PRACTICAL REGULAR EXPRESSION MINING AND ITS INFORMATION QUALITY APPLICATIONS.

(Practice-Oriented Paper)

Sergei Savchenko
DecisionsToday Inc. Canada
ssavchenko@decisionstoday.com

Abstract: Regular expressions are convenient devices representing common patterns in collections of text strings that can be used as filters insuring information quality in textual data. An algorithm inducing a representative regular expression given a set of text strings (possibly containing errors) is described. Such an algorithm is useful in estimating information quality and performing automated cleansing of legacy data or the data obtained by the means of automated sensing (e.g. OCR). A number of practical heuristics improving algorithm's real-life performance are introduced. A framework employing this algorithm is outlined.

Key Words: Data Quality, Information Quality, TDQM, Regular Expression, Data Mining, Automaton Induction

INTRODUCTION

Regular expressions are convenient devices to express common patterns in collections of text strings. These can be used as information quality filters insuring that a text string meets certain criteria.

To recall, a regular expression over some alphabet (e.g.: {*a*,*b*,*c*}) is:

- an atomic expression emitting a string consisting of a single symbol from its alphabet (e.g. *a*,*b* or *c*);
- if r_1 and r_2 are regular expressions then so is the sequence of expressions $r_1 r_2$ producing strings where the prefixes are emitted by the first expression followed by symbols produced by the second expression;
- if r_1 and r_2 are regular expressions then so is a disjunction of the expressions $r_1 \mid r_2$ producing strings of either expression r_1 or r_2;
- if *r* is a regular expression then so is r^* where symbols produced by *r* are repeated *0* to *n* times.

Additionally, brackets are used in a conventional fashion to disambiguate operator precedence. Although the constructive rules described above are sufficient, commonly software systems that use regular expressions introduce some syntactic additions. It is for instance to denote rr^* as r^+ thus describing a regular expression occurring *1* to *n* times. Also common are character classes such as, for instance, in the expression [*a*-*z*] to denote a disjunction of all letters between *a* and *z* (i.e. *a*|*b*|…|*z*).

As an example, a regular expression $(ab^+)^+$ matches any string such as *ab*, *abb*, *abab*, *abbbabb*, *ababbbabb* etc. but does not match a string such as *abcab*.

An interesting problem is that of *regular expression induction* or *regular expression mining*. Given a set of text strings we want to automatically derive a regular expression that is likely to describe strings in the set. If that is possible, there are obvious information quality applications. For example, given a large collection of unknown strings we can use a small portion of this collection as a training set to derive the most likely regular expression that can be further used to filter the entire string collection.

There are theoretic results attesting to regular expression induction being a difficult problem. It was shown by Gold in [3] that induction of a regular expression from positive examples only is in fact undecidable. Additional negative examples or ability of the algorithm to make membership queries are also necessary. Both negative examples and queries are problematic in practical settings where the nature of the data in the training set is usually unknown.

Despite this strong negative result there are multiple available heuristic approaches producing satisfactory results in practical situations. Many of the heuristic algorithms rely on state merging methodology first introduced by Angluin in [1]. This article describes an algorithm that we have implemented that is based in part on the work presented in [2], [4], [5] and [6]. In particular, we inherited best first search approach similar to the one taken in [2]. Unlike in [2], however, we allow for a limited, one node deep, backtracking. We use much simpler node comparison criteria then the one outlined in [5] since it appears from our experiments that additional heuristics (outlined further) are of larger practical utility compared to a more complex node comparison criteria.

In the following sections the basis of the algorithm is described, a method to deal with dirty training sets is suggested followed by the description of important additional heuristics and the overall framework to use this algorithm in insuring data quality in large collections of text strings.

AUTOMATON INDUCTION

The proposed algorithm starts by constructing an over-specific automaton that represents all strings in the training set. Nodes in this automaton are marked with emitted symbols whereas directed edges indicate which nodes follow which nodes in the produced strings. An over-specific automaton is obtained when we create two special nodes that do not emit symbols and represent the beginning and the end. We can further represent each string as a node chain connecting the initial and the final nodes. Clearly, this automaton exactly describes the training set and nothing else.

To illustrate this, let us suppose we are given a training set consisting of only two strings:

abb

abab

An over-specific automaton for this set is presented in Figure 1, *(a)*.

The algorithm proceeds by merging pairs of similar nodes in the automaton. We are looking for such nodes that have very similar predecessor nodes, are similar themselves in terms of symbols that they produce and also have similar successor nodes. The comparison criteria could potentially be recursive or limited to the immediate neighbors.

The comparison criteria that we currently use is a weighted sum

$$c_1 \frac{|p_1 \cap p_2|}{|p_1 \cup p_2|} + c_2 \frac{|n_1 \cap n_2|}{|n_1 \cup n_2|} + c_3 |t_1 \cap t_2| + c_4 \frac{|s_1 \cap s_2|}{|s_1 \cup s_2|},$$

where p_1 and p_2 are sets of symbols emitted by the immediate predecessor nodes, n_1 and n_2 are sets of symbols produced by the nodes being compared, t_1, t_2 are types of symbols in compared nodes and s_1 , s_2 are sets of symbols produced by the immediate successor nodes. Ratios of the size of the intersection over the size of the union describe how similar individual components of the weighted sum are. Every node is only allowed a single type (such as a character or a number or a separator type). Thus, the intersection of types will be *1* if the nodes are of the same type and *0* otherwise. The weight coefficients used could be as simple as *.3, .3, .1* and *.3* for c_1, c_2, c_3 and c_4, respectively. The weighted sum gives a value between *0* and *1* so that it is close to *1* for very similar nodes and close to *0* for dissimilar ones.

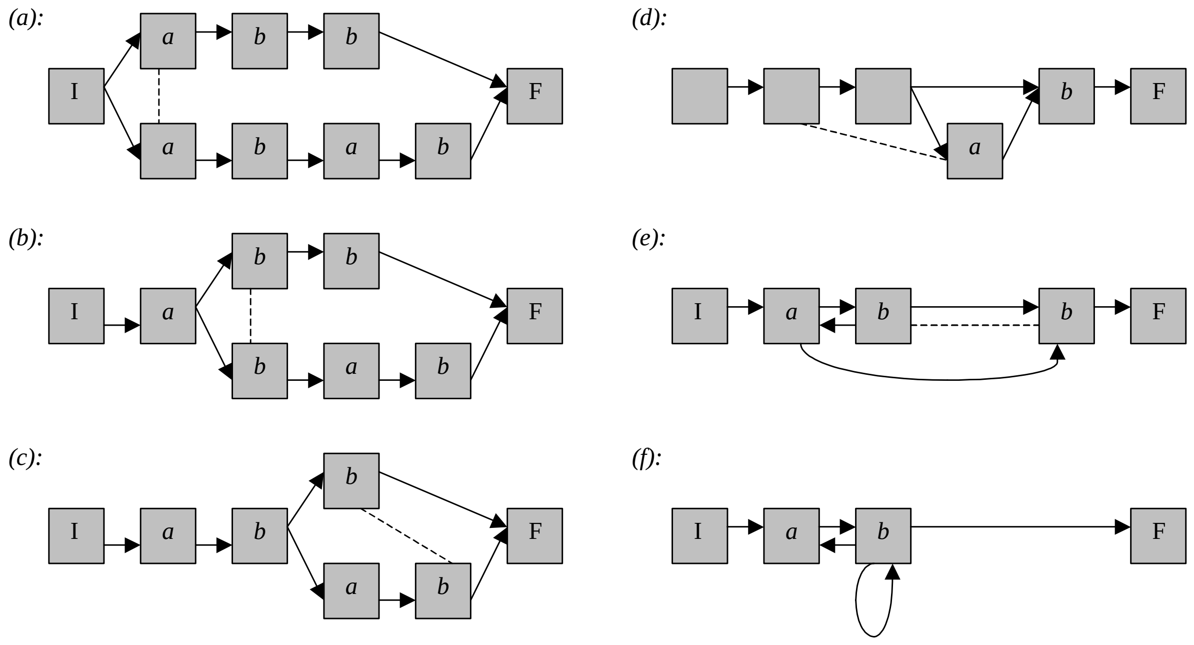

Figure 1: Automaton induction.

The algorithm proceeds iteratively, at each step merging a pair of nodes with the best comparison coefficient. When all comparison coefficients fall bellow a certain threshold, we stop the merging process.

Values in the range from *.4* to *.5* insure that the nodes with the comparison coefficient above this threshold are similar enough. The experimental observations seem to suggest that a larger threshold may prevent some important merges from occurring whereas lower threshold leads to too generous generalizations in the resulting regular expressions.

Figure 1 illustrates the merging process for a sample training set. Note that the broken line in the figures identifies nodes that are to be merged. For example, initially, in subfigure *(a)*, the two nodes producing symbol *a* are found to be very similar since they share the same predecessor node and their successor nodes emit the same symbol. Thus they are merged together.

To merge two nodes, we create a new node such that it emits any symbol emitted by the parents and shares all of the arriving and departing edges. Although originally, any node produces only a single symbol, it is possible that after a merge a node will be producing one of several symbols inherited from node's parents.

Loops could be obtained in the merging process if the merged nodes were connected to one another. *(see Figure 1 (f)).*

By looking at the derived automaton it is not hard to see that the result represents quite well the training set. Indeed it describes strings containing multiple blocks of repeating *b* symbols separated by single *a* symbols – exactly what the training set seems to suggest.

MANAGING DIRTY TRAINING SETS

A significant obstacle to obtaining usable regular expressions is the presence of data errors in the training sets. To identify strings with potential problems we use statistical properties of the data in the training set.

We record the number of distinct training set strings each edge originated from. When the over-specific automaton is created, every edge has a single parent string. When nodes are merged, the edge origin information is updated. For instance, if two nodes *i* and *j* that are to be merged share a common successor node *s*, when the merged node *m* is created we ensure that the edge that connects it to the successor node *s* is marked to originate from any of the strings edges from *i* to *s* and *j* to *s* originated from.

Consider the following example. Let's suppose we have a training set consisting of the following five numbered strings:

0: abb

1: abab

2: abb

3: abab

4: c

Figure 2 *(a)* illustrates the over-specific automaton built for this training set. Note that all edges originate from only one string. Once the automaton is derived, we can identify the least frequent edges. If their origin list is so small that they originate from only few training set strings, we assume that these edges are likely indicative of a data error.

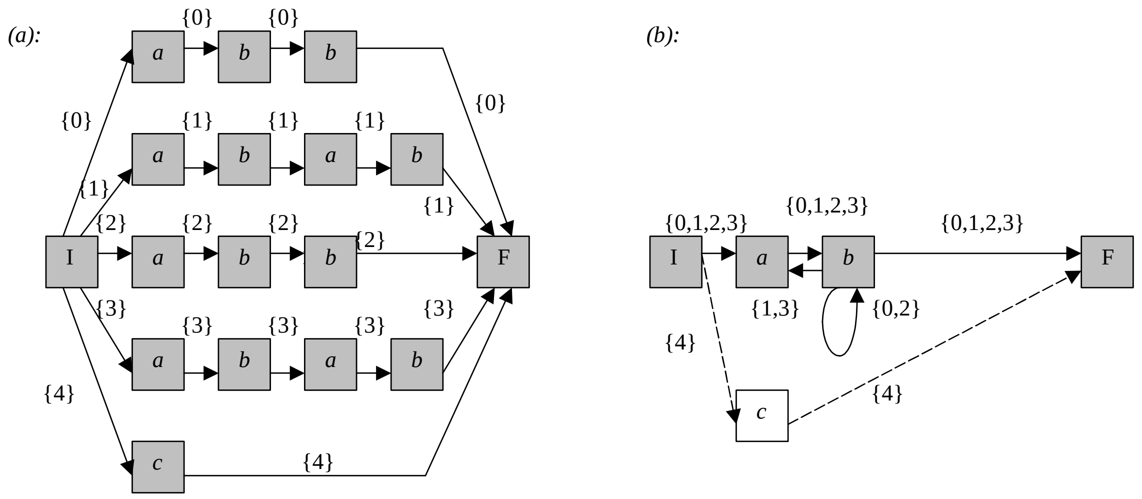

Figure 2: Identifying errors in the training set

Consider the derived automaton in Figure 2 *(b)*. In this figure the edges entering and leaving the node producing the character *c* have the shortest origin lists among all edges in the automaton. The parent string of these edges (the string with the id *4*) is thus deemed to contain a data error.

We must discard all potentially erroneous strings from the training set and regenerate the automaton from scratch for the remaining strings.

It should be noted that at this stage we could also recognize a situation when the derivation of a representative regular expression is not possible. If a significant percentage of training set strings have to be discarded at that stage we conclude that the training set doesn't contain a clear pattern and terminate the derivation process.

FINDING THE REGULAR EXPRESSION FROM THE AUTOMATON

Once the automaton is derived, we would like to convert it into a regular expression. The following algorithm can deduce a regular expression from an automaton.

We proceed by iteratively deleting every node in the automaton (with the exception of the initial and the final nodes) and replacing the deleted nodes by edges marked with symbols that the nodes used to emit.

For instance, consider Figure 3 *(a)*. It represents the situation where we want to delete a node producing symbol *s*. Since this node has a loop, when passing through the node, we will necessarily produce a sub-string of at least one repeating *s*. Thus, any two nodes that are connected via the current node can be joined directly by an edge marked with s^+ sub-expression. When this is done for all nodes connected via the current one, the node itself can be safely deleted.

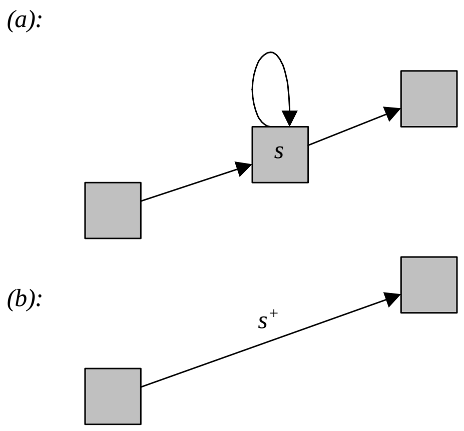

Figure 3: Easy case of node replacement.

It is beneficial to allow only a single marked edge to connect any two nodes. When another marked edge between the pair of nodes has to be created, we can instead update the expression marking the original edge. The new expression will simply be a disjunction of the new and the old expressions.

Figure 3 illustrated a trivial case. In a general case represented in Figure 4 there could already exist marked edges and marked loops. Since we are imposing a condition to allow only one edge to connect any two nodes in one direction there may not be more than a single marked and a single unmarked loop. Figure 4, thus, represents the general transformation to delete a node by replacing it with marked edges.

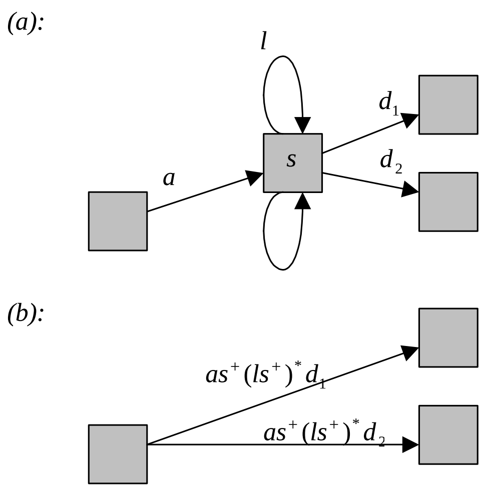

Figure 4: General case of node replacement.

Assuming that the arriving edge was marked by an expression *a*, the node itself emits an expression *s*, the departing edge is marked by an expression *d* and that there is also a loop marked with the expression *l* as well as an unmarked loop, the resulting expression to be placed on the new marked edge will be $as^+(ls^+)^*d$. It is not hard to see why this is so. On entering the node, the unmarked loop can be taken multiple times (represented by first s^+ in the expression). At an arbitrary point, the marked loop can be taken *0* to *n* times. However on every new arrival back to the node, the unmarked loop may be used once again.

Thus, to delete a node, every arriving edge must be reconnected with every departing edge using the procedure just described. In the end, all nodes are deleted with the exception of the initial and the final nodes that will end up connected by a single edge marked with the expression we are seeking.

Figure 5 illustrates the entire derivation process for the example presented earlier.

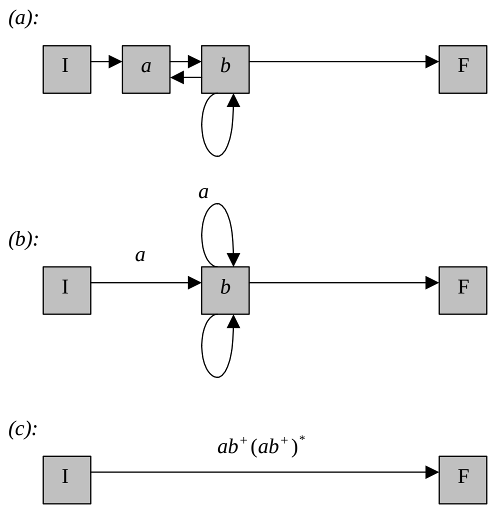

Figure 5: Regular expression derivation.

It should be noted that some post-processing could shorten the resulting expression even further. In the particular case of the example it can be shortened to just $(ab^+)^+$.

ADDITIONAL HEURISTICS

Although presented algorithm works quite well on small alphabets, in unmodified form it is not particularly useful on typical sets of strings occurring in industrial applications.

A number of extensions were introduced to improve practical performance. With the extensions we were able to successfully mine regular expressions from multiple typical string sets often occurring in the real-life situations, such as lists of email addresses, domain names, postal codes and dollar amounts.

The first extension allows us to recognize character classes. It is based on the observation that regular expressions often contain such character classes as [*a-z*], [*A-Z*] and [*0-9*]. To obtain such character classes we can explicitly examine every node in the automaton during the derivation. If a subset of predecessor (or successor) nodes produces a significant portion of some character class we can perform a generalization operation by collapsing all these nodes together into a single node marked with corresponding character class expression.

This generalization heuristic is only applied if there are no appropriate merge candidates.

The second extension is based on the observation that many types of strings are of a finite length. In this case we want to avoid excessive generalizations. For instance, a zip code is a number occurring five times – not an arbitrary number of times. If a training set is determined to have a predominant length (or lengths) we activate a restriction to disallow merges that result in loops in the automaton.

The third extension relies on the special nature of separator characters. These often occur exactly once in the string families we are likely to encounter in many industrial applications. For example @ symbol occurs exactly once in email addresses as does *$* symbol in dollar amounts or a dot in decimal numbers.

Thus, when the training set is read we analyze the strings and identify such predominantly unique separator symbols. This process also permits discarding some training set strings where a predominantly unique separator occurs multiple times. Additionally, in the run time we disallow merges that result in loops containing the unique separator.

The heuristics outlined above have a significant impact on the quality of regular expressions that were able to produce. They often disallowed unreasonable generalizations and helped to make reasonable ones.

Consider the following test case. A set of about two thousand strings representing dollar amounts was obtained. We have randomly selected a small sub-set containing only 15 strings.

Figure 6, shows the output of our standalone mining tool working on the selected training set of strings.

```
Regular expression Miner (c) DecisionsToday 2002.
Training set:
(00)=<$298.51>
(01)=<$76.95>
(02)=<$719.31>
(03)=<$147.51>
(04)=<$59.l0>
(05)=<$453.51>
(06)=<$921.51>
(07)=<$45.54>
(08)=<$15.1O>
(09)=<$12.45>
(10)=<$31.89>
(11)=<$$9.45>
(12)=<$32.10>
(13)=<$56.31>
(14)=<$90.22>
Unique separators: <$><.>
No predominant lengths
String (11)=<$$9.45> disqualified due to separator composition
Run 1: Merges: 59; Generalizations: 3+2;
String (04)=<$59.l0> disqualified due to potential data errors
String (08)=<$15.1O> disqualified due to potential data errors
Run 2: Merges: 47; Generalizations: 3+2;

Regular expression: <$[0-9]+.[0-9][0-9]>
```

Figure 6: Sample output.

The training set in Figure 6 contained several data errors that were identified at different stages in the mining process. One string was disqualified early on due to its separator composition whereas two other strings were disqualified since they have produced edges with short origin lists. Note that two unique separators were discovered.

The obtained regular expression was subsequently used to filter the entire set of two thousand strings representing dollar amounts. This filtering process identified many (but not all) optical character recognition related errors as well as a few probable typing errors. We estimated that in this particular case over 90% of all data errors were identified.

ALGORITHM'S COMPLEXITY

The automaton derivation part of the described algorithm runs in $O(n^4)$ time where *n* is the number of symbols in the training set. The asymptotic O-notation implies that the number of elementary steps the algorithm will require on the problem of input size *n* will be bounded from above by the function n^4, which is thus the worst-case complexity.

This high order of complexity is because during automaton derivation to determine appropriate merge candidates it is necessary to compare every node with every remaining node (thus n^2 in the worst case), every node may be linked (in the worst case) to all other nodes in the automaton thus the cost of comparison could be as bad as *2(n-1),* and there could be as many as *n-1* pairs of nodes to merge (thus n^4). If we used some recursive comparison criteria the complexity would have been even higher.

It should be noted however that the use of hashing techniques might reduce the effective order of complexity by at least one degree. Indeed, every merge on practice induces only a minor change to the structure of the automaton and thus we can reuse during subsequent iterations the results of comparisons for the nodes that were remote from the merged nodes.

The running time of regular expression derivation is $O(m^3)$ where *m* is the number of nodes in the resulting automaton (which we assume will be on average significantly less than *n*). It is because in regular expression extraction process we have to iteratively delete *m-2* nodes. When deleting a single node it could have as many as *m-1* predecessors and *m-1* successors. Since every predecessor edge has to be connected to every successor edge, deletion of an edge has the complexity of $O(m^2)$ thus giving the overall complexity of $O(m^3)$.

CONCLUSION

The described algorithm is employed in a tool that is being developed to perform automated data profiling and cleansing of industrial databases. This tool is a part of a framework that is being developed by *DecisionsToday* to automate building data warehouses.

In the framework, we employ multiple mining methods to identify patterns in data. The identified patterns could be subsequently used to filter the data to either give an estimate of data quality or to find all potential data errors.

Thus, for regular expression pattern we perform the following:

- Members of the training set are randomly selected. We require 30-50 strings to attempt regular expression mining.
- An attempt is made to derive a regular expression.
- If a regular expression is derived, it can be examined and modified by the user.
- Given a regular expression several features are enabled:
    - A larger set of strings can be selected and matched against the regular expression to give a data quality estimate.
    - Alternatively, all strings can be filtered to report potential data errors.

Note that we require a training set of a very moderate size (30-50 entries). Every string contains a significant amount of information and quite often a reasonable regular expression could be derived from even such a small training set. Assuming that the data we are analyzing does have a recoverable pattern, the approach to use a small portion of the data to deduce a regular expression and further use this expression to filter the entire data set appears quite attractive for practical applications.

In general, regular expression mining is an important technique that can be used to insure information quality in large string collections. The algorithm described in this article augmented by outlined heuristics seems to be quite effective on certain kinds of data typical in industrial databases. The ability to identify data errors in the training sets and the ability to identify training sets where induction of a useful regular expression may not be possible increase the practical utility of our approach.

In the ongoing effort to develop this technique further we are looking for new heuristics as well as methods to speed up the algorithm.

We also look into mining other kinds of patterns that could also be used to estimate quality and cleanse data in a fashion similar to the one outlined for regular expressions.

REFERENCES

[1] Angluin, D., and Smith, C. Inductive Inference: Theory and Methods. Computing Surveys. 1983, 3(15).

[2] Carrasco, R., and Oncina, J., Learning Stochastic Regular Grammars by Means of State Merging Method. *Proceedings Second International Colloquium ICGI*. 1994.

[3] Gold, E., Complexity of Automaton Identification from Given Data. *Information and Control.* 1978.

[4] Miclet, L., Structural Methods in Pattern Recognition. Springer-Verlag. 1986.

[5] Stolcke, A., and Omohundro, S., Inducing Probabalistic Grammars by Bayesian Model Merging. *Proceedings Second International Colloquiym ICGI*. 1994.

[6] Goan, T., Benson, N., and Etzioni, O., A Grammar Inference Algorithm for the World Wide Web. *Proceedings of the AAAI Spring Symposium on Machine Learning in Information Access*. 1996.