Predicting User Dissatisfaction with Internet Application Performance at End-Hosts

Diana Joumblatt*, Jaideep Chandrashekar†, Branislav Kveton†, Nina Taft†and Renata Teixeira*
*CNRS and UPMC Sorbonne Universites
†Technicolor Research

***Abstract*—We design predictors of user dissatisfaction with the performance of applications that use networking. Our approach combines user-level feedback with low level machine and networking metrics. The main challenges of predicting user dissatisfaction, that arises when networking conditions adversely affect applications, comes from the scarcity of user feedback and the fact that poor performance episodes are rare. We develop a methodology to handle these challenges. Our method processes low level data via quantization and feature selection steps. We combine this with user labels and employ supervised learning techniques to build predictors. Using data from 19 personal machines, we show how to build training sets and demonstrate that non-linear SVMs achieve higher true positive rates (around 0.9) than predictors based on linear models. Finally we quantify the benefits of building per-application predictors as compared to general predictors that use data from multiple applications simultaneously to anticipate user dissatisfaction.**

I. Introduction

Users today expect high quality performance of computers that run demanding applications such as VoIP and video streaming. If end-host tools could anticipate when a user might be dissatisfied with the performance of an application, they could proactively troubleshoot performance problems thereby improving the user's experience.

Previous research at tracking and improving the end-user experience has followed two different directions. In the first, network performance degradations are determined by using heuristics over raw network performance metrics such as delay, available bandwidth, losses, reordering, and such [1], [2]. The obvious shortcoming here is that the actual user response and perception is not taken into account, and the heuristics may not really track these well. To illustrate with an example, most modern video streaming applications incorporate enough of a buffer to tie over transient network outages. Tracking the performance metrics would reflect the transient outages, even though the end-user might be completely oblivious to them. In the second category, the research explicitly engages the end user but is focused on a few niche applications such as VoIP [3], [4], online gaming [5], video streaming [6], [7], and IPTV [8], [9]. While these are able to exploit application level semantics to build richer models, the results cannot be generalized to other applications. Building such application specific models is time consuming and requires a deep knowledge of the application (this information is often not available publicly). Despite recent interest in explicitly leveraging user feedback to achieve more general mappings of QoS to QoE [10]–[12], our community still doesn't have techniques that remain application agnostic and can predict user satisfaction given typical network-level metrics.

In this paper, we analyze passive measurements of system and network performance, which are annotated with a limited set of samples of end-user perception of network performance. Our user labeled performance data was collected from a deployment of 19 users who agreed to run our monitoring software on their laptops for periods varying from two weeks to six months. The basic problem we face is to design a binary classifier (satisfied or not) based on a key set of features that are culled from detailed low-level measurements. Having labeled data (user feedback) allows us to consider supervised learning techniques. However there are numerous challenges in building a training set for such techniques, that are likely to arise in many QoS to QoE mapping problems. First, users don't want to spend much time on giving feedback and thus provide at most a few samples per day (e.g. less than 5). Clearly the volume of user input is many orders of magnitude less than the data volume produced by monitoring tools that collect performance data on sub-second time scales. Second, because poor performance episodes are rare, we typically end up with far more samples labeled as satisfied than dissatisfied.

Our first contribution is a methodology for dealing with these issues in order to enable supervised learning techniques to be applied to this type of problem. Our second contribution is to provide a solution for aggregating fine-grained machine data into a few meaningful features that are correlated with user perception. One of the key ideas in our solution is to compute our selected features over two different windows of time, and use those simultaneously as input to our predictor. We propose LDA and SVM based predictors, and show that non-linear SVM outperforms other simpler linear predictors. Our third contribution focuses on the issue of whether one should build one predictor for each application, or one predictor for multiple applications. The advantage of the former case is a better predictor, but the disadvantage is that it requires building many predictors. We select specific applications as use cases and quantify the performance of both specific and general predictors.

II. HostView measurements

The data used in this paper was collected using a tool we built, called HostView, that runs directly on end-user laptops. We conducted a user study with 19 users, mostly computer scientists, who kindly agreed to run our tool for variable amounts of time ranging from 2 weeks to 6 months (starting in November of 2011). For a more detailed description of the data collection methodology, please refer to our earlier work [13].

Our study is based upon data from 19 users, a typical number for user studies. We acknowledge that this is a small set of users. However building models for these users raises a number of issues that are likely to come up in a large population, and thus we believe that the problems we face in our task here are typical and require general solutions. We also point out that because monitoring tools such as HostView get so close to the user, it is hard to do large scale studies as many users perceive the tool as a privacy threat.

Performance metrics. To capture the *actual* performance that applications experience, HostView collects packet traces with libpcap; the traces are then processed offline using `tcptrace` [14] to extract detailed TCP connection metrics (RTT, jitter, resets and retransmissions) and data rates for TCP and UDP connections. Concurrently, we periodically poll for CPU load and WiFi signal strength from the host OS. Table I lists all the performance metrics extracted along with the manner in which the metrics are obtained.

Application context. HostView also records the application context for each connection, i.e., it identifies the process executable associated with the connection, and the service being used (for web traffic, this is the top level domain). The process executable is obtained using the `gt` tool [15], and the top level domain is extracted by parsing DNS replies in the traces and associating them with the destination addresses.

Recording User (Dis)satisfaction. HostView collects user feedback with two complementary mechanisms: a system-triggered questionnaire based on the Experience Sampling Methodology [16], and the "I'm annoyed!" button (which is engaged by clicking an icon that is always present, but unobtrusive, in a corner of the screen). In both mechanisms, the user is presented with the same questionnaire: (i) rate your internet speed from 1 (slow) - 5 (very fast), (ii) identify any applications (from a list of those running) that they are unhappy with, (iii) indicate the problem (from a set of choices), and (iv) express any other additional information via a freeform text box.

Based on the answers to these questions, we categorize each user feedback sample into one of two classes: the user is considered *dissatisfied* if she notes an application as problematic, or if she explicitly selects a problem among the pre-defined list of problems, or if she rates the Internet speed below three. Otherwise, the user is considered *satisfied* at the moment the feedback is given. All of the measurements from the lower layers collected in a small time window (discussed later) before the user supplies a completed questionnaire are labeled as satisfied or dissatisfied according to the user's input.

In the data from our user study, we received 1278 surveys indicating a satisfied user, and 422 indicating a dissatisfied user. About 55% of the times when users supplied feedback, they did not explicitly indicate any particular application being used. Table II enumerates all the feedback reports obtained from users, along the the specific "applications" that were identified as problematic by the end-users; the frequencies are also reported. Note here that some of the user labels point to *online services* (e.g. `youtube`, `tf1.fr`), while others refer to application executables on the host machine.

Applications	**Dissatisfied**	**Satisfied**	**Users**
Firefox	51	384	8
Mail	23	332	4
Google Chrome	22	104	3
Safari	21	46	7
Skype	19	260	5
SSH	12	227	7
Adium	8	385	3
YouTube	7	18	3
Totem	6	10	1
tf1.fr	5	0	1
GTalk, iCal, iTerm, git	8	23	4
Thunderbird, Dropbox	2	317	2

Table II
SUMMARY OF USER LABELS PER APPLICATION

III. OVERVIEW

A. Prediction problem formulation and evaluation metrics

Our end goal is to build a predictor that can anticipate whether or not a user is satisfied with the performance of their machine, as a function of the state of the machine at the time. We take "state" to be captured by the metrics collected (see Table I). Before realizing such a predictor, there are two challenges to address. First, we need to "summarize" the raw data and convert it to a form that can be used by a learning algorithm, typically a vector of features. If we suppose that a user feedback sample was obtained at time t, we denote $\mathbb{X}_t$ as the feature vector that is generated by the raw data logged in time interval $[t-\delta, t]$. Second, we need to find the prediction function f which has the following property.

$$f(\mathbb{X}_t) = \begin{cases} 1 & \text{if user is } \textbf{dissatisfied} \text{ at } t \\ 0 & \text{if user is } \textbf{satisfied} \text{ at } t \end{cases}$$

Since we have data samples that are associated with labels, and considering that the predictor function f returns a binary value, we look at *supervised* machine learning methods to build binary classifiers. For every data sample, the predictor returns a binary value, and there are four possibilities to consider and these are enumerated below, along with the common notations for each.

		Output of $f(\mathbb{X}_t)$	
		0	1
User label at time t	Satisfied	True Negative (TN)	False Positive (FP)
	Dissatisfied	False Negative (FN)	True Positive (TP)

The performance of the predictors is characterized by True and False Positive Rates which are defined as follows.

$$TPR = \frac{TP}{(TP+FN)}, FPR = \frac{FP}{(FP+TN)}$$

A low TPR indicates that the predictor will miss poor performance episodes and be ineffective, whereas a high false positive rate indicates that the system will often, and incorrectly, trigger troubleshooting mechanisms on the host

	Metrics	Method	Freq.	Comments
Network	RTT	pcap+tcptrace	continuous	average per second
	Jitter	pcap+tcptrace	continuous	Difference between two consecutive RTTs of a connection, average per second
	Resets	pcap+tcptrace	continuous	Only resets that follow a SYN segment
	Retransmissions	pcap+tcptrace	continuous	-
	Per-connection data rates (up and down)	pcap+tcptrace	continuous	TCP and UDP
	Host data rates (up and down)	pcap+tcptrace	continuous	All TCP and UDP connections
Machine	CPU	top	30 secs	-
	SNR	airport (Mac OS) / iwconfig (Linux)	60 secs	$SNR_{dB} = RSSI_{dB} - Noise_{dB}$

Table I
PERFORMANCE METRICS

and is thus inefficient and leads to very high overhead. Thus, we are looking for predictors with high TPR and low FPR.

IV. HOW TO BUILD TRAINING DATA

We now describe how we process the raw traces to obtain post processed data that can be input to a learning algorithm.

Extracting Feature Vectors. Processing data to train a classifier commonly involves three stages: (i) quantization, (ii) feature selection, and (iii) labeling.

Quantization: The first step is to select a time window (or bins) of interest around the time of the user feedback. Small windows will localize the problem, but not track problems that the user does not immediately perceive. Larger windows could “smooth” out any transients that affect the user. We use δ to denote the bin size, and consider sizes of 1, 2 and 5 minutes. At the same time, it might be a good idea to incorporate history since the particular problem that is frustrating the user at time t might be the accumulation (or end result) of performance anomalies that manifested in the past. To this end, we also define a longer window of time (going back into the past) denoted Δ. Our intuition is that by incorporating state from both δ and Δ, it might be easier to find a portion of the feature space where user dissatisfaction is apparent.

Feature selection: Having selected the right time granularity, we need to summarize the raw data into features that represent each bin. We select a number of features, based on our previous knowledge, that are very likely to affect the network performance of applications. We consider the metrics listed in Table I and process as follows: for RTT, jitter, CPU load and SNR, we compute four descriptors of the distribution (computed with data observed only inside the window), which are the mean, median, std. dev., and 95%-ile. For resets and retransmissions, we simply report the total seen in the bin. For metrics related to data rates, we compute both outgoing and incoming and add it to the vector. Thus, in total, for each bin $[t-\delta, t]$, we generate vector $\mathbb{X}_t$ containing 20+20 features (corresponding to the small window and the history duration).

Another important step in feature extraction deals with the granularity which is tracked (should we consider applications individually, or aggregate them). One possibility is to compute the RTT, jitter, throughput statistics using all connections from all applications. Alternatively we could separate the traffic for each application and generate these features at the granularity of application. Since it is difficult to know which method is best ahead of time, we do both and extract two different sets: in the first, which we call, *machine-level feature extraction*, we aggregate all the traffic and do not break out that of any applications (here, each vector contains 40 features). In the second, *application-level feature extraction*, we partition the traffic by application and then extract features. Here, the RTT and jitter measurements reflect only the performance of the particular application. Note that we introduce two new features in this case, corresponding to *application* throughput (in each direction). Thus, the feature vectors we generate with application level feature extraction have 44 features.

Labeling Feature Vectors: The final step is to assign a label to each feature vector $\mathbb{X}_t$ by the label obtained from the end-user when the feedback was collected. Thus, each vector $\mathbb{X}_t$ is associated with a label that is either *satisfied* or *dissatisfied*.

Dealing with unbalanced data. Our data is severely unbalanced because we have an order of magnitude more "satisfied" reports than "dissatisfied" reports. If not dealt with, the resulting predictors will be biased and more likely to predict the windows when users are satisified (which corresponds to a low TPR). In order to deal with this, we employ a trick that is commonly used in the literature for situations like this [17]. In the training phase, we randomly repeat data samples for the dissatisfied reports until the two classes are more or less even. This rebalancing is only done in the training phase when building the detector. When testing, the duplicated instances are removed and only the original data is used.

Per-application case studies. To build per-application predictors, we select the five applications with the most user feedback in Tab. II (excluding web browsers): Mail, Skype, SSH, YouTube, and Adium. We exclude web browsers because they are used to carry out very different tasks, and interact with a diverse set of services. Hence, we only consider the user reports for browsers when the user has explicitly indicated an online service as problematic (for example, in the case of YouTube). In these cases, we collect and associate the browsers traffic to the service in question; however, we use the online service as the name of the application.

V. MULTI-FEATURE PREDICTOR

In this section, we compare the performance of three fully supervised machine learning techniques: Linear Discriminant

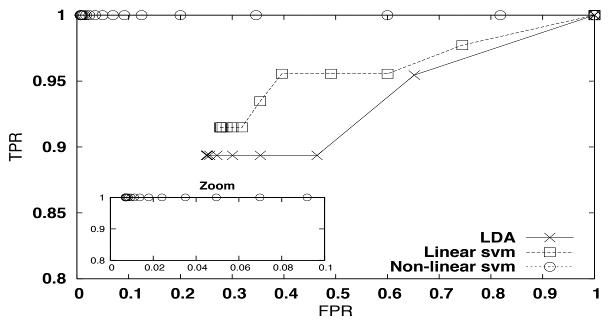

Figure 1. ROC CURVE FOR THE GENERAL PREDICTOR (δ=5, Δ=60)

Analysis (LDA) [18], Linear Support Vector Machines (l-svm) [19] and Non-Linear Support Vector Machines (nl-svm) [19]. We find that, among the three, nl-svm produces optimal results with our data set. We construct three *families of predictors*, one based on machine-level features, and two based on application-level features and compare these while varying the parameters (δ, Δ). In the best case, $(\delta = 5, \Delta = 60)$, we find that the nl-svm based predictor that was built with application-level features outperforms the other two families.

A. Comparison of predictors

We construct three families of predictors using LDA, l-svm, and nl-svm. *Machine* predictors are constructed by training on machine-level features across all machines, and the testing is carried out for similar data. *General* predictors are built by training on application-level features, but with data that is the union of all the applications (and all the hosts). The predictor is tested on data from particular applications. Finally, *application specific* predictors are built by training on data that is specific to that application (from all the hosts), and performance is tested on data of that particular application. Note that the first family of predictor works with vectors of 40 features; the other two use 44 features. Note from Sec. II that traffic from browsers is excluded except in select cases.

We experimented with various settings for δ, Δ and selected the optimal settings using leave-one-out cross validation. In figure 1, we show a representative RoC curve where we hold the window parameters fixed $(\delta = 5, \Delta = 60)$. The best performing predictor is found at the top-left of the graph (high TPR and low FPR). Unsurprisingly the plot shows that the nl-svm based predictor is consistently above and to the left of the curves for l-svm and LDA, indicating that overall nl-SVM outperforms the other two methods (the take-away was identical when we generated curves varying δ and Δ). The l-svm predictor achieves a respectable TPR of roughly 91%, however it does so with a FPR approaching 30%. This RoC illustrates that by using a non-linear SVM instead of a linear one, we can significantly lower the false positive rate.

B. Optimal Predictor Parameters

In order to determine the ideal value for δ, we compare three different settings $\delta = 1, 2, 5$ using the nl-svm predictor

	15 minutes		30 minutes		60 minutes	
Predictor	**TPR**	**FPR**	**TPR**	**FPR**	**TPR**	**FPR**
Mail	0.89	0.04	0.95	0.03	1	0.03
Skype	1	0.09	1	0.06	1	0.06
SSH	1	0.01	1	0.01	1	0.01
YouTube	1	0	1	0	1	0
Adium	1	0	1	0	1	0
Machine	0.60	0.13	0.60	0.11	0.62	0.11

Table III
AFFECT OF VARYING Δ, FOR NON-LINEAR SVM WITH HISTORY, AND FIXED δ=5

(the most powerful one). We find that a window of 5 minutes returns the best performance.

Subsequently, with the ideal window size, we then look at the effect of varying history lengths (i.e, value of Δ) upon performance. We select δ=5 and compare performance for the three predictor families varying Δ=15, 30 or 60 minutes. Table III reports these results and it is very clear incorporating history improves both TPR and FPR across the board. We also answer the "how much" question. In Table III $\Delta = 60$ brings the greatest benefit. This is intuitive considering that when $\delta = 5$ minutes, $\Delta = 60$ implies adding 11 additional little bins. In previous experiments, we also considered $\Delta = 0$ (no history is used) and this performed quite poorly. Interestingly, the *machine* predictor does not improve even as history is taken into account. These results imply that distinguishing applications is important in predicting user dissatisfaction.

C. Application predictor vs. general predictor

We now compare the general performance of three different predictors built with application-level features. Here we are trying to understand if the predictors that see more samples of application behavior, albeit from many different heterogenous applications have an edge over those that only see homogenous data (from a single application). We build *application specific* predictors for the five applications indicated in Table IV, and compare them with the other two *general* predictors. In the first of these, denoted "general", the training data mixes feature vectors from *all* the applications; in the second, denoted "general, no Mail" (the training data incorporates data from all other apps save for Mail). In both cases, the testing is done against the application-level feature vectors corresponding to Mail. To read the table, consider that the training data for the predictor is as denoted in the "predictor" column, and the testing data is as denoted in the "tested app" column.

We see in the table that for 3 applications, namely Mail, SSH and Adium, the per-application predictor outperforms a general predictor. In these cases, the improvement of a per-application predictor over a general one is significant in terms of TPR. Interestingly, for YouTube and Skype, the per-application and general predictors perform quite similarly. They both achieve full true positive detection (TPR=1) and only differ slightly in the FPR. In general, we would advocate for building per application predictors, especially frequently used ones, if there is enough data. Although the general ones do very well in some cases, the per-application predictors are more consistently good across multiple applications.

Tested App	Predictor	TPR	FPR
Mail	**Mail**	1	0.03
	general	0.84	0.06
	general, no Mail	0.05	0.03
Skype	**Skype**	1	0.06
	general	1	0.03
	general, no Skype	0.07	0.01
SSH	**SSH**	1	0.01
	general	0.75	0.27
	general, no SSH	0	0.05
YouTube	**YouTube**	1	0
	general	1	0.06
	general, no YouTube	0	0
Adium	**Adium**	1	0
	general	0.5	0.005
	general, no Adium	0	0.1

Table IV
APPLICATION PREDICTORS VS. GENERAL PREDICTOR (NON-LINEAR SVM, δ= 5, Δ=60)

VI. RELATED WORK

The general area of network performance monitoring and QoS has been extremely prolific. Similarly, the area of user quality and QoE is fairly mature (for example, ITU-T's standards for voice quality measurement in the telephone system are over a decade old [20]). QoS studies focus only on raw performance metrics, whereas QoE studies mostly map user experience to application metrics (e.g, channel zapping time in IPTV). In this paper, we aim to bridge the gap between these two fields and map QoS metrics to QoE.

A number of prior studies have built models of QoS to QoE by using application-level metrics [3]–[5], [7]–[9]. These models, however, won't apply for other applications. We instead take an application-agnostic approach to better capture the user online experience. Most related to our work are a number of efforts to collect and characterize network and system performance data annotated with user feedback from laptops and desktops [10]–[13], [21] as well as smartphones [22]. HostView [13], which provided the dataset used in this paper, is one example of these measurement efforts. These efforts have led to interesting preliminary insights on the relationship between QoS and QoE, but not to general models or predictors of QoE based on QoS parameters as we do in this paper. The only exception is OneClick [23], which uses a Poisson regression model to correlate network metrics of one application with the rate that users click on a button to indicate dissatisfaction with this specific application. The model is based on asking users to input feedback while watching or listening pre-recorded traces with different loss rates. In contrast, our solution can flag user dissatisfaction under real network conditions.

VII. DISCUSSION AND FUTURE WORK

We discuss here some open issues that must be addressed for our predictor to be integrated into an online diagnosis system.

Implementation of online prediction and diagnosis. This paper relied on the traffic traces processed offline to extract connection metrics and generate the feature vector $\mathbb{X}_t$ per application. In an actual system implementation, we must generate $\mathbb{X}_t$ online, predict user dissatisfaction at every time bin, and launch a diagnostic tool when a time bin is labeled as dissatisfied. The design and implementation of a system that performs all these tasks in real-time without overloading the user's machine in terms of CPU or storage represents a challenge that we plan to address in our future work.

Configuration of online predictor. The online predictor has to be configured with the models we obtain during our training phase. In the ideal scenario, we would train the predictor once and apply the same model to all users. For that, we need to conduct further research to better understand the generality of our predictors. In our future work, we plan to explore other related questions. Does a predictor learned from one user apply to another? How often do we need to re-train predictors?

Acknowledgements. This work was supported by the European Community's Seventh Framework Programme (FP7/2007-2013) grant no. 258378 (FIGARO) and the Agence National de la Recherche grant C'MON.

REFERENCES

[1] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang, "PlanetSeer: Internet Path Failure Monitoring and Characterization in Wide-area Services," in *Proc. USENIX OSDI*, 2004.

[2] J. Sommers, P. Barford, N. Duffield, and A. Ron, "Accurate and Efficient SLA Compliance Monitoring," in *Proc. ACM SIGCOMM*, 2007.

[3] W. Jiang and H. Schulzrinne, "Modeling of packet loss and delay and their effect on real-time multimedia service quality," in *NOSSDAV*, 2000.

[4] K. Chen, C. Huang, P. Huang, and C.-L. Lei, "Quantifying skype user satisfaction," in *Proc. ACM SIGCOMM*, 2006.

[5] K.-T. Chen, P. Huang, and C.-L. Lei, "How sensitive are online gamers to network quality?," *Commun. ACM*, vol. 49, no. 11, pp. 34–38, 2006.

[6] S. Tao, J. Apostolopoulos, and R. Guérin, "Real-time monitoring of video quality in IP networks," in *Proc. ACM NOSSDAV Network and Operating System Support for Digital Audio and Video*, 2008.

[7] R. K. Mok, E. W. Chan, X. Luo, and R. K. Chang, "Inferring the QoE of HTTP video streaming from user-viewing activities," in *Proc. SIGCOMM WMUST*, 2011.

[8] A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao, "Towards Automated Performance Diagnosis in a Large IPTV Network," in *Proc. ACM SIGCOMM*, 2009.

[9] H. H. Song, Z. Ge, A. Mahimkar, J. Wang, J. Yates, Y. Zhang, A. Basso, and M. Chen, "Q-score: proactive service quality assessment in a large iptv system," in *Proc. Internet Measurement Conference*, 2011.

[10] J. S. Miller, A. Mondal, R. Potharaju, P. A. Dinda, and A. Kuzmanovic, "Understanding end-user perception of network problems," in *Proc. SIGCOMM WMUST*, 2011.

[11] R. Schatz and S. Egger, "Vienna surfing: assessing mobile broadband quality in the field," in *Proc. SIGCOMM WMUST*, 2011.

[12] D. Joumblatt, R. Teixeira, J. Chandrashekar, and N. Taft, "Performance of Networked Applications: The Challenges in Capturing the User's Perception," in *Proc. SIGCOMM WMUST*, 2011.

[13] D. Joumblatt, R. Teixeira, J. Chandrashekar, and N. Taft, "HostView: Annotating End-Host Performance Measurements with User Feedback," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 3, pp. 43–48, 2011.

[14] "TcpTrace." http://www.tcptrace.org.

[15] F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso, and K. Claffy, "Gt: picking up the truth from the ground for internet traffic," in *ACM SIGCOMM Computer Communication Review*, 2009.

[16] S. Consolvo and M. Walker, "Using the Experience Sampling Method to Evaluate Ubicomp Applications," *IEEE Pervasive Computing Magazine*, vol. 2, no. 2, 2003.

[17] R. Duda, P. Hart, and D. Stork, *Pattern Classification*. Wiley, 2000.

[18] R. Duda and P. Hart, *Pattern Classification and Scene Analysis*. Wiley, 1973.

[19] V. Vapnik, *The Nature of Statistical Learning Theory*. New York, NY: Springer-Verlag, 1995.

[20] "Itu-t recommendation p.862. perceptual evaluation of speech quality (pesq): An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs," tech. rep., 2001.

[21] T. Karagiannis, C. Gkantsidis, P. Key, E. Athanasopoulos, and E. Raftopoulos, "Homemaestro: Distributed monitoring and diagnosis of performance anomalies in home networks," Tech. Rep. MSR-TR-2008-161, Microsoft Research, 2008.

[22] K. W, S. Ickin, J. Hong, L. Janowski, M. F., and A. Dey, "Studying the experience of mobile applications used in different contexts of daily life," in *Proc. SIGCOMM WMUST*, 2011.

[23] K.-T. Chen, C. C. Tu, and W.-C. Xiao, "Oneclick: A framework for measuring network quality of experience," in *Proc. IEEE INFOCOM*, 2009.