An Internet Protocol Address Clustering Algorithm

Robert Beverly
MIT CSAIL
rbeverly@csail.mit.edu

Karen Sollins
MIT CSAIL
sollins@csail.mit.edu

ABSTRACT

We pose partitioning a b-bit Internet Protocol (IP) address space as a supervised learning task. Given (*IP, property*) labeled training data, we develop an IP-specific clustering algorithm that provides accurate predictions for unknown addresses in $O(b)$ run time. Our method offers a natural means to penalize model complexity, limit memory consumption, and is amenable to a non-stationary environment. Against a live Internet latency data set, the algorithm outperforms IP-naïve learning methods and is fast in practice. Finally, we show the model's ability to detect structural and temporal changes, a crucial step in learning amid Internet dynamics.

1. INTRODUCTION

Learning has emerged as an important tool in Internet system and application design, particularly amid increasing strain on the architecture. For instance, learning is used to great effect in filtering e-mail [12], mitigating attacks [1], improving performance [9], etc. This work considers the common task of clustering Internet Protocol (IP) addresses.

With a network oracle, learning is unnecessary and predictions of e.g. path performance or botnet membership, are perfect. Unfortunately, the size of the Internet precludes complete information. Yet the Internet's physical, logical and administrative boundaries [5, 7] provide structure which learning can leverage. For instance, sequentially addressed nodes are likely to share congestion, latency and policy characteristics, a hypothesis we examine in §2.

A natural source of Internet structure is Border Gateway Protocol (BGP) routing data [11]. Krishnamurthy and Wang suggest using BGP to form clusters of topologically close hosts thereby allowing a web server to intelligently replicate content for heavy-hitting clusters [8]. However, BGP data is often unavailable, incomplete or at the wrong granularity to achieve reasonable inference. Service providers routinely advertise a large routing aggregate, yet internally demultiplex addresses to administratively and geographically disparate locations. Rather than using BGP, we focus on an agent's ability to *infer network structure from available data*.

Previous work suggests that learning network structure is effective in forming predictions in the presence of incomplete information [4]. An open question, however, is how to properly accommodate the Internet's frequent structural and dynamic changes. For instance, Internet routing and physical topology events change the underlying environment on large-time scales while congestion induces short-term variance. Many learning algorithms are not amenable to on-line operation in order to handle such dynamics. Similarly, few learning methods are Internet centric, i.e. they do not incorporate domain-specific knowledge.

We develop a supervised address clustering algorithm that imposes a partitioning over a b-bit IP address space. Given training data that is sparse relative to the size of the 2^b space, we form clusters such that addresses within a cluster share a property (e.g. latency, botnet membership, etc.) with a statistical guarantee of being drawn from a Gaussian distribution with a common mean. The resulting model provides the basis for accurate predictions, in $O(b)$ time, on addresses for which the agent is oblivious.

IP address clustering is applicable to a variety of problems including service selection, routing, security, resource scheduling, network tomography, etc. Our hope is that this building block serves to advance the *practical* application of learning to network tasks.

2. THE PROBLEM

This section describes the learning task, introduces network-specific terminology and motivates IP clustering by finding extant structural locality in a live Internet experiment.

Let $\mathbf{Z} = (x_1, y_1) \dots (x_n, y_n)$ be training data where each x_i is an IP address and y_i is a corresponding real or discrete-valued property, for instance latency or security reputation. The problem is to determine a model $f : X \to Y$ where f minimizes the prediction error on newly observed IP values.

Beyond this basic formulation, the non-stationary nature of network problems presents a challenging environment for machine learning. A learned model may produce poor predictions due to either structural changes or dynamic conditions. A structural change might include a new link which influences some destinations, while congestion dynamics might temporarily influence predictions.

In the trivial case, an algorithm can remodel the world by purging old information and explicitly retraining. Complete relearning is typically expensive and unnecessary when only a portion of the underlying environment has changed. Further, even if a portion of the learned model is stale and providing inaccurate results, *forgetting stale training data may lead to even worse performance.* We desire an algorithm where the underlying model is easy to update on a continual basis and maintains acceptable performance during updates. As shown in §3, these Internet dynamics influences our selection of data structures.

2.1 Terminology

IPv4 addresses are 32-bit unsigned integers, frequently represented as four "dotted-quad" octets (`A.B.C.D`). IP routing and address assignment uses the notion of a prefix. The bit-wise AND between a prefix p and a netmask m denotes the network portion of the address (m effectively masks the "don't care" bits). We employ the common notation p/m as containing the set of b-bit IP addresses inclusive of:

$$p/m := [p, p + 2^{b-m} - 1] \tag{1}$$

For IPv4 $b = 32$, thus p/m contains 2^{32-m} addresses. For example, the prefix 2190476544/24 (130.144.5.0/24) includes 2^8 address from 130.144.5.0 to 130.144.5.255.

We use latency as a per-IP property of interest to ground our discussion and experiments. One-way latency between two nodes is the time to deliver a message, i.e. the sum of delivery and propagation delay. Round trip time (RTT) latency is the time for a node to deliver a message and receive a reply.

2.2 Secondary Network Structure

To motivate IP address clustering, and demonstrate that learning is feasible, we first examine our initial hypothesis: sufficient secondary network structure exists upon which to learn. We focus on network latency as the property of interest, however other network properties are likely to provide similar structural basis, e.g. hop count, etc.

Let distance d be the numerical difference between two addresses: $d(a_1, a_2) = |a_1 - a_2|$. To understand the correlation between RTT and d, we perform active measurement to gather live data from Internet address pairs. For a distance d, we find a random pair of hosts, (a_1, a_2), which are alive, measurable and separated by d. We then measure the RTT from a fixed measurement node to a_1 and a_2 over five trials.

We gather approximately 30,000 data points. Figure 1 shows the relationship between address pair distance and their RTT latency difference. Additionally, we include a *rnd* distance that represents randomly chosen address pairs, irrespective of their distance apart. Two random addresses have less than a 10% chance of agreeing within 10% of each other. In contrast, adjacent addresses ($d = 2^0$) have a greater than 80% probability of similar latencies within 20%. The average disagreement between nodes within the same class C ($d = 2^8$) is less than 15%, whereas nodes in different /8 prefixes disagree by 50% or more.

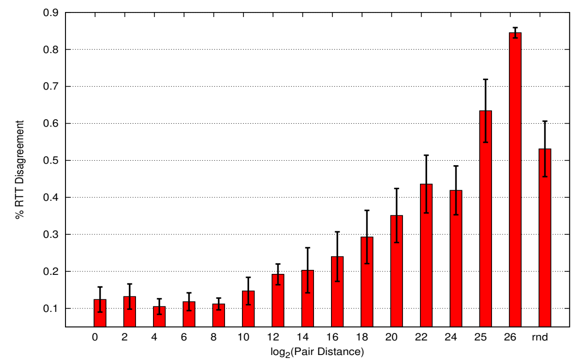


Figure 1: Relationship between d-distant hosts and their RTT latency from a fixed measurement point.

3. CLUSTERING ALGORITHM

3.1 Overview

Our algorithm takes as input a network prefix (p/m) and n training points ($\mathbf{Z}$) where x_i are distributed within the prefix. The initial input is typically the entire IP address space (0.0.0.0/0) and all training points.

Define split s as inducing 2^s partitions, p_j, on p/m. Then for $j = 0, \ldots, 2^s - 1$:

$$p_j = p + j2^{32-(m+s)}/(m + s) \tag{2}$$

Let $x_i \in p_j$ iff the address of x_i falls within prefix p_j (Eq. 1). The general form of the algorithm is:

1. Compute mean of data point values: $\mu = \frac{1}{n}\sum y_i$
2. Add the input prefix and associated mean to a radix tree (§3.2): $R \leftarrow R + (p/m, \mu)$
3. Split the input prefix to create potential partitions (Eq. 2): Let $p_{s,j}$ be the j'th partition of split s.
4. Let N contain y_k for all $x_k \in p_{s,j}$, let M be y_i for $x_i \notin p_{s,j}$. Over each split granularity (s), evaluate the t-statistic for each potential partition j (§3.3): $t_{s,j} = ttest(N, M)$.
5. Find the partitioning that minimizes the t-test: $(\hat{s}, \hat{j}) = \underset{s,j}{\operatorname{argmin}}\ t_{s,j}$
6. Recurse on the maximal partition(s) induced by $(\hat{s}, \hat{j})$ while the t-statistic is less than *thresh* (§3.5).

Before refining, we draw attention to several properties of the algorithm that are especially important in dynamic environments:

• *Complexity:* A natural means to penalize complexity. Intuitively, clusters representing very specific prefixes, e.g. /30's, are likely over-fitting. Rather than tuning traditional machine learning algorithms indirectly, limiting the minimum prefix size corresponds directly to network generality.

• *Memory:* A natural means to bound memory. Because the tree structure provides longest-match lookups, the algorithm can sacrifice accuracy for lower memory utilization by bounding tree depth or width.

• *Change Detection:* Allows for direct analysis on tree nodes. Analysis on these individual nodes can determine if part of the underlying network has changed.

• *On-Line Learning:* When relearning stale information, the longest match nature of the tree implies that once information is discarded, in-progress predictions will use the next available longest match which is likely to be more accurate than an unguided prediction.

• *Active Learning:* Real training data is likely to produce an unbalanced tree, naturally suggesting active learning. While guided learning decouples training from testing, sparse or poorly performing portions of the tree are easy to identify.

3.2 Cluster Data Structure

A radix, or Patricia [10], tree is a compressed tree that stores strings. Unlike normal trees, radix tree edges may be labeled with multiple characters thereby providing an efficient data structure for storing strings that share common prefixes.

Radix trees support lookup, insert, delete and find predecessor operations in $O(b)$ time where b is the maximum length of all strings in the set. By using a binary alphabet, strings of $b = 32$ bits and nexthops as values, radix trees support IP routing table longest match lookup, an approach suggested by [13] and others. We adopt radix trees to store our algorithm's inferred structure model and provide predictions.

3.3 Evaluating Potential Partitions

Student's t-test [6] is a popular test to determine the statistical significance in the difference between two sample means. We use the t-test in our algorithm to evaluate potential partitions of the address space at different split granularity. The t-test is useful in many practical situations where the population variance is unknown and the sample size too small to estimate the population variance.

3.4 Network Boundaries

Note that by Eq. 1, the number of addresses within any prefix (p/m) is always a power of two. Additionally, a prefix implies a contiguous group of addresses under common administration. A naïve algorithm may assume that two contiguous ($d = 1$) addresses, $a_1 = 318767103$ and $a_2 = 318767104$, are under common control. However, by taking prefixes and

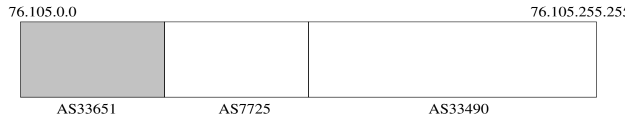


Figure 2: True allocation of `76.105.0.0/16`. Maximal valid prefix splits ensure generality.

Table 1: Examples of maximal IP prefix division

128.61.0.0 $\rightarrow$ 128.61.255.255	128.61.0.0 $\rightarrow$ 128.61.4.1	16.0.0.0 $\rightarrow$ 40.127.255.255
128.61.0.0/16	128.61.0.0/22 128.61.4.0/31	16.0.0.0/4 32.0.0.0/5 40.0.0.0/9

address allocation into account, an educated observer notices that: a_1 (18.255.255.255) and a_2 (19.0.0.0) can only be under common control if they belong to the large aggregate 18.0.0.0/7. A third address $a_3 = 18.255.255.155$, separated by $d(a_1, a_3) = 100$, is further from a_1, but more likely to belong with a_1 than is a_2.

We incorporate this domain-specific knowledge in our algorithm by inducing splits on power of two boundaries and ensuring maximal prefix splits.

3.5 Maximal Prefix Splits

Assume the t-test procedure identifies a "good" partitioning. The partition defines two chunks (not necessarily contiguous), each of which contains data points with statistically different characteristics. We ensure that each chunk is valid within the constraints in which networks are allocated.

Definition 1. *For b-bit IP routing prefixes* $p/m; p \in \{0,1\}^b$ $m \in [0, b]$ *is valid iff* $p = p \,\&\, \left(2^b - 2^{b-m}\right)$.

If a chunk of address space is not valid for a particular partition, it must be split. We therefore introduce the notion of *maximal valid prefixes* to ensure generality.

Consider the prefix 76.105.0.0/16 in Figure 2. Say the algorithm determines that the first quarter of this space (shaded) has a property statistically different from the rest (unshaded). The unshaded three-quarters of addresses from 76.105.64.0 to 76.105.255.255 is not valid. The space could be divided into three equally sized 2^{14} valid prefixes. However, this naïve choice is wrong; in actuality the prefix is split into three different autonomous systems (AS). The IP address registries list 76.105.0.0/18 as being in Sacramento, CA, 76.105.64.0/18 as Atlanta, GA and 76.105.128.0/17 in Oregon. Using maximally sized prefixes captures the true hierarchy as well as possible given sparse data.

We develop an algorithm to ensure maximal valid prefixes along with proofs of correctness in [3], but omit details here for clarity and space conservation. The intuition is to determine the largest power of two chunk that could potentially fit into the address space. If a valid starting position for that chunk exists, it recurses on the remaining sections. Otherwise, it divides the maximum chunk into two valid pieces. Table 1 gives three example divisions.

3.6 Full Algorithm

Using the radix tree data structure, t-test to evaluate potential partitions and notion of maximal prefixes, we give the complete algorithm. Our formulation is based on a divisive approach; agglomerative techniques that build partitions up are a potential subject for further work. Algorithm 1 takes a prefix p/m along with the data samples for that prefix: $\mathbf{Z} = (\mathbf{x}, \mathbf{y}) \forall x_i \in p/m$. The threshold defines a cutoff for the t-test significance and is notably the only parameter.

Algorithm 1 *split*$(p/m, Z, thresh)$:

R, an IP prefix table
$b \leftarrow 32 - m$
$\mu \leftarrow \text{mean}(\mathbf{x})$
$R \leftarrow R + (p/m, \mu)$
5: **for** $i \leftarrow 1$ to $32 - m$ **do**
  for $j \leftarrow 0$ to $2^i - 1$ **do**
    $p_j \leftarrow p + j2^{b+i}/(m-i)$
    for $x \in X$ **do**
      if $x_{ip} \in p_j$ **then**
10:        $N \leftarrow N + x_{ip}$
      else
        $M \leftarrow M + x_{ip}$
      $t_{i,j} \leftarrow \text{ttest}(N, M)$
$t_{best}, i_{best}, j_{best} \leftarrow \underset{i,j}{\text{argmin}}\ t_{i,j}$
15: **if** $t_{best} < thresh$ **then**
  $last \leftarrow p + 2^b - 1$
  $start \leftarrow p + (j_{best})2^{b+i_{best}}$
  $end \leftarrow start + 2^{b+i_{best}} - 1$
  $P \leftarrow start/(m - i_{best})$
20: **if** $start = p$ **then**
    $P \leftarrow P+$ divide$(end + 1,\ last)$
  else if $end = last$ **then**
    $P \leftarrow P+$ divide$(p,\ start - 1)$
  else
25:    $P \leftarrow P+$ divide$(end + 1,\ last)$
    $P \leftarrow P+$ divide$(p,\ start - 1)$
for $p_d/m_d \in P$ **do**
  $Z_d \leftarrow (x_i, y_i) \forall x_i \in p_d/m_d$
  split$(p_d/m_d, Z_d, thresh)$
30: return R

The algorithm computes the mean μ of the y input and adds an entry to radix table R containing p/m pointing to μ (lines 1-4). In lines 5-12, we create partitions p_j at a granularity of s_i as described in Eq. 2. For each $p_{i,j}$, line 13 evaluates the t-test between points within and without the partition. Thus, for s_3, we divide p/m into eighths and evaluate each partition against the remaining seven. We determine the lowest t-test value t_{best} corresponding to split i_{best} and partition j_{best}.

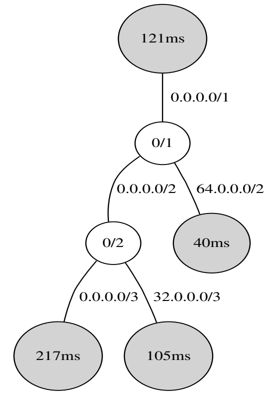


Figure 3: Example radix tree cluster representation

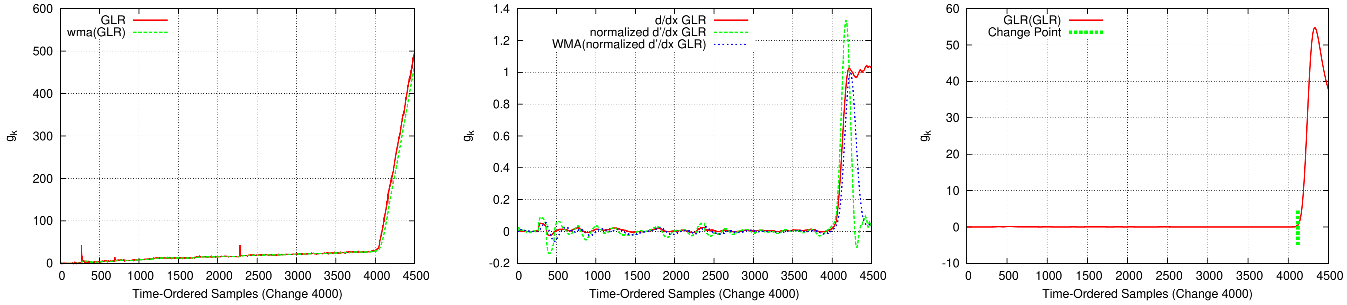
If no partition produces a split with t-test less than a threshold, we terminate that branch of splitting. Otherwise, lines 16-25 divide the best partition into maximal valid prefixes (§3.5), each of which is placed into the set P. Finally, the algorithm recurses on each prefix in P.

The output after training is a radix tree which defines clusters. Subsequent predictions are made by performing longest prefix matching on the tree. For example, Figure 3 shows the tree structure produced by our clustering on input $\mathbf{Z} =$ (18.26.0.25, 215.0), (18.192.1.34, 205.0), (60.1.2.3, 100.0), (60.99.2.4, 110.0), (69.4.5.6, 45.0), (70.4.5.6, 39.0).

4. HANDLING NETWORK DYNAMICS

An important feature of the algorithm is its ability to accommodate network dynamics. However, first the system must detect changes in a principled manner. Each node of the radix tree naturally represents a part of the network structure, e.g. Figure 3. Therefore, we may run traditional change point detection [2] methods on the prediction error of data points classified by a particular tree node. If the portion of the network associated with a node exhibits structural or dynamic changes, evidenced as a change in prediction error mean or variance respectively, we may associate a cost with retraining. For instance, pruning a node close to the root of the tree represents a large cost which must be balanced by the magnitude of predictions errors produced by that node.

When considering structural changes, we are concerned with a change in the mean error resulting from the prediction process. Assume that predictions produce errors from a Gaussian distribution $N(\mu_0, \sigma_0)$. As we cannot assume, a priori, knowledge of how the processes' parameters will change, we turn to the well-known generalized likelihood ratio (GLR) test. The GLR test statistic, g_k can be shown to detect a statistical change from μ_0 (mean before change). Unfortunately, GLR is typically used in a context where μ_0 is well-known, e.g. manufacturing processes. Figure 4(a) shows g_k



(a) g_k and WMA(g_k) of prediction errors (b) $\frac{d}{dt}g_k(t)$ and $\frac{d^2}{dt^2}g_k(t)$ (c) Impulse triggered change detection

Figure 4: Modified GLR to accommodate learning drift; synthetic change injected beginning at point 4000.

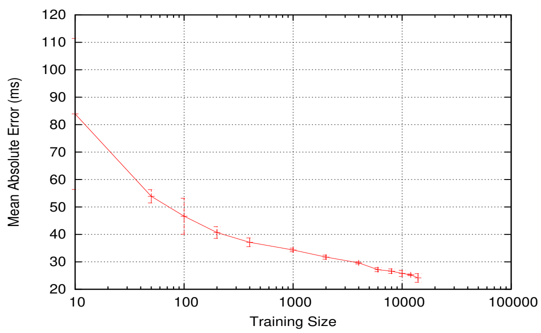


Figure 5: Latency regression performance

as a function of ordered prediction errors produced from our algorithm on real Internet data. Beginning at the 4000th prediction, we create a synthetic change by adding 50ms to the mean of every data point (thereby ensuring a 50ms error for an otherwise perfect prediction). We use a weighted moving average to smooth the function. The change is clearly evident. Yet g_k drifts even under no change since μ_0 is estimated from training data error which is necessarily less than the test error.

To contend with this GLR drift effect, we take the derivative of g_k with respect to sample time to produce the step function in Figure 4(b). To impulse trigger a change, we take the second derivative as depicted in Figure 4(c). Additional details of our change inference procedure are given in [3].

5. RESULTS

We evaluate our clustering algorithm on both real and synthetic input data under several scenarios in [3]; this section summarizes select results from live Internet experiments.

Our live data consists of latency measurements to random Internet hosts (equivalent to the random pairs in §2.2). To reduce dependence on the choice of training set and ensure generality, all results are the average of five independent trials where the order of the data is randomly permuted.

Figure 5 depicts the mean prediction error and standard deviation as a function of training size for our IP clustering algorithm. With as few as 1000 training points, our regression yields an average error of less than 40ms with tight bounds – a surprisingly powerful result given the size of the input training data relative to the allocated Internet address space. Our error improves to approximately 24ms using more than 10,000 training samples to build the model.

To place these results in context, consider a fixed-size lookup table as a baseline naïve algorithm. With a 2^p-entry table, each training address a/p updates the latency measure corresponding to the a'th row. Unfortunately, even a 2^{24}-entry table performs 5-10ms worse on average than our clustering scheme. More problematic is this table requires more memory than is practical in applications such as a router's fast forwarding path. In contrast, the tree data structure requires $\sim$ 130kB of memory with 10,000 training points.

A natural extension of the lookup table is a "nearest neighbor scheme:" predict the latency corresponding to the numerically closest IP address in the training set. Again, this algorithm performs well, but is only within 5-7ms of the performance obtained by clustering and has a higher error variance. Further, such naïve algorithms do not afford many of the benefits in §3.1.

Finally, we consider performance under dynamic network conditions. To evaluate our algorithm's ability to handle a changing environment, we formulate the induced change point game of Figure 6. Within our real data set, we artificially create a mean change that simulates a routing event or change in the physical topology. We create this change only for data points that lie within a randomly selected prefix. The game is then to determine the algorithm's ability to detect the change for which we know the ground truth.

The shaded portion of the figure indicates the true change within the IPv4 address space while the unshaded portion represents the algorithm's prediction of where, and if, a change occurred. We take the fraction of overlap to indicate the false negatives, false positives and true positives

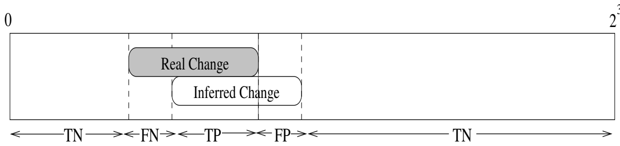


Figure 6: Change detection: overlap between the real and inferred change provide true/false negatives (tn/fn), and true/false positives (tp/fp).

with remaining space comprising the true negatives.

Figure 7 shows the performance of our change detection technique in relation to the size of the artificial change. For example, a network change of /2 represents one-quarter of the entire 32-bit IP address space. Again, for each network size we randomly permute our data set, artificially induce the change and measure detection performance. For reasonably large changes, the detection performs quite well, with the recall and precision falling off past changes smaller than /8. Accuracy is high across the range of changes, implying that relearning changed portions of the space is worthwhile.

Through manual investigation of the change detection results, we find that the limiting factor in detecting smaller changes is currently the sparsity of our data set. Further, as we select a completely random prefix, we may have no a priori basis for making a change decision. In realistic scenarios, the algorithm is likely to have existing data points within the region of a change. We conjecture that larger data sets, in effect modeling a more complete view of the network, will yield significantly improved results for small changes.

6. FUTURE WORK

Our algorithm attempts to find appropriate partitions by using a sequential t-test. We have informally analyzed the stability of the algorithm with respect to the choice of optimal partition, but wish to apply a principled approach similar to random forests. In this way, we plan to form multiple radix trees using the training data sampled with replacement. We then may obtain predictions using a weighted combination of tree lookups for greater generality.

While we demonstrate the algorithm's ability to detect changed portions of the network, further work is needed in determining the tradeoff between pruning stale data and the cost of retraining. Properly balancing this tradeoff requires a better notion of utility and further understanding the time-scale of Internet changes. Our initial work on modeling network dynamics by inducing increased variability shows promise in detecting short-term congestion events. Additional work is needed to analyze the time-scale over which such variance change detection methods are viable.

Thus far, we examine synthetic dynamics on real data such that we are able to verify our algorithm's performance against a ground truth. In the future, we wish to also infer real

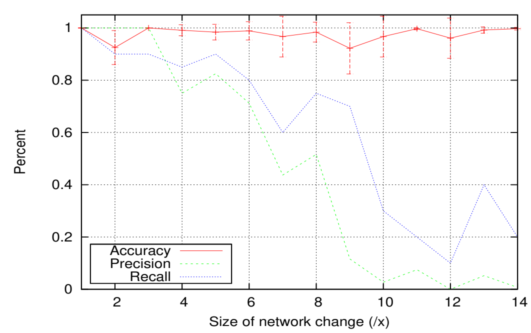


Figure 7: Change detection performance as a function of changed network size.

Internet changes and dynamics on a continuously sampled data set. Finally, our algorithm suggests at many interesting methods of performing active learning, for instance by examining poorly performing or sparse portions of the tree, which we plan to investigate going forward.

Acknowledgments

We thank Steven Bauer, Bruce Davie, David Clark, Tommi Jaakkola and our reviewers for valuable insights.

7. REFERENCES

[1] J. M. Agosta, C. Diuk, J. Chandrashekar, and C. Livadas. An adaptive anomaly detector for worm detection. In *Proceedings of USENIX SysML Workshop*, Apr. 2007.

[2] M. Basseville and I. Nikiforov. *Detection of abrupt changes: theory and application*. Prentice Hall, 1993.

[3] R. Beverly. *Statistical Learning in Network Architecture*. PhD thesis, MIT, June 2008.

[4] R. Beverly, K. Sollins, and A. Berger. SVM learning of IP address structure for latency prediction. In *SIGCOMM Workshop on Mining Network Data*, Sept. 2006.

[5] V. Fuller and T. Li. Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632 (Best Current Practice), Aug. 2006.

[6] W. S. Gosset. The probable error of a mean. *Biometrika*, 6(1), 1908.

[7] K. Hubbard, M. Kosters, D. Conrad, D. Karrenberg, and J. Postel. Internet Registry IP Allocation Guidelines. RFC 2050 (Best Current Practice), Nov. 1996.

[8] B. Krishnamurthy and J. Wang. On network-aware clustering of web clients. In *ACM SIGCOMM*, 2000.

[9] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Proceedings of USENIX OSDI*, Nov. 2006.

[10] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[11] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, Jan. 2006.

[12] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A bayesian approach to filtering junk e-mail. In *AAAI Workshop on Learning for Text Categorization*, July 1998.

[13] K. Sklower. A tree-based routing table for berkeley UNIX. In *Proceedings of USENIX Technical Conference*, 1991.