Comorbidities

Documentation for the project
Under the guidance of
Professor **I.V. Ramakrishnan**

At
State University of New York at Stony Brook
Department of Computer Science
December 24, 2016

Table of Contents

1. Abstract

Comorbidities are diagnosed using a set of rules that compare certain lab dictionary values against assigned threshold limits. In the current comorbidity diagnosis system, all theses comparisons are done using static functions defined as a part of the CERNER system. Adding a new comorbidity requires changes in the backend javascript code, making the system rigid. We aim to come up with a rule builder application that allows users to create rules for newer comorbidities. These rules will then be used in the existing comparison based method to identify what comorbidities are present. This will make the system more user friendly as the intended user can write the rules using a GUI that makes creating the rules as simple as writing it in plain language. There is no need of any programming knowledge. These changes seek to make the system flexible and more dynamic in nature.

2. Detailed Functionality

2.1. Rule Builder

2.1.1. Create Rule

This is the first tab of the home page of the rule builder application. This page allows the user to create new rules to be used for diagnosing comorbidities. To create a new rule, the user needs to populate all the lab dictionary values being used for comparisons against their threshold values. The tool also allows the user to add subcategory for each comorbidity and the set of rules for them individually. The conditions assigned for each rule are also interpreted in plain language which makes it easy for the user to validate the created rule with the intended rule functionality.

2.1.2. View / Edit rule

This is the second tab of the home page of the rule builder application. This page displays all the rules created thus far. It also allows the user to modify an existing rule with functionality like adding/removing comparisons, changing the threshold values used for comparison with lab dictionary values. The changes made from here are reflected in the globally persisted rules across the entire application.

2.1.3. Test Rules

This is a part of the rule builder to test its working in a local set up environment. This tab displays all the Comorbidities for which rules are created thus far. Selecting a comorbidity displays the lab dictionary values on the right and allows the user to provide their corresponding values as input. It then invokes the rule

builder code to execute the associated rules and then provide the output in the right hand part of the page.

2.2. Rule Interpreter

The rule interpreter is a complete workflow tool that processes the rules and reports the comorbidities present. A ‘rule’ here is a set of individual or complex comparison based expressions of lab dictionary values with certain threshold values. Comorbidities can be diagnosed based on the result of the evaluations of the rules.

This section encompasses all the functionality related to interpretation and evaluation of comorbidities rules. The design of the interpreter has been influenced by Douglas Crockford’s Top Down Operator Precedence.[1]

2.2.1. Interpreter: rule.interpreter.interpreter.js

Processes all the defined rules and invokes lexer, parser, evaluator and related functions. The functionality is mainly provided by the following functions:-

➔ **processRulesJson()**:- Processes all the defined rules and invokes the parseRule() function.

It takes as its input - *ruleJSON*, which is an array of all the rules in JSON (key,value) format where the key is the name of the comorbidity. The corresponding value is dependent on whether the comorbidity contains any subcategories:

- If the comorbidity does not contain subcategories, then the value for the key is the rule defined for that comorbidity and parseRule() is called on that rule. The result gets stored in the 'result' variable. The *result* variable is another JSON object with the key as the comorbidity name and the value as True or False depending upon the result of parseRule().
- If the comorbidity contains subcategories, then the value is an array of JSON objects. These nested JSON objects have the subcategory of the comorbidities as the key and the corresponding rules as the values. The function iterates over each subcategory and parseRule() is invoked for each subcategory’s rule. The result of the parseRule() is stored in the *result* variable with the primary key as the comorbidity name and the secondary key as the subcategory name. The value here is True or False

[1] "Top Down Operator Precedence - Douglas Crockford's Javascript." 21 Feb. 2007, http://javascript.crockford.com/tdop/tdop.html. Accessed 23 Dec. 2016.

depending on whether the subcategory of the comorbidity is present or not.

➔ **parseRule()**:- Invokes the lexer, parser and evaluator related functionality on the rule being evaluated.
The parseRule function takes as input a rule, the *variables* containing the values of all the lab dictionary parameters and *processedResult*, which contains the rules for previously defined comorbidities. The function calls the lexer with the rule as the input and extract tokens from the rule. It then creates an Abstract Syntax Tree (AST) by passing the tokens as input to the parse() function. The AST is then passed on as input, along with *variables* and *processedResult* to the evaluate() function, which analyzes and evaluates the conditions in the rule. It returns as the output - True or False depending on whether comorbidity was evaluated as being present or not respectively.

2.2.2. Lexer: rule.interpreter.lexer.js

The lexer splits the incoming rules into tokens for further processing. The main functionality can be found in the following function/s:-

➔ **lexer()**:- Generate tokens from the rule.
This function initially declares all the tokens that the rule could possibly get divided into. The tokens are defined using Regular Expressions to standardize the format of each token. The incoming rule is split based on the '\s' (,i.e., space character) to get all the tokens. It defines several utility functions to classify the token type as number or identifier, rule etc.. All the tokens generated by splitting the rule are then iterated over and passed to the classifying functions to create a list called *tokens*, where each element is a JSON object of key - type of token, and value - the token itself. The lexer also adds appropriate tokens for the any/or conditions. The list is returned to the calling function (parseRule(), in our case) for further processing.

Consider the rule for the Comorbidity - Acid Base Disorder for subcategory Metabolic Acidosis - minbicarbonate < 18.0
The lexer() tokenizes the rule to provide a list(*tokens*) as follows:-

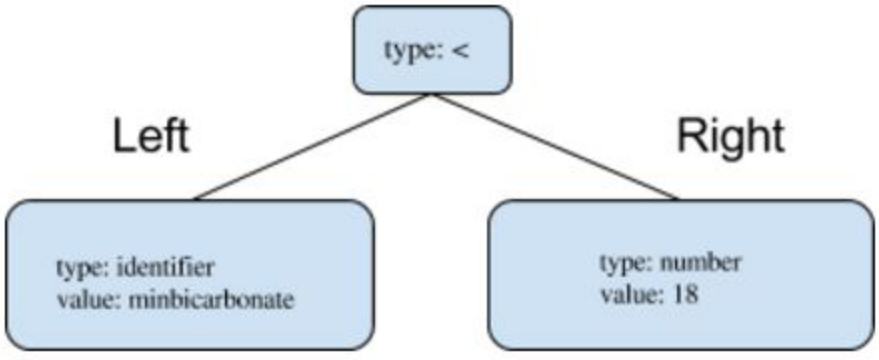
Index	type: (Key)	value: (Value)
0	identifier	minbicarbonate
1	<	undefined

2	number	18
3	(end)	undefined

2.2.3. Parser: rule.interpreter.parser.js

Generate the Abstract Syntax Tree (AST) from the tokens for evaluation and interpretation. The main functionality is as follows:-

➔ **parse()**:- The parser() function takes as input the generated tokens from the lexer function call and creates a list of expressions of the format of <Left Hand Side> <Operator> <Right Hand Side>. To note, each of LHS and RHS can be individual expressions.
The parser utilizes the concepts of Null Denotation(nud), Left Denotation(led) and Left Binding Power(lbp). Every operator token gets associated with an lbp attribute based on the preference according to standard arithmetic rules, so that the parsing can happen and the expression is created as intended by the rule creator. The nud and led are functions associated with the tokens. The nud method gets associated with tokens of type - Number and prefix operators. The led function gets associated with the infix and suffix operator tokens.
The parser invokes the nud function on the first token. The lbp of the next token is then checked. If it is as at least as large as the given binding power, then the led function for that token gets called. Operator tokens have higher binding power and led methods invoke the parses again to get the result of subexpressions. These are used with the left value (if it was of type Number) to compute a final left. The parsing of the expression continues till the end is reached. The ending of the token list is represented here as the (end) token. The parse tree generated in this manner is returned as output. The rule for Acid Base Disorder leads to a AST of type -



2.2.4. Evaluator: rule.interpreter.evaluator.js

➔ **evaluate()**:-
The evaluate function takes the list of expressions generated from the parse() function. It iterates over each expression and resolves the operation based on the type of the operator in the expression by using the parseNode() function.

➔ **parseNode()**:-
The parseNode() function takes in the node as input, and based on the node type, evaluates the value of the node. To evaluate an expression, it recursively calls parseNode() function on the left node and the right node to get the respective values. It applies the arithmetic operation on the corresponding values to get the output. It should be noted that the left and the right node of an expression can be a complex expression in itself, but because of the recursive nature of the calls, the parseNode() function is able to successfully evaluate the outcome of the expression.

Moreover, the parseNode() function also takes an input parameter ‘anyKLevel’, which is specific to OR(||) operations. This is to be used in cases where the user specifies ‘any 2/3/4/N of the following conditions are true’ in case of comorbidity rules. The evaluation for this is slightly different and was done by maintaining a count of the number of children expressions that are evaluated as true. Every time a child expression is evaluated to be true, the count is decremented by 1. Finally, the count is compared against 0 and it the parent expression is True only if the final count is less than or equal to 0.

2.3. Cerner Integration

2.3.1. Comorbidity.js

This file links the Cerner application to the Rule Builder tool.

Adding a new rule to the integrated system is described as follows:-

- The rules are specified in JSON format at the beginning of the file. Care must be taken while writing the rules, any incorrect syntax or extra spaces or missing brackets can render an error. Before testing for any new rule, the rule must be added to this JSON variable.
- All the variables that are used to verify the rules are populated with the values received from the reports.

➔ **buildCCBody()**

In the function buildCCBody, we declare a dictionary *variables* that stores the values of the rules' variables in one place. This is used at a later time to check all rules and return the rules that satisfy the conditions. This dictionary must be populated with the variables present in the rule you want to add.

➔ **processRules()**

The buildCCbody function invokes the processRules function. This function lies within the Interpreter code and is used to update the variables used in our rule verification. The update includes checking for minimum or maximum constraints, calculating a third value from two given values and checking for null values. Error handling for any new rules must be taken care of here. It returns the updated 'variables' dictionary.

At the end of the processRules function call, we get a result dictionary that contains the disease name, its subcategories (if present) and the result of the evaluation. Next, we go through each rule one by one, and check for the variables that are present in this rule.

- If a rule has a subcategory, its result type will be an object and not a Boolean. In that case, we iterate through the subcategories and see if any of them are true; if yes, we display them in blue, otherwise in black.
- If a rule doesn't have a subcategory, we simply check the Boolean value; if it is true, we display the disease in the final display in blue, otherwise we don't display it at all.

The display code is common and similar for all the disease types. We basically add rows to a table body and add checkboxes to denote if the disease is present.

3. Testing

The testing of the rule builder functionality was done in a controlled environment (MOCK server) separate from a production server. A sample set of 70 patients were considered. These patients were diagnosed with a large number of comorbidities, combinations of multiple comorbidities included as well. The output of the rule builder for the list of comorbidities was cross validated against the existing comorbidity diagnosis system that is in use. The format of the output generated by this rule builder was also matched against the existing format to maintain consistency across the application. Other details of each comorbidity such as the lab dictionary values, date recorded and the rule used to diagnose the comorbidity are also displayed as a mouse hover over functionality.

4. Conclusion

So far in this project, the rule builder application has been implemented in completion. It allows users to create new rules, with a user friendly interface, providing facilities to read the rules while they get created, to select lab values from an existing dictionary, as well as to export the rules in JSON format. Additionally, the user can view as well as edit an existing rule, using the View/Edit tab. The tool has been tested locally on all rules.

Also, the rule builder code has been integrated with the Cerner system's code. When selecting a patient's data, one can view the rule builder results by clicking the 'Rule Builder' tab on the left navigation pane. Results are displayed in the exact same format as Cerner displays results, so that there are no variations. Testing for this has also been done on 70 patients, as described in section 3.

5. Future Work

Currently, each comorbidity has its own display format and needs to be implemented for every comorbidity that appears as a result (Diagnosis: True/Comorbidity Present). In the future, we have to come up with a new template that reads all the comorbidities, for which a rule is defined and shows the output in a standard format. This should be a part of the rule created by the physician. In this, the physician chooses what details (For example:- lab dictionary values, date recorded etc.) are required to be shown for the comorbidity.

Currently, the rules JSON for the comorbidities generated from the rule builder is read statically by the code of the CERNER system. We need to come up with a mechanism that allows the rules JSON created from the rule builder to persist globally in the system. Additionally, as a part of the CERNER system, we need to implement a mechanism for the system to read these persistent rules JSON on the fly.

6. Code Repository Link

- The code for the rule builder integrated with the CERNER system can be found at the path: https://bitbucket.org/jugudannie/cernerrulebuilder

- The code for the stand alone version of the rule builder (to run on a local setup) can be found at the path :https://bitbucket.org/ankitmit/rulebuilder

7. References and Related Work

[1] "Top Down Operator Precedence - Douglas Crockford's Javascript." 21 Feb. 2007, http://javascript.crockford.com/tdop/tdop.html. Accessed 23 Dec. 2016.

[2]Simple Top-Down Parsing
http://effbot.org/zone/simple-top-down-parsing.htm

There are two other documents for setting up a VPN to access Cerner. They are present in the same repository.

- Comorbidities - Documentation - Sriganesh Navaneethakrishnan
- Comorbidities Project Documentation