Bayesian Detection of Router Configuration Anomalies

Khalid El-Arini
Computer Science Department
Carnegie Mellon University
kbe@cs.cmu.edu

Kevin Killourhy
Dependable Systems Laboratory
Computer Science Department
Carnegie Mellon University
ksk@cs.cmu.edu

Abstract

Problems arising from router misconfigurations cost time and money. The first step in fixing such misconfigurations is finding them. Previous efforts to solve this problem have depended on an *a priori* model of what constitutes a correct configuration and are limited to finding deviations from this model, but fail to detect misconfigurations that are uncommon or unexpected.

In this paper, we propose a method for detecting misconfigurations that does not rely on *a priori* expectations of their form. Our hypothesis is that misconfigurations in router data can be identified as statistical anomalies within a Bayesian framework.

We present three detection algorithms based on this framework and show that they are able to detect errors in the router configuration files of a university network. We show how these algorithms detect certain types of misconfiguration successfully, and discuss how they could be extended to detect more subtle misconfigurations.

1 Introduction

On January 23, 2001, Microsoft's websites went down for nearly 23 hours. The next day, Microsoft spokesman Adam Sohn attributed the failure to a "configuration change to the routers on the DNS network" [1]. This example highlights the critical problem of router misconfiguration. Since companies rely on the availability of their networks, such misconfigurations are costly. They are also extremely common, as networks typically contain between 10 and 1000 routers. Each router is individually configured with its own *router configuration file*, which can contain several thousand lines of commands. As a network evolves over time, each file is individually edited to add and remove commands. While the syntactic correctness of each file can be verified, determining semantic correctness and consistency across all the router configuration files in a network is a much harder problem. Due to the magnitude of this problem, our goal is to develop a method that automatically identifies semantic mistakes among the set of router configuration files that define each network.

Previous approaches to this problem require an *a priori* expectation of what these configuration files should look like. For example, if the expectation is that BGP is enabled in all routers, the prior approach would be to write a tool to check that BGP actually is enabled. Our work differs in that we make no such assumptions about the structure of router configurations. By estimating the probabilities of certain configurations, we can detect potential errors as anomalies. For example, we could detect that it is highly unlikely for BGP to be disabled.

We designed and tested three anomaly detection algorithms on router configuration data obtained from the Carnegie Mellon University campus network. This group of configuration files has been modified extensively over the years, and those familiar with this process indicate that errors are to be expected. For this reason, it is an appropriate data set on which to evaluate our algorithms.

2 Related Work

Our work is largely inspired by a suggestion made by Caldwell et al. [2]. They noted that network design choices might be inferred by commonalities across router configuration files, and that these might be learned automatically and codified as rules. As an example, they noted that "if 99 of 100 routers have the finger daemon disabled, the inferred rule would be 'the finger daemon should be disabled' and the exception would be the one non-compliant router." Our work does not directly employ a rule learner, but our algorithms detect such misconfigurations as statistical anomalies just the same.

Feldmann and Rexford [3] parsed Cisco IOS configuration files and performed checks for known misconfigurations or inconsistencies. For example, the authors knew that in the AT&T environment, BGP should always be enabled. As such, they are able to check all the router configuration files to see that the command enabling BGP is present. Our work is more general in that it discovers unknown patterns in the data and detects when those patterns are violated (e.g., detecting that BGP is disabled in one router but enabled in 99 others).

Maltz et al. [4] also used Cisco IOS configuration files in their study of routing design in operational networks. Like Feldmann and Rexford, they extracted information that they knew would be useful in their analysis. For example, they wanted to investigate the use of access lists and whether these were applied to internal traffic or just traffic crossing AS borders. To complete this investigation, they extracted the access lists from the router configuration files.

3 Problem and Approach

Router misconfigurations are a problem due to the high cost to network administrators in money, time, and efficiency. Previous efforts to solve this problem have depended on an *a priori* model of what constitutes a correct configuration and are limited to finding deviations from this model. Thus, they fail to detect misconfigurations that fall outside of this model. Our hypothesis is that misconfigurations in router data will be identified as statistical anomalies within a Bayesian framework. Building detectors that abide by this hypothesis precludes the need to specify the exact types of misconfiguration when performing the search.

Our approach is to build three detectors using this Bayesian framework and to evaluate those detectors on the Carnegie Mellon dataset. These detectors were named according to the probability model that they use: naïve Bayes, joint Bayes, and structured Bayes. Select misconfigurations in the data were manually identified and the performance of the algorithms was measured by how well, and with what overhead, these misconfigurations were detected.

4 Router Data

The router configuration files of 24 Cisco IOS routers were obtained from the Carnegie Mellon campus network. Passwords and SNMP community names were stripped from these files beforehand. Figure 1 shows an excerpt of a Cisco IOS file. The IOS file format is highly unstructured. A configuration is a sequence of commands, many with multiple attributes and some with no attributes. Many attributes appear after the command name, some appear before the command name, while others appear among the separate words in the command name.

By converting these Cisco IOS configuration files to a language-independent canonical form, we make further manipulations of the files easier. Further, by settling on a canonical form for router configuration files, we leave open the possibility that our tools can be applied to other router configuration file formats (e.g., Juniper's JUNOS).

5 Detection Algorithms

Three detection algorithms were designed and implemented to test our hypothesis: naïve Bayes, joint Bayes, and structured Bayes. All three algorithms consist of a training phase and a detection phase. The training phase examines each line of every configuration file and computes a set of key frequencies describing the commands and their arguments. The detection phase makes a second pass through the files, using these frequencies to find anomalies. Our algorithms are akin to outlier detection, where a first phase to compute mean and covariance is followed by a second phase to identify outliers. In both cases, it is meaningless to run the second phase on different data than the first phase. This process differs from techniques that demonstrate generalization, e.g., cross-validation, that use separate training and test data sets.

5.1 Naïve Bayes

The first algorithm makes the simplifying assumptions that (1) each line of a configuration file is independent of every other line and (2) for a given command, each attribute is independent of every other attribute. While these assumptions do not hold, in practice this naïve Bayes algorithm has been shown to perform well in many such domains (e.g., text classification). This algorithm estimates the probability of seeing a specific instance of a command (i.e., a single line in the configuration file). Consider a line L in a configuration file that consists of a command c and attributes $(a_1, a_2, \ldots, a_n)$. This algorithm estimates the probability of the line given the command $P(L \mid c)$ as the product of the conditional probabilities of each attribute given the command:

$$\begin{aligned} P(L \mid c) &= P(a_1, a_2, \ldots, a_n \mid c) \\ &= \prod_{i=1}^{n} P(a_i \mid c) \end{aligned}$$

During the training phase, the algorithm estimates these conditional probabilities from the router data.

```
1: version 12.1
2: no service pad
3: service timestamps debug datetime msec localtime show-timezone
```

Figure 1: A segment of a Cisco IOS configuration file

For each attribute a_i and command c, the probability of an attribute given the command is estimated as the fraction of instances of the command c that contain a_i. If we use $\#(c)$ to denote the number of times c appears in the router data and $\#(a_i \mid c)$ to denote the number of times a_i appears as an attribute of c, then the probability $P(a_i \mid c) = \frac{\#(a_i \mid c)}{\#(c)}$.

During the detection phase, the algorithm computes the conditional probability of each line of a configuration file using these estimates. If the conditional probability of this line is significantly below its expected value, the algorithm classifies the line as an anomaly. Specifically, the algorithm makes the following comparison

$$P(L_i \mid c) < \alpha E[P(L \mid c)]$$

where L_i is the i-th line of the file and α is an empirically determined multiplier. Due to the fact that individual commands vary in the number of attributes, a general threshold across all commands in the configuration file would not produce accurate results. The expected value allows the threshold to adapt to specific distributions. The α variable provides a tunable parameter allowing the user to control the sensitivity of the algorithm.

5.2 Joint Bayes

The second algorithm keeps the simplifying assumption that each line of a configuration file is independent of every other line. However, unlike naïve Bayes, the algorithm does not assume that attributes of a single command are independent of one another. Consider a line L in a configuration file that consists of a command c and attributes $(a_1, a_2, \ldots, a_n)$. This algorithm estimates the probability of the line given the command $P(L \mid c)$ as the joint probability of all the attributes given the command

$$P(L \mid c) \quad = \quad P(a_1, a_2, \ldots, a_n \mid c)$$

During the training phase, the algorithm estimates these probabilities as follows. For each line L with command c and attributes $a_1, a_2, \ldots, a_n$, the probability of the line given the command is estimated as the fraction of instances of the command c that contain the entire sequence of attributes a_1 through a_n. If we again use $\#(c)$ to denote the number of times command c appears and we use $\#(a_1, \ldots, a_n \mid c)$ to denote the number of times the sequence of attributes appears for command c, then the probability $P(a_1, a_2, \ldots, a_n \mid c) = \frac{\#(a_1,\ldots,a_n \mid c)}{\#(c)}$.

The detection phase is similar to that of the naïve Bayes algorithm. We calculate the probability of a line $P(L_i \mid c)$ using these estimates. The major difference between this algorithm and the previous one is how this probability is used to determine whether a line is an anomaly. This decision must depend on more than simply the probability of the line. Consider the example of two commands, c_1 and c_2. Each command appears 24 times, and each takes a single argument. The command c_1 appears once with argument x_1 and 23 times with argument x_2. The command c_2 appears once with each argument y_i, for $i \in \{1, \ldots, 24\}$. The lines "c_1 x_1" and "c_2 y_1" both have equal probability of occurring (one in 24). However, "c_1 x_1" seems to be an anomaly while "c_2 y_1" does not. To differentiate between these two scenarios, we use entropy, a measure of how predictable a distribution is. Specifically, we compute the entropy of each command,

$$H(c) = -\sum_{\langle a_i \rangle \in \mathcal{A}} P(\langle a_i \rangle \mid c) \log P(\langle a_i \rangle \mid c),$$

where $\mathcal{A}$ is the set of possible sequences of attributes for this command and $\langle a_i \rangle = (a_1, a_2, \ldots, a_n)$ is a particular sequence. In our example, c_1 has low entropy while c_2 has high entropy, thus a threshold weighted by entropy will differentiate between the two cases.

If the conditional probability of this line is significantly below the inverse of the entropy, the algorithm classifies the line as an anomaly. Specifically, the algorithm makes the following comparison

$$P(L_i \mid c) < \frac{\alpha}{H(c)}$$

where again L_i is the i-th line and α is an empirically determined multiplier.

5.3 Structured Bayes

The third algorithm also assumes that each line of the configuration file is independent of every other line. However it tries to strike a middle ground between the first two algorithms, treating some attributes of a command as mutually dependent and others as independent. We manually selected attributes that appear to be mutually dependent (e.g., the IP address and subnet mask) and treated them as a single attribute. While this algorithm does require a model of how arguments appear grouped on a command line, it does not require knowledge of what a correct configuration file should contain. Our dependency is on the syntax of the configuration language grammar rather than the semantics of the configuration file. During the training phase, the probabilities of these attributes are estimated as in the naïve Bayes algorithm. During the detection phase, the probability of each line is calculated using these estimates, and then compared to the entropy-based threshold as defined for joint Bayes.

6 Methodology

The Carnegie Mellon router configuration files were used to assess the ability of these three algorithms to detect misconfigurations. First, a set of potential misconfigurations were found in the files and vetted by a domain expert. Then, these three detection algorithms were measured by their ability to find these misconfigurations among the rest of the commands.

6.1 Finding Ground Truth

From the literature [2, 3], we defined three critical types of misconfiguration, which we refer to as lone, suppressed, and dangling commands.

Lone Command: We define a line to be a lone command if it is a unique usage of a popular command. For instance, if a command appears five or more times and, in all occurrences but one, it takes one set of attributes, the unique occurrence is called a lone command. While a lone command is not necessarily a misconfiguration, it is a strong indicator of a potential misconfiguration (e.g., due to a typographical error).

Suppressed Command: We define a line to be a suppressed command if the following two conditions hold. First, it must be an `access-list` command that permits or denies a certain traffic pattern. Second, there is a prior `access-list` command with the same group number that performs the opposite action (i.e., deny or permit) on the exact same traffic pattern. Such a command is evidence of a misconfiguration since it contradicts other commands in the configuration file.

- Lone Commands
  1. `ip ospf authentication null`
     (in `pod-b-cyh`)
  2. `exec-timeout 0 0`
     (in `rtrbone`)
  3. `version 12.2`
     (in `rtrbone`)
- Suppressed commands
  1. `access-list 2 permit any`
     `access-list 2 deny any`
     (in `campus`)
  2. `access-list 2 permit any`
     `access-list 2 deny any`
     (in `rtrbone`)
- Dangling commands
  1. `ip access-group 198`
     (in `pod-c-cyh`)
  2. `ip access-group 133`
     (in `core255`)

Table 1: Table of the potential misconfigurations found in the Carnegie Mellon router data

Dangling Commands: We define a line to be a dangling command if it is an `access-group` command that applies an access list to an interface, but that access list's group number is never defined later in the configuration file. Such a command is evidence of a misconfiguration because it applies access restrictions that do not exist.

We built tools to manually identify occurrences of these misconfigurations in the router data. By applying these tools to the Carnegie Mellon router data, we found seven potential misconfigurations. They are listed in Table 1. These tools were not used by our detection algorithms. Rather, we evaluate our algorithms by measuring their success at detecting the misconfigurations discovered by these tools. In fact, such tools that detect misconfigurations of a specific form are exactly what our algorithms avoid.

To confirm that these potential misconfigurations would be interesting to a network administrator, an expert familiar with these configuration files and the campus routing infrastructure reviewed them, and agreed that they should be brought to the attention of Computing Services.

	Naïve Bayes	Joint Bayes	Structured Bayes
Lone 1	3666	2544	4503
Lone 2	2513	2	2612
Lone 3	2513	2	2612
Average	2897.333	849.333	3242.333
Supp 1	3418	2543	1957
Supp 2	3418	2543	1957
Average	3418	2543	1957
Dang 1	5550	5598	5852
Dang 2	5071	4740	5706
Average	5310.5	5169	5779

Table 2: A breakdown of the number of anomalies generated in order to detect each of the seven misconfigurations

6.2 Evaluating Performance

Having found the seven occurrences of lone, suppressed, and dangling commands in the Carnegie Mellon data, we evaluate the sensitivity of our algorithms by determining how many false positives are detected along with each misconfiguration. False positives are defined to be every line that is not a lone, suppressed, or dangling command. However, such lines may be evidence of a different type of misconfiguration.

Each of the three detectors was run on all 24 router configuration files. As mentioned above, the training phase models the probability distribution of each command, and the detection phase classifies individual commands as anomalies using these probabilities. For each line, the minimum value of α necessary to classify the line as an anomaly is computed. For each of the potential misconfigurations, we determine the number of commands that have a lower minimum α value. This count signifies the number of commands that would also be classified as anomalies in addition to the misconfiguration.

7 Results and Analysis

There are 11,128 commands across all the configuration files, and thus 11,128 potential anomalies to be detected. Table 2 shows the minimum number of lines that are also classified as anomalies for each potential misconfiguration by each detector. For instance, naïve Bayes only detects the first lone command (Lone 1) as one of 3,666 other anomalies. Clearly, the ideal case is when an algorithm detects no anomalies other than the misconfigurations. The worst case is that all 11,128 lines must be detected as anomalies in order for the misconfiguration to be found.

On average, joint Bayes detects the misconfigurations with fewer spurious anomalies. In all cases, dangling commands are the hardest misconfiguration to find. Structured Bayes has the interesting characteristic that it is the only algorithm to detect suppressed commands before lone commands.

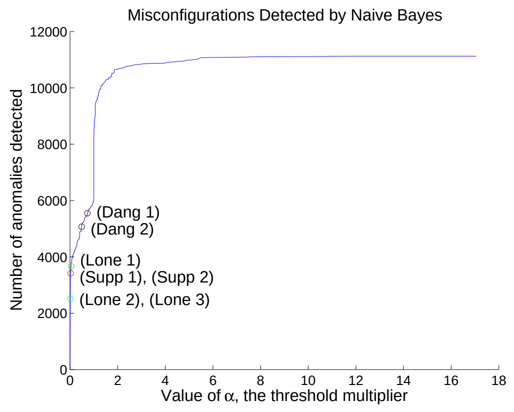
The series of graphs in Figure 2 shows a more precise breakdown of the results for each detector. Figure 2 (a) shows the number of anomalies detected by naïve Bayes over the full range of threshold multipliers (α). The minimum α value needed to detect each of the potential misconfigurations is plotted with annotation on the curve. Figure 2 (b) shows the same graph of anomalies detected for joint Bayes and Figure 2 (c) for structured Bayes.

Note that the shape of these three curves is less important to the operation of the detector as the placement of the potential misconfigurations on this curve. The position of these points along the horizontal axis determines the value of α needed to detect the potential misconfigurations with minimum overhead. The position along the vertical axis determines this overhead. Also, note that on Figures 2 (b) and (c), the vertical jump in the number of anomalies detected at the tail of each curve is a result of the entropy-based threshold. Since lines that always occur in the same format have an entropy of zero, they will never be detected as anomalies by either detector. Thus, they appear at the tail.

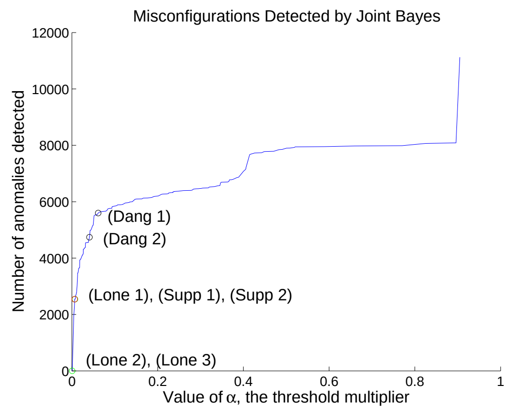
8 Discussion

Our results show that joint Bayes is able to detect potential misconfigurations without also detecting other anomalies. Lone commands 2 and 3 are immediately detected by this detector while the other two detectors are only able to detect them along with over 2,500 others. This type of misconfiguration is specifically that described by Caldwell et al. [2] as important to detect.

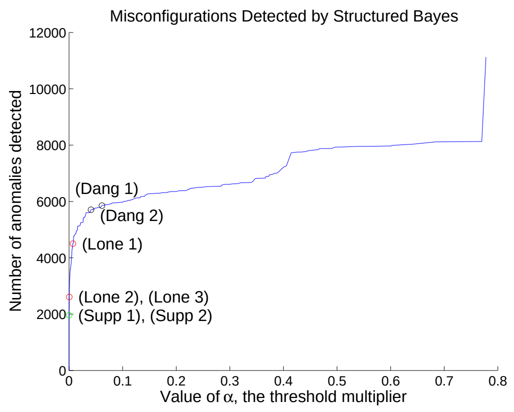
Further progress might be made in the detection of router misconfigurations as anomalies if the assumption of independence between commands is relaxed. For instance, local context could help in the detection of suppressed command misconfigurations. An algorithm that is allowed to assume dependencies between adjacent lines would be able to detect when one line contradicts the other. Similarly, global context could help in the detection of dangling commands. An algorithm that is aware of dependencies between the interface configuration and the access list declarations could detect when an access list is missing. However, whereas local context could be added while maintain-



(a): Naïve Bayes



(b): Joint Bayes



(c): Structured Bayes

Figure 2: A comparison of the threshold multiplier α to the number of anomalies detected by each of the three classifiers. The point at which each of the seven potential misconfigurations is detected is plotted on the curve.

ing the generality of these methods, global context seems feasible only in very controlled settings, where the structure of the files is taken into account.

9 Conclusion

The goal of this work was to determine whether router misconfigurations could be detected without prior knowledge of their form. Three detectors were designed, implemented, and evaluated in this task. These detectors were able to successfully detect certain types of potential misconfiguration in real-world router configuration data.

10 Acknowledgments

The authors are grateful to David Maltz for his significant contribution in providing us with data, reviewing our results, and revising this paper.

References

[1] D. McCullagh, "How, why Microsoft went down," *Wired News*, January 25, 2001. `http://www.wired.com/news/technology/0,1282,41412,00.html`.

[2] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford, "The cutting EDGE of IP router configuration," in *Proceedings of the ACM SIGCOMM HotNets Workshop*, 2003.

[3] A. Feldmann and J. Rexford, "IP network configuration for intradomain traffic engineering," *IEEE Network Magazine*, pp. 46–57, September/October 2001.

[4] D. A. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjalmtysson, and A. Greenberg, "Routing design in operational networks: A look from the inside," in *Proceedings of the ACM SIGCOMM 2004*, 2004.