Title Text

Subtitle Text, if any

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

This is the text of the abstract.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

1. Learning strategy

A primary concern in any machine learning type task is to minimize both false negatives and false positives. In the context of configuration file verification, a false positive is when ConfigC reports an error on a valid confiuration file and a false negative is when ConfigC fails to reports an error on an invalid confiuration file. Too many false positives will cause users to ignore the reported error(). However, since the cost of system failure is so high from a misconfiguration, ConfigC propritzes the minimization of false negatives.

While a traditional classififcation learning machine learning approach can reduce both of these situations, generally, there is can be no garuntee that all false negatives will be eliminated. Instead of building classification models over the learning set (such as an SVM), we learn the largest set of rules that all correct configuration files satisfy. In this way, ConfigC can garuntee that, over the set of rules we consider, there will be no false negatives that could have been caught with the given learning set. The only case of a false negative can be when there was no evidece of such a rule in the learning set - we cannot generate rules from nothing.

$\mathcal{L}$ = Learned Rules
$\mathcal{C}$ = Correct Configuration File Learning Set
$\mathcal{R}$ = Reported Rules
$\mathcal{T}$ = True Rules

For probablistic learing $\mathcal{C}'$ = Incorrect Configuration File Learning Set

That is, taking the following definitions:

$$\begin{aligned}
&\mathcal{T} = \{r \mid \\
&\quad \forall file \in \mathcal{C},\ r(file)\ * \wedge \\
&\quad \exists file \in \mathcal{C},\ r(file) \text{ is non-trivial} \wedge \\
&\quad \text{if } \neg r(userfile) \text{ then system crash}\} \\
&\mathcal{L} = \{r \mid \\
&\quad \forall file \in \mathcal{C},\ r(file) \wedge \\
&\quad \exists file \in \mathcal{C},\ r(file) \text{ is non-trivial}\} \\
&\mathcal{R} = \{r \mid \\
&\quad r \in \mathcal{L} \wedge \\
&\quad \neg r(userfile)\}
\end{aligned}$$

We can then conclude that ConfigC is complete but unsound. The key to showing that ConfigC can be complete lies in proving $\mathcal{R}$ is indeed the set it claims to be. In order to do this, we must closely examine our implementation.

$$\forall r \in \mathcal{T}, r \in \mathcal{R} \qquad \text{[Complete]} \tag{1}$$

$$\exists r \in \mathcal{R}, r \notin \mathcal{T} \qquad \text{[Unsound]} \tag{2}$$

The learned rule set, $\mathcal{L}$, is represented as a type, where the type must support a particular interface (called a typeclass in Haskell) to be compatible with our system. The three methods of this typeclass and the associated specification will help to show completeness of our system. The functions of this typeclass will be used, invisibly to the user, to make the overall system run. As long as the specifications for each function are met, ConfigC can garuntee completness.

The typeclass can support anything that is Foldable, which roughly means the user can use any datastructure they prefer. In fact, in our implementation, two rules are implemented with lists, and two others use hashmaps.

```
class Foldable t => Attribute t a where
  learn :: IRConfigFile -> t a
  merge :: t a -> t a -> t a
  check :: t a -> IRConfigFile -> Error
```

2. Rules

2.1 learn

For a single given file in the intermediate representation format, learn the full set of rules on that file. By overfitting to each file, we can eventually garuntee the completness of ConfigC. The specification of this function is the obvious reduction of the Considered Set definition.

learn $file = \{r | r(file) \wedge r$ is non-trivial$\}$

2.2 merge

Merging the sets of rules from two files to build a new set that is true over both files is the most difficult and important function a rule must implement. This is generally implemented as a filter over the union of the two set, but may vary slightly. The second predicate of formal specification states that the rule cannot conflict with other existing rules.

$$\begin{aligned}\text{merge* } &Set1\ Set2 = \{r \mid \\ &r \in \text{Set1} \cup \text{Set2} \wedge \\ &\exists file\ \forall r' \in \text{Set1} \cup \text{Set2}, r(file) \wedge r'(file)\}\end{aligned}$$

2.3 check

To check a file by using a rule set, we simply take all the rules that are releveant to the user's file. Rules that are relavent are the ones where both parts of the ordering are present. We learn the rule set for the user file, and every rule in the learned set must be present in the user file.

References