A Modular Machine Learning System for Flow-Level Traffic Classification in Large Networks

YU JIN, University of Minnesota
NICK DUFFIELD, JEFFREY ERMAN, PATRICK HAFFNER, and SUBHABRATA SEN, AT&T Labs – Research
ZHI-LI ZHANG, University of Minnesota

The ability to accurately and scalably classify network traffic is of critical importance to a wide range of management tasks of large networks, such as tier-1 ISP networks and global enterprise networks. Guided by the practical constraints and requirements of traffic classification in large networks, in this article, we explore the design of an accurate and scalable machine learning based flow-level traffic classification system, which is trained on a dataset of flow-level data that has been annotated with application protocol labels by a packet-level classifier. Our system employs a lightweight *modular architecture*, which combines a series of simple linear binary classifiers, each of which can be efficiently implemented and trained on vast amounts of flow data in parallel, and embraces three key innovative mechanisms, *weighted threshold sampling*, *logistic calibration*, and *intelligent data partitioning*, to achieve scalability while attaining high accuracy. Evaluations using real traffic data from multiple locations in a large ISP show that our system accurately reproduces the labels of the packet level classifier when runs on (unlabeled) flow records, while meeting the scalability and stability requirements of large ISP networks. Using training and test datasets that are two months apart and collected from two different locations, the flow error rates are only 3% for TCP flows and 0.4% for UDP flows. We further show that such error rates can be reduced by combining the information of spatial distributions of flows, or *collective traffic statistics*, during classification. We propose a novel two-step model, which seamlessly integrates these collective traffic statistics into the existing traffic classification system. Experimental results display performance improvement on all traffic classes and an overall error rate reduction by 15%. In addition to a high accuracy, at runtime, our implementation easily scales to classify traffic on 10Gbps links.

Categories and Subject Descriptors: C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*; I.5.2 [**Pattern Recognition**]: Design Methodology—*Classifier design and evaluation*

General Terms: Algorithms, Measurement, Theory

Additional Key Words and Phrases: Communications network, traffic classification, machine learning

ACM Reference Format:
Jin, Y., Duffield, N., Erman, J., Haffner, P., Sen, S., and Zhang, Z.-L. 2012. A modular machine learning system for flow-level traffic classification in large networks. ACM Trans. Knowl. Discov. Data 6, 1, Article 4 (March 2012), 34 pages.
DOI = 10.1145/2133360.2133364 http://doi.acm.org/10.1145/2133360.2133364

The work of Z.-L. Zhang was supported in part by the NSF grants CNS-0905037 and CNS-1017647, and an AT&T VURI gift grant.
Authors' addresses: Y. Jin, N. Duffield, J. Erman, P. Haffner, and S. Sen, AT&T Labs – Research, AT&T Shannon Laboratory (Building 103), 180 Park Avenue, Florham Park, NJ 07932; email: {yjin, duffield, erman, haffner, sen}@research.attt.com; Z.-L. Zhang, Department of Computer Science and Engineering, University of Minnesota, 4-192 Keller Hall, 200 Union Street SE, Minneapolis, MN 55416; email: zhzhang@cs.umn.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permission may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 1556-4681/2012/03-ART4 $10.00
DOI 10.1145/2133360.2133364 http://doi.acm.org/10.1145/2133360.2133364

1. INTRODUCTION

Motivation. Today's large IP networks, such as tier-1 ISPs and global enterprise networks, carry a mixture of traffic from a wide range of diverse applications. The ability to accurately classify network traffic, namely map traffic to the types of applications that generate them, is critical to the operations and management of such networks. Security monitoring includes detection of known attacks, or unexpected growth in an application usage that may indicate a new exploit. Traffic engineering in an ISP requires better classification and prediction of peer-to-peer and video application traffic, due to their high and growing bandwidth consumption. Identifying emerging applications and estimating their bandwidth requirements can also help ISPs better forecast and adapt to application trends, and plan network capacity accordingly. Meeting and monitoring service level agreements (SLAs) for business customers and other profit-generating applications, as well as traffic policing and prioritization for performance-sensitive applications, also require accurate traffic classification.

While a variety of network traffic classification methods have been proposed (see, e.g., Sen et al. [2004], Karagiannis et al. [2004], Haffner et al. [2005], and Ma et al. [2006]), most were developed for monitoring at the level of end hosts or enterprise networks, and either do not meet the accuracy requirements of network management tasks, or are too expensive to be scalably deployed in such large networks. For example, the simplest and cheapest approach is to use port numbers; however, attribution of applications based on port numbers alone is widely known to be problematic, limited by non-conventional use of port numbers, amongst other reasons [Sen et al. 2004]. While packet-based traffic classification techniques (e.g., Haffner et al. [2005] and Ma et al. [2006]) are generally very accurate, they additionally employ application protocol level information, and hence require visibility beyond the packet's network and transport layer headers. At the line speed of high-capacity network links, special traffic measurement devices are needed for such packet-based classifiers, making them costly for large-scale deployment that could classify all network flows.

System Design Considerations. Guided by the practical constraints and requirements of traffic classification in large IP networks, in this article we focus on the design of an accurate and scalable traffic classification system that uses only IP *flow-level* network data. Unlike packet-level data, flow-level data (derived from IP and transport layer packet headers) is collected routinely and widely in large networks, and used for various network management tasks. Their availability notwithstanding, the amount of flow data is still enormous and thus poses challenging scalability issues in the design and operations of network traffic classification systems. Hence, a central research question which this article attempts to address is the following: Is it possible to develop a flow-level traffic classification system that meets the scalability and other requirements (e.g., can be inexpensively deployed in a large-scale network, producing reliable, stable and interpretable results without frequent inputs or tracking by human operators), while attaining the accuracy of a packet-level traffic classification scheme? We employ flow records created at two locations in a large Tier-1 ISP using specialized measurement devices. These flow records are annotated with application labels by an operational packet-level classification system that utilizes application protocol level information. Using these flows as ground truth, we then explore (supervised) machine learning (ML) techniques to train a flow-level classification system that accurately and scalably reproduces the packet-level classification outcomes.

The main contribution of this article lies in the development of a machine-learning-based flow-level traffic classification system, which utilizes a lightweight *modular architecture*, together with several innovative mechanisms, for large-scale and accurate traffic classification. The modular architecture combines a series of simple linear

binary classifiers each of which can be efficiently implemented and trained on vast amounts of flow data in parallel and integrates them in such a manner that it attains the accuracy of more sophisticated (but computationally more demanding, thus less scalable) ML classifiers. This is achieved via three innovative mechanisms, (i) *weighted threshold sampling*, which allows each of the simple binary classifiers to be adequately and extensively trained on large training data despite the inherently imbalanced data in terms of traffic class distributions; (ii) *logistic calibration*, which effectively handles the differences in traffic class distributions of both training and testing data to attain high accuracy when the outputs of the binary classifiers are integrated; and (iii) *intelligent data partitioning*, which utilizes domain knowledge to intelligently partition data based on certain features to further enhance the scalability and accuracy of the system. This architecture is very naturally amenable to parallelization, which is becoming increasingly prevalent with the advent of multicore machines. We therefore implemented our classifier as a multithreaded system to exploit such parallelism wherever available.

Our second contribution is to propose a novel two-step-model to incorporate spatial traffic class distribution information, or *collective traffic statistics*, as additional features to further enhance traffic classification accuracy. By representing flows in the network as a colored traffic activity graph (colored TAG) [Jin et al. 2010a], these new features correspond to the closeness of different traffic classes on the graph. The proposed two-step model seamlessly integrates such neighborhood information into the existing traffic classification system. At the first *bootstrapping* step, we use the existing system to conduct flow-level classification and construct a colored TAG with initial labeling from aggregating flow-level classification results. In the second *graph-based calibration* step, we employ the inherent neighborhood and local properties of the edges in the TAG to calibrate (re-enforce) the initial edge labeling.

Preview of Evaluation Results. We evaluate our system's performance along three principle directions: spatial and temporal stability, classification accuracy, and training and runtime scalability.

Extensive evaluations show that our flow-level classifier remains accurate over long period of time as well as across different geographical locations. For example, for training and testing data collected two months apart and from different locations, the error rates are only 3% for TCP flows and 0.4% for UDP flows. Even for a larger time difference of one year, the corresponding error rates are still only 5.5% for TCP and 1.2% for UDP traffic. By incorporating collective traffic statistics as features using the two-step approach, we can further improve the accuracy on all traffic classes, and reduce the overall error rate by 15%. The temporal persistence of accuracy suggests that, once trained, the classifier can be effectively used for a long time without frequent retraining. The spatial persistence of accuracy suggests that the classifier trained at one location can be used at other places, obviating the needs for a deployment of a packet level classification capability everywhere for collecting training data.

The accuracy of the classifier is a function of both the amount of the training data and the number of iterations used by the machine learning algorithm. The training component of our system is designed to be configurable. Given the time budget, it can determine the amount of training data needed and the number of iterations of the learning algorithm. For example, to get error rates of 3% for TCP and 0.4% for UDP, the training time of our system was only around 2 hours.

We evaluated the runtime scalability of our system on different hardware architectures under realistic traffic conditions for a large ISP. On a heavily loaded machine monitoring two 1Gbps links, with only one thread, our system was able to handle a load of up to 800K new flows arriving per minute. This figure is about 2 to 4 times

higher (depending on the time of the day) than the normal new flow arrival rate on those links. On a multicore machine, using 10 threads, we were able to handle up to 6.5 million new flow arrivals per minute, which far exceeds the capability required for classification on a 10Gbps link with similar utilization as the previous case.

We note that even though we evaluate our method using large ISP network data, the challenges and requirements in other large networks, such as large enterprise networks, are similar to ISPs and hence we believe our method is also applicable to these networks.

Outline. The remainder of the article is organized as follows. Section 2 motivates the problem by discussing the unique operational requirements and challenges. We then give an overview of the proposed modular system architecture in Section 3, explain the technical details of the design choices in Section 4, and evaluate each of them in Section 5. We discuss how to use the two-step model to improve the classification accuracy based on the collective traffic statistics in Section 6. Section 7 presents the details of implementation and optimization employed, and evaluates the accuracy, scalability and stability of the whole system. Finally, Section 8 concludes the article.

1.1. Related Work

Most supervised learning-based approaches [Jiang et al. 2007; Moore and Zuev 2005] use manually labeled data for training purposes and have major focus on accuracy. Moore and Zuev [2005] extensively study the suitability of a Naïve Bayes classifier for Internet traffic classification. This work is continued in Jiang et al. [2007] by adapting their approach to real-time traffic classification. Williams et al. [2006] compare five population machine learning algorithms. Our work shares with theirs the same observation that training time varies greatly across different machine learning algorithms and shall be taken into consideration in practice.

Because labeled training data is hard to obtain, many works propose using unsupervised/semi-supervised learning methods for traffic classification. Bernaille et al. [2006] use unsupervised learning of application classes by clustering of flow features and a derivation of heuristics for packet-based identification. Similarly, Crotti et al. [2007] use packet sizes, interarrival times, etc. of the first N packets as features for their classifier. Erman et al. [2007] propose a semi-supervised ML approach based on clustering flow statistics. All these approaches demand manual identification of the traffic class for each cluster and hence are not scalable to large networks. Moreover, these methods are known to be less accurate than supervised learning.

Automated construction of packet-level application signatures via ML was proposed in Haffner et al. [2005]; by contrast, our focus is on automated construction of flow level signatures, learning from the outcomes of packet-level classifiers.

In comparison to building a general traffic classification framework, many works apply machine learning techniques to classify traffic under specific application scenario. For instance, But et al. [2007] proposed the ANGEL system for detecting and then prioritizing game traffic in ISP network. Iliofotou et al. [2009b] developed a graph-based framework for classifying P2P traffic. Nguyen and Armitage [2006a, 2006b] proposed training the classifier on a combination of short subflows for fast traffic classification and demonstrated that good classification results could be achieved with subflows formed by as few as only 25 packets.

In contrast to much previous work, the major contribution in our article is that we take into account many practical challenges, for example, scalability and stability, for deployment of machine-learning-based traffic classification techniques in large networks. Extensive evaluations demonstrate that our system meets the practical requirements for traffic classification in such large networks.

In additional to utilizing packet-level or flow-level features for network traffic classification, several recent studies [Iliofotou et al. 2009a; Karagiannis et al. 2005; Trestian et al. 2008; Xu et al. 2005] have examined the problems of endpoint traffic characterization from network traffic data as well as other information. Of particular interest is the multi-level traffic classification scheme (called BLINC) proposed in Karagiannis et al. [2005], which also exploits characteristics of traffic exchanged with other hosts. Nonetheless, the goal of BLINC is primarily on characterizing the traffic of endpoints, not the application classes used between two endpoints. Further, unlike our two-step methodology for utilizing collective traffic statistics which is developed based on sound machine learning models, BLINC is purely heuristic-driven, using ad hoc (unsupervised) graph analysis. One difference in applicability of our work compared with graph level classification of Iliofotou et al. [2009a] is that the latter is not expected to work well for small traffic classes, for example, `NetNews` or `FTP`, for which relatively few edges are observed, and for which hence the graph level features are not statistically meaningful.

In terms of analyzing spatial patterns from network traffic data, McDaniel et al. [2006] studies the so-called (host-level communities-of-interest (COIs) in network traffic, and use the historical communications of COIs as reference profiles for detecting propagation of malware. The notion of traffic activity graphs (TAGs) come from the two recent studies [Iliofotou et al. 2007; Jin et al. 2009] which investigate the properties of various application-specific TAGs. The two-step methodology in this paper was inspired by the findings in Jin et al. [2009].

Collective classification or inference [Sen et al. 2008] refers to a class of algorithms that extend machine learning classifications techniques to network or graph data. These techniques have been extensively studied on web data and social networks but we are not aware of their application to IP traffic graphs. Besides its scale, a traffic graph presents another challenge to traditional collective inference techniques: edge (or node) labels can change with time. The only class of collective classification algorithms that would scale to our data is iterative classification [Sen et al. 2008]. Compared to iterative classification, our proposed two-step method is significantly simpler and is especially well suited for time varying-graph data, as we shall see in Section 7.6. We have applied the two-step method in Jin et al. [2010a] for classifying network level traffic where port numbers are missing. In this article, we adopt the same approach and demonstrate its usefulness in improving the classification accuracy and stability of the flow-level traffic classifier. In comparison to Jin et al. [2010a], this article also describes in detail the novel modular machine learning architecture for building the flow-level traffic classification system and evaluates extensively the system in a large-scale ISP network.

2. PROBLEM SETTING

We motivate the design of our flow-level classification system by discussing the target operational environment and the unique challenges arising in large ISP networks, before overviewing our proposed solution in Section 3.

2.1. Operational Requirements

Monitoring traffic in large IP networks presents challenges in speed and scale. They typically span multiple geographically dispersed sites, including city level router centers housing 10s or 100s of routers equipped with high-speed (Gbps or higher) interfaces. Passive traffic flow measurement commonly takes place at routers, using, for example, Cisco NetFlow. A traffic flow is a sequence of packets with a common key, namely, the standard 5-tuple of IP protocol, source and destination IP addresses, and

Table I. TCP/UDP Broad Application Classes

Index	TCP/UDP	Class	Exemplary Applications
1	TCP/UDP	Business	Middleware, VPN, etc.
2	TCP/UDP	Chat	Messengers, IRC, etc.
3	TCP/UDP	DNS	DNS application
4	TCP/UDP	FileSharing	P2P applications
5	TCP	FTP	FTP application
6	TCP/UDP	Games	Everquest, WoW, Xbox, etc.
7	TCP	Mail	SMTP and POP
8	TCP/UDP	Multimedia	RTSP, MS-Streaming, etc.
9	TCP/UDP	NetNews	News
10	TCP	SecurityThreat	Worms and trojans
11	TCP/UDP	VoIP	SIP application
12	TCP	Web	HTTP application

TCP/UDP ports, that are localized in time. Flow measurements comprise summary statistics that aggregate information derived from a flow's packet headers (including the key, aggregate packet and byte counts for the flow, and timing information) that are exported as IP flow records to a collector. Router-based flow measurement is widely available in routers and provides a low-cost approach to network traffic measurement.

To serve the various needs network management tasks, such as traffic accounting and security monitoring, we must classify each IP flow accurately into one of a number of broad application classes (using one or both of TCP and UDP), each of which is associated with one or more specific applications; see Table. I.

Since traffic flows are the only available data at most network sites, a flow-based traffic classification system is a natural choice. In selecting an approach, we are drawn to machine learning, which has been used to automatically derive classification rules in settings when data sizes makes this infeasible as a manual task; see Section 1.1. Thus, we focus on designing *a machine learning based flow-level traffic classification* solution. The basic idea depends on having a training dataset comprising flow records, which in addition to the usual flow level information, are also labeled with their applications and classes from Table I. As described more fully here, such labels are derived from packet-based classifiers that can key off application protocol level information in packets, in addition to the usual packet header fields. Using this training data, machine learning algorithms construct a new set of classification rules that can operate purely at the (unlabeled) flow level, but that reproduce, at least approximately over a population of flow records, the desired application class labels.

Training datasets of the type just described are typically acquired through packet-level monitoring at special purpose measurement devices; this is described more fully in Section 2.2. The expense of widespread deployment of such devices network wide provides a powerful incentive for our approach. A limited deployment of such functionality in one (or a small number of) representative network sites furnishes training data that can be used to construct flow-level classifiers that can operate on flow level measurement network wide.

The design of such a machine learning based flow-level traffic classification system for the target operational environment must meet the following requirements.

— *Accuracy*. The flow-level classifier should attain similar performance to the packet-based classifier, i.e., mapping flows to a set of application/class categories ("labels") with low error rates.
— *Scalability* has two aspects. First, training should not require prohibitively expensive computational resources. When retraining is necessary to maintain accuracy (periodically or in response to events) the time required should be controllable.

Table II. Datasets from Two Sites

Site 1 (s_1)		Site 2 (s_2)	
w_1	05/02/2008 to 05/08/2008	w_1	06/12/2008 to 06/18/2008
w_2	05/09/2008 to 05/15/2008	w_2	06/19/2008 to 06/25/2008
w_3	05/16/2008 to 05/22/2008	w_3	06/26/2008 to 07/02/2008
w_4	06/09/2008 to 06/15/2008	w_4	07/03/2008 to 07/09/2008
w_5	06/16/2008 to 06/22/2008	d_1	04/12/2009
w_6	06/26/2008 to 07/02/2008	d_2	04/13/2009
w_7	07/03/2008 to 07/09/2008		

Second, at runtime, classification must be able to keep up with the high traffic volume and rates at each site, without requiring sampling prior to classification. In practice, bursts of around 1 million new flows per minute must be accommodated on Gbps links.

— *Stability*. The classifier is expected to remain accurate for a long duration without retraining or human intervention. Likewise, the same classifier is expected to be deployed at different sites without site-specific retraining or parameter tweaking. Such temporal and spatial stability are particularly important, as retraining requires collection and labeling of new packet-level training data, which may be computationally expensive.

— *Versatility*. In order to accommodate changing needs of network management tasks and the emergence of new applications, the traffic classifier should be easily reconfigurable to utilize different feature sets or new class labels, without having to rebuild the entire system. Such capability is especially valuable, once the classifier has been deployed in multiple different locations.

2.2. Data Characteristics and Challenges

Characteristics of the available training data also present several unique challenges for our system design. Our datasets comprise flow records, annotated by application labels, that were created by special purpose traffic measurement devices operating at two geographically dispersed sites of a large ISP network. As in Haffner et al. [2005] and Karagiannis et al. [2005], the labels are generated in an automated way by the measurement device, using a set of packet-level rules based on combinations of packet signatures that operate on layer-4 packet header information, and layer-7 application protocol signatures. The flow records do not include any application data; neither do they report any user identity information. Due to the huge traffic volume, sampling is employed in the creation of flow records, with 1 out of 20 flows reported on, sampling over the standard flow level 5-tuples. However, for each sampled flow, the flow record aggregates header information from all its packets, without further sampling. The datasets contain flow records from approximately 40,000 ISP network endpoints gathered at two sites, representing several hundred Terabytes of network traffic. No endpoint is represented at both sites. The datasets are summarized in Table II, where $w_i s_j$ denotes the whole ith week dataset from site j, and $d_k s_j$ to indicate the whole kth day dataset from site j.

The distribution of flows over the application classes of Table I is highly unbalanced. The largest classes—`Web` and `FileSharing`—account for 60% to 80% of the total flows in different weeks, while the smallest classes (e.g., `NetNews` and `SecurityThreat`) contain only a few thousand flows out of millions. In addition, a portion of flows (29.4% of total flows representing 19.9% of total bytes) cannot be classified using the packet-based classifier, that is, they do not match any rule. This can be caused by encryption of application level information, or the presence of new applications or security threats for which signatures are not yet developed. We call these flows `Unknown`

Table III. Flow-Level Features

Name	Type	Name	Type
lowsrcport	numeric	lowdstport	numeric
highsrcport	numeric	highdstport	numeric
duration	numeric (*)	packet	numeric
mean_packet_size (mps)	numeric (*)	byte	numeric
mean_packet_rate (mpr)	numeric (*)	tos	numeric
toscount	numeric	numtosbytes	numeric
tcpflags	text	srcinnet	{0,1}
dstinnet	{0,1}		

hereafter. The *labeled* flow records are used both as training and testing data, and serve as the "ground truth" for our study. We exclude `Unknown` flows in our experiment. (However, because we are adding new manual rules over time, our ground truth on the training data is also expanding. The data classified by these new rules can, in some sense, serve as ground truth for evaluating the accuracy of our classification system on `Unknown` flows. Please see the Appendix Section for details).

The *flow-level* features used in our study are listed in Table III.[1] The numeric values for the two (related) features *lowsrcport* and *highsrcport* are set using the following rule: If the source port of a flow, say, x, is above 1024, then *lowsrcport* is set to -1, and *highsrcport* is set to x; otherwise, *lowsrcport* is set to x and *highsrcport* is set to -1. *lowdstport* and *highdstport* are set similarly. *Duration*, *packet* and *byte* represent the length of the flow, number of packets and bytes in the flow, respectively. *Mean_packet_size* is the average bytes per packet, and *mean_packet_rate* is the average packet interarrival time in seconds. The *tcpflag* feature contains all possible TCP flags in the packets. The TOS (type of service) related features *tos*, *toscount* and *numtosbytes* are the predominant TOS byte, the number of packets that were marked with tos, and the number of different tos bytes seen in a flow, respectively. The last two features *srcinnet/dstinnet* equals 1 if the source/destination address belongs to the ISP network, and 0 otherwise.

The training architecture must accommodate both the large size and the imbalance of the training data, while producing accurate flow level classifiers in an acceptable time. In addition, we must select the appropriate features from amongst the many possible in order to avoid the poor accuracy which can result from overfitting. Even though a number of powerful machine learning algorithms, in particular nonlinear and kernel approaches, are capable of achieving high classification accuracy in general, we learned during our study that a "black box" approach of employing existing machine learning methods in a standard way does not provide a practical solution that meets the accuracy, scalability, stability and versatility requirements. Instead, we must understand both the specific constraints and unique requirements of our setting, and the inner workings of machine learning algorithms, then judiciously exploit domain knowledge to guide our design. In the next section, we provide an overview of our solution: a lightweight modular architecture for large-scale network traffic classification, and discuss several innovative mechanisms to address the operational requirements and challenges.

3. SYSTEM DESIGN OVERVIEW

We now present an overview of the proposed solution. We focus on the design choices for satisfying various operational requirements. We will provide a formal description

[1]The features marked (*) are not reported directly in flow records, but computed from quantities thereof.

of the whole classification process and explain the technical details of different components in Section 4.

Our key idea to address all these requirements is through *modularization*, that is, we decompose the original monolithic *multiclass* traffic classification task into a series of much simpler subtasks. Exploiting the availability of distributed systems and multicore machines, subtasks can be executed in parallel, resulting in high efficiency both during the training phase as well as during real-time classification operations. Further, the reduction in complexity due to partitioning makes possible that we can apply more training data for a given total computational resource than in the unpartitioned case; thereby enabling higher accuracy and training scalability.

3.1. Two-Level Modularization

Our system uses two levels of modularization. Figure 1(a) shows the first level partitioning of the system: we exploit domain knowledge to intelligently partition the flow data into m nonoverlapping flow sets according to certain flow features. For example, the flow data can be partitioned based on protocols or flow sizes. We then run m multiclass (k-class) classifiers in parallel, where each one classifies flows in one of the m partitions exclusively. This first level partitioning also enhances accuracy by training a classifier particularly specific for the flows in each partition. We will discuss the flow-size based partitioning strategy in Section 5.5, which is designed specifically for enhancing the byte accuracy.

Zooming into one of the m multiclass classifiers, we show the architecture for the second level modularization in Figure 1(b): this divides each k-class classifier into k binary classifiers, that is, each binary classifier is a *one-vs-rest* classifier that separates examples belonging to the associated broad traffic class from all others. For each flow, the k binary classifiers produce k scores, indicating the likelihood that the flow belongs to each of the k traffic classes. We then combine the binary classification results for multiclass classification by assigning the flow to the traffic class with the highest score. We note that we can also decompose the multiclass classification problem in a finer granularity, for example, based on applications, which we will discuss in Section 5.5.

Previous studies have found the deviation from optimality resulting from a decoupled two-stage approach does not cause significant increase in error in practice [Rifkin and Klautau 2004]. In our setting, the practical advantages of using one-vs-rest classifiers are considerable. Compared to the single k-class classifier, training k binary classifiers can be both cheaper and faster. Up to k times less memory is required, and with parallelization on a k-core machine, one can train the k binary classifiers at least k times faster. Another advantage is flexibility. We can select unique features or classification algorithms for different traffic classes for better accuracy or scalability. We can also easily accommodate new class labels by adding new binary classification modules for them.

3.2. Choice of Machine-Learning Algorithms

Any machine-learning algorithm can potentially serve as a binary classifier. However, to strike a good balance between accuracy and scalability, we use state-of-the-art classification algorithms that linearly combine first order predicates over features, since these can be efficiently implemented without requiring immense computational power. In particular, Adaboost [Freund and Schapire 1995] has execution time that is easy to control, and it performed most accurately on our datasets. We also considered alternative algorithms with more power to represent and learn complex non-linear separations between classes. However, using higher order predicates, for example,

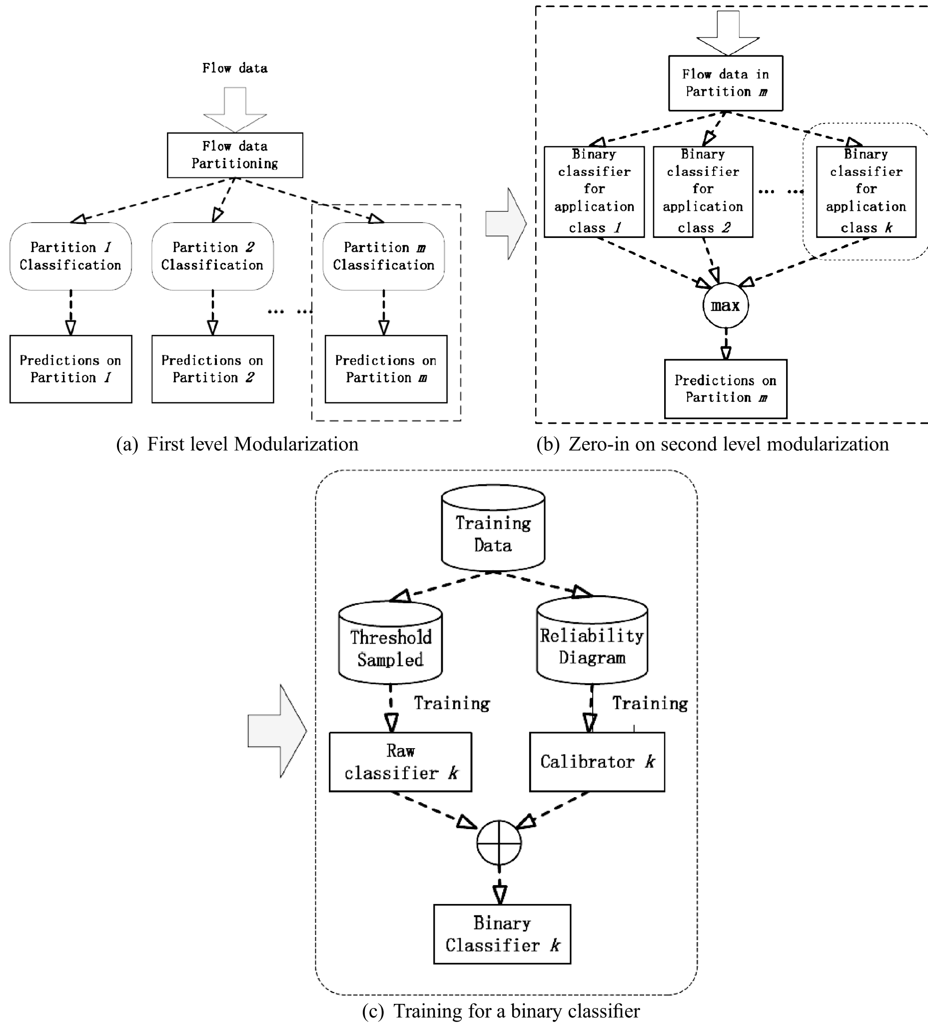


Fig. 1. A schematic representation of the modular traffic classification architecture.

classification trees, did not noticeably improve accuracy, while being much less scalable both in training and real time operations.

3.3. Sampling and Calibration

The training architecture for individual classifiers, see Figure 1(c), was designed for scalability. Due to the imbalance of the amount of training data across different application classes, some class' trainers would still be swamped by their training data, even after partitioning into multiple one-vs-rest binary classifiers. Simple uniform sampling cannot simultaneously reduce the data of the large classes to a workable size while still retaining sufficient samples to maintain acceptable accuracy for small classes. Instead, we propose a nonuniform *weighted threshold sampling* to create a small but more balanced training set; see Section 4.3.

The nonuniform sampling technique, while enhancing accuracy of the overall system, can skew the distribution of the training data away from the actual traffic

ALGORITHM 1: Training the flow-level classifier.

1: Parameters: Flow set $\mathcal{F}$, number of classes K, partitioning criteria;
2: Output: classifiers $\{c_{ij}\}$ and calibrators $\{cl_{ij}\}$, $1 \le i \le K, 1 \le j \le p$;
3: Partition $\mathcal{F}$ according to the partitioning criteria into $\{\mathcal{F}_j\}$, $1 \le j \le p$;
4: **for** each $\mathcal{F}_j$ **do**
5:   **for** i from 1 to K **do**
6:     Create a training data set D_{ij}^{tr} from $\mathcal{F}_j$ using weighted threshold sampling (see Section 4.3);
7:     Train a classifier c_{ij} using D_{ij}^{tr};
8:     Run c_{ij} on $\mathcal{F}_j$ and get results R_{ij};
9:     Create a reliability diagram using $\mathcal{F}_j$ and R_{ij} (see Section 4.4);
10:     Train a calibrator cl_{ij} based on the reliability diagram;
11:   **end for**
12: **end for**

ALGORITHM 2: Flow-level classifier in operation.

1: Parameters: flow set $\mathcal{F}$, partitioning criteria, classifiers $\{c_{ij}\}$ and calibrators $\{cl_{ij}\}$, $1 \le i \le K, 1 \le j \le p$;
2: Output: multi-class classification result
3: **for** each flow record $x \in \mathcal{F}$ **do**
4:   Run c_{ip} on x and get a score $f_{c_{ip}}(x)$;
5:   Calibrate $f_{c_{ip}}(x)$ using cl_{ij} and get the posterior probability $P(C_i|x)$ corresponding to the application class C_i;
6: **end for**
7: Find $k = argmax_q P(C_q|x)$, $1 \le q \le K$;
8: Classify flow record x into application class C_k;

distribution. This potentially violates the independent and identically distributed (IID) assumption in the machine learning algorithms, and leads to problems when we combine the outputs from the binary classifiers (subtasks) to produce the final (multiclass) classification results. We solve this problem by a *logistic calibration* method in Section 4.4. At training time, we train a calibrator for each binary classifier. Then, at runtime, the prediction from a binary classifier will be adjusted by the associated calibrator first before combination.

We summarize the training and operation of the proposed system in Algorithm 1 and Algorithm 2. We note that we present the algorithm in a sequential way. However, all the procedures within the for loop are ready to be parallelized and the performance of the system can be improved significantly (see Section 7). The next section details the different system components. In Section 5, we justify our design choices through extensive experiments. In Section 7, we use test sets of labeled flow records to evaluate the performance of the whole system for scalability, accuracy, and stability.

4. DETAILS OF THE COMPONENTS

In this section, we introduce the technical details for building our machine learning-based traffic classification system. For ease of exposition, we first formalize the classification process in our system as follows.

Let $\mathcal{F} = \{(\mathrm{x}_1, y_1), (\mathrm{x}_2, y_2) \cdots, (\mathrm{x}_n, y_n)\}$ be a set of n flow records. Each $\mathrm{x}_i = \{x_{ij}, 1 \le j \le r\}$ is an r-dimensional vector representing a flow record of r features, where x_{ij} stands for the jth feature in the ith flow record. x_{ij} can be either categorical (e.g., tcpflag)

or numerical (e.g., flow duration). $y_i \in \{C_1, C_2, \cdots, C_k\}$ stands for the label of the corresponding flow record, from the predefined k classes, for instance, `Web` class and `Multimedia` class. Our goal is to find an efficient yet accurate classification mapping that can be used to attribute an application class y to any x.

At the first level of modularization, the flows are partitioned into m nonoverlapping subsets, that is, $\mathcal{F} = \{\mathcal{F}_h\}$, where $1 \leq h \leq m$. For each flow partition $\mathcal{F}_h$, we train a separate k-class classifier. At the second level of modularization, we divide each of the k-class classifiers into k binary classifiers. Upon the arrival of each flow record, each of the k binary classifiers will produce a prediction, or the posterior probability $P(C_j|\mathrm{x})$, indicating the likelihood of the given flow belonging to the corresponding application class C_j. We then compare the k posterior probabilities, and assign the flow to the application class C, where $C = \mathrm{argmax}_{C_j} P(C_j|\mathrm{x})$. In the ideal case, this assignment exactly corresponds to the Bayes optimum for the multiclass classification problem [Duda et al. 2000].

4.1. Modularization Strategies

By default, the first level of modularization partitions flows by protocol into TCP flows and UDP flows.[2] Section 5.5 studies a partitioning strategy based on flow size, designed specifically to improve byte accuracy.

At the second level of modularization, our default partitioning strategy is based on broad application classes, that is, we partition the TCP (and UDP) multiclass classifiers into 12 (and 8) binary classifiers, respectively. We can also partition more finely, based on specific applications, if network management tasks require this detail. For example, instead of using a single binary classifier for the entire `FileSharing` class, we use multiple binary classifiers corresponding to individual applications within the class, such as `BitTorrent`, `eMule`, etc. Results under application-based partitioning are in Section 5.5.

4.2. Adaboost Algorithm as a Building Block

Almost all machine-learning algorithms can handle the binary classification problems that we have described. To fulfill the requirements of scalability, accuracy and stability, we prefer simple yet powerful linear algorithms that minimize the number of features they use. A small number of features yields faster classification times, better generalization by avoiding idiosyncratic features, and easier model analysis. The *Adaboost* [Schapire and Singer 2000] algorithm fulfills this requirement with a greedy incremental approach that can be restricted to learn a limited number of features (with implicit L_1 regularization). To strike a balance between accuracy and scalability, we choose the decision stump (the simplest decision tree, having one level) as the weak learner. We refer to *Adaboost* with decision stumps as *BStump* in the rest of the article.

Figure 2 gives a schematic view of *BStump*. During the training phase, we specify the number of iterations T (or the number of the weak learners) used by the algorithm. At iteration t, the algorithm selects one particular flow feature and the corresponding feature value δ that best partitions the weighted training data into positive (target class) and negative (other classes) instances. The algorithm creates a decision stump using the selected feature as the weak learner, which we denote as h_t, and the classification result from the weak learner on flow record x is represented as $h_t(\mathrm{x})$.

[2]Other IP protocols constitute a negligible proportion of the total flows and bytes.

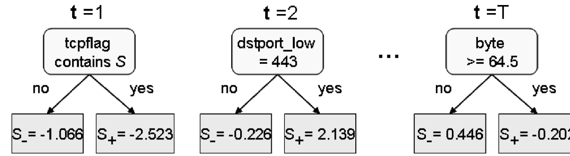


Fig. 2. Weak learners for the `Voip` class.

Each weak learner outputs S_- for a feature value below δ (for continuous feature) or not equal to δ (for categorical feature), and outputs S_+ otherwise. A total score corresponding to a combination of weak learners is computed and a threshold is applied to compute a binary outcome. The data weights are adjusted in order to best reproduce the ground truth on all flows. The process is iterated until T weak learners are generated.

At runtime, for each flow x, T scores are generated by the weak learners from the binary classifier corresponding to target class C and these scores are summed up as the prediction $f_C(\mathrm{x}) := \sum_{t=1}^{T} h_t(\mathrm{x})$. The binary prediction is determined based on the sign of $f_C(\mathrm{x})$, having a confidence that increases monotonically with $f_C(\mathrm{x})$.

Due to the simplicity of the *BStump* algorithm, the training time and accuracy of *BStump* and hence the entire system can be controlled by varying the training data size and the number of iterations; see Section 7.4. One of the main limitations of *BStump* is that it may overfit on noisy data. However, *BStump* performs well on data with little noise [Haffner et al. 2005], and this is the case in our setting.

One potential problem of *BStump* (and for other algorithms except L_1-*Maxent*, that are considered in Section 5.1) arises when k *BStump*'s are combined for the final multiclass prediction. The score output ($f_C(\mathrm{x})$) from *BStump* ranks examples belonging to the target class, but is not a good approximation of $P(C|\mathrm{x})$. Choosing C to maximize $f_C(\mathrm{x})$ does not in general yield a solution that minimizes classification error for the equivalent full k-class training problem. We need a method to remap $f_C(\mathrm{x})$ to $P(C|\mathrm{x})$. This problem can be solved by the calibration method proposed in Section 4.4.

4.3. Weighted Threshold Sampling

In this section, we propose a weighted threshold sampling method that creates smaller but more balanced training sets from an immense number of imbalanced flow records. Given a threshold θ, if the number of flows $|C_i|$ in a class C_i is below the threshold θ, we keep all flows from C_i. But if $|C_i| > \theta$, we perform simple random sampling on flow records from C_i with sampling rate $\theta/|C_i|$, thus yielding θ flows on average. As we shall see in the experiment, the parameter θ can be easily determined from the training data distribution and the available computation resources.

By creating a balanced sample, weighted threshold sampling introduces bias since the distributions across applications are different for the training and testing data sets, essentially violating the IID assumption that machine learning algorithms rely on. This problem can again be solved by the calibration method in Section 4.4.

4.4. Logistic Calibration

Calibration addresses two issues: (1) different distributions in training data and testing data, (2) remapping of the score outputs $f_C(\mathrm{x})$ to the posterior probabilities $P(C|\mathrm{x})$. Calibration is based on the observation that the relation between the predictions from the binary classifiers and the posterior probabilities tends to follow a logistic curve [Platt 1999]. This can be visualized from the reliability diagrams in Figure 3. Each

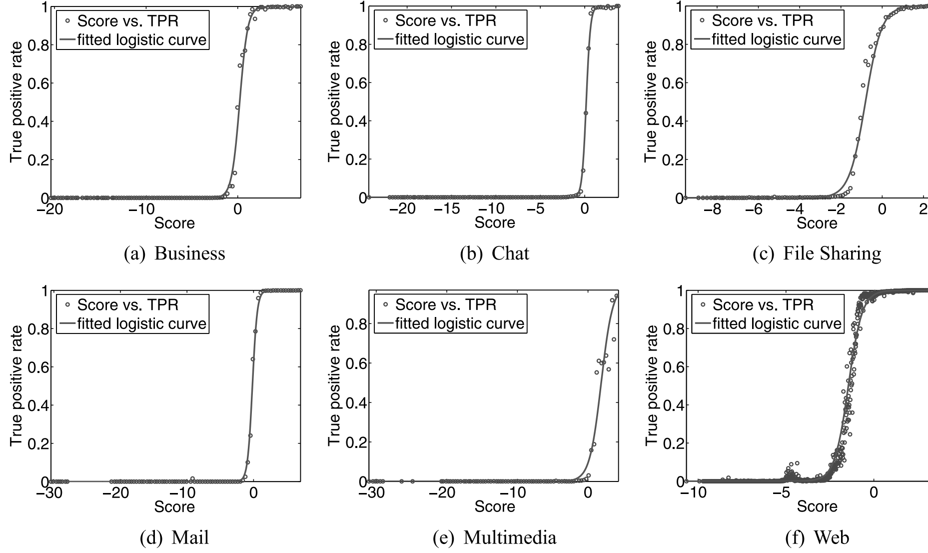


Fig. 3. Calibration results on selected TCP classes (similar observations for other TCP and UDP classes).

plot corresponds to the classification results for a specific binary *BStump* classifier,[3] where the x-axis represents the output score intervals, and the y-axis represents the empirical posterior probability value (true positive rate), which is computed as the number of positive samples divided by the total number of samples within the same score interval. For a well-calibrated classifier in which the output scores match the posterior probabilities perfectly, we expect all the points in the associated reliability diagram to be along the diagonal line. However, we observe that all the points actually follow a logistic curve. Let $f_C(\mathrm{x})$ denote the predicted score of flow record x from a binary classifier C, the logistic relation can be expressed as:

$$P(C|\mathrm{x}) = \frac{1}{1 + exp(-\alpha f_C(\mathrm{x}) - \beta)}. \quad (1)$$

In practice, we use the entire flow set from which the training data is generated as our calibration set to address the different distributions in the training set and the testing set. In particular, we run each binary classifier on the calibration set. We then create a reliability diagram based on the classification results to estimate α and β for the calibrator by fitting a logistic curve of all the points on the diagram. These logistic curves are also displayed in Figure 3. In the process of computing α and β, we need to choose the number of score intervals to construct the reliability diagram. A large enough number of intervals is required to accurately fit a logistic regression curve, but too many intervals leads to more outliers. Empirical studies indicate that the number of intervals between 50 and 100 generally provides satisfactory results. Therefore, we always use 100 intervals for fitting the logistic curves during our experiments.

[3]Similar results are observed for *BStump* classification results on other TCP/UDP classes and the results of other binary classifiers except *BTree*, which we will discuss in Section 5.2.

5. EVALUATING DESIGN CHOICES

This section provides extensive evaluations of our design choices for the machine learning architecture. We present our evaluation in a bottom-up manner. We first compare the binary classification accuracy and scalability of *BStump* with three alternate algorithms. We then aggregate the binary classification results to determine multi-class classification performance, and demonstrate the effectiveness of our calibration method. Lastly, we discuss and evaluate different modularization strategies.

5.1. Alternative Algorithms

We use the Boostexter [Schapire and Singer 2000] implementation for the selected *BStump* algorithm, using the default number of iterations, namely, 400. As we shall see in Section 7, generally, increasing the number of iterations will result in reducing the flow error rate, and vice versa. We select 400 iterations for *BStump* because the flow error rates become stable after 400 iterations on most training datasets. For the purpose of comparison, we select three popular machine learning algorithms. The parameters for the alternative machine learning algorithms are selected in a similar way.

Maximum Entropy. Maxent classifiers aim to approximate the probability distributions for each class C_j with the simplest possible distribution, corresponding to maximum entropy. While most Maxent algorithms directly optimize the conditional probability $P(C_j|\mathrm{x})$, the specific L_1-*Maxent* algorithm [Phillips et al. 2004] optimizes the joint probability $P(C_j, \mathrm{x})$. It consists of a sequential procedure that greedily adds weak learners in a similar way to *Adaboost*. L_1-*Maxent* converges to an optimum that maximizes the likelihood while minimizing the L_1 norm of the weight vector, and thus the number of non-zero feature weights. Theory and experiments show convergence even with very noisy data. In this paper, L_1-*Maxent* will refer to an implementation using stumps.[4]

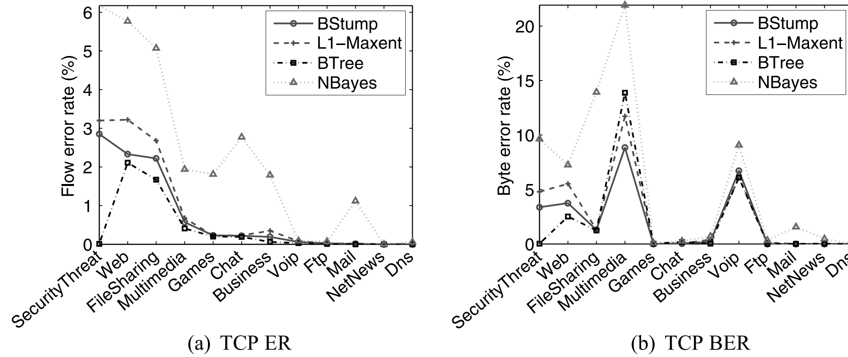
Boosting Decision Trees. Decision trees are a generalization of decision stumps, comprising a hierarchy of threshold decisions over different features. They are supported by *Adaboost*, in an algorithm we denote by *BTree*. While a boosted decision tree is still a linear classifier on decision tree features, it has the representation power of a more complex non-linear classifier that can handle conjunction of features. For *BTree*, we use the Weka [Witten and Frank 1999] implementation (version 3.6.1) with number of iterations equal to 100 to avoid overfitting; other parameters take default values.

Naïve Bayes as a Baseline. As a baseline for comparison we trained a Naïve Bayes classifier (*NBayes*), the most widely used machine learning algorithm in the traffic classification domain [Jiang et al. 2007; Moore and Zuev 2005]. A Naïve Bayes classifier models the distribution function $P(\mathrm{x}|C_j)$ for each traffic class C_j and obtains the prediction by using the Bayes theorem. A kernel approach is also applied to fit $P(\mathrm{x}|C_j)$ with multidimensional Gaussian mixtures. We use the Weka implementation (version 3.6.1) of the *NBayes* classifier with the kernel option on and other parameters as default.

5.2. Binary Classification Accuracy

A fair comparison of the learning algorithms requires use of the same training set. The size of the training set is limited by *BTree*, which can only accommodate 100K training examples under our hardware configuration. Therefore, we create a training set with approximately 100K flows from a whole week flow data w_1s_1 using weighted threshold sampling with parameter $\theta = 10{,}000$ for TCP and $\theta = 15{,}000$ for UDP. A

[4] `http://www.cs.princeton.edu/~schapire/maxent/`.



(a) TCP ER (b) TCP BER

Fig. 4. TCP per-class error rates.

complete week's data is used in these experiments in order to eliminate any day-of-week variation. Another whole week's data (w_2s_1) serves as our test set.

Given a flow record x, let $f_C(\mathrm{x})$ be the corresponding output score from a binary classifier of class C. A positive prediction—that x belongs to C—is made if $f_C(\mathrm{x})$ exceeds a threshold δ. Hence, δ controls the operating point and performance of the classifier. We adjusted δ individually for each classifier to determine performance at its break-even point, that is, when the numbers of flow false positives and flow false negatives are equal.

Figures 4(a) and 4(b) shows TCP per-class flow error rate (ER) and byte error rate (BER) for all the binary classifiers, expressed as the percentages of misclassified samples at the break-even point. The classes are arranged by the corresponding *BStump* flow error rates in a descending order. In general, the non-linear classifier *BTree* yields the lowest error rate; however, *BStump* and L_1-*Maxent* perform nearly as well as *BTree*, while being at least 10 times as fast as *BTree* on the same data sets. (Scalability is discussed in detail in Section 5.3.) *NBayes* has the worst performance, likely because of the unrealistic independence assumption on the flow features. In terms of per class byte accuracy, `Multimedia` and `Voip` have the largest byte error rates across all classifiers, due to the existence of large and long duration flows in such traffic [Chen et al. 2010]. Similar behavior holds for UDP traffic.

5.3. Scalability of Binary Classifiers

In this section, we evaluate the scalability of different machine-learning algorithms. Here we focus on training scalability; with proper optimization, all four classifiers can achieve similar runtime scalability, discussed for *BStump* in Section 7.3.

Table IV reports the longest training time for a single one-vs-rest binary classifier. Since different binary classifiers can be trained in parallel, this time corresponds to the training time on a multi-core computer in which one core is allocated to each of the 12 classes (see Section 7.1 for details of the lab machine for training). Imposing a limit of training time 2-3 hours, *BStump* or L_1-*Maxent* can support up to 1 million training flows, in comparison to 100K for *BTree*, and less then 500K for *NBayes*. The training time for *BStump*[5] or L_1-*Maxent* is linear in the size of the training data,

[5] We also test the training time of the Weka implementation of *BStump*, which is around 3 times slower than the Boostexter implementation. However, the linear relationship between the number of training samples and training time still holds.

Table IV. Training Time for Different Classifiers

classifier	training data (#. of flows)	training time(mins) TCP	training time(mins) UDP
BStump	100K	17.3	13
	1M	138	122
L_1-*Maxent*	100K	13	13.3
	1M	198	189
NBayes	100K	13	9
	500K	924	459
BTree	100K	115	92.9

superlinear for *NBayes*,[6] while the Weka implementation of *BTree* cannot accommodate more than 100K training flows. In practice, to attain a better classification accuracy often requires an increase of the training samples or an extension of the training time (see Section 7 for the evaluation on training scalability). In this case, *BStump* and L_1-*Maxent* appear to be a more favorable choice to their linear relationship between the training data and the training time.[7]

5.4. Multiclass Classification Performance

A Single k-class Classifier vs. A Combination of k Binary Classifiers. We compared the performance of a directly trained single multiclass classifier against our binary combination approach. For a given memory size and total available training time, the direct multiclass classifier had lower accuracy, principally because fewer iterations could be performed, taking 17 times as long per iteration as the binary case. For example, with 400 iterations during training, the flow error rate for a direct multiclass classifier is around 4.1%, which is worse than the 3.07% error rate from the proposed modular approach.

We note that we use the implementations of Adaboost.MH and Adaboost.M1 as the multiclass *BStump* classifier and the single class *BStump* classifier, respectively. Let k be the number of classes. The training complexity of Adaboost.MH is $O(k^2)$ in time and $O(1)$ in space, in comparison to the $O(1)$ complexity in both time and space for Adaboost.M1. Due to the high time complexity, at the same level of convergence, we could not conduct full experiment (with millions of training samples) using the multiclass classifier. However, on smaller cases, at the same level of convergence, Adaboost.MH with k classes never performed significantly better than k binary Adaboost.M1.

Benefit of Calibration. Recall calibration involves attributing a flow to the class with the highest posterior probability, instead of the class with the highest score. Table V displays the flow and byte error rates (ER and BER) with and without calibration. Except for *BTree*, calibration decreases ER by a factor 2 or better for both TCP and

[6]We note that when discretization is used instead of kernel-density functions, *NBayes* only requires half of the training time as *BStump*. However, the overall ER (BER) for *NBayes* increases in this case from 13.8% (36.5%) to 28.8% (41.7%) on TCP traffic.
[7]We note that our evaluation is based on the existing algorithms without any optimization involved. Other implementation of these algorithms, for example, optimization based on specific machine architectures, memory caches and smart data structures, can potential increase the training scalability of these algorithms. As an ongoing work, we are experimenting on a more efficient implementation of the *BTree* algorithm for the traffic classification task.

Table V. TCP/UDP Multiclass ER (BER) in Percent

	Before Calibration		After Calibration	
Classifier	TCP	UDP	TCP	UDP
BStump	7.64 (41.3)	1.34 (27.5)	3.07 (30.1)	0.34 (18.3)
BTree	8.37 (45.8)	1.97 (54.5)	35.2 (66.6)	2.66 (77.6)
NBayes	26.9 (52.7)	10.3 (72.9)	13.8 (36.5)	1.77 (73.3)
L_1-*Maxent*	18.7 (52.5)	3.40 (39.4)	3.87 (40.1)	0.44 (16.4)

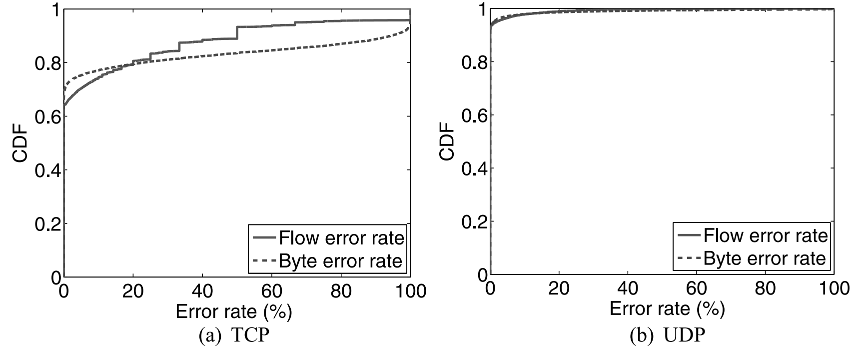


Fig. 5. Distribution of per endpoint accuracy.

UDP traffic. Using calibration, the best performance is obtained with *BStump*, with an ER of 3.07% for TCP and 0.34% for UDP, L_1-*Maxent* being a close second.[8]

Both *BStump* and L_1-*Maxent* have the added advantage of speed (see the next section) and model simplicity. The calibration process actually impairs the classification results of *BTree* classifiers, because *BTree* does not exhibit a logistic relation between the output scores and the posterior probabilities on the reliability diagrams.[9]

Comparing Byte Accuracy across Methods. In Table V, we observe a much higher byte error rate than the flow error rate for each method. This is partly a reflection of our methods, which are optimized to minimize the flow error rate. In fact, the comparison in Table V is limited by use of a smaller dataset to accommodate the relatively slow *BTree* classifier. Focusing on our best method, *BStump*, we will obtain better accuracy by efficient use of more training data, and by a partitioning of the training based on flow size that enables us to improve accuracy on the relatively small number of large flows by handling them separately; see Section 5.5. On of our future works in on developing efficient algorithms that directly optimize byte accuracy.

Byte Accuracy Distribution across Endpoints. Network operation problems that involve attribution of network usage, require accurate estimation of traffic volume per endpoint. Figures 5(a) and 5(b) shows the CDF over endpoints of the BER of classification with *BStump*, for all TCP/UDP flows associated with the endpoint. We see that majority of the byte errors are contributed by only a small portion of endpoints. For 60% of the endpoints, we can achieve 100% TCP byte accuracy; while for more than 90% of the

[8]Most errors are due to `Multimedia` flows misclassified into `Web` classes. Those flows utilize port 80 for communication and share similar characteristics of `Web` traffic. We note that such result can be improved by applying our two-step model as introduced in Section 6.

[9]We note that, without boosting, a good tree classifier, for example, C4.5, can achieve a TCP flow error rate of 4.3% after calibration. We believe that, after an appropriate calibration process, we shall expect a much better accuracy result from *BTree*. We leave this as our future work.

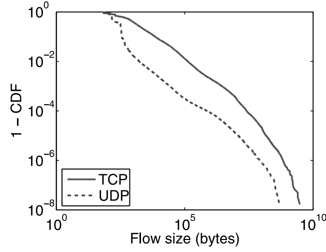


Fig. 6. Byte dist.

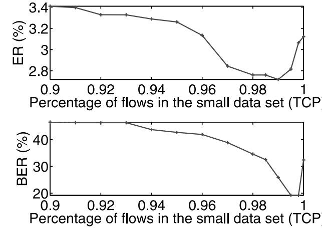


Fig. 7. TCP ER.

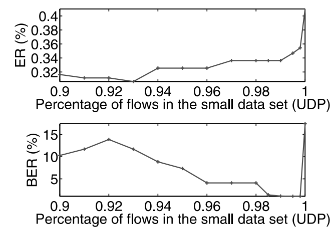


Fig. 8. UDP ER.

endpoints, we can obtain 100% UDP byte accuracy. Comparing with Table V, we conclude that overall byte errors tend to be dominated by those of a relative small number of endpoints. Investigation on these endpoints demonstrates that they concentrate on mostly `Multimedia` applications which use nontraditional port numbers for communication. This implies that these users may be unfairly penalized if the classification result is used for policing or prioritization. A classifier focuses on minimizing the maximum endpoint byte error rate can solve this problem. Searching for and evaluating such algorithms will be our future work. This effect was also found in our subsequent experiments for *BStump*. Hence, in the remainder of this article we shall report on, and differentiate performance by, the overall byte accuracy.

5.5. Alternative Modularization Strategies

In this section, we discuss alternative modularization strategies to improve both accuracy and efficiency in training and testing. In machine learning, this kind of partitioning is known as a Mixture of Experts [Jacobs et al. 1991]. However, unlike traditional methods that randomly partition the data, our proposed method incorporates domain knowledge to determine the feature for partitioning, and then apply automated search for the optimal partitions.

Flow-Size Based First Level Modularization. Our analysis of the cause of high byte error rates in Section 5.4 suggests incorporating more large flows. This agrees with Figure 6, which depicts the flow size distributions (log-log plot) for the TCP and UDP traffic from $w_1 s_1$, respectively. Overall byte accuracy is very dependent on the correct classification of a relatively small number of large flows. To address this issue, we use a threshold value to partition the data set into sets of small and large flows, each of which is to be separately trained and classified.

To determine the threshold for partitioning, we apply a line search approach on a small data set from $w_1 s_1$. For a given threshold, we apply weighted threshold sampling separately to the sets of small and large flows (using $\theta = 10K$ for TCP and $\theta = 15K$ for UDP) to create two training sets with roughly 100K flows. We correspondingly partition $w_2 s_1$ into test sets of small and large flows. We train and test separately on large and small flows, and compute composite ER (and BER) as an average weighted by the total flows (or bytes) in each set. Figure 7 and Figure 8 show the composite ER/BER for TCP and UDP traffic as a function of the threshold as indicated by the proportion of flows that are deemed small. The threshold corresponding to the minimal byte error rate is selected, which is 85,761 (99%) for TCP and 1,703 (98.5%) for UDP.

We now apply our optimal size threshold, to partition a larger dataset of 1 million samples for training and testing (the calibration set and the test set remain the same). The corresponding flow and byte error rates are summarized in Table VI. We can see over 12% decrease in flow error rates for TCP and 6% for UDP. More noticeably, size

Table VI. Error Rates for Partitioning at the Minimal Byte Error Rate Point (as Percent)

	TCP		UDP	
	Before	After	Before	After
ER	3.13	2.75	0.35	0.33
BER	29.3	21.3	17.0	12.1

Table VII. Error Rates for Application Partitioning

	TCP		UDP	
	Before	After	Before	After
ER	3.13	2.79	0.35	0.24
BER	29.3	25.5	17.0	17.1

partitioning rewards us with a significant improvement in byte accuracy, with BER reduced by more than 27% for both TCP and UDP.[10]

Application-Based Second-Level Modularization. Another partitioning strategy is to decompose the multiclass classification at the level of individual applications, rather than the coarser broad application classes of Figure 1(b). For example, the `Mail` class can be further divided into applications, such as `smtp` and `pop3`, etc. Since we have 57 TCP and 34 UDP applications, we apply the weighted threshold sampling at the subclass level with $\theta = 20K$ for TCP and $\theta = 30K$ for UDP to create training sets with around 1M flows. Binary classifiers are then trained for each application, with calibration being also at the application level. At the classification stage, the classifier combines the binary classification results and determines the associated application for each flow, and then assigns the flow to the corresponding application class. The effectiveness of this method is summarized in Table VII. We again observe over 10% decrease in flow error rates. No improvement of the UDP byte error rate has been observed, since our application partitioning method does not guarantee more large flows are included in the training dataset.

In summary, the modular architecture enables us to further break down the traffic classification task for both efficiency and optimizing certain application requirements. These examples illustrate how we incorporate domain knowledge to find partitioning criteria that maximize the flow accuracy and byte accuracy, which performs well in practice. For other application needs, specific partitioning methods can also be designed, which we leave for future work.

6. IMPROVING ACCURACY WITH COLLECTIVE TRAFFIC STATISTICS

In this section, we study the possibility of enhancing classification accuracy by incorporating a new set of features, the collective traffic statistics. We demonstrate the concept of collective traffic statistics through colored traffic activity graphs (colored TAGs), which is introduced in Jin et al. [2010a, 2010b]. Figure 9 illustrates an example colored TAG containing 2000 edges. The nodes in the TAG stand for the hosts in the network; while each edge describes the interaction (communication) between a pair of hosts (For better visualizing smaller traffic classes, Figure 9(b) is constructed by removing `FileSharing` edges from Figure 9(a). Figure 9(c) removes both `FileSharing` and `Web` edges). Classifying traffic at the level of color TAGs enable us to utilize a set of new features, which is called the collective traffic statistics, which captures the spatial

[10]The increase of overhead during training can be minimized by parallelization and almost no overhead is introduced during operation.

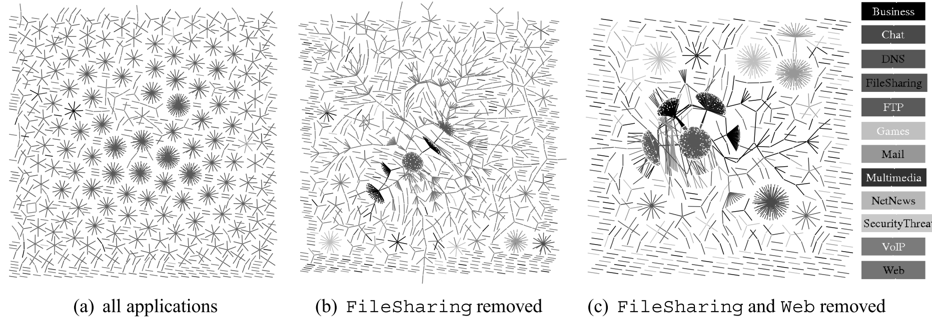


Fig. 9. TAGs containing 2000 edges, where different applications are represented with different colors.

distribution of application classes in the network. For example, we see that the edge colors tend to be clustered together – where edges incident on some nodes are all of the same color – and hence regions of the TAG seem to have the same color. This seems to suggest that certain groups of hosts tend to generate application traffic in a similar way (e.g., exchanging traffic with the same set of web servers), thereby showing up with the same color on the TAG.

To utilize such information incorporated in spatial distribution of application classes for the classification task, we apply a two-step methodology, which was initially developed to improved performance when protocol level information (such as port number) is not available [Jin et al. 2010a]. In this case, a 50% reduction in the error rate was observed. Here we use the two-step method as an additional calibration step. We present the methodology in this section and demonstrate in Section 7 the improvement in both accuracy and robustness to our flow-level traffic classification system after incorporating this new calibration step.

6.1. Problem Definition

Let $\mathcal{G} := \{\mathcal{N}, \mathcal{E}\}$ denote a particular TAG constructed over a specific time period T, where $\mathcal{N}$ denote the set of all hosts in the network and each edge $e_{ij} \in \mathcal{E}$ represents the aggregation of all traffic between host i and host j. In our edge classification problem, we assume that each edge $e_{ij} \in \mathcal{E}$ belongs to one of K predefined application classes, C_k, $1 \leq k \leq K$ (with $K = 12$).[11] However, what class e_{ij} belongs to is unknown and to be determined. Let $L : \mathcal{E} \to \{C_k, 1 \leq k \leq K\}$ denote the edge class mapping, $L(e_{ij}) = C_k$ for some k. Our problem is to infer this edge class mapping L, given the unlabeled $\mathcal{G}$ and the collection of the edge attribute sets, $\{\mathrm{x}_{ij} : e_{ij} \in \mathcal{E}\}$. To solve this problem, we assume a supervised machine learning environment, where we are given a training dataset, that is, a labeled $\mathcal{G}$ (constructed from the traffic within a certain time period) where the class of each edge is given. The inference problem becomes the following learning problem: Can one learn a function f that returns an estimate of the edge class mapping each edge e_{ij} (Eq. (2))

$$\tilde{L}(e_{ij}) = f\left(\mathrm{x}_{ij}, L(e_{i\cdot}), L(e_{\cdot j})\right), \tag{2}$$

[11]We note that 99.5% of the communications between two hosts only involved a single application class throughout a day (either communications are between clients and servers and servers usually host only one type of application, for example, `Web` or `DNS`, or both communicating hosts use p2p `FileSharing` applications). Hence in the model defined here only one traffic class is associated with each edge. In this case, the error rate based on flows is similar to the error rate based on edges in our experiment, we therefore report edge errors when the two-step model is applied. Moreover, the proposed model can be readily extended for inference on multicolored edges.

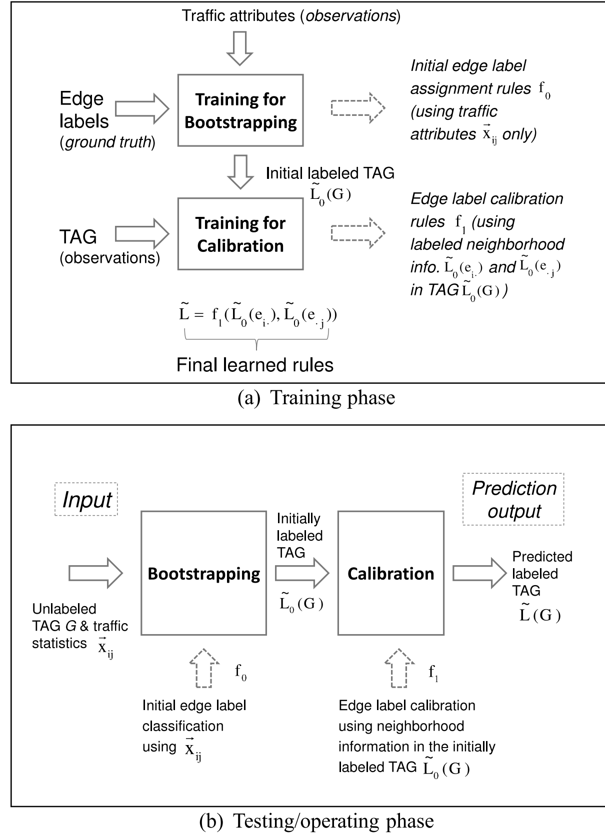


Fig. 10. Architectures for training and testing using the proposed two-step model.

where the traffic features x_{ij} contain the traffic statistics for edge e_{ij}; $e_{i\cdot}, e_{\cdot j} \subset \mathcal{E}$ represent the edges incident on the endpoint h_i and those incident on the endpoint h_j, respectively; the *neighborhood features* $L(e_{i\cdot})$ and $L(e_{\cdot j})$ are obtained through aggregation of the corresponding edge classes.

6.2. Incorporating Collective Traffic Statistics with the Two-Step Model

Exact learning of $\tilde{L}(e_{ij})$ would require the knowledge of all the classes of the neighborhood edges, which are obtainable only when the model $\tilde{L}(e_{ij})$ is known. The simplest approximation based on the training of local classifiers is *iterative classification*, when one starts from an initial estimate $\tilde{L}_0(e_{ij})$ and runs the following iteration t until the edge classes stabilize:

$$\tilde{L}_{t+1}(e_{ij}) = f\left(\mathrm{x}_{ij}, \tilde{L}_t(e_{i\cdot}), \tilde{L}_t(e_{\cdot j})\right). \tag{3}$$

In this article, we propose a simplified variant of the iterative classification algorithm, which we call the two-step model. It offers excellent results while adding little more computational load. The schematic view of the training phase and the testing phase for the proposed model is depicted in Figure 10.

The proposed model consists of two components. The first step, which we refer to as bootstrapping, treats edges features as unknown and infers edge classes according to only the traffic features x_{ij} associated with each edge, regardless of any structural

properties of the TAG. The initial classification from the bootstrapping step is in Eq. (4)

$$\tilde{L}_0(e_{ij}) := f_0(\mathrm{x}_{ij}). \tag{4}$$

Bootstrapping provides us with the initial labels for all edges, though the accuracy of these labels depend on the available traffic information in different application scenarios and hence can be inaccurate in certain situations. In a study where key traffic features (e.g., port numbers, protocol number, etc.) are absent, the bootstrapping step can only reach around 80% accuracy. In comparison, when all traffic features are accessible, the bootstrapping step can achieve an accuracy of over 96%.

The second step, referred to as graph-based calibration or calibration in short, incorporates the inherent neighborhood and local properties of the edges in the TAG to calibrate (re-enforce) the initial edge classification provided by the bootstrapping step. For example, given an edge with class C_1 and all the neighborhood edges with class C_2, the calibration step may follow some edge clustering rule (as observed in Jin et al. [2009]) and change the edge class into C_1. The calibration process is expressed in Eq. (5):

$$\tilde{L}(e_{ij}) := f_1\left(\tilde{L}_0(e_{i\cdot}), \tilde{L}_0(e_{\cdot j})\right). \tag{5}$$

Why do we deprive classifier f_1 from traffic features x_{ij}? The explanation is that we want the classifier to focus on the neighborhood features, which, for a given node or IP address, only change slowly over time. This means that if we use the neighborhood features only, test data that has been collected from the same graph as the training data (but later in time) may still have a distribution that is close to the training data. On the other hand, the traffic features suffer from a much greater time variability, and can become undesirable noise when one has access to neighborhood features. In a preliminary study, we always obtain a higher error rate by incorporating traffic features in the calibration step.

As we do not have traffic features to stabilize the iterative classification process, we found that a single calibration step was enough to obtain the best performance, hence the simplification of the algorithm into a two-step approach. Therefore, from Eq. (5), the edge classification from the proposed model is expressed as a combination of the bootstrapping step and the calibration step. We note that the inference on the class of a particular edge e_{ij} is based on the initial (inaccurate) classification of the neighborhood edges $(\tilde{L}_0(e_{i\cdot}), \tilde{L}_0(e_{\cdot j}))$ from the bootstrapping step. Therefore, the training of the calibration function f_1 also depends on the initial classification provided by the function f_0 in the bootstrapping step, instead of depending on the ground truth. In the following, we discuss the training and operating of the proposed two-step model.

6.3. Training and Operating the Two-Step Model

Taking advantage of the ground truth that we have for the network flow data, we formulate both the bootstrapping step and the calibration step as classical multiclass classification problems. Hence, the bootstrapping function f_0 and the calibration function f_1 correspond to two multiclass classifiers. We note that these two classifiers (f_0 and f_1) only differ in the feature sets, and both of these classifiers can be efficiently learned using the existing machine-learning architecture proposed in Section 3.

Overall Training and Operating Architecture. The training architecture is presented in Figure 10(a) and the whole procedure is summarized in Algorithm 3. Given the ground truth of edge classes in the training data, we first learn a multiclass classifier f_0, which maps traffic features x_{ij} corresponding to each edge e_{ij} to the initial classification $\tilde{L}_0(e_{ij})$. We then generate initial classification for the entire TAG, $\tilde{L}_0(\mathcal{G})$ and learn the classifier

ALGORITHM 3: Training the two-step model.

1: Parameters: Flow set $\mathcal{F}$;
2: Output: edge-level classifier f_0 and graph calibrator f_1;
3: Construct a TAG $\mathcal{G} := \{\mathcal{N}, \mathcal{E}\}$ using $\mathcal{F}$, with each edge $e_{ij} \in \mathcal{E}$ labeled as $L(e_{ij})$;
4: Using $L(e_{ij})$ as ground truth and traffic statistics associated with e_{ij} as features, train f_0;
5: **for** each edge $e_{ij} \in \mathcal{E}$ **do**
6: Classify e_{ij} using f_0 and get $\tilde{L}_0(e_{ij})$;
7: Create collective traffic statistics based on the labels of the neighboring edges $\tilde{L}_0(e_{i\cdot})$ and $\tilde{L}_0(e_{\cdot j})$ using histograms;
8: **end for**
9: Using $L(e_{ij})$ as ground truth and $\tilde{L}_0(e_{i\cdot})$ and $\tilde{L}_0(e_{\cdot j})$ as features, learn f_1;

ALGORITHM 4: Two-step model in operation.

1: Parameters: flow set $\mathcal{F}$, edge-level classifier f_0, graph calibrator f_1;
2: Output: multiclass classification result $\tilde{L}(e_{ij})$;
3: Construct a TAG $\mathcal{G} := \{\mathcal{N}, \mathcal{E}\}$ from $\mathcal{F}$;
4: **for** each edge $e_{ij} \in \mathcal{E}$ **do**
5: Classify e_{ij} using f_0 as $\tilde{L}_0(e_{ij}) = f_0(e_{ij})$;
6: Construct collective traffic statistics from neighboring edges, $\tilde{L}_0(e_{i\cdot})$ and $\tilde{L}_0(e_{\cdot j})$, using histograms;
7: Calibrate the label of e_{ij} as $\tilde{L}(e_{ij}) = f_1(\tilde{L}(e_{i\cdot}), \tilde{L}(e_{\cdot j}))$;
8: **end for**

f_1 for the calibration step, which maps initial classification to the true classification based on the classes of the neighbors of individual edges.

After learning two classifiers f_0 and f_1, at the operating time (Figure 10(b)), given a TAG $\mathcal{G}$ created from the test dataset, we first apply f_0 to obtain initial classes for all the edges in the TAG, namely, $\tilde{L}_0(\mathcal{G})$. We then encode the neighborhood information of all the edges into histograms and apply the classifier f_1 for the calibration purpose, which will produce the final prediction $\tilde{L}(\mathcal{G})$ after calibration. The operation process is summarized in Algorithm 4.

We note that f_0 uses traffic features associated with individual edges. The available traffic features depend on specific applications, which we will discuss in detail in Section 7.6. In the following, we focus on explaining how we encode the neighborhood information as features for constructing f_1.

Encoding Neighborhood Information for Graph-based Calibration. Given the fact that an edge may have an unbounded number of neighborhood edges connected to the end nodes, we encode the neighborhood information as histograms. More specifically, for an edge e_{ij}, let $|C_k|$ denote the number of edges connected to h_i which are labeled as C_k, $1 \leq k \leq K$. We then define K features corresponding to the neighborhood edges connected to the end host h_i as $|C_k|/\sum_j |C_j|$, representing the percentage of edges connected to h_i that are labeled as C_k. Similarly, we define K features to encode the neighborhood edges connected to h_j. In addition, we include the degrees of h_i and h_j as two additional features. For $K = 12$ (the number of predefined application classes in Table I), in both the two application scenarios in this article, we create a total of 26 features to encode the neighborhood information of individual edges. Despite the loss of structural information, encoding objects as histograms has enabled a fast deployment of machine-learning solutions to many real world problems, with surprisingly good results.

6.4. Discussion

In our traffic activity graph (TAG), we want to classify the edges into a set of applications. This differs from most graphs studied in the collective classification literature, where the nodes have to be classified. This difference is not significant, and we assume that most techniques can easily be adapted from edge classification to node classification.

A more significant difference is the way we distinguish the train and test graphs. The two most common formulations [Gallagher and Eliassi-Rad 2007] are *between-network classification*, where the train and test graphs are disconnected, and *within-network classification*, where they are connected, and where the test data usually consists of the unlabeled nodes. Adding a temporal dimension to the graph requires a new types of approach. For instance, to deal with time-stamped IMDB movies [Neville and Jensen 2000], one has to consider a graph where nodes are added over time. In our problem, not only new nodes and edges are created over time, but the edge labels may change. While we treat the test TAG as a different graph from the train TAG, this time dependency will critically influence the choice of our model.

Our two-step approach differs from standard iterative collective classification algorithms in two ways. First, the calibration step is iterated once only, as our experiments show that a second calibration step is always detrimental. Second, the calibration classifier uses graph features only, and does not use traffic features. The reason for this is that it significantly improves performance when the training and the testing data come from the same site, which is our most common setting. In the less common cross site setting, the training and test graphs are clearly separated, corresponding to the standard collective classification setting. In this case, we verify that traffic features help in the calibration step.

7. SYSTEM EVALUATION

In this section, we present the evaluation of the whole modular flow-level traffic classification system. We first discuss the evaluation environment and implementation details. We then evaluate the runtime scalability and training scalability. At the end of the section, we focus on evaluating the accuracy and stability of the system.

7.1. Evaluation Environment

To test real-time performance, we implemented the flow-based classifier on the same machine where the packet-level classifier operates. This machine (referred to as *Test*) directly connects to a Gbps link. The new flow arrival rate on the link ranges from about 200K to 450K new flows per minute, apart from isolated bursts of up to 1.2 million new flows per minute.

The Test machine is a Sun X4200, equipped with 2 dual-core CPUs. The total memory size is 8GB, shared by the 4 cores via dynamic allocation. The Test machine is heavily CPU loaded before we start our system, because traffic collection, flow aggregation and packet level classification on the Gbps link are all performed directly on this machine, and hence we need to set high priority to the aforementioned modules to ensure the proper function of these modules. Therefore, we can only utilize the remaining limited computation power for the flow-level classifier. This enables us to study the impact on the classifier's performance when many applications compete for resources, for example, during the peak hours.

Multicore technology is widely available and multithreading is a common tool for improving performance. Here we evaluate the potential performance gains it can provide for our system. Since the Test machine contains four cores and is also used by other applications, we also conduct the multithreading experiments on a lab machine.

The lab machine is a X4600 M2 machine with eight Quad-Core AMD Opteron processors (2.7 Ghz), which contains 64GB RAM, with 2x 146 GB 10,000 RPM SAS Drives and 2x Single-Channel 4Gb FC-AL HBA PCIx. We copy traffic collected on the Test machine to the lab machine and replay it there for our experiments. The lab machine is also used for training purposes.

7.2. Implementation and Optimization

The training part for our system is implemented using Python. We allow two tunable parameters: the sampling threshold θ and the number of iterations for the *BStump* algorithm. Our system automates the following training process. First, the system creates training data using weighted threshold sampling with parameter θ. It then forks k instances of the Boostexter [Schapire and Singer 2000] implementation of the *BStump* algorithm to train k binary classifiers in parallel for the specified number of iterations. Finally, the system trains k calibrators for these binary classifiers. We note the training time is measured as the running time of all Boostexter threads, no the execution time of the entire Python script.

During the real-time evaluation, we implement the flow-based classification system using C++. The whole system contains 12 TCP and 8 UDP binary classifiers and is parameterized with the number of threads. On the Test machine, due to resource constraints from other applications, we set the number of threads to be one, where the 12 TCP or 8 UDP binary classifications for each flow are conducted sequentially and then combined for the final multiclass prediction. On the lab machine, we vary the number of threads to study the impact of multithreading on the runtime performance of the system, where these binary classification tasks are distributed evenly to multiple threads. More specifically, assume q threads are running in parallel, each one will handle $\lfloor 20/q \rfloor$ to $\lceil 20/q \rceil$ binary classifications.

For better scalability, we further optimize the binary classifier implementation. Originally, each binary classification needs to match T rules, where $T = \sum_1^r T_i$ is the number of decision stumps (weak learners) derived iteratively during training and T_i is the number of decision stumps associated with feature i (in r features). However, the complexity for classification can be reduced from $O(T)$ to $O(\sum_1^r \log T_i)$, where $r \ll T$ in general.

We illustrate the optimization for continuous features; categorical features are treated with a slight modification. Recall that in a binary classifier, each decision stump bi-partitions the real range of a particular feature with a threshold δ. The decision stump assigns a score S_- to the flow if the corresponding feature value falls below δ, and a score S_+ is assigned otherwise. The T scores (from all decision stumps) are summed up as the final prediction for the flow. Since $r \ll T$, instead of doing T rule matches at runtime, we can partition the real range of each feature into intervals according to all the decision stumps associated and precompute the aggregated scores corresponding to each interval. Therefore, at runtime, we can directly obtain the score associated with each feature value by mapping the feature value to the correct interval through a binary search tree (with complexity $O(\log T_i)$). Hence, a maximum of $\sum_1^r \log T_i$ comparisons is needed for each binary classification and a performance gain of 10 to 20 times has been observed in our experiments.

7.3. Runtime Scalability

Real-Time Performance. We test the flow classification rate during the busiest hours (7pm to 9pm) everyday for a week. Recall the new flow arrival rate at the busiest hours is around 450K flows per minute. Without multithreading and with the presence of other heavily loaded applications, our classifier processed up to more than 800K new

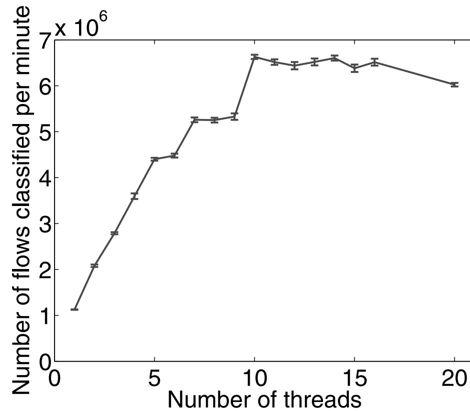


Fig. 11. Flow classification rates vs. number of threads.

flows per minute, which is twice as much as the mean flow rate on the two Gbps links. However, in less than 0.5% of the 1-minute time intervals, bursts of more than 1 million new flows occur. Such bursts can be accommodated by temporarily storing the unprocessed flows to disk before classification. Multithreading can also help, as we now discuss.

Performance Gain with Multithreading. The performance of multithreading will depend on the differing resource requirements of classifiers, the manner in which they are distributed over threads, and the manner in which threads are distributed over cores. We focus on the dependence on number of threads, which we vary from 1 to 20 on the lab machine. We run each experiment for 100 times and report the average and standard deviation of the flow classification rate in Figure 11.

The best performance is achieved when 10 to 14 threads are running in parallel; here the flow-based classifier can scale up to 6.5 million flows per minute, an order of magnitude higher than the normal flow rate and 4 times higher than the flow burst rate (our result is faster than the result of 54.7K flows per second reported by Williams et al. [2006], plausibly accredited to parallelization and our explicit optimization). Assuming a linear scale up, this means our classifier would be able to keep up with new flow arrivals on a 10Gbps link. This result is conservative in the sense that our system would perform at least as well on a platform with no competing applications, and because the optimization we applied were relatively simple. When the number of threads is greater than 14, we observe the general performance drop by adding more threads. We believe this is likely due to the overhead of constantly switching threads among CPUs. An even better performance is expected if each CPU is running one thread exclusively.

7.4. Evaluation on Training Scalability

A major hurdle in the deployment of machine learning solutions is the fact that training time often turns out to be long and unpredictable. We found out that our modular approach based on Adaboost training makes it possible to optimally allocate computer cluster resources to deliver the best performing model under the constraint that training should be executed in T hours.

The key observation is that the training time is at worst linear in the number of iterations, features, and examples. It is obviously linear in the number of iterations. The time spent in each iteration is linear in the number of features, as each of them have

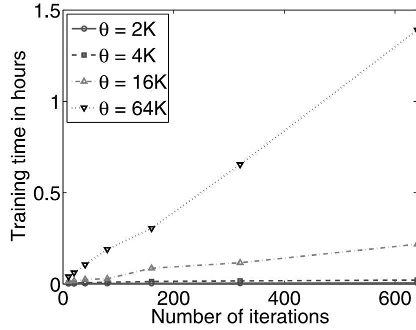


Fig. 12. Training time with different parameters.

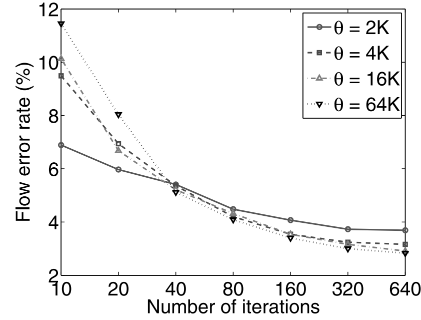


Fig. 13. Flow error rate with different parameters (the x-axis is in log scale).

to be considered for weak classifiers. For each numerical feature, one has to consider all stump threshold candidates, as many as there are training examples in the worst case. Note the number of examples is determined by the parameter θ in the weighted threshold sampling method. Assume the number of features is fixed, we train 12 TCP binary classifiers in parallel and report the training time as the one for the slowest thread. Figure 12 shows the changes of training time by varying the number of iterations and θ, where the x-axis represents the number of iterations and y-axis stands for the training time in hours (note the x-axis is in log scale). The training time is linear to both the number of iterations and the training data size. With parallelization, we can finish training in less than 1.5 hours with 700K training samples ($\theta = 64K$) using 640 iterations.

Using w_2s_1 as the test set, we evaluate the accuracy of the classifier by varying training parameters. In Figure 13, the x-axis represents the number of iterations and the y-axis represents the flow error rate on the test data. A much faster convergence for the flow error rate is expected with a smaller training data size. However, if a higher accuracy is crucial, we need to incorporate more training data and allocate much more computation resource for the training process. For example, if the training time is constrained to be less than 15 minutes, the optimal setting is to use around 180K flows ($\theta = 16K$) for training with the number iterations set to 640. In addition, the maximum achievable accuracy depends (sublinearly) on the training data size. For example, when the number of iterations equals 640, the flow error rate when $\theta = 64K$ (i.e., around 600K training samples in total), the flow error rate is close to 3.1%; in comparison, when $\theta = 2K$ (i.e., around 24K training samples in total), the flow error rate is around 3.7%. Considering the large amount of traffic to be classified in a large ISP network, such a decrease of the number of misclassified flows is worth the cost of a longer training time.

7.5. Evaluation of Stability of Accuracy

This section focuses on evaluating the temporal and spatial stability of classifier accuracy. Robustness of learning across different times and measurement locations reduces costs by limiting the need for frequent retraining and enabling the use of a single model for different locations. Here we observe the degree of performance degradation when the testing and training sets are drawn from a different time periods or locations.

Temporal Stability: Cross-Time Evaluation. To test the cross-time stability of our classifier, we train three different classifiers using the 1M training data from w_1s_1, w_2s_1 and w_3s_1, respectively, and testing on data from the subsequent weeks from site 1.

Table VIII. Cross-Site Error Rates (as Percent)

	w_1s_1-w_2s_1	w_1s_1-w_5s_2	w_5s_2-w_1s_1	w_5s_2-w_6s_2
TCP ER	3.13	2.03	3.48	1.80
TCP BER	29.3	20.0	33.4	23.8
UDP ER	0.35	0.70	0.75	0.46
UDP BER	17.0	39.4	27.5	37.3

In a two-month time interval, the flow error rates are $3 \pm 0.5\%$ for TCP traffic and $0.4 \pm 0.04\%$ for UDP traffic, even with up to a 6-week gap between the collection of the training and testing data sets. There is a slight ascending trend in the flow error rate when the time gap between the training data set and the testing data set becomes larger. This reveals the inherent change of network traffic distribution, which is against the IID assumption of the traffic classifier and results in the growing of error rates. For a longer period stability test, we recently collected 7 weeks data from 04/12/2009 to 05/30/2009. After one year, our classifier still achieves a TCP flow error rate of $5.48 \pm 0.54\%$ and a UDP flow error rate of $1.22 \pm 0.2\%$. The BER remains stable in both experiments.

Spatial Stability: Cross-Site Evaluation. We selected two weeks of flow data from each of two geographically separated sites, w_1s_1, w_2s_1, w_2s_1 and w_2s_2. Both sites are parts of a large ISP network.[12] Note there is also a month time-gap between collection dates of the first week data at the two sites. We label each experiment as w_is_j-w_ps_q, in which a 1M training data set is created from data set w_is_j and the entire w_is_j and w_ps_q sets is used for calibration and testing purpose, respectively.

We summarize the cross-site evaluation in Table VIII. The flow-level classifier exhibits a strong spatial stability, with the TCP flow error rates less than 3.5% and the UDP flow error rates less than 0.75%. There are slight increases of the error rates for UDP traffic when tested on the second site, as well as a little decrease of the error rates for TCP traffic. Details of per-class error rates reveal that the major causes of the error rate increase are `Business` and `Multimedia` for TCP traffic and `Business`, `FileSharing` and `Games` for UDP traffic. This, in some sense reflects, the locality property of traffic data, since the classification accuracy for "global" activities such as `Mail` and `Web` remains stable.

In summary, the classifier exhibits strong temporal and spatial stability across different sites and temporal separations of up to 6 weeks. In particular, these timescale are far longer than the 2-hour training time for the classifier reported in Section 5.3. The stability of our flow-level classifier is likely attributed to the implicit L_1 regularization of *BStump*, which eliminates the features that are unstable.

7.6. Adding Collective Traffic Statistics

In this section, we add a calibration step to follow the two-step methodology presented in Section 6. We discuss the implementation details of this step and report performance improvements.

Flow-Level Bootstrapping. We note that our proposed approach is designed for the inference on the edges in a TAG. However, in order to utilize flow-level features like port numbers, we need to conduct the bootstrapping step at the flow-level. After classifying

[12]When two sites are different in nature, for example, one is ISP network and the other one is a campus network, a new calibrator shall be trained to address the difference in the traffic class distribution. The classifier, since it depends only on characteristics of application classes, which are expected to be stable across locations and change slowly over time, does not require retraining.

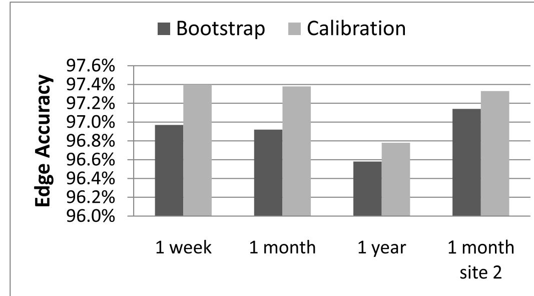


Fig. 14. Accuracy and stability.

flow records with the features in Table III, we choose the dominant flow predictions as the predictions of individual edges. In particular, the dominant prediction of e_{ij} is defined as the predicted traffic class that is associated with the largest number of flows observed on e_{ij}. Similarly, we use the dominant flow-level labels as the ground truth for the edges of the calibration method.

Implementation of the Calibration Step. The proposed method can be implemented in a real-time manner. The basic idea is to maintain a historical TAG for the calibration step. More precisely, let T be the time window length to construct the TAG for the calibration step. To classify an edge e_{ij} in real-time at t_0, we need to maintain the traffic statistics and neighborhood edges of e_{ij} in a past time window from $t_0 - T$ to t_0. By maintaining the edge histogram features as running averages, their computation cost is independent of the window size. In general, the number of flows collected within a certain time window is approximately 4 times the number of edges within the same time window. Furthermore, due to the lower number of features and rules for the graph-based calibration step, calibrating one edge only requires 1/3 the computation time for classifying a flow record. In other words, turning on the graph-calibration step only results in 8% (1/12) additional computation overhead. In this case, our traffic classification system can still scale up with 10Gbps links. Similarly, the additional training cost for the new calibrator is also small. In practice, around 1/4 iterations is required compared to the time for building the original flow-level traffic classifier.

Accuracy and Stability. Using a one-hour dataset (05/03/2008 10-11AM) for training and three datasets from the same site that are 1 week, 1 month and 1 year away from the training dataset for testing, we evaluate the accuracy and the stability of our approach. In addition, we employ a one-hour dataset collected from the second site 1 month later from the training dataset.

Figure 14 shows the accuracy after bootstrapping and calibration for the four testing datasets mentioned above. Interestingly, the calibration step still increases the accuracy by 0.5% from the bootstrapping step (14% to 15% reduction in the errors) even after the one-month time period. Even when the time gap between the training data and the test data is one year apart or when the two datasets are collected from two different sites, we still observe a 5% to 6% error rate reductions through calibration.

Per-Class F1 Scores. We also analyze the per-class classification performance by looking at the per-class F1 score from the bootstrapping step and the calibration step. Figure 15 shows all F1 scores corresponding to different traffic classes. Though the F1 scores slightly drop for *FTP* and *Games*, overall the F1 scores increase for all traffic

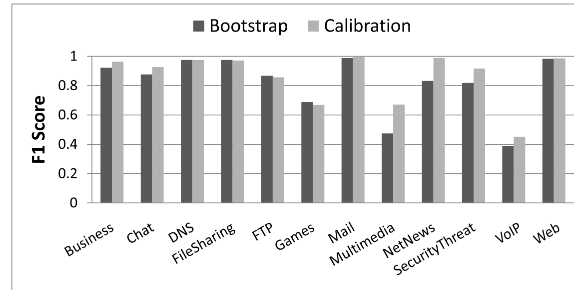


Fig. 15. F1 scores.

classes. This again indicates that the graph information on TAGs indeed boosts the accuracy of all traffic classes and hence the overall traffic classification accuracy.

8. CONCLUSIONS

In this article, we designed a machine learning-based flow-level traffic classification system as a solution to the large-scale network traffic classification problem. Utilizing a novel modular architecture, our system decomposed the classification task into multiple subtasks and employed classical machine-learning algorithms as well as weighted threshold sampling, calibration and intelligent data partitioning methods to solve these subtasks to achieve both accuracy and scalability. Through extensive online and offline experiments, we demonstrated that our system attained a low-flow error rates of only 3% of TCP and and 0.4% for UDP traffic, which remain stable for a few months. Furthermore, such error rate was further reduced by 15% with collective traffic statistics using our novel two-step approach. More importantly, on a heavily loaded machine with one thread and simple optimization, our system can scale up with the new flow arrival rate on two 1Gbps links in an operational ISP network. With the help of multithreading on multicore machines, our system can easily handle traffic classification on 10Gbps links. We note that further parallelization methods can be designed for improving the scalability.

The flow records employed for this study were not packet sampled, that is, all packet headers from each sampled flow contributed to the aggregate flow record. Their flow-specific features (packet/byte counts and duration) will have a different distribution to flow records produced by (packet) Sampled NetFlow in wide use. Our main future work is to adapt our system to learn flow based classifiers for packet sampled flows, that can then be applied directly to classify Sampled NetFlow measurements. We also plan to develop an automatic feature selection method that focus deliberately on improving stability. Our preliminary results show that by removing *tos* and *tosnumbyte*, we decrease the runtime complexity by 13.3% for TCP and 14.3% for UDP. In addition, the TCP flow/byte error rates are reduced by 16%/21% and the UDP error rates remain unchanged. We shall further refine this feature selection method.

APPENDIX

Appendix: Classifying Unknown Flows

The efficacy of traffic classification is limited by packet encryption and the lack of rules for emerging applications, resulting in a subset set of flow labeled as `Unknown` by the packet-level classifier. In contrast, the flow-level features are always available, allowing our system to make inference concerning the class of these `Unknown` flows. In this section, we evaluate the performance of our flow-level classifier on `Unknown` flows.

Constructing Training/Evaluation Sets. For evaluation, we must furnish as set of flows that are `Unknown` at the packet-level, but to which we can, nevertheless, attribute a class. We achieved this by constructing the evaluation set from the second type of `Unknown` flows. In particular, using two versions of the packet-level classifier developed at different times. The first one is from 07/07/2008 (R_0), while the second one is from 04/13/2009 (R_1). Compared with R_0, R_1 contains 15,545 more packet-level rules covering 63 diverse new applications–such as `Games/wii`, `Web/iphone`, `FileSharing/pando`–distributed over all 12 of the broad application classes. Most of the new rules and associated traffic relate to TCP, hence we only focus on TCP traffic in the experiment. Therefore, R_1 can classify a portion of flows (from new applications) that is "unknown" to R_0! We utilize this portion of flows as our evaluation set. In a more formal way, let $\mathcal{F}$ be a set of flows. By applying R_0 on $\mathcal{F}$, we obtain $\mathcal{F} = \mathcal{K} \cup \mathcal{U}$, where $\mathcal{K}$ and $\mathcal{U}$ represent the flows that can be classified by R_0 and the flows unknown to R_0, respectively. We then run R_1 on $\mathcal{U}$. R_1 can classify an additional flow set $\mathcal{U}_E \subset \mathcal{U}$, which we use as the evaluation set. In our experiments, we construct the evaluation set $\mathcal{U}_E$ from two whole-day flow sets d_1s_2 and d_2s_2 (Table II), which accounts for 0.7% of the total flows.

Classification Results on Unknown Flows. We train the classifier using dataset w_1s_1 (with ground truth from R_0) in the default way and test on the evaluation set $\mathcal{U}_E$ mentioned previously. The overall flow error rate on the evaluation set is 24.0%, a good result on these `Unknown` flows which otherwise can not be classified at all.

Examining classification results for individual applications in $\mathcal{U}_E$, the system achieves a close to 0 flow error rate on most applications. This reflects that most of the new applications in each broad class have similar flow feature values to other applications in the same class, and our flow-level classification system can take advantage of this to accurately classify these new applications. We do observe a large error rate on `Games/wii`, which is almost all classified into `Web` class, resulting an error rate of 25.7% for the `Web` class in $\mathcal{U}_E$. In fact, this is also the major contributor to the large overall flow error rate, since removing `Games/wii` results in an overall error rate of only 3.7%. We suspect this is because the traffic characteristics of `Games/wii` are significantly different from other applications in `Games/wii`. We validate this by retrain the classifier using the ground truth from R_1 and the overall flow error rate reduces to 2.8%.

REFERENCES

BERNAILLE, L., TEIXEIRA, R., AND SALAMATIAN, K. 2006. Early application identification. In *Proceedings of CoNext'06*. ACM.

BUT, J., NGUYEN, T., STEWART, L., WILLIAMS, N., AND ARMITAGE, G. 2007. Performance analysis of the angel system for automated control of game traffic prioritisation. In *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support For Games (NetGames)*. 123–128.

CHEN, A., JIN, Y., CAO, J., AND LI, L. 2010. Tracking long duration flows in network traffic. In *Proceedings of the 29th Conference on Information Communications (INFOCOM)*. 206–210.

CROTTI, M., DUSI, M., GRINGOLI, F., AND SALGARELLI, L. 2007. Traffic classification through simple statistical fingerprinting. *ACM SIGCOMM Comput. Comm. Rev. 37,* 1, 5–16.

DUDA, R. O., HART, P. E., AND STORK, D. G. 2000. *Pattern Classification*. Wiley-Interscience.

ERMAN, J., MAHANTI, A., ARLITT, M. F., COHEN, I., AND WILLIAMSON, C. L. 2007. Offline/realtime traffic classification using semi-supervised learning. *Perform. Eval. 64*, 9–12, 1194–1213.

FREUND, Y. AND SCHAPIRE, R. E. 1995. A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the 2nd European Conference on Computational Learning Theory (EuroCOLT)*.

GALLAGHER, B. AND ELIASSI-RAD, T. 2007. An examination of experimental methodology for classifiers of relational data. In *Proceedings of the 7th IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE Computer Society Press, 411–416.

HAFFNER, P., SEN, S., SPATSCHECK, O., AND WANG, D. 2005. ACAS: Automated Construction of Application Signatures. In *Proceedings of the SIGCOMM Workshop on Mining Network Data (MineNet)*. ACM.

ILIOFOTOU, M., PAPPU, P., FALOUTSOS, M., MITZENMACHER, M., SINGH, S., AND VARGHESE, G. 2007. Network monitoring using traffic dispersion graphs (TDGS). In *Proceedings of the ACM Internet Measurement Conference (IMC)*.

ILIOFOTOU, M., FALOUTSOS, M., AND MITZENMACHER, M. 2009a. Exploiting dynamicity in graph-based traffic analysis: techniques and applications. In *Proceedings of CoNext'09*. ACM.

ILIOFOTOU, M., KIM, H., FALOUTSOS, M., MITZENMACHER, M., PAPPU, P., AND VARGHESE, G. 2009b. Graph-based p2p traffic classification at the internet backbone. In *Proceedings of the 28th IEEE International Conference on Computer Communications Workshops (INFOCOM)*. 37–42.

JACOBS, R. A., JORDAN, M. I., NOWLAN, S. J., AND HINTON, G. E. 1991. Adaptive mixture of local experts. *Neural Computat. 3*, 79–87.

JIANG, H., MOORE, A. W., GE, Z., JIN, S., AND WANG, J. 2007. Lightweight application classification for network management. In *Proceedings of the 2007 SIGCOMM Workshop on Internet Network Management (INM)*. ACM.

JIN, Y., SHARAFUDDIN, E., AND ZHANG, Z.-L. 2009. Unveiling core network-wide communication patterns through application traffic activity graph decomposition. In *Proceedings of SIGMETRICS'09*. 49–60.

JIN, Y., DUFFIELD, N., HAFFNER, P., SEN, S., AND ZHANG, Z.-L. 2010a. Inferring applications at the network layer using collective traffic statistics. In *Proceedings of the 22nd International Teletraffic Congress (ITC'22)*.

JIN, Y., DUFFIELD, N., HAFFNER, P., SEN, S., AND ZHANG, Z.-L. 2010b. Inferring applications at the network layer using collective traffic statistics (extended abstract). In *Proceedings of ACM SIGMETRICS'10*.

KARAGIANNIS, T., BROIDO, A., FALOUTSOS, M., AND CLAFFY, K. 2004. Transport layer identification of P2P traffic. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.

KARAGIANNIS, T., PAPAGIANNAKI, K., AND FALOUTSOS, M. 2005. BLINC: Multilevel traffic classification in the dark. In *Proceedings of ACM SIGCOMM'05*. ACM.

MA, J., LEVCHENKO, K., KREIBICH, C., SAVAGE, S., AND VOELKER, G. M. 2006. Unexpected means of protocol inference. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.

MCDANIEL, P., SEN, S., SPATSCHECK, O., DER MERWE, J. V., AIELLO, B., AND KALMANEK, C. 2006. Enterprise security: A community of interest based approach. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS)*.

MOORE, A. W. AND ZUEV, D. 2005. Internet traffic classification using Bayesian analysis techniques. In *Proceedings of ACM SIGMETRICS'05*.

NEVILLE, J. AND JENSEN, D. 2000. Iterative classification in relational data. In *Proceedings of the AAAI Workshop on Learning Statistical Models from Relational Data*. AAAI.

NGUYEN, T. AND ARMITAGE, G. 2006a. Synthetic sub-flow pairs for timely and stable IP traffic identification. In *Proceedings of the Australian Telecommunication Networks and Application Conference*. 293–297.

NGUYEN, T. AND ARMITAGE, G. 2006b. Training on multiple sub-flows to optimise the use of machine learning classifiers in real-world IP networks. In *Proceedings of the 31st Conference on Local Computer Networks*. IEEE.

PHILLIPS, S. J., DUDÍK, M., AND SCHAPIRE, R. E. 2004. A maximum entropy approach to species distribution modeling. In *Proceedings of the 21st International Conference on Machine Learning (ICML)*. ACM.

PLATT, J. 1999. Probabilistic outputs for support vector machines and comparison to regularized likelihood methods. In *Proceedings of the 13th Conference on Neural Information Processing Systems (NIPS)*.

RIFKIN, R. AND KLAUTAU, A. 2004. In defense of one-vs-all classification. *J. Mach. Learn. Res.*, 101–141.

SCHAPIRE, R. E. AND SINGER, Y. 2000. Boostexter: A boosting-based system for text categorization. *Mach. Learn. 39*, 2–3, 135–168.

SEN, S., SPATSCHECK, O., AND WANG, D. 2004. Accurate, scalable in-network identification of P2P traffic using application signatures. In *Proceedings of the 13th International World Wide Web Conference (WWW)*. ACM.

SEN, P., NAMATA, G., BILGIC, M., GETOOR, L., GALLAGHER, B., AND ELIASSI-RAD, T. 2008. Collective classification in network data. *AI Mag. 29*, 3.

TRESTIAN, I., RANJAN, S., KUZMANOVI, A., AND NUCCI, A. 2008. Unconstrained endpoint profiling (Googling the Internet). In *Proceedings of ACM SIGCOMM '08*.

WILLIAMS, N., ZANDER, S., AND ARMITAGE, G. 2006. A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. *SIGCOMM Comput. Comm. Rev. 36*, 5–16.

WITTEN, I. H. AND FRANK, E. 1999. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann.

XU, K., ZHANG, Z.-L., AND BHATTACHARYYA, S. 2005. Profiling internet backbone traffic: Behavior models and applications. In *Proceedings of ACM SIGCOMM*.

Received June 2010; revised March 2011, June 2011; accepted June 2011