. Hence, we rephrase the problem directly in terms of the attribute values. We modify the one-dimensional f-measure that applies to $\mathcal{A}(F)$ to a multidimensional f-measure that applies to many binary features.

Definition 1. *For fixed interval I and a set of features F we define the* multidimensional f-measure *as follows:*

$$Q(I, F) = 2\frac{mprecision(I, F) \times mrecall(I, F)}{mprecision(I, F) + mrecall(I, F)} \quad (2)$$

We define true positives mtp, false negatives mfn and false positives mfp as follows:

$$mtp = \{r_i : f_1^i \bigwedge f_2^i \bigwedge, \ldots, f_k^i = 1 \ \& \ \lambda_i = 1\}$$

$$mfn = \{r_i : f_1^i \bigwedge f_2^i \bigwedge, \ldots, f_k^i = 0 \ \& \ \lambda_i = 1\}$$

$$mfp = \{r_i : f_1^i \bigwedge f_2^i \bigwedge, \ldots, f_k^i = 1 \ \& \ \lambda_i = 0\}$$

Then $mprecision = \frac{|mtp|}{|mtp|+|mfp|}$ *and* $mrecall = \frac{|mtp|}{|ntp|+|mfn|}$.

Multidimensional f-measure has many desirable properties. For example, $0 \leq Q(I, F) \leq 1$. Adding feature f_a which correlates perfectly with one of the features in F does not change the function, that is, $Q(I, F) = Q(I, F')$ where $F' = \{F, f_a\}$. On the other hand, $Q(.,.)$ does not capture certain inverse relationships. For example, if f_a is the complement of f_b, that is, any tuple with $f_a = 1$ $(= 0)$ means $f_b = 0$ $(= 1$, respectively), then $Q(I, \{f_a, f_b\})$ is same as $Q(I, f_a)$. In our application, in order to overcome such situations we add inverts for binary features.

Similarly to f-measure, multidimensional f-measure is not convex and hard to optimize. In particular, consider example described in Table 2. Here we are interested in interval $I = \{1\}$. The score $Q(.,.)$ of single feature f_1 is greater than score of any other single feature available, *i.e.,* $Q(L_i = 1, f_1) > Q(L_i = 1, f_2)$, however features f_2 and f_3 taken together are perfect predictor for I: $Q(L_i = 1, f_1) < Q(L_i = 1, \{f_2, f_3\})$.

f_1	f_2	f_3	L
0	0	1	0
0	0	1	0
1	1	1	1
1	1	1	1
1	1	1	1
0	1	1	1
0	1	0	0
0	1	0	0

Table 1: Non-monotonicity of $Q(.,.)$

Reduction to the Binary Case.

In our analysis, we will use the multidimensional f-measure applied to *binary features*, however *attributes* of requests are often not binary. Attribute values can be translated to binary format in many different ways, *e.g.,* binary or unary encoding, split into folds of equal length or mass. Different binary feature formation will result in different capabilities of the analysis. We use the following encoding process: for *categorical attributes*, we create separate feature for each attribute value. For *continuous attributes*, the attribute value range is split between several features, each feature corresponding to an interval of value. For a *binary feature* we include feature itself and its complement, to be able to detect situations when *absence* of some factor is decisive. For example, requests of size 15 kb and *not* from USA are processed within 50-75 msec. Observe that since the range of values of attribute are split into non-overlapping features we have:

LEMMA 1. *Let f_1 and f_2 be features that correspond to the same attribute, then if $Q(I, f_1) > 0$ and $Q(I, f_2) > 0$, then $Q(I, \{f_1, f_2\}) = 0$.*

Hence, no two features f_1 and f_2 that correspond to the same attribute will be in the multidimensional f-measure based output we generate.

3. FORMAL PROBLEM DEFINITION

We model the workload of a service as a sequence of requests $\mathcal{R}$. Each request r_i has m features $F = (f_1, f_2, \ldots, f_m)$ and observed latency L_i. W.l.o.g, we assume that each feature is binary, because of the reduction. Given the largest observed $L = \arg\max_i\{L_i\}$ that is seen in practice, each latency L_i falls in the range $[0, L]$.

DEFINITION 2. *We say the subset of features $F^* \subseteq F$ explains latency interval I if F^* maximizes $Q(.,.)$ on I: $F^* = \arg\max_{A \subseteq F} Q(I, A)$. We refer to F^* as* explanation.

The entire range $[0, L]$ is not likely to be explained by the same set of features. Hence, we approach it in two steps: (1) for fixed latency interval and scoring function find set of features that form explanation of the interval; (2) given large set of intervals with explanations, select (small) subset of intervals that provides concise summary for interesting patterns found in the data. We formalize these problems as follows.

PROBLEM 1. ***[Single Interval Analysis]*** *For fixed interval I and scoring function $Q(.,.)$ find the explanation F^*, s.t., $F^* = \arg\max_{A \subseteq F} Q(I, A)$.*

PROBLEM 2. ***[Latency Range Analysis]*** *Split the entire latency domain into non-overlapping intervals $\{I_1, I_2, \ldots, I_l\}$, s.t., total score $\sum_{i=0}^{l} \max_{A \subseteq F} Q(I_i, A)$ is maximized over all possible splits of latency domain.*

Both problems have interesting variations. For instance, in Problem 1 we are considering the case in which only one explanation is found for the interval, however one can easily extend this formulation to the case, in which single interval can have up to k explanations. For Problem 2, one can consider choosing overlapping intervals. However, in practice we observed that even without overlaps one can detect feature sets that have overlapping latency intervals. In this paper we consider only versions of problems as defined above. kur

4. SINGLE INTERVAL ANALYSIS

In this section, to solve Problem 1, we adopt and evaluate two classical algorithms: branch-and-bound [11, 17] and forward feature selection [5, 15].

Branch-and-Bound Algorithm. This algorithm is effectively a backtracking algorithm that uses branch-and-bound approach to reduce the search space. The pruning for exhaustive search relies on two observations. First, recall of a set of features F is always lower or equal to the recall of any subset of features $F' \subset F$. Thus, if we have a lower-bound γ on acceptable $Q(.,.)$ for an interval, then we can use that to lower bound acceptable recall r, and prune candidate sets: $r \geq \frac{\gamma}{2-\gamma}$. Notice, that similar observation does not hold for precision: combination of features can, and often does have higher precision than any individual feature of that set.

Nevertheless we can avoid selecting subsets of nearly identical features, by following the branch if support of the corresponding explanation reduces significantly, while still preserving recall. More formally, let S_I be the support of the interval I and $\mathcal{R}(f_1, f_2, \ldots, f_i) \subseteq \mathcal{R}$ be a subset of all requests that have features $f_1, \ldots, f_i$ satisfied, Suppose, that explanation with precision ρ exists. Thus such explanation would have support of at most $\mu = |S_I|/\rho$. Indeed, any explanation which has support more than μ, can not possibly have precision of ρ. We use this observation to obtain a bound on how much each added feature should reduce the support of requests in order to achieve aimed precision ρ. We show that the set can be populated in such a way that adding feature f_{i+1} to first i features, reduces the support $|\mathcal{R}(f_1, \ldots, f_i)|$ by at least

$$\frac{\mu - |\mathcal{R}(f_1, \ldots, f_i)|}{k - i}$$

We formalize and prove this in the following lemma.

LEMMA 2. *Suppose explanation contains a set of features $(f_1, f_2, \ldots, f_k)$ and precision of an explanation is at least ρ. Then there exists an ordering $l_1, \ldots, l_k$, s.t., for all $i \geq 0$*

$$|\mathcal{R}(l_1, \ldots, l_i)| - |\mathcal{R}(l_1, \ldots, l_{i+1})| \geq \frac{\mathcal{R}(l_1, \ldots, l_i) - |S_I|/\rho}{k - i}$$

Proof. Fix $t \geq 0$ and suppose sequence $l_1, \ldots, l_i$ satisfying the lemma has been built for all $i \leq t$. Obviously when $t = 0$, such condition is trivially true. Let $l_1, \ldots, l_t$ be one such order, and without loss of generality assume $l_j = j$ for $j \leq t$. Let $T_i = S_I(f_1, \ldots, f_i)$. We have $T_i \subseteq T_{i-1}$ for all $i > 0$. Therefore since $T_k \leq \mu$, there exists $j \geq t$ such that

$$|T_j| - |T_{j+1}| \geq (T_t - \mu)/(k - t) \quad (3)$$

But $|T_t| = |T_j| + |T_t \setminus T_j|$ and

$$|T_t \cap \mathcal{R}(f_{j+1})| = |T_{j+1}| + |(T_t \cap \mathcal{R}(f_{j+1}) \setminus T_j|.$$

Now subtracting former from later we have:

$$|T_t| - |T_t \cap \mathcal{R}(F_{j+1})| = |T_j| - |T_{j+1}| + |T_t \setminus T_j| - |(T_t \cap \mathcal{R}(f_{j+1}) \setminus T_j|$$

Using the fact that

$$|T_t \setminus T_j| \geq |(T_t \cap \mathcal{R}(f_{j+1}) \setminus T_j|$$

we have

$$|T_t| - |T_t \cap \mathcal{R}(F_{j+1})| \geq (T_t - \mu)/(k - t)$$

Hence feature $l_{t+1} = f_{j+1}$, satisfies the conditions of the lemma. ■

Forward Feature Selection Algorithm. We have presented an efficient branch and bound approach which guarantees global optimality. However, if we allow suboptimal solutions, it is possible to improve running time even further. We present intuitive greedy algorithm based on forward feature selection: start with empty set of features $F^* = \{\}$ and expand it by adding a feature f_i *s.t.,*

$$f_i = \arg\max_{f_i \in F} Q\left(I, \{F^* f_i\}\right)$$

We keep growing F^* while $Q\left(I, F^*\right)$ is strictly growing.

The intuition for this method rests on two properties of multidimensional f-measure. First, as we have observed before, recall of single feature is always greater or equal to the recall of any super set. Secondly, given subset of requests $\mathcal{R}'$, set of features F' and the number of requests in the interval I we can calculate global recall.

5. LATENCY RANGE ANALYSIS

Let *split point* $r_i \in [0, L]$ be a point that separates latency range into smaller intervals. Assume we have s potential split points $\{r_o = 0, r_1, r_2, \ldots, r_s = L\}$. Let $\Theta(r_i, r_j) = \max_{A \subset F} Q\left([r_i, r_j), A\right)$ be the score of the interval $[r_i,\ r_j)$. The goal is to select subset of split points $\{r_1^*, r_2^*, \ldots, r_k^*\}$ to maximize total score $\sum_{i=1}^{k-1} \Theta(r_i^*, r_{i+1}^*)$. Notice, we do not make any assumptions about the shape of the function $\Theta(\cdot, \cdot)$.

This problem can be solved efficiently using the following dynamic programming formulation. Let $D(i)$ denote the best score for a solution that covers interval $[0, r_i)$, with initial state $D(0) = 0$. The update step is:

$$D(i) = \max_{1 \leq j < i} (\Theta(r_j, r_i) + D(j)) \tag{4}$$

In other words, to construct solution for $[0, r_j)$ we search for such a pair i, j, s.t., $\Theta(r_j, r_i) + D(j)$ is maximized. We repeat iterative step s times. This dynamic programming gives $O(|s|^2)$ running time solution.

Proof that $D(i)$ indeed finds the best score at each step i is a simple application of induction argument. Consider smallest i such that there exists a solution $D'(i) > D(i)$ then we consider the last interval that spans j and i in $D'(i)$. By induction hypotheses $D(j)$ is optimal and thus $D(i) \geq D(j) + \Theta(j, i)$, which contradicts with our assumption.

Problem variations. This problem has several variations. For instance, we implicitly assume that every interval of $[0, L]$ that was not included into final solution has $\Theta(\cdot, \cdot) = 0$. However, one can penalize for gaps in the final solution by making implicit score negative. Considering formulation in which intervals in final solution are allowed to overlap is particularly interesting. It can be solved by considering slightly different dynamic programming formulation: let $D(i, j, k)$ denote the best solution with k intervals, that covers interval $[0, r_i)$, and the latest point which has overlap is j. We do not explore this modification further in this paper.

6. EXPERIMENTS

We test our framework on two types of input: (1) synthetic and (2) real data. For the absence of labeled data, we use synthetic data to evaluate performance, robustness and running time of our algorithms. We run our algorithms on real data to learn more about abilities, perspectives and utility.

6.1 Synthetic Data

Generation.

There are multiple ways one can obtain synthetic data for experiments. For instance, one can modify data collected from real services: fix service, obtain data set, then take one feature, modify it's values, and "translate" changes to the latency value for each request instance in the data set. This approach requires extensive knowledge of structure of the service together with relations between different parameters: does change of single attribute value affects only latency value? does it affect any other attributes? Another approach is to modify running service in order to obtain data, this approach cannot be carried out in practice for majority of services. Therefore, we generate synthetic data to model situations in which irregular latency values occur when several factors occur simultaneously. For the data set we generate 2 types of requests: *Base requests* correspond to normal state of the system, *i.e.,* system behaves as expected. We assume that their latencies follow some normal distribution[2] with fixed parameters. The other type of requests contains a *pattern* or co-appearance of particular values for some set of features. Further, we assume that for each pattern latency is coming from a particular normal distribution, *e.g.,* if request i originates from USA and has request size of 100KB or more, then $L_i \sim N(\mu_j, \delta_j^2)$. It is only natural to expect that system behaves differently under different conditions. Potentially, synthetic data set can contain multiple patterns, patterns can be overlapping or not in terms of features or latency intervals.

To generate data, we fix distribution for base requests N, number of patterns p, each with predefined distribution $\{N_j\}$, and the size of the data set (number of requests n and number of binary features m). Data generation is straightforward and on the higher level is as follows. First, m features are generated: we generate attributes, each consisting of 3 to 6 binary features, together with multinomial distributions, according to which values of the attributes (and features) will be sampled. Second, we generate patterns, where pattern is essentially the bag of features $P_k = \{f_a, f_b, \ldots, f_c\}$. Finally, we sample requests by sampling values of the attributes. If sampled request i contains the pattern, then we sample it's latency from corresponding

[2]More realistic assumption would be to use log-normal distribution, however it does not change the results aside from changing the latency scale to normal.

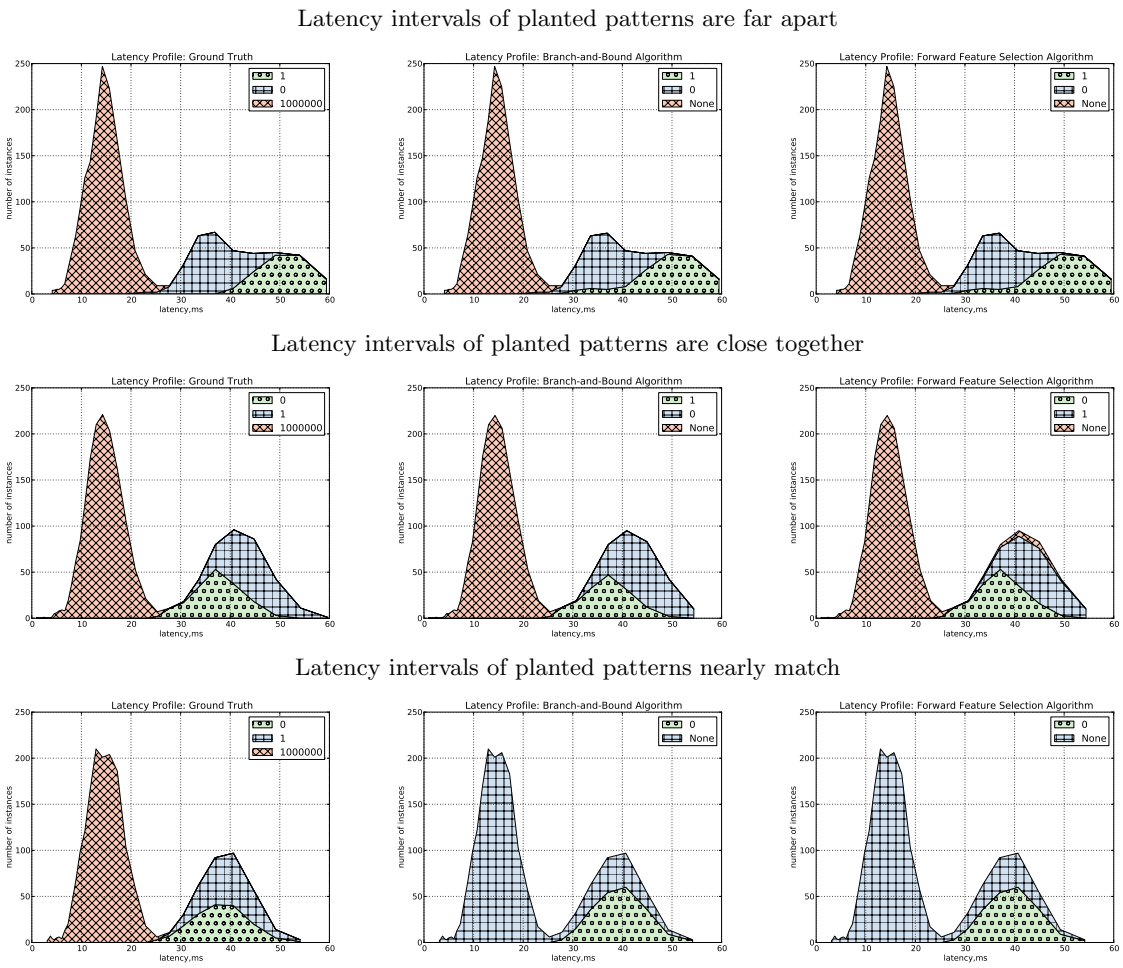

Figure 2: Tests on synthetic data: quality of analysis as latency associated with patterns "1" and "2" are brought closer and closer together. Requests with no pattern found in them are labeled with "None". Each row represents independent experiment. The left column shows ground truth, middle column contains output for branch-and-bound algorithm while the right most column shows results forward feature selection. The branch-and-bound algorithm successfully solves first and second inputs. Forward feature selection, solves the first input perfectly, finds almost perfect solution for the second one. Both methods fail to detect both patterns when latency associated with patters is virtually identical.

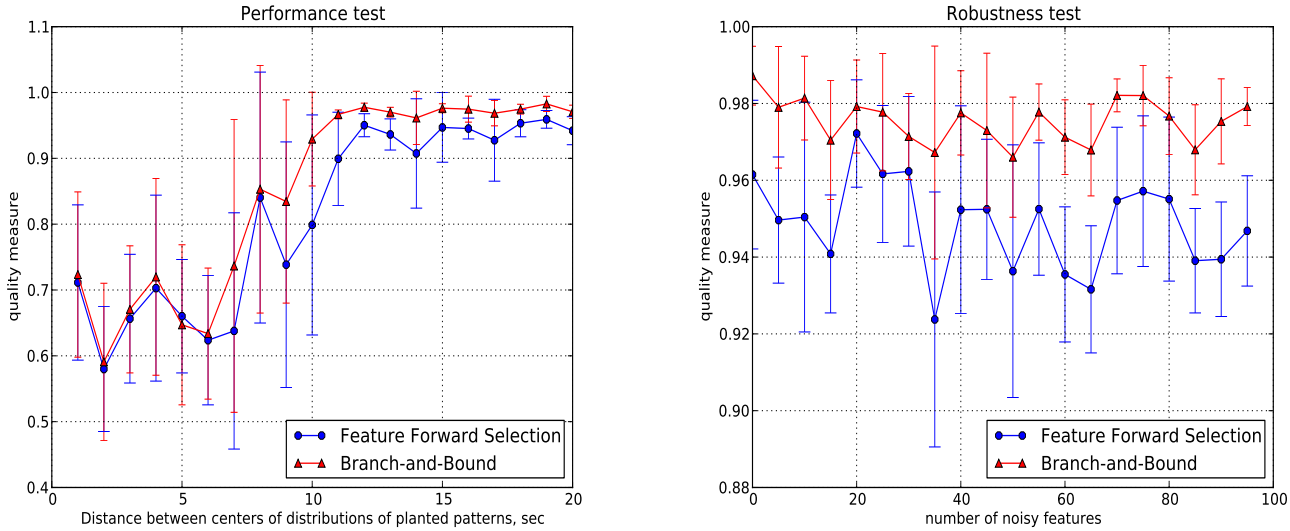

Figure 3: Left graph shows average quality measure as function of the distance between two planted patterns. The right graphs shows average quality measure as function of the number of noisy features.

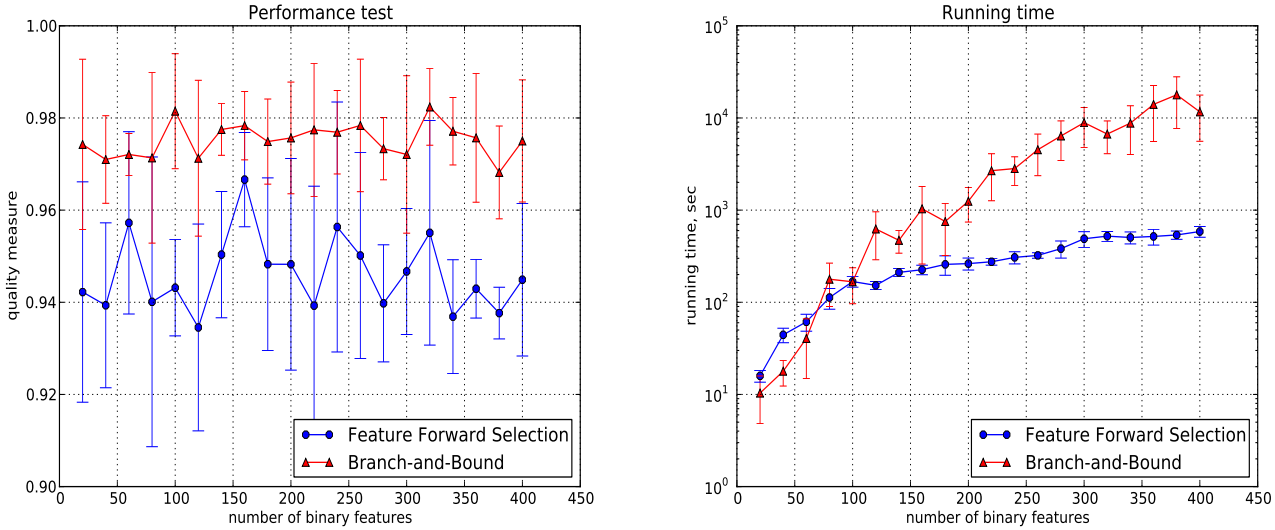

Figure 4: The left graph shows average quality measure as the number of features grows. The right plot shows running time of two algorithms as the number of features grows.

pattern latency distribution N_j, otherwise we sample the latency from N. The description of the process can be found in Algorithm 1. We further extend our generation process in order to be able to test robustness of our algorithms. We generate number of *noisy features*, each associated with some particular pattern: it co-occurs (has value 1) with the pattern with probability p, if pattern is absent it occurs with probability $1 - p$.

Algorithm 1 Synthetic Data Generation

```
1: procedure FindExplanation(n, m, {N_1, ..., N_i}, N)
2:     F ← generateFeatures(m)
3:     P ← generatePatterns({N_1, ..., N_i})
4:     for i ∈ {1, ..., n} do
5:         r_i ← instantiateRequest(F)
6:         if p_k ← findPattern(r_i, P) then
7:             L_i ← sampleLatency(N_k)
8:         else
9:             L_i ← sampleLatency(N)
10:        end if
11:    end for
12:    return {R, F, P}
13: end procedure
```

Synthetic Data Set.

Each synthetic data set consists of 1000 generated requests, each described by m attributes and g noisy features as defined in Section 6.1. Default number of (binary) features is 70, except for experiment in which we vary number of features in order to compare running times of two approaches. Default number of noisy features is $g = 15$, except for experiment in which we vary g in order to see its influence on the algorithms' performance. Level of noise p is constant and is set to 0.9 in all experiments. Each attribute consists of [3..5] features, number of features per attribute is selected uniformly at random. Each data set has 2 planted patterns: "0" and "1". Each pattern consists of [2..4] features. Base requests or requests in which analysis did not detect any patterns are labeled as "None". We find that this setup closely mimics the type of request data observed in real world. All figures describing performance of the algorithms contain averaged out quality scores defined below. To obtain a data point on the quality of analysis, we run each of our algorithms 5 times, and take the average. For each point on the diagrams we also plot standard deviation as confidence interval.

Figure 2 shows generated latency profiles and analysis outputs produced by the framework. Each row, of three, represents analysis result for a distinct data set. The first column corresponds to the ground truth, the second column corresponds to the branch and bound based analysis results, and the third column shows forward feature selection based analysis results. In a series of experiments below we investigate how well our algorithms work when we vary one parameter at a time: number of noisy features, number of features, or separation of patterns.

Quality score. Result of our analysis is essentially clustering, where requests are assigned to groups. To measure success rate of the analysis we match each of the ground truth planted patterns p_i with the best matching cluster C_j from the algorithm output. We then compute f-measure for clustering results. For that we, first compute total number of requests G that are true positive (*e.g.,* request contains planted pattern and analysis placed it to the corresponding cluster). Then we compute overall precision and recall $p = G/|\cup C|$ and $r = G/|\cup P|$, respectively. We compute f-measure as in Eq. 1.

Experiment 1. Pattern separation.

In this experiment we want to explore when our algorithms will be able to discover planted patterns, *e.g.,* do latency interval and its specific pattern must be well separated from any other requests or will our analysis be able to detect pattern when it is mixed with others? To test this we run series of trials: in each data set we plant 2 patterns. In the first trial latency intervals corresponding to patterns are well separated (see Figure 2, row 1), we proceed experiments by bringing them closer and closer together to the point when they are almost equal (see Figure 2, row 3).

The results of experiment are highlighted in Figure 2 and effect to the quality of analyses is shown in Figure 3. In most cases branch-and-bound algorithm produces slightly better results, that is to be expected given that it effectively searches the whole hypothesis space. It is also interesting that in a few points greedy algorithm provided better results. We explain this by the fact that our target metric (comparison with ground truth) is different from the metric we seek to maximize — total f-measure based on the interval selection.

Experiment 2. Noisy features effect.

It is intuitive to think that as the number of features increases, it might get harder for algorithms to discover patterns. Even more so for noisy features, as they are designed to confuse our methods. In this experiment we vary number of noisy features, in order to see how robust our algorithms are, and results are shown on Figure 3. Performance of two algorithms is comparable, and the quality of results does not degrade or depends on the number of noisy features.

Experiment 3. Running time.

From the previous experiment we can see that branch-and-bound algorithm in general performs slightly better than forward feature selection. However, it is significantly slower as the number of features grows (see Figure 4). In practice, both methods are very easy to implement using map-reduce framework: on the map step separate the search space by features, on the reduce step choose the combination of features that scored the best.

Running time analysis. In the worst case scenario branch-and-bound algorithm can run exponential time, whereas forward feature selection algorithm running time is bounded by $O(m^2|\mathcal{R}|)$, where m is the number of attributes and $|\mathcal{R}|$ is number of requests. In Figure 4 we show the running time as the number of features increases. In practice we found that the running time of branch-and-bound highly varies: in some cases it terminated quicker than forward feature selection, however tests that we performed indicate exponential running time trend.

6.2 Experiments on Real Data

In order to obtain more intuition about usefulness, perspective of the framework we test it on several data sets that are obtained from several backend services of a large search engine. Algorithms do detect unusual and interesting patterns in the data. Each data set contains requests sent to a single service. Traces were collected by Dapper [20] tracing

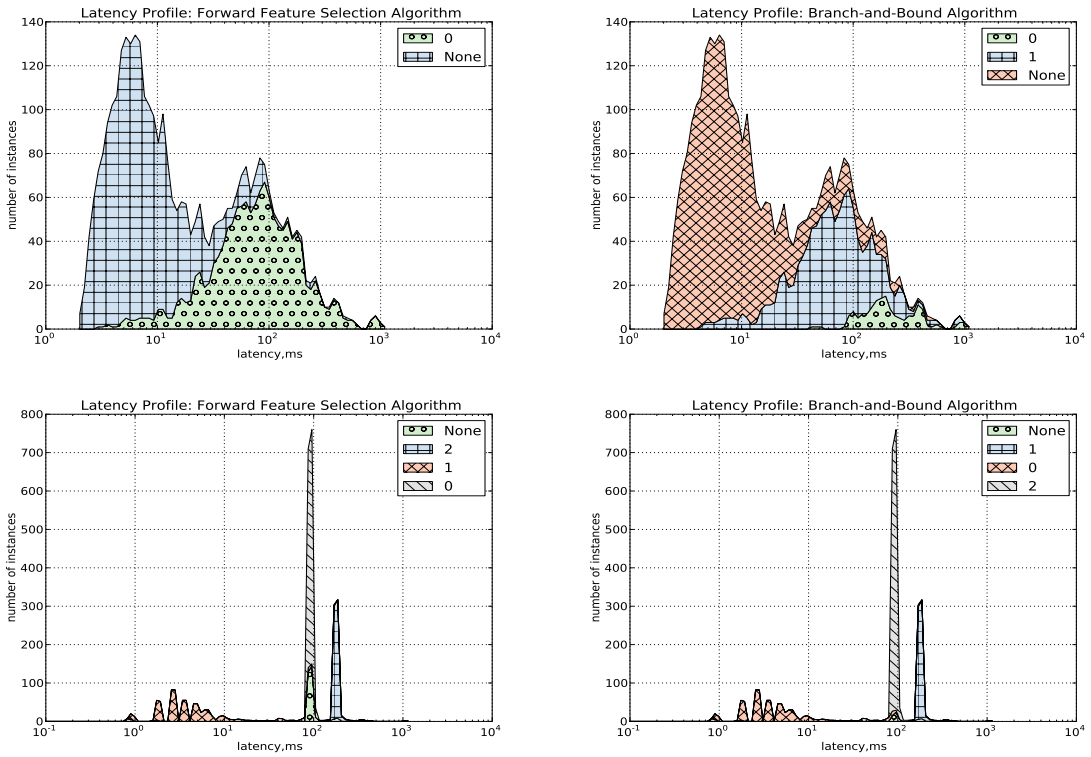

Figure 5: Latency distributions for two backend services at a large search engine.

service, each trace describes invocation of one of the services related to the request. To obtain requests, we take traces, and group them by request. Attributes for the dataset are also obtained from traces, *e.g.,* request and response sizes, service name, custom annotations, observed CPU and disk loads, latency. Depending on the service, the number of attributes is different, starting from 50. We translate values of attributes to binary format as described in Section 2.

First data set contains 1850 requests, each having 640 features. It's latency profile can be found in top row of Figure 5. One can see that requests form 2 groups: one group has latency of around 100 ms or less, the second group of requests is significantly slower. Both methods identified slower group of requests. While we are not able to go into greater detail regarding the service, we mention that the explanation produced for the slower requests indicated that the culprit for low performance were particular type of operation occurring several layers below in the execution stack. In addition to that branch and bound identified slowest of requests can be explained by the same operation but for requests with session.

Second data set contains 1500 requests each having 140 features. It's latency profile indicates 3 distinct peaks and can be found in bottom row of Figure 5. While again we cannot disclose lower level details, we mention here that analysis results of two methods while being a little bit different have the same bottom line: there is single event, occurrence of which causes latency to fall into range of 2 peaks on the right, while additional event allows to distinguish between right and left peaks (its absence and presence, respectively). Note, that in this particular case the quicker fraction of requests is uniform and was described by single pattern (same pattern for both methods).

6.3 Interval Generation

There are multiple ways to form intervals. One simple approach is to greedily form all possible intervals. If every request r_i has distinct latency L_i, then we end up with $|\mathcal{R}|^2$ of intervals. The other extreme would be to generate only few good candidates. However, it is hard to know what is a "good" interval, and how to detect it automatically. We leave this direction open to further research. Instead, in our experiments we use local minima approach that generates potential *split points* by building latency histogram for the service and adding all local minima into the set. Set of intervals is formed by taking all pairs of split points. Set of split points is also used by dynamic programming.

7. RELATED WORK

Cloud and web services abound, for instance Amazon Web Services, Google and Bing. Nearly all providers of such services have dashboards tracking vital statistics of the systems for detecting anomalies, failures and also monitoring performance measures (such as QPS, etc). Latency of services is one of metric to track. There are several low-overhead techniques that help to monitor large scale services. Tools, such as X-trace [7, 6] and Dapper [20] allow RPC level data collection. Traces or RPC calls capture time and path of requests, and allow rich statistical analysis.

There is very little prior work on latency analysis in large scale systems. Path profiling [2], modeling [3], or call tree reconstruction [13] became a prominent directions in latency

analysis. Such systems are of great help in order to model services, building prediction models. In [19] authors develop approach to differentiate between anomalies and behavioral changes. However, they do not help to explain which attributes have played major role in some particular latency range. In contrast, our focus here is on explaining the latency variations and we take a combinatorial approach of identifying subset of monitored attributes that engineers can later troubleshoot.

There are many general statistical and data mining tools that apply to the problem of explaining an observed feature (*e.g.,* latency). One is based on correlation of features [9]. However, in our case we want to be able to detect synergy between different features, that is while separate features might not appear to be correlated with performance metric, together they can serve as a prefect explanation for the interval. Moreover, single set of features can be explaining only some interval of latencies, and not the whole range. Another approach uses decision trees [17]. There are also approaches that combine correlation and decision trees [4].

One can consider finding the explanation for a fixed interval to be *feature subset selection* [10] problem: select a subset of at most m features from set of features of size n, s.t,. the score is maximized. Here is conceptual difference between our problem, and feature selection problem. While in feature selection the goal is to select a set of features, s.t., some family of algorithms works well on data sets of some particular type. In our case we are interested in selecting set of features that optimizes the score on a *particular* data set. We are not concerned about over-fitting, or the fact that selected set of features is useless for analysis of the other data set.

Our approach is derived from f-measure. f-measure is frequently employed in machine learning, also in feature selection, to evaluate results [8]. f-measure is known to be non-convex, and hard to optimize. There are number of studies to optimize f-measure for family of methods, *e.g.,* [16, 22] We proposed a multidimensional f-measure which is even more challenging to compute. We believe this measure is of inherent interest and other uses for this measure will be found. Algorithms we adopted and evaluated for finding explanation for the fixed interval are classical branch-and-bound [11, 17], and greedy forward feature selection [5, 15]. For branch-and-bound we exploit properties of f-measure in order to significantly reduce the search space. The latter algorithm does not bound to find optimal solution, however can be used in practice when running time is crucial.

Along with feature selection *frequent itemset* and *association rule mining* are relevant. Their goal is to find subsets of features that frequently co-occur together. Algorithms such as a-priori [1] and it's modifications relay of support of generated sets and usually applied to complete data sets. Association rules appear to be a natural solution, in which we build one association rule for each interval of interest. However, in our case, feature set found for the interval can be also frequent outside of the interval. Or set of features can be infrequent on the scale of the data set, but be a perfect predictor for the interval of latency. Most importantly, association rules express causation relation, however in our case we do not want to restrict ourselves to this kind of relationship between features and latency attribute.

8. CONCLUSION

Cloud services are essential for modern online world, and users perceive latency as critical.Latency varies a lot due to variety of factors. There is very little prior work on latency analysis in large scale systems. We addressed the problem of understanding the latency profile and explaining it in terms of few, monitored attributes. We proposed a combinatorial quantity that we call the *multidimensional f-measure* and presented algorithms to use that to identify the suitable subset of attributes that explain latency. Understanding the latency profile of cloud services is a fundamentally important problem, and much remains to be done.

There is a growing trend in data mining that the output of analysis be meaningful to the end user. For example, in rank based data mining, there has been recent focus on picking rows and columns that explain the rank of the matrix rather than looking at linear combinations, because the rows and columns correspond to samples and features respectively which are easier to reason for end users [12]. Our work adds to this perspective. Also, the multidimensional f-measure, like the standard one-dimensional version, is difficult to handle computationally. It will be interesting to obtain approximations that are more efficient to compute. We also believe this measure will find other applications.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc 20th Int Conf Very Large Data Bases VLDB*, volume 1215, pages 487–499. Citeseer, 1994.

[2] T. Ball and J. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, 1996.

[3] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th conference on Hot Topics in Operating Systems-Volume 9*, pages 15–15. USENIX Association, 2003.

[4] A. Destrero, S. Mosci, C. De Mol, A. Verri, and F. Odone. Feature selection for high-dimensional data. *Computational management science*, 6(1):25–40, 2009.

[5] P. Devijver and J. Kittler. *Pattern recognition: A statistical approach*. Prentice/Hall International, 1982.

[6] R. Fonseca, M. Freedman, and G. Porter. Experiences with tracing causality in networked services. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 10–10. USENIX Association, 2010.

[7] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[8] G. Forman. An extensive empirical study of feature selection metrics for text classification. *The Journal of Machine Learning Research*, 3:1289–1305, 2003.

[9] I. Guyon. Practical feature selection: from correlation to causality. *Mining Massive Data Sets for Security: Advances in Data Mining, Search, Social Networks and Text Mining, and their Applications to Security*, pages 27–43, 2008.

[10] R. Kohavi and G. John. Wrappers for feature subset selection. *Artificial intelligence*, 97(1):273–324, 1997.

[11] A. Land and A. Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[12] M. Mahoney and P. Drineas. Cur matrix decompositions for improved data analysis. *Proceedings of the National Academy of Sciences*, 106(3):697–702, 2009.

[13] G. Mann, M. Sandler, D. Kruschevskaja, S. Guha, and E. Even-dar. Modeling the parallel execution of black-box services. *USENIX/HotCloud*, 2011.

[14] M. Marshak and H. Levy. Evaluating web user perceived latency using server side measurements. *Computer Communications*, 26:2003, 2003.

[15] A. Miller. *Subset selection in regression*. Chapman & Hall/CRC, 2002.

[16] D. Musicant, V. Kumar, A. Ozgur, et al. Optimizing f-measure with support vector machines. In *Proceedings of the Sixteenth International Florida Artificial Intelligence Research Society Conference*, pages 356–360, 2003.

[17] P. Narendra and K. Fukunaga. A branch and bound algorithm for feature subset selection. *Computers, IEEE Transactions on*, 100(9):917–922, 1977.

[18] K. Ostrowski, G. Mann, and M. Sandler. Diagnosing latency in multi-tier black-box services. 2011.

[19] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 4–4, Berkeley, CA, USA, 2011. USENIX Association.

[20] B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. *Google Research*, 2010.

[21] C. Van Rijsbergen. Information retrieval, 1979.

[22] N. Ye, K. Chai, W. Lee, and H. Chieu. Optimizing f-measures: A tale of two approaches. 2012.