Bachelor Thesis

Autocompletion for Network Configurations

Ahsan Mahmood

Date: June 4, 2018

Advisor: Prof. Aaron Gember-Jacobson

Department of Computer Science
Colgate University
Hamilton, New York

Abstract

In this thesis, we state the need for an auto-completion engine for writing network configurations. The tools and technology available today allow for synthesizing configurations in large batches or verifying existing configurations. We wish to target the middle of the development life-cycle, to help network operators as they create new configurations or edit existing ones. We propose a simple, yet powerful model inspired by code completion techniques and NLP research. We perform various evaluations which show that the current state of the model gives encouraging results but, also challenge some of our assumptions about network configurations. We also outline additional work that is required to tune our model specifically for network configurations before we can truly realize our goal. Once completed, we believe our engine will be a strong first step in creating a holistic tool similar to IDEs that can assist network operators.

Acknowledgments

I would like to thank Professor Aaron Gember-Jacobson for his continuing guidance and support throughout the year. I would also like to thank the Colgate Computer Science department for allowing this senior thesis to be carried out.

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Colgate University, 05/16/2018

. . . . . .

(Ahsan Mahmood)

Contents

1 Introduction

A network's backbone is its routing control plane: a set of rules and distributed routing protocols that describe how the network should operate. A control plane is thus defined through configuration files present on every individual routing device in the network. These configurations are written in vendor specific languages (e.g. Cisco IOS and Juniper Junos) and describe very low level behaviours of a particular router. Network operators tasked to configure control planes are required to satisfy various 'policies' that the owning organization wants to enforce: e.g certain devices should always be blocked from communicating with higher privileged devices.

Research has shown that configuring control planes can be extremely complex in modern networks [1]. Consequently, this causes configurations to be prone to errors, many of which are only uncovered during operation after a failure has already dealt significant damage [24].For example in 2012, failure of a router in a Microsoft Azure data center triggered previously unknown configuration errors on other devices, degrading service in the West Europe region for over two hours [23]. Additionally, surveys done by researchers in the past [25] have shown that network administrators report router/switch software bugs as the most common symptoms of network failure. Similarly, [16] discovered a number of errors in the routing policies applied to BGP sessions. More specifically they observed missing BGP communities, typos in the definitions of existing communities and missing import or export policy. These examples highlight a need to develop highly resilient configurations that perform reliably.

1.1 Existing Tools

Even though networks are getting larger and more complex, administrators use simple tools for configuring network devices, such as CLIs on the routing devices or networking management tools such as NetMRI or SolarWinds (further described in section 3.3). Additionally, it is notoriously hard to debug problems in a network; operators often have to rely on rudimentary tools such as ping and traceroute. Apart from these existing tools, there are two

main areas of research that are trying to enable network resilience: configuration synthesis and network verification. Synthesis tools like NetComplete [6], Zeppelin and SyNet [7] are working towards perhaps the most ambitious solution: automatically generating all configurations in one batch. However, all synthesis tools require comprehensive input from the network operators. They usually have to fully specify the high level requirements of the policies the organization wishes to impose, ahead of time. This can be a time consuming and nuanced task for the operators. It is difficult to ascertain what existing policies the network follows and there is a lot of room to miss some rule or policy.

This high-input nature of synthesis systems makes them much more suitable during the initial stages of developing a network, given that the operators are willing to put in the work. However, the return rate of configuration synthesis falls sharply if the network operators are concerned with incremental changes. Whenever there is a new router being added to the network, the networking policies are essentially changed in accordance with this new router. Thus network operators will have to re-run the synthesis system to generate configurations for the entire network regardless of whether a router needs to be updated or not. This can make synthesis an unnecessarily time consuming process as all routers have to be updated, which may lead to more downtime in the network.

In contrast, network verification and group repair with synthesis (e.g ARC [9]) can be extremely useful to determine whether the current configurations comply with the existing policies of the network. Again, these policies could be defined by the operators or, as in the case of ARC, they could be inferred via a snapshot of the network. However, verification by design is a post development tool i.e. it will catch errors after they have already been produced. It does not help operators avoid producing these errors in the first place. Furthermore, even though it is possible for certain tools (like ARC) to generate a possible fix, it is not a guarantee made by verification tools in general. In essence, the operators will have to retrace the error and manually edit the faulty configurations. We thus wish to explore a solution that sits in between configuration synthesis and verification.

One existing approach that can be utilized by network operators in the middle of the development lifecycle is templating. Currently, operators try to minimize extraneous work by reusing existing configurations that have been known to work in the past. When creating a network, operators typically write templates containing specific configuration lines that define a base set of behaviours for different router roles [1]. These templates are then used to specialize individual routers to achieve objectives for their respective part of the network. Due to varying router specifications, the template systems used allow network operators to fill in parameters with appropriate information each time the template is used.

However in practice, as the customer requirements of the organization change over time, the networks start to get more complex. [2] describes how configuration templates in use constantly grow in number and become increasingly specialized as a service grows and as customer requirements evolve. The paper shows how operators will have to write up new templates or edit existing ones eventually. We envision our engine would ease this process by providing convenient suggestions while operators are editing configurations (regardless of whether they are templates or not). Our approach thus serves to complement existing techniques for writing routing configurations.

1.2 Motivation for a Completion Engine

The analyses on two campus networks performed by [15] shed light on how networks evolve. If we consider the types of changes that operators perform, we observe some homogeneity. Not only are many changes security related but, each campus has distinct security practices that use specific router configuration commands. This seems to hold true across organizations, where certain design and operational practices tend to be followed. Thus, a completion engine could exploit this regularity by learning from a history of changes and suggest the routines and practices that an organization tends to follow. Our penultimate goal would be a "writing assistant" for network configurations, one that can complete entire stanzas (explained in section 2.1) and recommend more concise syntax. It could also proactively offer "negative" recommendations. If the operator attempts to change a line or stanza of the network configuration that is not frequently changed, the system could discourage the modification by alerting the operator. We postulate that developing an effective token recommendation system would be a concrete first step towards such a goal.

We consider the problem of writing network configurations to be analogous to writing software code. Most configurations are written using vendor specific languages, that make use of rules and keywords similar to traditional programming languages. We envision an interactive system inspired by code completion engines that could be invoked by network operators as they are writing router configurations to offer them suggestions for what to put in next, or list the options available from the invocation point.

1.3 Challenges

There are a few challenges that arise when undertaking such a project. Firstly, routers tend to play different roles in the network. For example in our dataset, two common roles that we encountered were "core" and "edge". Core routers are designed to form the backbone of the networks and are configured to handle connections between all the devices within the network. Edge routers can be thought of sitting on the edge of this backbone, connecting multiple core routers to each other and also to routers outside the organization. Since the roles vary in configuration, our main concern was whether we could build a model that could cater to all the roles in the network.

Secondly, routing configurations tend to have a lot of parameters such as IP addresses, subnet masks, interface names etc. that add a lot of variance between configurations. We would thus need a way to clean our data and ensure that we can reasonably suggest a generic parameter type without trying to guess the exact value. Another challenge is minimizing the size of data required for the model. If an engine requires copious amounts of data, then it might be unsuitable to be used by some organizations who do not regularly store snapshots of their network's configurations. The model should thus perform reasonably accurately in situations of data scarcity.

We explain how our model addresses some of these challenges in section 4. In general, we saw that using a Natural Language Processing approach helped answer a lot of our challenges.

1.4 Completion Using NLP

Recent research on software systems has shown that codebases tend to contain regularities, much like natural languages [11]. This has motivated further research on using traditional Natural Language Processing techniques for code completion and token suggestion, resulting in fairly accurate models [11, 20]. We hypothesize a similar regularity for network configurations, especially since they tend to be homogeneous by design, reusing the same set of keywords/tokens. In Section 3.1, our analysis of router configurations from a large research university showed that configurations shared between 85% and 99% of tokens across different routers, which seems to support our claim. This prompted us to explore simple NLP techniques that could leverage these token similarities to produce useful suggestions or completions.

Another reason we gravitated towards NLP as a basis of our model is the flexibility it entails. Traditional code-completion techniques capitalize on the grammar rules of the languages to build their models. An analogous version for network configurations would require us to reverse engineer the configuration grammar, presumably exploiting some existing parser.[1] Moreover, as different vendors have different grammars, we would effectively have to repeat this process for every vendor and ostensibly rebuild the engine. On the other hand, an NLP approach is theoretically language agnostic. For example, in this paper we perform tests on Cisco configurations but, we could easily train and test the model on configurations written for Juniper. Conceding that it is almost exclusively data dependent, we maintain that an NLP model is a good candidate for an elegant solution to the token completion problem. We thus consider our work as a feasibility analysis for this particular strategy. In general, our results show that using an off-the-shelf NLP algorithm with minor modifications can give us up to 93% accuracy for some configurations. These are encouraging results and in Section 4 we discuss some additional analyses we performed that make us optimistic about the utility of this approach.

1.5 Organization

This paper outlines the necessary details for building a completion engine for network configurations. Section 2 provides background information about network configurations, existing network management tools and code completion techniques. Section 3 briefly discusses all the work pertinent to our research. Section 4 describes our token analysis results and explains how our model works. Section 5 and 6 describe and discuss all the analyses we performed using our model, followed by a conclusion in Section 7.

[1] In fact, we tried to extract this information ourselves, hoping to compare our model to tab-completion which seemed as a reasonable ground truth to try and improve on. However, accurately parsing the parse tree proved to be a tedious, time consuming process. We eventually had to drop this line of analysis, as we were unable to get the system to a point where we were satisfied by the tab-completion output.

2 Background

Generating token predictions for network configurations requires knowledge from multiple domains. In this section, we introduce router configurations and the tools available for writing as well as managing them. We also expand on some code completion techniques that have been successful for programming languages and form a basis for our engine.

2.1 Network Configurations

Router configuration files are often written in a vendor specific language, the popular ones being provided by Cisco and Juniper systems. These files often exist as plain text on the routers and are composed of different types of 'stanzas'. A stanza is defined as the largest contiguous block of commands that encapsulate a piece of the router's functionality. The most important types of stanzas include routing protocol, access-control list (ACL), and interface. Each stanza describes the router's particular role in relation to the stanza type. Network operators will configure these stanzas to define how the routers interact with each other. For example, operators might specify which devices the given router is connected to and what protocol it should follow when communicating with such devices. Additionally, they could enforce security measures by using access-control lists to block certain hosts from entering or leaving a network.

Consider the configuration file for the router with hostname A in Figure 2.1. We can see an ACL stanza near the top which is configured to deny communication from any IP address beginning with 12. A few interface stanzas follow right below it, which define how this router is connected to other routers with some details about the connections (such as costs associated with using those routes). Lastly, at the bottom, we see a routing protocol stanza which states that the router uses the OSPF protocol to connect to two subnets. Here we can see some inklings of general purpose programming languages. We have certain keywords to set parameters (such as cost) for stanzas and define particular aspects of the router. We also have some notion of variables, where we can reuse predefined values. For example the last line in the first interface stanza, we use the access list with ID 1 that was defined in

the ACL stanza above it. The similarities with programming languages that we observe motivate us to consider code completion techniques to help guide the development of our engine.

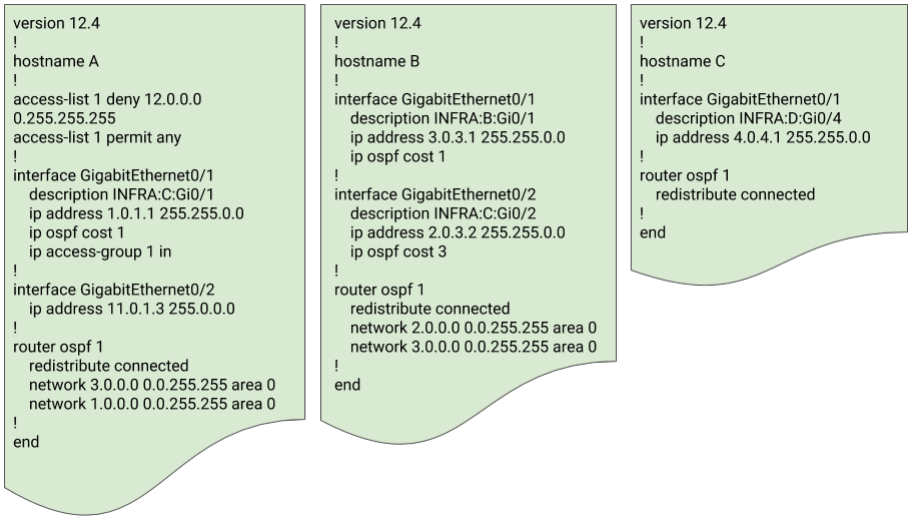

Figure 2.1: A set of simplified configurations for a small network with four routers employing a single OSPF protocol.

2.2 Code Completion

Traditional completion techniques, such as those seen in IDEs [13], generate context-aware models of program histories. This means the code completion engines have to be aware of the grammar of the programming language and make suggestions based off that. Additionally, they maintain a record of past objects, their types and method calls invoked on them. If the user starts typing code with a similar type structure, they use this model to generate and rank the suggestions. These solutions offer fairly respectable accuracies but come with their idiosyncrasies. Popular IDEs, such as IntelliJ [12] or Eclipse [5], use relatively simple type based inferential techniques to suggest all methods available for an object, usually sorted in alphabetical order. Researchers, on the other hand, have proposed more 'intelligent' forms of code completion techniques in the past. Early work started by adopting rule based approaches where a database of predefined rules could be continuously queried to carry out possible completion tasks [14]. Other researchers explored how to make use of program history to offer suggestions based on what users had done in the past [21].

Eventually people started applying machine learning techniques, such as K-Nearest Neighbours, to extract patterns from existing code bases and building models that could be used to rank possible predictions for a given input vector [3]. All these techniques, however, require some form of context extraction, so that information about the codebase can be stored e.g. in form of a feature vector. They heavily leverage the existing code structure and require knowledge about the grammar of the programming language. A similar methodology for network configurations would require carefully extracting information from the parse-tree to ensure that the context of the tokens was properly understood. We go into a little more detail about some code completion techniques in Section 5.2.

Natural Language Processing (NLP) techniques, on the other hand, can generate predictions based on token usage and do not need to be explicitly aware of the grammar. This allows us to use these techniques independent of vendor specific configuration languages. Our inspiration for using NLP techniques primarily came through Hindle,*et al.* 2012 [11]. This paper provides an excellent insight into the regularity of software code. The authors draw parallels between natural languages and codebases, and show that software is just as predictable as many human languages. They used an n-gram model to demonstrate high regularity in a dataset of Java projects, compared to an English corpus. They also proved that these results arose directly from the 'natural' regularity of the codebases rather than being an artefact of the programming language being syntactically simpler than English. Similar to Hindle,*et al.*, Raychev *et al.* 2015 [20] took an NLP inspired an approach to generating code completions. They reduced the problem to predicting probabilities of sentences, performing static analysis on the code and feeding the results to two statistical language models: N-gram and Recurrent Neural Networks (RNN). They collect a history of method calls and treat them as sentences to synthesize suggestions. Interestingly, using RNNs had a negligible effect on their accuracies even though it increased the training time by many folds. Consequently, we have decided to use N-gram models as they seemed to work surprisingly well for both Raychev *et al.* and Hindle,*et al.*. We detail this model in Section 3.

2.3 Network Management Tools

It is important for us to recognize existing tools for managing networks and their shortcomings. One of the main motivations for pursuing this research is the lack of resources available for network operators to write configurations. Most routers offer some form of simple built-in Command Line Interface (CLI), where operators can use vendor specific languages to update router configurations. Often, these CLIs will offer rudimentary tab

completion, where they will alphabetically suggest all the options available for a token from the invocation point. These are sometimes unhelpful as the user then has to search for the desired completion.

Enterprise network management tools, on the other hand, are built to assist network operators as they design, maintain and monitor large sets of router configurations. These tools often also offer various functionality to help write router configurations. NetMRI [18], for example, allows users to use existing templates or write their own scripts to automate simple changes across the network. These changes might include adding ACLs, updating the router OS etc. SolarWinds [22], a similar product, claims to simplify and standardize complex configuration changes by creating a single vendor-neutral script that can be scheduled and executed on multiple devices.

A recurring drawback of these tools is that they focus mostly on updating existing configurations. They do not provide any additional functionality for writing new configurations other than utilizing templates. Even in the latter case, the operators will have to fill in the templates appropriately or write their own custom templates for specialized router roles. Our work acknowledges that in practice no network's functionality can be captured by templates alone. Thus, there is a need for an engine that can distinguish itself from these existing tools by being agnostic towards where it is used in the network development life cycle. We expect our engine to perform consistently whether invoked while writing new configurations or updating existing ones.

3 Related Work

In this section we discuss areas of research that are pertinent to our work.

3.1 Configuration Complexity

Perhaps the most relevant work on this topic is done by Benson *et al* 2009. In this paper, the authors develop a family of complexity models and metrics that describe the complexity of the design and configuration of an enterprise network.They describe three key metrics for measuring the complexity of designing networks: referential complexity, inherent complexity, and router roles. Referential complexity is the measure of the number of reference links from one router to other routers in the network. Inherent complexity tries to measure how policies dictating the function of a router impact the performance of the network. However, the metric most pertaining to our work is router roles. The authors recognized that while creating a network, the operators will define a base set of behaviours that will be present across all routers in the control plane. They proceed to argue that networks become more complex to manage as router roles increase and as routers start to play multiple roles. As we have mentioned before in this paper's introduction, a common cause for network outages is configuration errors, which stem from the complexity of the network. If we can facilitate the building of new configurations, then perhaps we can also help reduce common mistakes. Highly complex routing designs leave more opportunity for configuration errors to occur [4] which, as we mentioned in our introduction, are the leading cause for network outages.

3.2 Code Completion Tools

There are many code completion engines for software codebases. We gave a brief overview in Section 2 but here we go into a little detail about how they work.

Kaiser *et al.* 1988 was one of the earliest works done on auto-completion for code. The authors present an architecture that provides intelligent assistance by automating certain tasks like compiling or completing missing parameters to a function. The assistant maintains of a database of all the entities in the software system, such as modules, procedures, types etc., and a comprehensive collection of rules that define the conditions in which the assistant tools may carry out a possible task. Overall, the architecture presented in this paper takes a rule-based approach reminiscent of expert systems. If a rule is not implemented by the developers, then the engine is unable to provide any assistance. This architecture seems as a precursor to modern software development technologies which offer much of the same assistance. IDEs such as IntelliJ and Eclipse also offer tab completion, which ranks all possible completions from the invocation point in alphabetical order. These IDEs often use type inference to collect all possible methods available to a certain variable.

Robbes *et al.* 2008 and Bruch *et al.* 2009 showcase how program history can be used for maintaining a model of existing code. They store information about changes made in groups of work that are written close together in time. The researchers used various algorithmic techniques to generate code completions. Usually, these algorithms employ some form of variable contextualization. Robbes *et al.* 2008 makes use of static type inference with a clever session based history which first finds similar code segments and then recommends what was written in the same time frame. Bruch *et al.* 2009 on the other hand, extracts the context of variables and encodes them as a feature vector for those particular variables. Features may include the type of the variable, methods already invoked on it, the method in which it is called etc. Their algorithm (which is similar to KNN) can then use these feature matrices along with the input context from the user to retrieve possible recommendations which are determined by their distances from the input vector. Completion techniques involving program histories offer fairly respectable accuracy results but come with their idiosyncrasies.

3.3 Configuration Synthesis

In recent years, researchers have developed network configuration synthesis tools that tackle the problem of generating network-wide configurations. These tools can build router configurations entirely from scratch by using high level policies the owning organization wants to enforce, which are provided by network operators as input. Configuration synthesizers can be extremely useful to avoid bugs, guarantee policy compliance, and sometimes even offer network resilience.

SyNET [7] and Zeppelin are two prominent examples of such tools. SyNet accomplishes this by modeling the routing protocols as a stratified Datalog program, and synthesizing inputs such that they satisfy certain policies or path requirements that comply with the operator's requirements. Zeppelin on the other hand, employs a two phase solution: first synthesizing policy compliant paths, and then generating configurations guided by those concrete paths. Zeppelin offers increased connectivity resilience over configurations generated by SyNet as minimizing the number of static routes used in the configurations. There are other tools such as Propane and Cocoon that similarly use high level specification to generate low level configurations, though we did not study them extensively.

These systems demonstrate that it is possible to use policy constraints to guide configuration creation. However, they require well-defined and thorough policies to be made by the operators, which may be a tedious task for larger networks. Additionally, depending on the complexity of the networks and the policies , synthesis from scratch can take long periods of time and would have to be repeated whenever a new policy is introduced or an existing one is changed. Finally, these systems require the replacement of the entire current network control plane, which can incur significant overhead and network downtime.

It is perhaps possible for these systems to be used in conjunction with code completion techniques to provide a specialized network configuration completion engine. It would require us to develop some intermediate system between the policy definitions and the synthesis techniques proposed by SyNet and Zeppelin. For now, we simply consider configuration synthesis as a potential area to explore future improvement to our engine.

4 Preliminaries

In this section we detail all the work that has been done on building our model which is the backbone of our engine. We start by describing the intuition behind using Natural Language Processing, then introduce N-gram models as our NLP technique of choice and close by describing how we incorporated that into our completion engine.

4.1 Token Analysis

Prior research in computer networking hints that we should expect network configurations to share a common set of stanzas and thus implicitly tokens [2]. As Hindle,*et al.* 2012 [11] pointed out, regularities in bodies of texts can be easily exploited by NLP techniques. We hope to find and use such regularities in network configurations to generate suggestions.

In [1] researchers identified a few key design decisions commonly made by network operators. Network configurations are designed to be homogeneous as a means of easy maintenance, where some operators start off with common configuration templates with varying parameters. They may then tweak these templates to achieve specialized routing roles if needed. Thus one can posit that configurations across devices in a given network may share a lot of the same tokens, subnets and sometimes even complete stanzas (such as Access Control Lists). To confirm our hypothesis, we took configurations from a large university (University A as described in Section 4) network and split up each configuration file into a list of tokens. Tokens included all keywords and subnets with punctuation and newline characters stripped off. For every file we then plotted the percentage of tokens and statements that exist in other router configuration files.

Our results show that most of each file could be rebuilt from existing statements in routing configurations due to the amount of tokens they share. The statement similarity is particularly interesting as it shows that some of these router configurations had a large percentage of complete statements that they shared with the rest of the network. Given our results, and the observations made by [1] about how networks are configured, we can confidently

hypothesize that most token suggestions can be generated from analyzing other existing configurations. This effectively makes all router configuration histories a part of the search space for our NLP model.

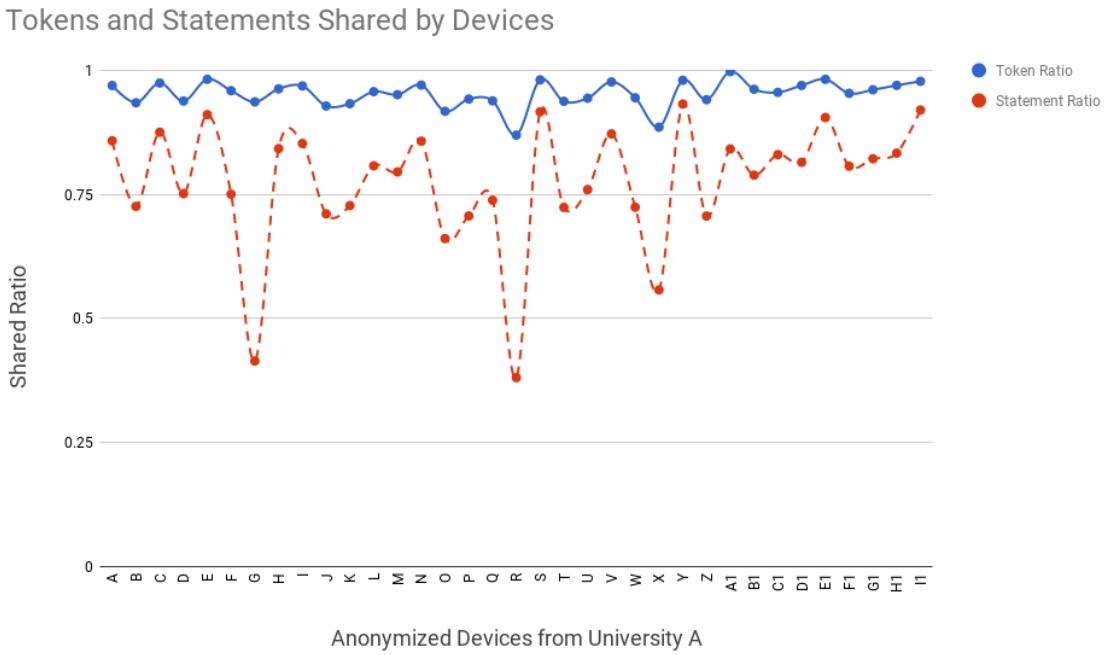

Figure 4.1: The plot shows how many tokens and statements a router configuration holds in common with the rest of the network. The data was taken from a large research university.

4.2 N-gram Models

Given that token regularities existed in network configurations, we employ NLP techniques to make use of token histories. As mentioned in Section 2.2, we use N-gram models as a basis of our engine. Here we briefly describe the theory behind these models. Consider a sequence of tokens in a body of text (in our case, network configurations). We can statistically model how likely tokens are to follow other tokens. We accomplish this by calculating the conditional probabilities of certain tokens appearing in the text. Given a sequence of tokens $a_1, a_2, a_3, \ldots, a_n$, we can calculate the probability of a_2 occurring given that a_1 has already occurred i-e $p(a_2|a_1)$. We continue by calculating the probability of a_3 given a_2, and so on. Since we looked at two tokens at a time, this is called a bigram model. More generally,

predicting how likely a token is to show up based on the previous $n-1$ tokens is called an N-gram model. In our work, we used bigram and trigram models. Any higher number of N-grams would have required a larger data set to train.

The probabilities for suggesting a token pair are estimated by using some scoring function. Often a function may simply count the frequency by which a given pair occurs in the training data and calculate the probability as a ratio of frequency of the pair to the total token count. However, we utilize likelihood estimators as they provided good accuracies for Hindle,*et al.* and have been known to work well for N-gram models [17]. Likelihood estimators have an added advantage of generally being more appropriate for sparse data than other tests. Our system internally uses the Manning and Schutze (5.3.4) version of likelihood estimators [17].

One problem that often emerges in N-grams is that high frequency and low variance can be accidental. If the two constituent words of a frequent. For example, some bigrams like "ip address" may show up together just because are frequently "ip" and "address" are highly occurring words in of themselves even if they do not form a collocation. Researchers will thus use various forms of hypothesis testing to obtain values that better inform the model whether word pairs occur together more often than chance. We chose likelihood ratios as our hypothesis tests of choice. The two hypotheses for a bigram w_1w_2 in our case would be:

$$\text{Hypothesis 1 } (H_1)\text{: } P(w_2|w_1) = p = P(w_2|\neg w_1)$$
$$\text{Hypothesis 2 } (H_2)\text{: } P(w_2|w_1) = p_1 \neq p_2 = P(w_2|\neg w_1)$$

Hypothesis 1 is a formalization of independence (the occurrence of w_2 is independent of the previous occurrence of w_1). Hypothesis 2 informs us of the dependence between the words which is good evidence for an interesting collocation. Assuming a binomial distribution ($b(k;n,x)$), the likelihood functions are as follows:

$$L(H_1) = b(c_{12};c_1,p)b(c_2) - c_{12};N-c_1,p)$$
$$L(H_2) = b(c_{12};c_1,p_1)b(c_2) - c_{12};N-c_1,p_2)$$

Here, we use maximum likelihood estimates for p, p_1 and p_2 with c_1, c_2, c_{12} representing the counts for w_1, w_2 and w_1w_2 respectively, and N being the sample size:end

$$p = \frac{c_2}{N}$$
$$p_1 = \frac{c_{12}}{c_1}$$
$$p_2 = \frac{c_2-c_{12}}{N-c_1}$$

A likelihood ratio would thus be the ratio of $L(H_1)$ and $L(H_2)$.

4.3 Placeholders

We next add a networking specific optimization by incorporating some preprocessing steps to substitute certain parameters with generic tokens. For example, since IP addresses and subnets tend to vary a lot and add noise to the data, we replaced them with placeholders. Having these generic tokens in lieu of parameters helps the engine suggest places where parameters may be entered without trying to predict exact values, which is extremely difficult given our pure NLP approach. In Section 6, we consider other approaches to help suggest parameters such as IP addresses. In Section 5, we also look at the effect of every individual placeholder on the accuracies. We should also note that our initial analysis showed that the engine was trying to predict what the user would enter after a complete configuration statement. We observed that there was little correlation between the end token of a line and the starting token of the next line. We thus altered our model to only suggest tokens that appear on the same line.

4.4 Implementation

N-gram models provided us with a strong foundation on which we could build a specialized completion engine. We developed a program in Python which reads in configuration files and builds a trigram model, using the NLTK package [19]. NLTK also allows us to easily incorporate likelihood ratios as a means of scoring the N-grams. We wrote additional scripts to perform analyses and generate graphs. For our analyses, all individual tests were run in parallel to cut down on runtime. Our scripts were run on a 22-Core Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz server with 128 GB of RAM running Ubuntu 16.04.4 LTS.

5 Evaluation Methodologies

There were a myriad of analyses we performed in order to test the efficacy of an Ngram-based engine as described in the previous section. This section looks at how our model responds to some of the challenges we mentioned in Section 1.3 and discusses the interesting results produced. Some of the main questions that we hoped to answer were as follows:

- How well did our placeholders perform?
- How does the model perform for smaller datasets?
- Does training separate models for different roles improve our accuracies?

We first describe our data, and then analyze the general trend we observed when implementing placeholders. Next, we outline how our analyses show that across both dimensions of time and length, the NLP algorithm is resistant to data scarcity. Furthermore, we showed how breaking up the configurations by router roles gave us unexpected results.

5.1 Configuration Data

We applied our framework to Cisco configurations of core, border, and distribution routers from three large university networks (Table 5.1). Figure 5.1 shows us the distribution by size for configurations from all universities combined. Additionally, we were also able to get extensive data from University A's version control histories. This allowed us to perform some tests based on snapshots sampled across time. We used monthly time intervals for such tests.

Univ.	No. of Configs	Total Lines	Avg Lines
A	35	73K	2.1K
B	26	61K	2.3K
C	24	67K	2.8K

Table 5.1: Configurations used in our evaluation

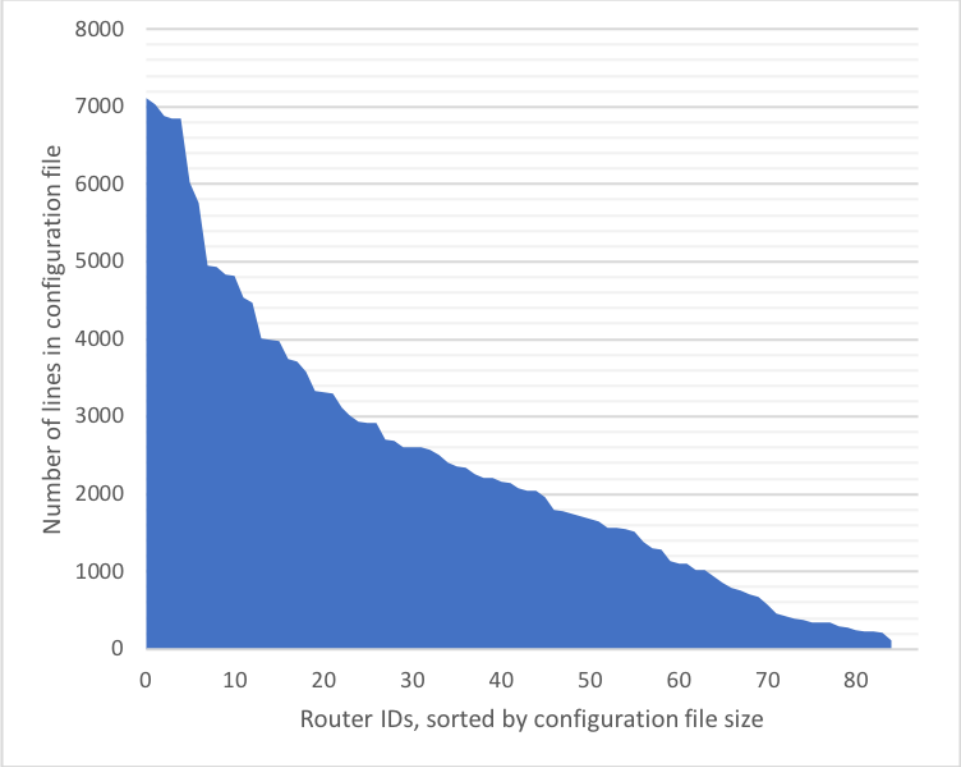

Figure 5.1: Our data has a good mixture of long and short configurations

5.2 Testing Methodology

To test the accuracy of our model, we perform Leave One Out (LOO) Cross Validation. This form of cross validation involves usingone observation as the validation set and the remaining observations as the training set. This is repeated for all combinations of training sets, allowing every observation to act as a validation set. For our analyses an observation could be one set of configurations or just one device configuration depending on the test. For example, for our analyses on length of histories, a validation set is comprised of all

device configurations for a given month whereas for an analysis on the number of devices, the validation set is the configuration of one routing device.

For a single test, we "walk through" rebuilding the validation set token-by-token, starting from the first keyword. For every line we do not predict the first token but invoke the model for every subsequent token. Figure 5.2 shows us what this test conceptually looks like. On the left handside we have a configuration that is being built using the model and on the right the complete original configuration. Consider the line pointed to by the red arrow. We assume that the first keyword ("ip") is given to us. We then use the model to generate three suggestions [1] for the next token and check against the original configuration see whether the correct token ("address") was within this list of suggestions. The next step would be to consider both "ip" and "address" to generate suggestions for the third token. As we are using trigrams, we do not consider more than two previous tokens at a time. We use the ratio of the number of correct predictions to the number of model invocations as a measure of the model's accuracy.

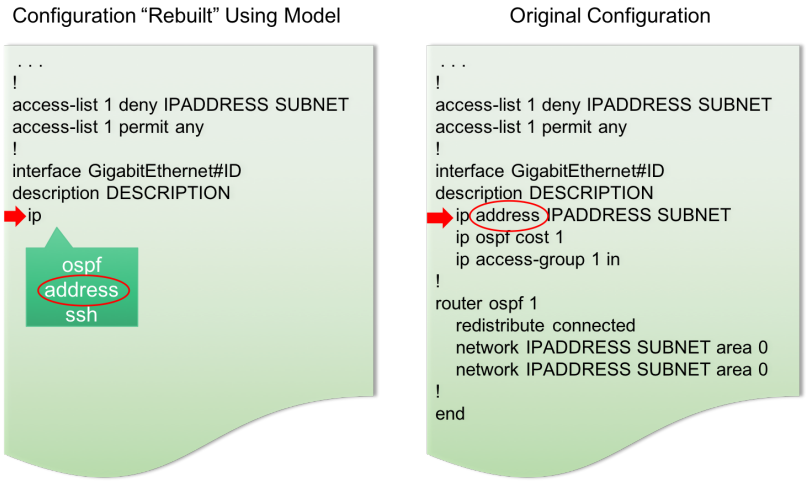

Figure 5.2: A visualization of how a validation test is carried out

[1] The code completion papers we read considered a prediction to be correct if it appeared within the top three to five suggestions. Accurately predicting the correct token within one suggestion every time can be extremely difficult for any code completion system, which is why developers give themselves some leeway when assessing a model's accuracy.

6 Experimental Results

6.1 Overall Performance

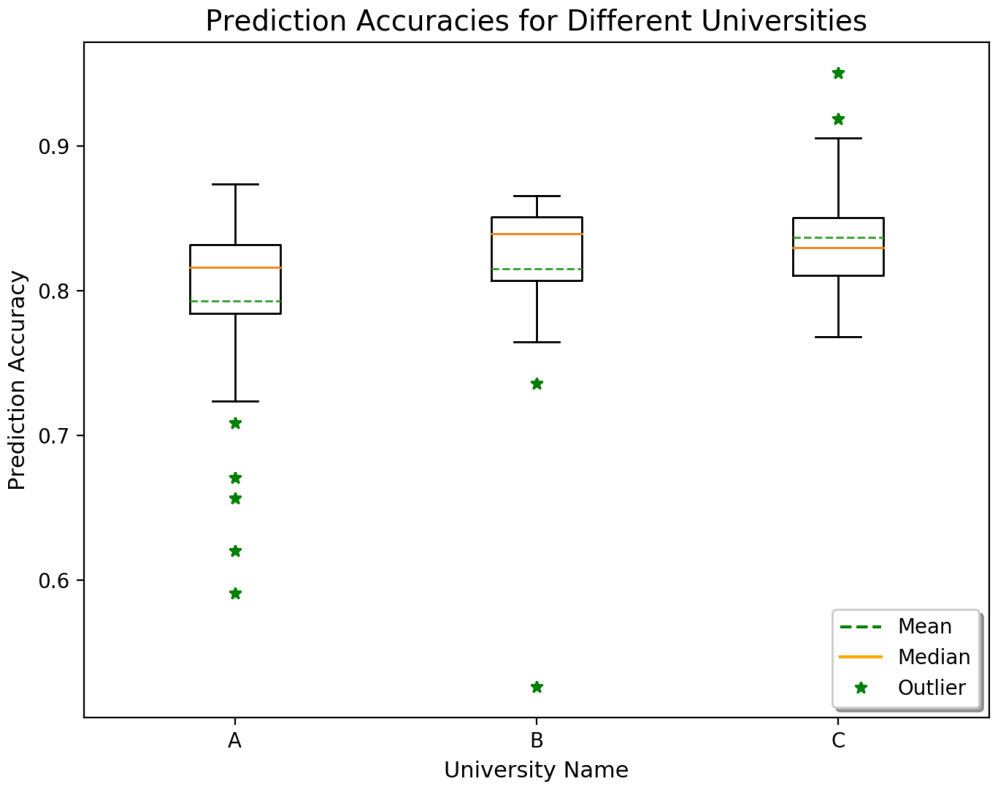

Figure 6.1: Overall accuracy of the model for each set of university configurations used as the validation set.

In Figure 6.1, we show the overall accuracy of our model. For each university, we chose one snapshot in time and ran a LOO Cross Validation on all the device configurations. The

the x-axis shows the name of the three anonymized universities used, and the y-axis shows prediction accuracies. The box plot is supposed to highlight the average (green dotted line), median (orange line) and upper (Q_U) and lower quartiles (Q_L). The box itself marks the Inter Quartile Range (IQR). The outliers (green stars) are all the data points that lie outside $Q_L - 1.5 * IQR$ and $Q_U + 1.5 * IQR$. Our results in Figure 6.1 are after preprocessing the data and we see accuracies as high as 95%, and an average of 80%. These results are very promising as we see high accuracies using a fairly unmodified preexisting NLP algorithm. We next analyze how modifications to the model and to the training data affect our accuracies.

6.2 Placeholders

Generally, we see our accuracies increase when placeholders are used to substitute certain parameters. For each analysis, we took monthly snapshots of University A and replaced all the keywords associated with the particular placeholder being tested. We performed one reference test which did not use any placeholders and used that as a baseline for our accuracy. For example, in the reference test our model will try to predict the exact IP address (e.g. 192.168.1.1) while in the IP Address placeholder test, the model will simply suggest a generic IPADDRESS token.

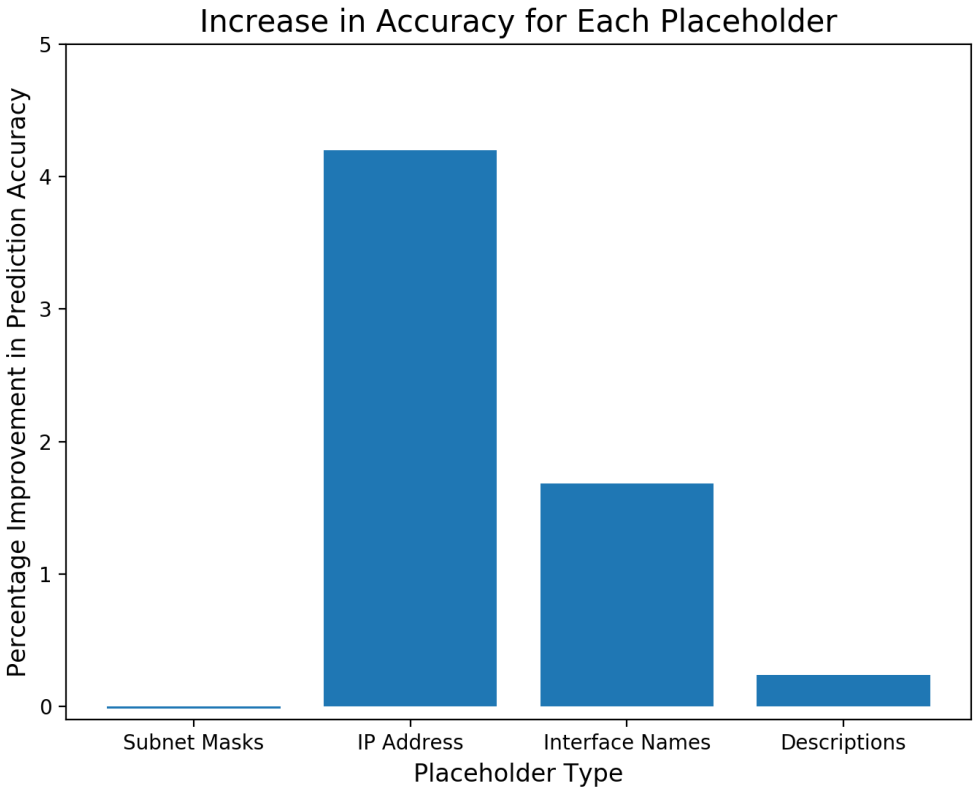

Figure 6.2: Effect on Accuracy by Every Placeholder

As Figure 6.2 shows, the biggest jump in accuracy is accomplished by using IP address placeholders. This is expected, as configurations are bound to consist of a number of unique IP addresses which it makes it extremely difficult to predict which one is going to be used. Other tokens, for example subnet masks, tend to be much more homogeneous (usually a handful of subnet masks are repeatedly used across a network). Thus, we see an almost negligible improvement when replacing subnets. As placeholders were resulting in diminishing returns, we decided to go ahead with the ones we had implemented. We leave additional replacements (such as placeholders for VLAN names and routing costs) as a possible future extension.

6.3 Length of Histories

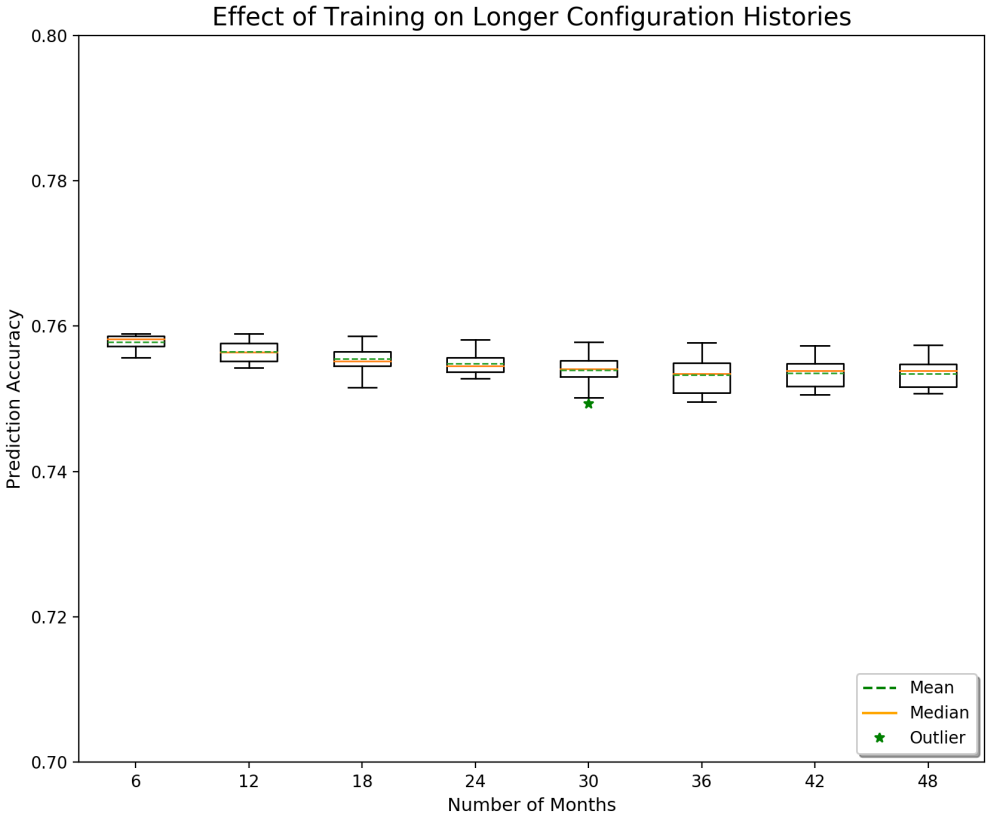

Figure 6.3: Longer histories may not result in higher accuracies.

As we had extensive data from University A's version control system, we analyzed the effect of selecting configurations across time on prediction accuracies. In Figure 6.3, the x-axis of the graph shows the number of months we chose to train on. As is apparent from the graph, if we train on longer configuration histories our accuracies stay about the same. In fact, we see a slight decrease which may be attributed to the variation introduced by the increased data points. It is a possibility that we might be overfitting on the parts of the configurations that remain unchanged over time. In a future analysis, one way to remediate this discrepancy would be to train on just the diffs of the configurations. We should note

that we do expect some change in the configurations over time as the university networks evolve. However, these changes tend to be small and homogeneous as the analyses from [15] inform us. Similarly, another study done by [10] observed that in most networks (80%) there were a small number of change events (no more than 10). Furthermore, the study reveals that most change events seen across networks are small: in about half of the networks, a change event affects only one or two devices (on average). Thus, informed by existing research and our own analyses, we can expect our model to perform accurately without needing to train on data across multiple years.

6.4 Number of Devices

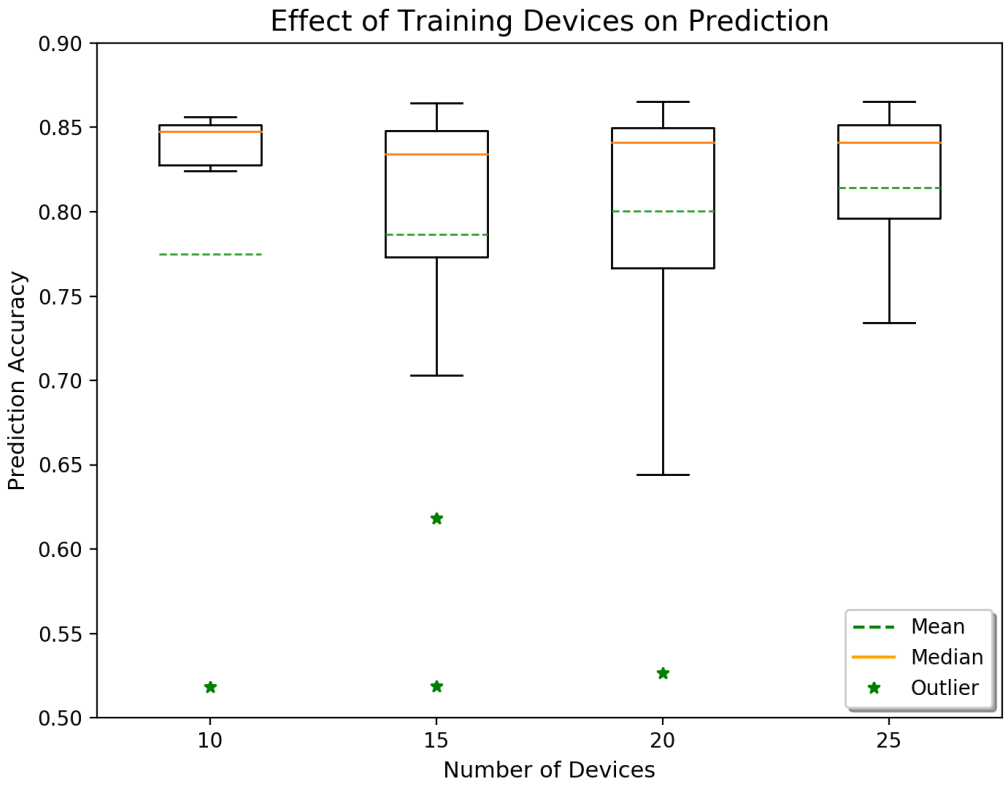

Figure 6.4: Training across more devices does improve accuracy to a certain point.

To obtain the results for Figure 6.4, we used configurations from University B and varied the number of devices to train on. We start by randomly selecting 10 devices for our initial training set. For each subsequent analysis, we continue randomly adding more devices to the existing training set. From the graph we can observe that as we add more device configurations, we see a slight increase in accuracy before it starts to give us diminishing returns. This may mean that the model has already seen most of the tokens that are often used by network operators. Every additional device contributes less to the overall model prediction set. In practice some devices being added may be different from the rest of the network, causing them to act as outliers which would be affecting our averages.

6.5 Role-Based

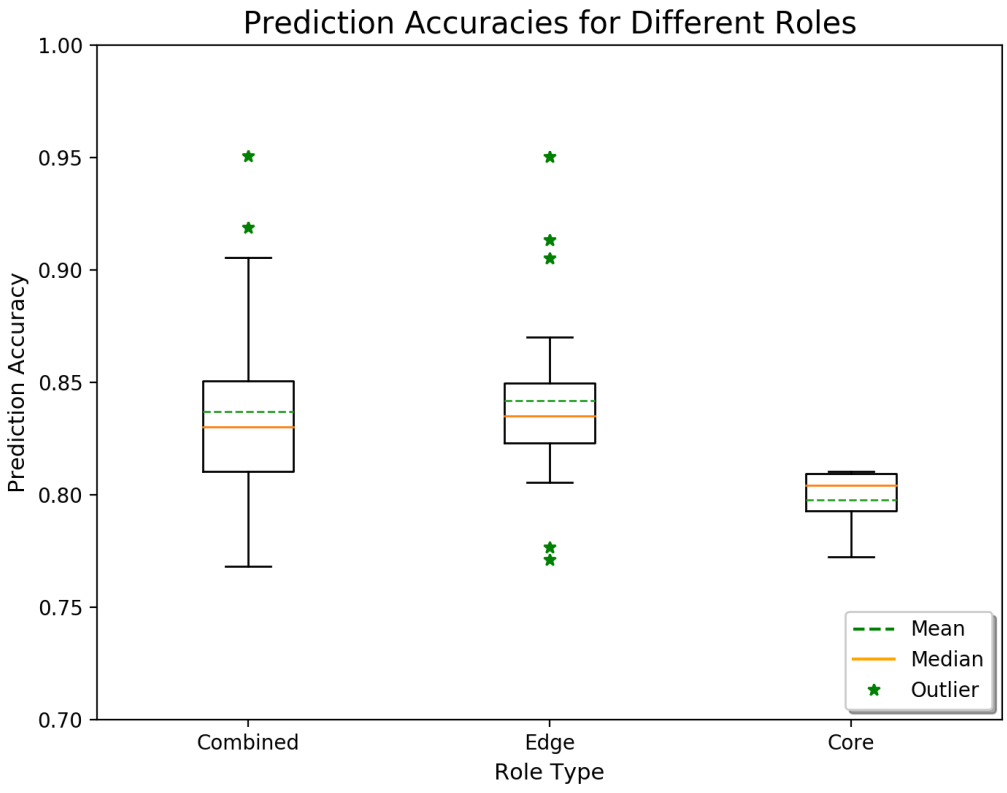

Figure 6.5: Accuracies of models were trained on core and edge routers only, with the first boxplot showing the combined training set

In Figure 6.5, we analyse the effects of splitting the devices by their roles (explained in Section 1.3) and training the model on each role. We used data from University C as the router configurations were clearly labeled "edge" and "core". From the graph, we observe that splitting by roles did not result in an improvement in accuracy like we expected. There are a few factors that could be contributing to this result. It might be possible that there is not much variation in the two roles to begin with. We were assuming that the suggestions generated by the "combined" version would contain the tokens for both edge routers and core routers which would reduce the chances of the correct one cropping up in the top 3 suggested tokens. However, the results show that after separation, edge routers did not fare any better and core routers actually did worse. We also believe that this may be a problem of the labels/roles being coarsely defined as we were relying on the names of the routers to separate them. It is entirely possible that network operators may be labeling the routers as a rough approximation of their intended roles. Thus researchers may need to figure out a more fine grained way about dividing them.

6.6 Key Observations

Our analyses show that the model performs fairly accurately after some preprocessing to the data in the form of placeholders. We can see that it is reasonably resilient to data scarcity across time and length, though we acknowledge that the homogeneity of the networking configurations may be a contributing factor to this observation. Finally, our analysis of router roles gives us an unexpected result. We believe further analyses on the contents of the configurations pertaining to different roles could help determine whether separation by roles is justified.

7 Conclusion

Compared to software developers, network operators are often left neglected when it comes to development tools. Our work tries to bridge that gap by providing a simple completion engine that could be incorporated into a more extensive tool. Our initial findings show that we can get fairly respectable accuracies with an off-the-shelf NLP technique. We provide analyses that help legitimize the potential of such a model to be used in a completion engine. Additionally, we propose a myriad of refinements to the model that could potentially improve our accuracies to desirable numbers.

7.1 Future Work

There are many directions in which we can expand on this work. An obvious next step would be to incorporate higher order N-gram models (quadgrams, 5-grams etc.) in to our engine and test their accuracies. Larger N-grams could help suggest complete statements which would be extremely useful for minimizing input from the network operators. Furthermore, when using N-gram models, researchers will often start at a higher number and fallback on lower order N-grams. For example, we could first use the trigram model to generate suggestions, and if the results are unsatisfactory we could invoke the bigram model. A combination of the two results should result in improved performance. It should also be relatively easy to add additional placeholders in the preprocessing step, such as for VLAN numbers and routing costs. Additionally, we acknowledge that we have limited ourselves to N-gram models as we deemed it an excellent starting point. However, it is imperative for us to explore other code completion and NLP techniques (such as Recurrent Neural Networks) before we can confidently declare N-grams as the final choice for our engine. In doing so, we also leave open the possibility of developing a hybrid model which offers the best of individual ones.

As we mentioned earlier in Section 3, there are some techniques that we could explore to generate custom completions for IP addresses and subnets. Currently, the model will generate all the addresses that it has seen before during training. It would be possible to

improve these results if we could store a mapping of all the subnets that the router is known to be connected to. Then if a network operator wants to add an IP address we would be able to suggest only those addresses that are relevant to that particular router.

One extremely useful addition to the model would be context awareness. We could generate customized completions for different stanza types in the configuration files. For example, a routing interface stanza uses certain keyword like neighbor and network, more often than other stanzas. Our engine should then weight these keywords higher if it is invoked within a routing stanza. Existing configuration parsers like Batfish [8] already have the functionality to be context-aware of stanzas. We would like to explore ways in which we can extract information using such parsers and incorporate it into our engine.

Lastly, we have additional plans for further evaluating our model. It should be relatively straightforward to train and test on router configurations belonging to non-academic organizations. It should be possible to scrape additional ones from online data sources such as router vendor documentations and publicly available Internet configurations. Additionally, we would like to ascertain the extent to which our model is generalizable. This would require multiple analyses across configurations that vary in owners, device types, size etc. This will allow us to see whether our model is confounded when tested on sources that are different in nature to the training set.tionFuture Work

Bibliography

[1] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. pages 335–348, 2009.

[2] T. Benson, A. Akella, and A. Shaikh. Demystifying configuration challenges and trade-offs in network-based isp services. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 302–313, New York, NY, USA, 2011. ACM.

[3] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[4] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford. The cutting edge of ip router configuration. *SIGCOMM Comput. Commun. Rev.*, 34(1):21–26, Jan. 2004.

[5] Eclipse. Eclipse oxygen. `https://www.eclipse.org/`.

[6] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, Renton, WA, 2018. USENIX Association.

[7] A. El-Hassany, P. Tsankov, L. Vanbever, and M. T. Vechev. Network-wide configuration synthesis. *CoRR*, abs/1611.02537, 2016.

[8] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, Oakland, CA, 2015. USENIX Association.

[9] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. August 2016.

[10] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. Management plane analytics. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 395–408, New York, NY, USA, 2015. ACM.

[11] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. T. Devanbu. On the naturalness of software. *Commun. ACM*, 59(5):122–131, 2016.

[12] JetBrains. Intellij. `https://www.jetbrains.com/idea/`.

[13] JetBrains. Intellij autocompletion documentation. `https://goo.gl/1MrE8o`.

[14] G. E. Kaiser and P. H. Feiler. An architecture for intelligent assistance in software development. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 180–188, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[15] H. Kim, T. Benson, A. Akella, and N. Feamster. The evolution of network configuration: A tale of two campuses. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 499–514, New York, NY, USA, 2011. ACM.

[16] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb. Detecting network-wide and router-specific misconfigurations through data mining. *IEEE/ACM Trans. Netw.*, 17(1):66–79, Feb. 2009.

[17] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[18] NetMRI. `https://www.infoblox.com/products/netmri/`.

[19] NLTK. `http://www.nltk.org/`.

[20] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. *SIGPLAN Not.*, 49(6):419–428, June 2014.

[21] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society.

[22] SolarWinds. `https://www.solarwinds.com/network-management-software`.

[23] Y. Sverdlik. Microsoft: misconfigured network device led to azure outage. `https://goo.gl/Y5sDov`.

[24] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. pages 159–172, 2011.

[25] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 241–252, New York, NY, USA, 2012. ACM.