Service Capacity Estimation Through Telemetry Analysis

Daniel De Freitas Adiwardana, Joe Wang

Capacity planning traditionally has been an imprecise process in which owners of large scale Internet services anecdotally make a guess as to the number of servers to procure and then increase that estimate by a factor of 10. Although overprovisioning services may satisfy user demand, it certainly is not the most cost efficient method by which to scale out. In this paper, we present a novel method, using machine learning, for estimating actual service capacity. As a crucial component to a larger service capacity model, this work should help produce a more exact (and therefore cost effective) solution to the problem of capacity planning.

Motivation

In the world of large scale services today, one of the critical investments a company must make is the purchase of hardware in the datacenter. Ideally, a service's capacity, or ability to handle user requests, should match its workload, or actual volume of user requests. A larger capacity than workload means wasted dollars powering unused servers, while a larger workload than capacity means slow response times and a poor experience for customers. From this observation, we hence derive two important aspects of a "capacity model" for any given service:

- *Workload Projection*, the ability to predict drivers of load on the system such as user initiated requests (or more generally user behavior), device or software initiated requests (such as downloading system updates), and so forth, and
- *Capacity Estimation*, the ability to estimate the maximum load a given service can handle, given its current and past performance characteristics.

In this paper, we present a unique method of addressing the second point.

Related Work

Before continuing further, it should be said that there currently does exist various strategies for estimating a service's capacity. In particular, the most common methodology for doing so is for engineers to write specific performance benchmarks to stress their own services [2] [3]. In doing so, the test would output a set of numbers correlating load (RPS or requests per second) with response time (latency) and processor utilization. These numbers would be able to fairly accurately predict the "maximum" workload it could handle.

Unfortunately, this methodology does not apply very well to the large scale, loosely coupled services backing the Internet properties found today [1]. First and foremost, it is very difficult to stress these services in isolation, as they often have several layers of upstream or downstream dependencies [4]; finding an upper bound on a particular service may not generate enough load to find the upper bound on downstream services, and conversely, a given "upper bound" on a given service may not be an actual upper bound, but rather the reflection of a bottleneck in a downstream service. Second, it is often very difficult if not impossible to replicate the entire set of service interdependencies just for testing—in fact, many stress tests actually run on production services to validate actual behavior an end user might see. That being said, engineers cannot allow these benchmarks to impact production performance, limiting their effectiveness. Finally, service code and topologies change fairly regularly, which makes writing specific tests rather time consuming and frankly inefficient. These are myriad other issues all show traditional performance testing to be cost prohibitive.

Rather than taking the traditional approach, we propose to utilize service QoS (quality of service) telemetry to build a machine learning model for predicting a service's estimated capacity.

Inputs and Target

To state our problem more formally, we wish to construct some hypothesis that takes a set of API features (some API identifier, the data size, etc) with an RPS and outputs a latency value. As input, these API characteristics can be derived solely from telemetry data—service logs events created when handling requests, while the RPS can be calculated as the count of all the requests processed during any given second.

Before diving into the data, however, there are some observations we can make to help choose the best route to take when developing our learning algorithm:

1. The latency logged by a particular API may be skewed upward if the server handling that API was under high load (due to requests to that API, or other available APIs).
2. At a very high levels, all APIs share common behavior (in the way requests are queued and handled), despite executing vastly different instructions.
3. The API identifier narrows down the set of possible behaviors an API may exhibit, but does not uniquely identify a single behavior.

At a first glance, it seems like we are working with a regression problem in which some of the inputs are continuous variables, while some are discrete categorical variables.

Datasets

In working on this project, we were collected two distinct datasets of sample production telemetry from an Internet scale property currently serving millions of users. The first dataset was collected early on and was used primarily to determine a baseline for the learning algorithms, while the second dataset was collected on a day with higher user traffic to provide a wider range of API behavior. The two datasets collected contain the following fields:

- *qoSId*. This is the unique identifier for a particular API being called on the service.
- *latencyMs*. This is the latency of this particular invocation of the API in milliseconds.
- *requestSizeBytes*. This is the size of the payload for this particular request in bytes.
- *protocolStatusCode*. This is a string identifier of the result (success/failure type) of the request.
- *rps*. This is the number of requests per second a particular server was handling for this API during this invocation of the API.

In the first dataset, up to 100 sample requests were collected per hour per API for 12 hours to be used as the training set; an additional hour of data was collected to be used as the test set. The second dataset contained 22 hours of training data and 2 hours of test data. Dataset sizes were 17 MB and 34 MB with training example counts 180,278 and 299,291 respectively.

A note on data quality.
An issue that we noticed early on was the general quality of the available data—because the collected telemetry events were the direct result of instrumentation by engineers, there were instances where fields had invalid values, or where certain fields seemed rather sparse. For instance, most of the *requestSizeBytes* did not have a value, and only some of the *protocolStatusCode* did. We realize that this will likely have an impact on how "good" of results we may be able to obtain, but acknowledge that this is just the state of the world in solving a challenging industry problem. We hope that as the data quality improves in the future, our learning models will become richer and more accurate.

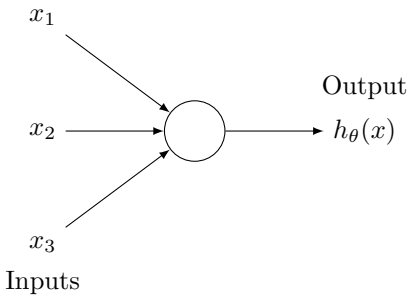
Methods

Linear Regression
Because our problem space required the computation of a model that could accurately predict a real valued latency given the input set of features, we naturally attempted to use a linear regression model for training our baseline. As inputs, we had both numerical features (*requestSizeBytes*, *rps*) as well as categorical features (*qoSId*, *protocolStatusCode*) for which the unique set of values were encoded as integers. A model output from the regression can be represented by a simple equation

$$y = \theta^T x + b$$

where x is a vector composed of the numerical features and encoded categorical features, θ is the weight vector assigned to the features, and b is the bias.

Neural Network
Neural networks are a class of models which can learn complex non-linear functions. Each neuron of the network takes a number of inputs x_i and outputs $h_\theta(x)$ where x is a feature vector with components x_i.



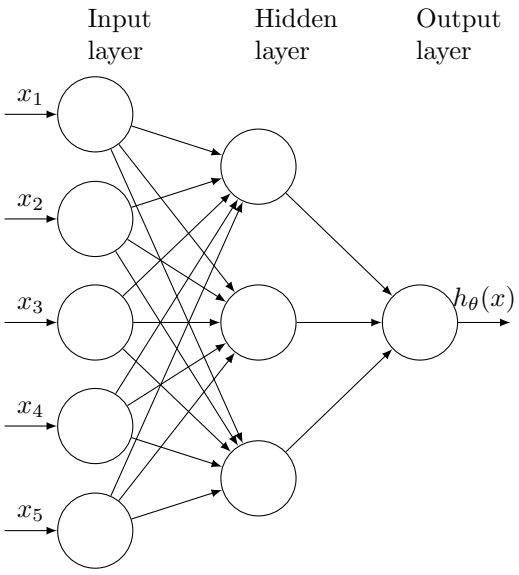
The figure above provides an illustration for three inputs, where $h_\theta(x)$ is a non-linear function of the linear combination of the inputs x_i, that is

$$h_\theta(x) = g\left(\sum_i w_i x_i\right)$$

where w_i are the neuron model parameters (a.k.a. weights) and g is some non-linear function called the activation function. A common choice for g is the sigmoid function

$$g(z) = \frac{1}{1 + e^{-z}}.$$

A neural network is a collection of such neurons. The neurons can be interconnected in various ways. One choice is the feed-forward architecture, in which neurons are organized into layers and the outputs of the neurons of one layer feed into the input of neurons of the next layer. The illustration below shows an example network with a single hidden layer containing 3 neurons.



The model parameters of a neural network are computed by optimizing the following cost function

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_\theta(x^{(i)}) - y^{(i)}\right\|^2\right)$$

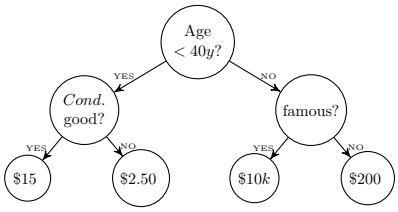
where $x^{(i)}$ and $y^{(i)}$ are the features and label of the i^{th} training example, respectively. Currently, the most commonly used algorithm for finding the parameters that minimize the given cost function is called backpropagation. The backpropagation algorithm randomly initializes the parameters and iterates over the following steps until convergence: (i) compute all the outputs of every neuron down to the output layer using the current parameter values, (ii) work backwards from the output layer towards the input layer to compute the gradient of the cost function J with respect to the parameters of each layer. This gradient can then be used to find new parameters values that minimize J for each layer in each iteration. For example, one could use gradient descent to update the layer parameters.

Boosted Decision Tree Regression

Decision tree regression is an algorithm that produces a regression search tree model. In this model, each node in the tree is a question about a given test point whose answer takes the search to exactly one of its child nodes. Eventually, if no data is missing in the test point, the search will reach a leaf. Each leaf in turn is associated with a possible predicted value for the test point. The tree is built starting with the complete training set from the root and then by picking a feature variable and question/split value in a way that it splits the training set to greedily minimize the sum of the squared errors of all leaves so far. The algorithm then proceeds recursively on the formed subtrees. Formally it finds a partition of the training set that greedily minimizes

$$J = \sum_{leaves}\sum_{i}(y^{(i)} - \mu_y)^2$$

where each $y^{(i)}$ is a training example associated to a particular leaf and μ_y is the mean of all such training examples. The following figure is an example of a fictitious binary regression decision tree for deciding the price of baseball card given its age, conditions and whether it features a famous player:



Boosting is a method to increase the accuracy of any learning algorithm by building an ensemble of "weak" models that together form one "strong" model. The present work utilizes the Gradient boosting method. The basic idea of the Gradient boosting algorithm is to, at each step, train a model, use a predefined loss function to measure its error, and correct for this error in the next model to be trained. So in a sense, each model of the obtained ensemble will have been trained in a way that complements the previous model's deficiencies by "focusing" on its errors.

Filter Feature Selection

Feature selection is one way of finding what features x_i of our dataset could be the most "relevant" to predict the target variable. Filter feature selection accomplishes this by first scoring each feature x_i according to some scoring function $S(i)$, which measures how informative each feature x_i is about the label y. The algorithm then outputs the k highest scoring features. In our work, we chose $S(i)$ to be the Mutual information between x_i and y, which can be defined as:

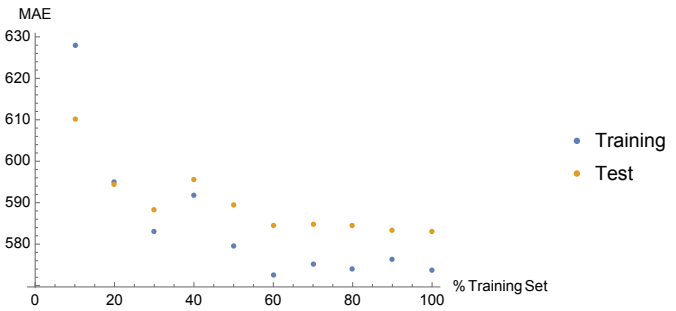
$$MI(x_i, y) = KL(p(x_i, y)||p(x_i)p(y))$$

where KL is the Kullback-Leibler divergence—a way of measuring the difference between the probability distributions $p(x_i, y)$ and $p(x_i)p(y)$. We picked Mutual information as the scoring function because it is capable of quantifying general relationships between variables, unlike say, Person's correlation, which is mostly used to quantify linear relationships.

Discussion

Baseline Linear Regression

Upon obtaining the first dataset, we set out to train our first model using batched linear regression, which constitutes our baseline. We also ran a diagnostic to determine our bias/variance, and whether our training set size was sufficient. The figure below shows the learning curve of our first run, where the error measure is the mean absolute error (MAE). We use MAE rather than root mean squared error (RMSE) because we care more that our predictions are mostly correct, rather than the magnitude of error for any particular prediction (i.e. we don't want large outliers to skew the error).



Original dataset baseline.

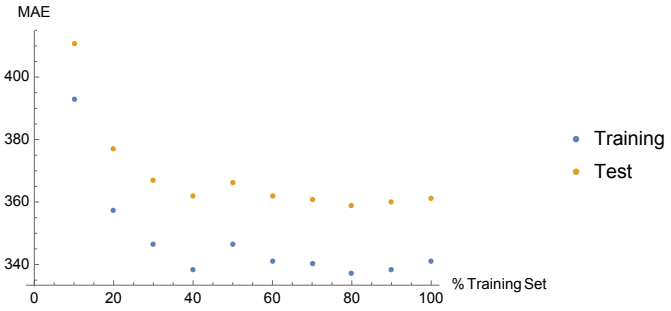
Unfortunate but not unexpected, we deemed the MAE of 584.44 to be too high. Armed with this information, we decided to (i) collect more data with better features and (ii) produce a more complex model.

Better Data

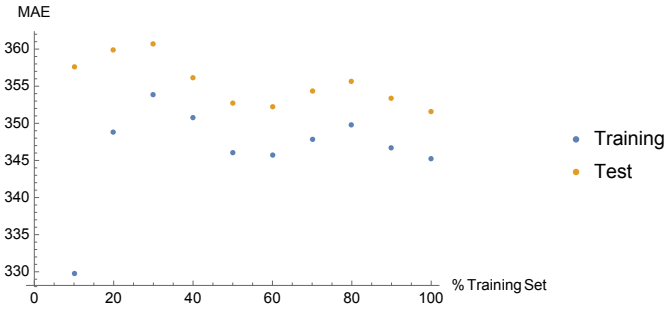
We knew going in that data quality was an issue, and so at this point, we decided to try to collect a more representative set for training. Our hypothesis revolved around the idea that by increasing RPS, a service's latency should increase; however, our dataset had very few instances of high traffic volume. Therefore, we decided to collect data from a day for which we suppose would produce a higher load (Black Friday). In this collection, we also decided to double the amount of training examples.

We also tried to add features and cleanup noise by including more metadata about each data point that we thought may be helpful (e.g. the service name, the average latency per API) and filtering out data points from non-production servers (e.g. testing and integration).

By doing these things, we were able to obtain the learning curves (for both datasets) below.



Original dataset, cleaned.



Black Friday dataset, cleaned.

We see that we were able to drop the test error significantly (to 355.65) just by obtaining a better dataset. However, we still weren't quite happy with these results. Given our results, we hypothesized that the model is most likely underfitting the training data, and so we needed models capable of better representing the intricacies of the underlying target function.

More Complex Models

We first tried a neural network because of its potential for learning complex non-linear models. Since we didn't have many features, we picked a neural network architecture comprised of a single hidden layer with a number of neurons in the same order of magnitude as our number of features as a heuristic to avoid overfitting our training set. So in our first experiment we tried 1 hidden layer with 10 neurons, 0.005 learning rate and 100 epochs. From this we obtained a training MAE of 515.33 and a test MAE of 503.33.

Since our training error was still high we attempted increasing the number of neurons tenfold to increase the learning capacity of the model. This actually increased the training MAE to 548.40, perhaps because the minimization had not yet converged. We attempted to diagnose the convergence of the model by modifying the number of iterations, which showed that it wasn't converging.

# iterations	MAE
25	668.83
50	661.66
100	548.10
125	571.33
150	610.47

We didn't invest much further into this approach given the long training time requirements of the neural network. These results indicated that we might actually not be underfitting an existing pattern, but that perhaps the pattern is mixed with a lot of noise with respect to our current feature set and/or data quality.

In light of this realization, we then applied a feature selection algorithm to select a subset of the 3 most relevant features out the 4 we had; this could reduce potentially noisy data. The feature selection algorithm we used was Filter based selection utilizing Mutual information as the scoring function. The following table shows the resulting feature relevance scores:

qoSId	0.82
requestSizeBytes	0.074
protocolStatusCode	0.051
rps	0.040

The feature selection scores point to the removal of the *rps* feature. However, given our domain knowledge, we believe that this feature is actually relevant, but the data we currently have is mostly for very low *rps* values. So in preparation for potentially higher *rps* values in future data we chose to preserve it. The second least relevant feature is *protocolStatusCode*. After removing this feature our linear regression model scored 336.47 test MAE.

Also, because *qoSId* seemed to be an important feature by which to slice the data, it was possible that the patterns inside each stratum defined by a unique *qosId* would be less noisy. So we tried an algorithm that could leverage this fact. The Decision tree regression algorithm could be expected to be able to start by splitting the data according to the various *qoSId* values. Concretely, given that the *qoSId*s appear to be very informative of the label values, then the *qoSId* values to branch on the lower nodes of the tree may also minimize the total squared error of the resulting set of leaves, which is the goal of the algorithm. On the other hand, decision trees are notorious for overfitting the training set. So we used a Boosted decision tree solution which is meant to generalize better, but preserve the decision tree's capability of learning non-linear functions.

As a starting point, we picked the following learning parameters: maximum number leaves per tree equal to 20, learning rate 0.2 and total number of trees in the boosting ensemble to be 100. It scored a 308.81 test MAE. So it already performs slightly better than our baseline. This algorithm trained a model fast enough that it allowed us to perform an automatic random sweep over the learning parameter space and measure it's training error. The best scoring model contained just 8 leaves, allowed a minimum of 1 instances per leaf, used a learning rate of 0.05 and 500 trees in the ensemble. It marginally performed better than the initial parameter settings producing a 304.83 test MAE.

Error Analysis
At this point, we weren't sure whether additional runs of the algorithms above would net us a better result, so we decided to dig a bit deeper and understand how our model performed with respect to each individual API. By aggregating by *qoSId* and averaging the training and test error, we discovered that we could logically group APIs with similar error. In particular, the following categories had the largest error:

- Security (encryption, decryption)
- Database writes
- File downloads
- Server timeout errors
- Poorly instrumented telemetry

while the following categories had the lowest error:

- Database lookups
- Frontdoor request handlers
- Service health endpoints

This made it very apparent that our original assumption about most APIs behaving in a similar manner to be false. The APIs that were bound to some factor (like CPU or disk) stood out in particular; these were requests that were simply more computationally expensive. We noticed that, in performing this qualitative analysis, it appeared that there may be multiple clusters of APIs that shared behavior, which our current single algorithm models wouldn't be able to fit very well.

Composed Models
The logical next steps here would have been to perform clustering of the data by *qoSId* to obtain features that could then be used as part of the regression. However, we didn't have enough time to run the needed experiments. As a first shot, we used domain knowledge to partition the dataset by the average latency of each *qoSId* into three categories: APIs with low (< 25 ms), medium (≥ 25 ms and < 2000 ms), and high (≥ 2000 ms) latency. We then trained each set individually in hopes of a better overall error. Unfortunately, this did not turn out as we would have liked—each set had errors that were just as poor as our original attempts.

Conclusion and Future Work

In tackling this particular problem of capacity estimation, we arrived at 2 findings:

1. The dataset we used may not have been representative of actual API behavior (the *rps* values were skewed too low, many of the features were sparse due to poor instrumentation).
2. A rich model that encompasses all the data needed to perform service capacity estimation requires a mixture of learning algorithms to produce desirable results.

The next steps, therefore, are to collect better data and try additional methods to tease out the behaviors of each API. In terms of data collection, we could apply a better sampling algorithm to obtain telemetry that better reflect the distribution of incoming requests to the service. For instance, we could first analyze the distribution of latencies on each API, and collect samples based on that distribution.

As for additional training, we have several ideas in mind:

- *Clustering.* We could try the idea of clustering APIs and using the clusters as features, like we described in the Composed Models section above.
- *Classification.* We could also try turning the problem into a classification one by partitioning the output latencies into some number of buckets. Despite the final model not being as precise, it could still produce useful data for capacity estimation.

There remains much more work to be done on this project, but if we find a successful solution, it has the potential to save millions in server costs.

References

[1] P. Brebner. Service-oriented performance modeling the mule enterprise service bus (esb) loan broker application. In *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, pages 404–411, Aug 2009.

[2] J. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. *Performance Testing Guidance for Web Applications: Patterns & Practices*. Microsoft Press, Redmond, WA, USA, 2007.

[3] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 1st edition, 2009.

[4] Newman Sam. *Building Microservices*. OReilly Media, 2015.