Intelligent Data Extraction System Using Regular Expressions

Ala' Hashesh, Othman Alkhamra, Ahmad Salameh and Abdel Salam Sayyad
Department of Electrical and Computer Engineering
Birzeit University
Birzeit, Palestine
Email: ala.hashesh@gmail.com, othmankhamra@gmail.com, ahmadsalameh93@gmail.com, asayyad@birzeit.edu

Abstract

Data is everywhere, but to extract specific information from huge data could be an exhausting process. However, there are many concepts introduced in computer science can be used to make this problem simpler, such as regular expressions. But, generating a regular expression capable of extracting a predefined string from a text is not an everyday task. In this research, Regular Expression are generated using Genetic Programming. The validity and correctness of a regular expression is decided by making it extract a set of positive examples and ignore another set of negative examples. We validate this method with three datasets related to IPv4 address extraction, article title extraction, and HTML Header extraction. The resulting regular expressions achieved very good accuracy of extraction for the given tasks.

Keywords: Genetic Programming; Regular Expressions; Text Extraction

1. INTRODUCTION

"There is a big data revolution," says Professor Gary King, but this revolution is not only in the quantity of data. "The big data revolution is that now we can do something with the data." [1] As the web continues to evolve, the amount of content and data that a typical user is subjected to increases exponentially. These days, data stream from daily life, from computers, televisions, websites, credit cards and phones [1]. Data production will be 44 times greater in 2020 than it was in 2009 [2]. Transforming this data into meaningful information wasn't and still isn't a trivial task. However, extracting and taking only the data that matters can surely ease this task. Even then, extracting meaningful information from this huge ambiguous data gives rise to many considerations. Regular expressions (Regex) are one of the most important tools in Computer Science, as they are the approach to perform many extraction tasks that may take a lot of time and effort in mere seconds.

Our problem can be summarized in trying to automatically generate a correct regular expression for a given data set, which contains positive and negative examples, using genetic programming. In other words, our problem is to find a correct regular expression, which is able to get all positive examples and ignore all negative examples in the dataset.

Composition of regular expression is not an easy task; since it requires merging different sub expressions with each other in the correct way. The merge is not an easy operation, and so our project will make this easier by using genetic programming (GP), which achieves the merge using both crossover

and mutation operations. GP programs (solutions) are usually represented as a syntax tree with each node belongs to either the function set or the terminal set. The most challenging and most important concept of GP is the fitness function. This function is used to determine how well a program is able to solve a problem [3].

The generated regular expression should have a 100% accuracy for examples outside of the dataset, but that will be impractical with insufficient examples in the datasets.

The importance of our work is to make lives easier. Since it is not easy to compose regular expressions in traditional ways even if you have a good background about regular expressions, our project is presented to average users to allow them to automatically generate the appropriate regular expression for a given data set by just clicking one button. And so, any user can get the appropriate regular expression, even if the user doesn't have any background about regular expressions.

2. RELATED WORK

2.1 Alberto Bartoli et al.

Automatic generation of regular expression from examples using genetic programming is a new subject. The first paper about it was published in 2014. Working at the Machine Learning Lab, University of Trieste, Alberto Bartoli et al. implemented a web application, which generates regular expression automatically from examples using genetic programming.

The approach of Machine Learning Lab is that the user provides a set of examples, each of which is composed by a pair of strings <t, s> where "t" is a text line and "s" is the substring of t that must be detected by the regular expression. A pair where s is empty, means that no string must be extracted from t, in other words t is a negative example. In their implementation, each individual is a tree, that represents a valid regular expression [4].

To begin their work, they defined a function set and a terminal set. The function set includes **the concatenator** which is a binary node that concatenates other nodes or leaves, **possessive quantifiers** (i.e. "*+", "++", "?+", and "m,n+"), **group operator** (i.e."()") and **character classes** (i.e. "[ ]" and "[^]") [4].

The terminal set includes **constants** (i.e. a single character, a number or a string), **ranges** (i.e. "a-z" or "A-Z"), **character classes** (i.e. "\w" or "\d") and **the wildcard** (i.e. the ".") [4].

To test the correctness of each generated program, they utilized two fitness definitions. The first one is given by the Levenshtein distance that defines the minimum number of operations (delete, insert or swap) that must be used to convert a given string to another. The second fitness function simply defines the length of an individual program (i.e. the length of the generated regex). A full run using their system goes through several steps that include:

1. Randomly separate data into a training, validation and testing sets.
2. Run J independent GP Jobs with the same controlling parameters (In practical a population of 500 and a number of generations of 1000).
3. Select the best individual on the training set within each job.
4. From the set of best individuals, test each one on the validation set and output the best one to the user.
5. Repeat evaluation and GP operators until an optimized solution is reached.

Using this approach their system was able to achieve greater results with less time in comparison with other previous work that used other techniques to tackle the same problem. Finally, its important to note that their implementation used only possessive quantifiers to avoid catastrophic backtracking and they used a special transformation mechanism to replace these quantifiers with greedy and lazy ones to add compatibility with JavaScript.

2.2 Yunyao Li et al.

Another contribution in this field was made by Yunyao Li et al. at the University of Michigan [5], where their learning algorithm takes in addition to the labelled examples an initial regular expression. This was done for two purposes, the first one was to provide a knowledge about the structure of the entity being extracted. The second was to restrict the range of the output regular expressions by properly defining their relationship to the input regular expression.

Their regex learning problem based on the identification of the instances of a given entity. "Let R_0 denote the input regex provided by the user and let $M(R_0, D)$ denote the set of matches obtained by evaluating R_0 over a document collection D. Let $M_p(R_0, D) = \{x \in M(R_0, D)$: x instance of E$\}$ and $M_n(R_0, D) = \{x \in M(R_0, D)$: x not an instance of $E\}$ denote the set of positive and negative matches for R_0" [5].

The procedure begins from the initial regular expression R_0 and compares it with a candidate regex R. R is a better extractor than R_0 for the same data collection D, if the set of matches produced by R are contained within the set of available labelled examples, i.e., if $M(R, D) \subseteq M(R_0, D)$ [5].

One of the concepts that was introduced by them is the regex transformations, in which functional set quantifier is replaced with restricted ranges. For example, the replacement of the occurrence of the + in the regular expression \d+ with specific ranges, such as {1, 2} or {3} so it becomes either \ d{1, 2} or \ d{3} [5]. Another concept is the negative dictionaries, in which specific strings are disallowed using the negative lookahead operator. Lookaheads are special constructs that allow a sequence of characters to be checked for matches against a regex without the characters themselves being part of the match [5].

3. OUR APPROACH

The user provides a set of positive and negative examples, where the positive examples represent the desirable outputs that the regular expression should extract, whereas the negative ones mustn't be extracted by the regular expression.

Our implementation is based on the Genetic programming, in which each valid regular expression in each population in each generation is represented as a tree. However, to be able to use Genetic Programming to generate valid regular expressions function set, terminal set and fitness function must be defined.

In general, and to ensure that the primitive set is sufficient for any extraction task, both the function set and terminal set should include every possible regex character and function. However, including all possible characters for an easy task can make it even harder, and so we decided to start with a basic and sufficient primitive set that could be expanded later on. First, the function set includes:

1. **Regex Quantifiers** (i.e. "*", "+" and "?").
2. **Concatenation function** that concatenates any type of nodes.
3. **List, repetition and Group operators** (i.e. "[]", "{}", and"()").
4. **Other operators** (i.e. "^", "$", "|", ...etc).
5. Second, the terminal set includes:
6. All possible common characters and symbols. (a, b, . . ., z, A, B, . . ., Z, 0, 1, . . ., 9, $, %, ...).
7. All special characters including an empty string character. ("\d", "\w", ".", ...).

The fitness function of our approach is the summation of two fitness definitions, the first being the length of the extracted output, using the valid solution, subtracted from the length of the example itself. The second is just the length of the extracted text from a negative example.

$$f_1(L) = \sum_{i=1}^{n}(L(P_i) - L(R_i)) \quad (1)$$

$$f_2(L) = \sum_{i=1}^{n} L(N_i) \quad (2)$$

Equation 1 represents our fitness function when evaluating a program against positive examples, where $L(P_i)$ represents the length of the example, and $L(R_i)$ represents the length of the output of the regular expression. This equation is applied only if the extracted output R_i is a part of the i-th positive example.

Equation 2 represents the fitness of a program against negative examples, where $L(N_i)$ represents the length of the extracted text from the negative example. For example, if we have "45a" as a negative example and "\d+" as a regular expression, then the extracted text is "45" and L("45") is 2, which will increase the fitness indicating an inaccurate regular expression. As far as the summation of both fitness functions become closer to zero, this raises an impression that the solution is closer than before. Ideally, when the summation becomes zero, then the solution is reached.

4. EXPERIMENTS

In all of our experiments, we used the EpochX framework [6], which is an open source java framework, designed to provide all the necessary tools and facilities to study and perform genetic programming operations.

4.1 IPv4 Address Extraction

The main purpose from this example was to test the correctness of the system, and its ability to solve a real problem at hand. The EpochX framework was used to generate a valid regular expression that extracts all valid IP's from a give text. To make our initial implementation simple we used a very simple and trivial fitness function. For a positive example, the score of the regex is either zero for a perfect extraction, one for a partial extraction and two for empty extraction; on the other hand, a score of three is given if the regex extracted any part of a negative example.

As previously stated we took a sufficient subset from the primitive set, that can help in solving this problem including: number ranges, the dot character, and several regex functions. To train programs to extract positive IP's we generated a training set of one-thousand random valid IP, and to reject a negative example, each program was trained with one-thousand random invalid IP's. To test an extractor, we used a testing set of a thousand valid IP, and one-thousand invalid IP.

4.2 Titles Extraction

In this example, the EpochX framework is used to generate a valid regular expression that extracts all valid titles which are defined as "title={Some Title}". For this example, we used the updated fitness definition given in (1) and (2), to increase the probability of solving the problem within the predefined period.

In this example, the positive training set includes one- hundred valid titles, and the negative training set includes one-hundred invalid titles. On the other hand, the positive testing set includes one-thousand valid titles, and the negative testing set includes one-thousand invalid titles.

The terminal set includes the dot character '.' and an empty string character. Next, we have the '\w' character, which matches a whole word, and the '\b' character, which specifies a word boundary; for example, "\ba" would match an \a" only if it's preceded by a word. Finally, we included "=", "{" and "}" characters.

The function set includes the already defined basic function set with an extra new function called the Curly Brackets Function "{}". This function specifies a specific amount of repetition, for example "3{2}" would match "33".

4.3 HTML headings extraction

As we moved from one example to another, we kept adding tweaks and small modifications to improve the efficiency of our runs and fix any previous bugs. So this example can be considered as a demo for what the system is truly capable of. Here in this example the task is to extract all HTML headings given in the format "<*h{number*} {*any parameters for the html*} > {*any text*} </{*number*}h>".

Each program was trained with one-hundred valid positive example, and one-hundred invalid positive example. Also the correctness of each program was tested against one-thousand valid example, and against one-thousand invalid example.

The function set for this example was the already defined function set, while the terminal set included the empty string character, the dot character ".", and any necessary string that can help the extraction including: "<", "h", ">", "/".

5. RESULTS

Based on long and thorough experiments, which last for one month, we found out that the average time needed by successful programs to reach a solution is 2 hours. Therefore, the runs were only allowed to be executed on the processor for 2 hours, which means that we consider the run in our computations only if it finished execution within 2 hours.

In order to obtain the best parameters (crossover rate, mutation rate, population size), we had to carry out runs with different range of parameters. And in **Table 1** we classified the best parameters' results.

The best parameters were chosen according to the best ratio between the number of successful executions to the total number. As each set of parameters was applied 20 times, the number of successful runs is the number of times it finished execution before the deadline. If two or more sets of parameters achieved the same number of successful runs, the time of each run is the decisive factor. To see the effect of parameter tuning, we excluded the IP experiment from it, by pre-setting the parameters, to the early obtained ones.

5.1 IPv4 Address Extraction

Preliminary results showed that the system can reach a solution but it takes a while to get there. Among our runs the best program was able to classify all positive IP's, and 70% of the invalid IP's. The best resulting regular expression is:

```
8+39$?09+[68]*8+3|[\d?5((46)0)]\d?[\d-
4*\.+\d?5(00)]* \.+\d?[\d?5(00)]?[\d0-5|ˆ0-2?5ˆ0-
[3?52]$-ˆ3|4|3?8]$
```

To test the correctness of the solution we ran tests on the testing dataset, producing an accuracy of 100% on the positive testing set and an accuracy of around 70% on the negative testing set. The parameters that were used to get this result are shown in **Table 2**.

5.2 Titles Extraction

Preliminary runs showed great results with high performance indicators, however we didn't want to rush a positive conclusion so we started a series of experiments designed to find the best parameters combination that can grantee good results. We tested several parameters combinations (around hundred) each of which was used to find a solution twenty times. The best parameters, which are shown in **Table 2**, were educed from the best running results shown in **Table 1**. The best resulting regular expression is:

```
ˆ title..(...\w* |{.||}||\w)*({*..\w*.}|ˆ.|(......... |)
(...\w*.\w* ||\w)*})
```

5.3 HTML Headings Extraction

Most of the time the system was able to get a fast and efficient solution, however, to increase confident we did the same thing that we did previously. We conducted several parameters experiments similar to the previous example, in which we end up with the best parameters shown in **Table 2**, these parameters give the results shown in **Table 1**. The resulting regular expression is:

```
(<h?+\d+.*?(.</h*\d*>+.*?
```

Table 1: Summarization of runs results

Task	Number of examples				Results				Time	
	Training		Testing		Accuracy	Precision	Recall	F-Measure	Min	Average
	Positive	Negative	Positive	Negative						
IPv4	1000	1000	1000	1000	85%	76.92%	100%	86.95%	47:37	01:56:48
Title Extraction	100	100	1000	1000	94.84%	99.53%	90.22%	94.83%	15:16	56:21
HTML Headings Extraction	100	100	1000	1000	100%	100%	100%	100%	00:16	01:19

Table 2: Best Parameters

	Crossover rate	Mutation rate	Population size
IPv4	0.800	0.070	350
Title Extraction	0.500	0.275	250
HTML Headings Extraction	0.870	0.080	550

6. CONCLUSION AND FUTURE WORK

As our results show in **Table 1**, we were able to get a good accuracy in the three applications that we did so far. Furthermore, the applications that used parameter tuning were able to get better results. The IPv4 address regex succeeded in recognizing all the valid IPv4 addresses, whereas it failed in determining some of the incorrect IP's but with an acceptable accuracy as discussed before.

For the second application, the resulting regular expression was able to extract all the strings that begin with "title" until the closing curly bracket. However, very small number of testing examples were not extracted by that regular expression, and that's why the accuracy is not 100%.

For the third application, the generated regular expression was able to extract all positive examples whether they are testing or training, and to ignore all negative ones. In fact, this application was done after all the updates in the fitness function, and it showed excellent results, and had an accuracy of 100%.

Overall, we can see that applying Genetic Programming concepts to solve the problem of generating a specific regular expression is a very good approach. Since, it can take characteristics of different good regular expressions to generate an even better one, while a brute force solution would require testing all possible combinations of regex alphabet to make this possible, which might increase the execution time significantly.

As a future work, we will work on the optimization of the execution time of our runs as much as possible, in addition to the optimization of the length of the resulting regular expression. Moreover, we will conduct experiments on additional datasets in order to further generalize and validate our approach. Basically, we will improve our results horizontally by including more datasets, and vertically by improving the technique to produce shorter, more efficient regular expressions within faster runtimes. Furthermore, we will work on making our system available for users by making an application with a graphical user interface, and this will help us in making our system more flexible and accurate.

ACKNOWLEDGEMENT

We would like to thank Eng. Ghaleb Salah for providing cloud computing resources that we used for the experiments.

REFERENCES

[1] J. Shaw, "Why "big data" is a big deal," Harvard Magazine, 2014.

[2] W. Koff and P. Gustafson, "Data revolution," http://www.csc.com/ innovation/ds/84818-data revolution, Csc Leading Edge Forum, 2011.

[3] "The gp tutorial," http://www.geneticprogramming.com/Tutorial/, [On- line; accessed 01-April-2015].

[4] Bartoli, Alberto, et al. "Automatic generation of regular expressions from examples with genetic programming." Proceedings of the 14th annual conference companion on Genetic and evolutionary computation. ACM, 2012.

[5] Y. Li, R. Krishnamurthy, S. Raghav, and S. Vaithyanathan, "Regular expression learning for information extraction," 2008, iBM Almaden Research Center, San Jose, CA 95120.

[6] "Quickstart guide to epochx," http://www.epochx.org/quickstart- guide. php/, [Online; accessed 01-April-2015].