RegExpMiner: Automatically discovering frequently matching regular expressions

Julien Rabatel[1], Jérôme Azé[1], Pascal Poncelet[1], and Mathieu Roche[1,2]

[1] LIRMM, CNRS UMR 5506, Univ. Montpellier 2, 34095 Montpellier, France
[2] UMR TETIS - Cirad, Irstea, AgroParisTech, 34093 Montpellier, France

Regular expressions (REs) are a very powerful and popular tool to manipulate string data in a variety of applications. They are used as search templates to look for the occurrences of a given piece of text in a document, or to define how a given piece of text should be formatted in order to be valid (e.g., to check that the value entered in an email field of a Web form is correctly formatted), or even to help solving more complex NLP tasks [NS07]. Their popularity in those various application domains arises from several reasons. First, they are easy to understand and manipulate for common usages, despite their wide expressiveness and power of abstraction. Second, they are natively usable within a large variety of programming languages, hence making them suitable to be integrated into every project addressing text processing tasks. Their usage often relies on a very limited amount of hand-crafted REs. It is indeed difficult to automatically obtain the REs matching with a given set of strings for which no *a priori* knowledge about their underlying formatting rules is given. Such an automatic discovery of REs would nonetheless offer some very interesting prospects. Regular expressions indeed have an interesting abstraction power as they are able to provide information about how textual content is formatted, rather than focusing on the actual sequences of characters. Having a more abstract description space for describing textual content then offers new insights. For instance, an application scenario consists in data cleaning problems. Given a database containing some textual content about entities (e.g., addresses, names, phone numbers, etc.), one may be interested in finding values contained in the database that are mistakes from the people who entered them. Such typos and formatting mistakes can easily be highlighted if they result in strings that do not match the same regular expressions as the majority of the other strings.

While regular expressions can be seen as interesting descriptors of textual data for various NLP and machine learning tasks, they are hard to obtain. The literature does not offer fully relevant solutions when one wishes to enumerate some REs to describe a given set of strings. Regular Expression learning [Fer05], for instance, consists in building a single regular expression matching with a given set of positive string examples. Such approaches typically do not allow exceptions w.r.t. the set of strings to be matched, hence losing their interest as soon as input data are noisy. Additionally, only one RE is learned while one can expect to obtain several REs reflecting the different templates that co-exist in the data. E.g., one cannot expect all the values of a list of international ZIP codes to respond to only one template, as each country may use a different one. Constructing one single RE matching with all of them will of-

ten lead to an over-generalization of the underlying templates that would make the obtained RE irrelevant in practical applications. On the other hand, the sequence mining literature, when applied to string data, offers the possibility to discover more various templates via frequent patterns, i.e., data fragments occurring in a sufficient amount of strings. While this general principle answers the problems above-mentioned for RE learning approaches, the type of extracted patterns (e.g., sequential patterns [AS95], episodes [MTV97]) is typically much less expressive than REs. Some efforts have however been put in allowing the generalization of sequence elements [PLL$^+$10] but extracted sequential patterns have little commonality with REs, as they only aim at discovering sequence elements that are frequently found in the same order.

We propose an approach for extracting regular expressions under the form of frequent patterns in textual data. To this end, we define a relevant pattern language that offers some interesting algorithmic properties. While we do not aim at exploiting all the characteristics and expressiveness of the RE language, we focus on providing a preliminary approach by keeping some of its main features. In particular, we fully consider the problem of allowing the generalization of characters via the use of predefined *character classes*, commonly used in REs[3]. Another aspect that this approach takes into account is the repetition of some characters in strings. For instance, we assume that the strings "*012*" and "*9876543*", should both be generalizable to the RE /[0−9]+/, i.e., a list of consecutive digit characters, even if they do not contain the same digits nor the same amount of digits. We define the *frequent regular expression pattern* mining problem by providing a theoretical framework linking together the RE and sequence mining worlds, and highlight some properties that, while inspired from known properties in sequence mining, are specific to the problem we consider study and employs them to design the `RegExpMiner` algorithm to mine such patterns.

References

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 3–14. IEEE, 1995.

[Fer05] Henning Fernau. Algorithms for learning regular expressions. In *Algorithmic Learning Theory*, pages 297–311. Springer, 2005.

[MTV97] Heikki Mannila, Hannu Toivonen, and A Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[NS07] David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, 2007.

[PLL$^+$10] Marc Plantevit, Anne Laurent, Dominique Laurent, Maguelonne Teisseire, and Yeow Wei Choong. Mining multidimensional and multilevel sequential patterns. *ACM Transactions on Knowledge Discovery from Data*, 4(1), 2010.

[3] Character classes are sets of characters. When tested against a string, a character class matches with any of the characters it contains. For instance, the character class [0−9] contains all the digit characters $0, 1, \cdots, 9$, which allows it to match with strings such as "3" or "8", but not with "A".