Regular Expression Learning for Information Extraction

Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan
IBM Almaden Research Center
San Jose, CA 95120
{yunyaoli, rajase, rsriram}@us.ibm.com, shiv@almaden.ibm.com

H. V. Jagadish[*]
Department of EECS
University of Michigan
Ann Arbor, MI 48109
jag@umich.edu

Abstract

Regular expressions have served as the dominant workhorse of practical information extraction for several years. However, there has been little work on reducing the manual effort involved in building high-quality, complex regular expressions for information extraction tasks. In this paper, we propose ReLIE, a novel transformation-based algorithm for learning such complex regular expressions. We evaluate the performance of our algorithm on multiple datasets and compare it against the CRF algorithm. We show that ReLIE, in addition to being an order of magnitude faster, outperforms CRF under conditions of limited training data and cross-domain data. Finally, we show how the accuracy of CRF can be improved by using features extracted by ReLIE.

1 Introduction

A large class of entity extraction tasks can be accomplished by the use of carefully constructed regular expressions (regexes). Examples of entities amenable to such extractions include *email addresses* and *software names* (web collections), *credit card numbers* and *social security numbers* (email compliance), and *gene and protein names* (bioinformatics), etc. These entities share the characteristic that their key representative patterns (features) are expressible in standard constructs of regular expressions. At first glance, it may seem that constructing a regex to extract such entities is fairly straightforward. In reality, robust extraction requires the use of rather complex expressions, as illustrated by the following example.

Example 1 (Phone number extraction)**.** *An obvious pattern for identifying phone numbers is "blocks of digits separated by hyphens" represented as* R_1 = `(\d+\-)+\d+`.[1] *While* R_1 *matches valid phone numbers like* 800-865-1125 *and* 725-1234*, it suffers from both "precision" and "recall" problems. Not only does* R_1 *produce incorrect matches (e.g., social security numbers like* 123-45-6789*), it also fails to identify valid phone numbers such as* 800.865.1125*, and* (800)865-CARE*. An improved regex that addresses these problems is* R_2 = `(\d{3}[-.\ ()]){1,2}[\dA-Z]{4}`.

While multiple machine learning approaches have been proposed for information extraction in recent years (McCallum et al., 2000; Cohen and McCallum, 2003; Klein et al., 2003; Krishnan and Manning, 2006), manually created regexes remain a widely adopted practical solution for information extraction (Appelt and Onyshkevych, 1998; Fukuda et al., 1998; Cunningham, 1999; Tanabe and Wilbur, 2002; Li et al., 2006; DeRose et al., 2007; Zhu et al., 2007). Yet, with a few notable exceptions, which we discuss later in Section 1.1, there has been very little work in reducing this human effort through the use of automatic learning techniques. In this paper, we propose a novel formulation of the problem of learn-

[*]Supported in part by NSF 0438909 and NIH 1-U54-DA021519.

[1]Throughout this paper, we use the syntax of the standard Java regex engine (Java, 2008).

ing regexes for information extraction tasks. We demonstrate that high quality regex extractors can be learned with significantly reduced manual effort. To motivate our approach, we first discuss prior work in the area of learning regexes and describe some of the limitations of these techniques.

1.1 Learning Regular Expressions

The problem of inducing regular languages from positive and negative examples has been studied in the past, even outside the context of information extraction (Alquezar and Sanfeliu, 1994; Dupont, 1996; Firoiu et al., 1998; Garofalakis et al., 2000; Denis, 2001; Denis et al., 2004; Fernau, 2005; Galassi and Giordana, 2005; Bex et al., 2006). Much of this work assumes that the target regex is small and compact thereby allowing the learning algorithm to exploit this information. Consider, for example, the learning of patterns motivated by DNA sequencing applications (Galassi and Giordana, 2005). Here the input sequence is viewed as multiple atomic events separated by gaps. Since each atomic event is easily described by a small and compact regex, the problem reduces to one of learning simple regexes. Similarly, in XML DTD inference (Garofalakis et al., 2000; Bex et al., 2006), it is possible to exploit the fact that the XML documents of interest are often described using simple DTDs. E.g., in an online books store, each book has a title, one or more authors and price. This information can be described in a DTD as $\langle \mathsf{book} \rangle \leftarrow \langle \mathsf{title} \rangle \langle \mathsf{author} \rangle + \langle \mathsf{price} \rangle$. However, as shown in Example 1, regexes for information extraction rely on more complex constructs.

In the context of information extraction, prior work has concentrated primarily on learning regexes over relatively small alphabet sizes. A common theme in (Soderland, 1999; Ciravegna, 2001; Wu and Pottenger, 2005; Feldman et al., 2006) is the problem of learning regexes over tagged tokens produced by other text-processing steps such as POS tagging, morphological analysis, and gazetteer matching. Thus, the alphabet is defined by the space of possible tags output by these analysis steps. A similar approach has been proposed in (Brill, 2000) for POS disambiguation. In contrast, our paper addresses extraction tasks that require "fine-grained" control to accurately capture the structural features of the entity of interest. Consequently, the domain of interest consists of *all characters* thereby dramatically increasing the size of the alphabet. To enable this scale-up, the techniques presented in this paper exploit advanced syntactic constructs (such as character classes and quantifiers) supported by modern regex languages.

Finally, we note that almost all of the above described work define the learning problem over a restricted class of regexes. Typically, the restrictions involve either disallowing or limiting the use of Kleene disclosure and disjunction operations. However, our work imposes no such restrictions.

1.2 Contributions

In a key departure from prior formulations, the learning algorithm presented in this work takes as input not just labeled examples but also an *initial regular expression*. The use of an initial regex has two major advantages. First, this expression provides a natural mechanism for a domain expert to provide domain knowledge about the structure of the entity being extracted. Second, as we show in Section 2, the space of output regular expressions under consideration can be meaningfully restricted by appropriately defining their relationship to the input expression. Such a principled approach to restrict the search space permits the learning algorithm to consider complex regexes in a tractable manner. In contrast, prior work defined a tractable search space by placing restrictions on the target class of regular expressions. Our specific contributions are:

- A novel regex learning problem consisting of learning an "improved" regex given an initial regex and labeled examples
- Formulation of this learning task as an optimization problem over a search space of regexes
- ReLIE, a regex learning algorithm that employs transformations to navigate the search space
- Extensive experimental results over multiple datasets to show the effectiveness of ReLIE and a comparison study with the Conditional Random Field (CRF) algorithm
- Finally, experiments that demonstrate the benefits of using ReLIE as a feature extractor for CRF and possibly other machine learning algorithms.

2 The Regex Learning Problem

Consider the task of identifying instances of some entity $\mathcal{E}$. Let R_0 denote the input regex provided by the user and let $\mathrm{M}(R_0, \mathcal{D})$ denote the set of matches obtained by evaluating R_0 over a document collection $\mathcal{D}$. Let $\mathrm{M_p}(R_0, \mathcal{D}) = \{x \in \mathrm{M}(R_0, \mathcal{D}) : x \text{ instance of } \mathcal{E}\}$ and $\mathrm{M_n}(R_0, \mathcal{D}) = \{x \in \mathrm{M}(R_0, \mathcal{D}) : x \text{ not an instance of } \mathcal{E}\}$ denote the set of positive and negative matches for R_0. Note that a match is positive if it corresponds to an instance of the entity of interest and is negative otherwise. The goal of our learning task is to produce a regex that is "better" than R_0 at identifying instances of $\mathcal{E}$.

Given a candidate regex R, we need a mechanism to judge whether R is indeed a better extractor for $\mathcal{E}$ than R_0. To make this judgment even for just the original document collection $\mathcal{D}$, we must be able to label each instance matched by R (i.e., each element of $\mathrm{M}(R, \mathcal{D})$) as positive or negative. Clearly, this can be accomplished if the set of matches produced by R are contained within the set of available labeled examples, i.e., if $\mathrm{M}(R, \mathcal{D}) \subseteq \mathrm{M}(R_0, \mathcal{D})$. Based on this observation, we make the following assumption:

Assumption 1. *Given an input regex R_0 over some alphabet Σ, any other regex R over Σ is a candidate for our learning algorithm only if* $\mathrm{L}(R) \subseteq \mathrm{L}(R_0)$. *(*$\mathrm{L}$*($R$) denotes the language accepted by R).*

Even with this assumption, we are left with a potentially infinite set of candidate regexes from which our learning algorithm must choose one. To explore this set in a principled fashion, we need a mechanism to move from one element in this space to another, i.e., from one candidate regex to another. In addition, we need an objective function to judge the extraction quality of each candidate regex. We address these two issues below.

Regex Transformations To systematically explore the search space, we introduce the concept of *regex transformations*.

Definition 1 (Regex Transformation)**.** *Let $\mathcal{R}_\Sigma$ denote the set of all regular expressions over some alphabet Σ. A regex transformation is a function $\boldsymbol{T} : \mathcal{R}_\Sigma \to 2^{\mathcal{R}_\Sigma}$ such that $\forall R' \in \boldsymbol{T}(R)$, $\mathrm{L}(R') \subseteq \mathrm{L}(R)$.*

For example, by replacing different occurrences of the quantifier $+$ in R_1 from Example 1 with specific ranges (such as `{1,2}` or `{3}`), we obtain expressions such as R_3 = `(\d+\-){1,2}\d+` and R_4 = `(\d{3}\-)+\d+`. The operation of replacing quantifiers with restricted ranges is an example of a particular class of transformations that we describe further in Section 3. For the present, it is sufficient to view a transformation as a function applied to a regex R that produces, as output, a set of regexes that accept sublanguages of $\mathrm{L}(R)$. We now define the search space of our learning algorithm as follows:

Definition 2 (Search Space)**.** *Given an input regex R_0 and a set of transformations $\mathcal{T}$, the search space of our learning algorithm is $\mathcal{T}(R_0)$, the set of all regexes obtained by (repeatedly) applying the transformations in $\mathcal{T}$ to R_0.*

For instance, if the operation of restricting quantifiers that we described above is part of the transformation set, then R_3 and R_4 are in the search space of our algorithm, given R_1 as input.

Objective Function We now define an objective function, based on the well known F-measure, to compare the extraction quality of different candidate regexes in our search space. Using $\mathrm{M_p}(R, \mathcal{D})$ (resp. $\mathrm{M_n}(R, \mathcal{D})$) to denote the set of positive (resp. negative) matches of a regex R, we define

$$\mathrm{precision}(R, \mathcal{D}) = \frac{\mathrm{M_p}(R, \mathcal{D})}{\mathrm{M_p}(R, \mathcal{D}) + \mathrm{M_n}(R, \mathcal{D})}$$

$$\mathrm{recall}(R, \mathcal{D}) = \frac{\mathrm{M_p}(R, \mathcal{D})}{\mathrm{M_p}(R_0, \mathcal{D})}$$

$$\mathcal{F}(R, \mathcal{D}) = \frac{2 \cdot \mathrm{precision}(R, \mathcal{D}) \cdot \mathrm{recall}(R, \mathcal{D})}{\mathrm{precision}(R, \mathcal{D}) + \mathrm{recall}(R, \mathcal{D})}$$

The regex learning task addressed in this paper can now be formally stated as the following optimization problem:

Definition 3 (**Regex Learning Problem**)**.** *Given an input regex R_0, a document collection $\mathcal{D}$, labeled sets of positive and negative examples $\mathrm{M_p}$(R_0,$\mathcal{D}$) and $\mathrm{M_n}$(R_0,$\mathcal{D}$), and a set of transformations $\mathcal{T}$, compute the output regex $R_f = \mathrm{argmax}_{R \in \mathcal{T}(R_0)} \mathcal{F}(R, \mathcal{D})$.*

3 Instantiating Regex Transformations

In this section, we describe how transformations can be implemented by exploiting the syntactic constructs of modern regex engines. To help with our description, we introduce the following task:

Example 2 (Software name extraction)**.** *Consider the task of identifying names of software products in text. A simple pattern for this task is: "one or more capitalized words followed by a version number", represented as R_5 =* `([A-Z]\w*\s+)+[Vv]?(\d+\.?)+`.

When applied to a collection of University web pages, we discovered that R_5 identified correct instances such as *Netscape 2.0*, *Windows 2000* and *Installation Designer v1.1*. However, R_5 also extracted incorrect instances such as course numbers (e.g. *ENGLISH 317*), room numbers (e.g. *Room 330*), and section headings (e.g. *Chapter 2.2*). To eliminate spurious matches such as *ENGLISH 317*, let us enforce the condition that "each word is a single upper-case letter followed by one or more lower-case letters". To accomplish this, we focus on the sub-expression of R_5 that identifies capitalized words, R_{5_1} = `([A-Z]\w*\s+)+`, and replace it with $R_{5_{1a}}$ = `([A-Z][a-z]*\s+)+`. The regex resulting from R_5 by replacing R_{5_1} with $R_{5_{1a}}$ will avoid matches such as *ENGLISH 317*.

An alternate way to improve R_5 is by explicitly disallowing matches against strings like *ENGLISH*, *Room* and *Chapter*. To accomplish this, we can exploit the negative lookahead operator supported in modern regex engines. 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. As an example, `(?!`R_a`)`R_b ("`?!`" being the negative lookahead operator) returns matches of regex R_b but only if they do not match R_a. Thus, by replacing R_{5_1} in our original regex with $R_{5_{1b}}$ =`(?! ENGLISH|Room|Chapter)[A-Z]\w*\s+`, we produce an improved regex for software names.

The above examples illustrate the general principle of our transformation technique. In essence, we isolate a sub-expression of a given regex R and modify it such that the resulting regex accepts a sub-language of R. We consider two kinds of modifications – drop-disjunct and include-intersect. In drop-disjunct, we operate on a sub-expression that corresponds to a disjunct and drop one or more operands of that disjunct. In include-intersect, we restrict the chosen sub-expression by intersecting it with some other regex. Formally,

Definition 4 (Drop-disjunct Transformation). *Let $R \in \mathcal{R}_\Sigma$ be a regex of the form $R = R_a\rho(X)R_b$, where $\rho(X)$ denotes the disjunction $R_1|R_2|\ldots|R_n$ of any non-empty set of regexes $X = \{R_1, R_2, \ldots, R_n\}$. The drop-disjunct transformation $\mathrm{DD}(R, X, Y)$ for some $Y \subset X$, $Y \neq \emptyset$ results in the new regex $R_a\rho(Y)R_b$.*

Definition 5 (Include-Intersect Transformation). *Let $R \in \mathcal{R}_\Sigma$ be a regex of the form $R = R_aXR_b$ for some $X \in \mathcal{R}_\Sigma$, $X \neq \emptyset$. The include-intersect transformation $\mathrm{II}(R, X, Y)$ for some $Y \in \mathcal{R}_\Sigma$, $Y \neq \emptyset$ results in the new regex $R_a(X \cap Y)R_b$.*

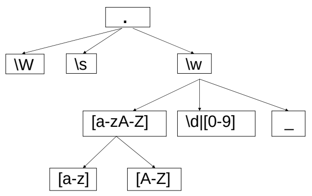

Figure 1: Sample Character Classes in Regex

We state the following proposition (proof omitted in the interest of space) that guarantees that both drop-disjunct and include-intersect restrict the language of the resulting regex, and therefore are valid transformations according to Definition 1.

Proposition 1. *Given regexes R, X_1, Y_1, X_2 and Y_2 from $\mathcal{R}_\Sigma$ such that $\mathrm{DD}(R, X_1, Y_1)$ and $\mathrm{II}(R, X_2, Y_2)$ are applicable, $\mathrm{L}(\mathrm{DD}(R, X_1, Y_1)) \subseteq \mathrm{L}(R)$ and $\mathrm{L}(\mathrm{II}(R, X_2, Y_2)) \subseteq \mathrm{L}(R)$.*

We now proceed to describe how we use different syntactic constructs to apply drop-disjunct and include-intersect transformations.

Character Class Restrictions Character classes are short-hand notations for denoting the disjunction of a set of characters (`\d` is equivalent to `(0|1...|9)`; `\w` is equivalent to `(a|...|z|A|...|Z|0|1...|9|_)`; etc.).[2] Figure 1 illustrates a character class hierarchy in which each node is a stricter class than its parent (e.g., `\d` is stricter than `\w`). A replacement of any of these character classes by one of its descendants is an instance of the *drop-disjunct* transformation. Notice that in Example 2, when replacing R_{5_1} with $R_{5_{1a}}$, we were in effect applying a character class restriction.

Quantifier Restrictions Quantifiers are used to define the range of valid counts of a repetitive sequence. For instance, `a{m,n}` looks for a sequence of `a`'s of length at least `m` and at most `n`. Since quantifiers are also disjuncts (e.g., `a{1,3}` is equivalent to `a|aa|aaa`), the replacement of an expression $R\{m, n\}$ with an expression $R\{m_1, n_1\}$ ($m \leq m_1 \leq n_1 \leq n$) is an instance of the *drop-disjunct* transformation. For example, given a subexpression of the form `a{1,3}`, we can replace it with

[2]Note that there are two distinct character classes `\W` and `\w`

one of `a{1,1}`, `a{1,2}`, `a{2,2}`, `a{2,3}`, or `a{3,3}`. Note that, before applying this transformation, wildcard expressions such as `a+` and `a*` are replaced by `a{0,maxCount}` and `a{1,maxCount}` respectively, where `maxCount` is a user configured maximum length for the entity being extracted.

Negative Dictionaries Observe that the include-intersect transformation (Definition 5) is applicable for every possible sub-expression of a given regex R. Note that a valid *sub-expression* in R is any portion of R where a capturing group can be introduced.[3] Consider a regex $R = R_a X R_b$ with a sub-expression X; the application of *include-intersect* requires another regex Y to yield $R_a(X \cap Y)R_b$. We would like to construct Y such that $R_a(X \cap Y)R_b$ is "better" than R for the task at hand. Therefore, we construct Y as $\neg Y'$ where Y' is a regex constructed from negative matches of R. Specifically, we look at each negative match of R and identify the substring of the match that corresponds to X. We then apply a greedy heuristic (see below) to these substrings to yield a *negative dictionary* Y'. Finally, the transformed regex $R_a(X \cap \neg Y')R_b$ is implemented using the negative lookahead expression R_a`(?! Y')`XR_b.

Greedy Heuristic for Negative Dictionaries Implementation of the above procedure requires certain judicious choices in the construction of the negative dictionary to ensure tractability of this transformation. Let $S(X)$ denote the distinct strings that correspond to the sub-expression X in the negative matches of R.[4] Since any subset of $S(X)$ is a candidate negative dictionary, we are left with an exponential number of possible transformations. In our implementation, we used a greedy heuristic to pick a single negative dictionary consisting of all those elements of $S(X)$ that *individually* improve the F-measure. For instance, in Example 2, if the independent substitution of R_{5_1} with `(?!ENGLISH)[A-Z]\w*\s*`, `(?!Room)[A-Z]\w*\s*`, and `(?!Chapter)[A-Z]\w*\s*` each improves the F-measure, we produce a negative dictionary consisting of `ENGLISH`, `Room`, and `Chapter`. This is precisely how the disjunct `ENGLISH|Room|Chapter` is constructed in $R_{5_{1b}}$.

[3]For instance, the sub-expressions of `ab{1,2}c` are `a`, `ab{1,2}`, `ab{1,2}c`, `b`, `b{1,2}`, `b{1,2}c`, and `c`.

[4]S(X) can be obtained automatically by identifying the substring corresponding to the group X in each entry in $\mathrm{M_n}(R,\mathcal{D})$

```
Procedure ReLIE($\mathcal{M}_{tr}$,$\mathcal{M}_{val}$,$R_0$,$\mathcal{T}$)
// $\mathcal{M}_{tr}$: set of labeled matches used as training data
// $\mathcal{M}_{val}$: set of labeled matches used as validation data
// $R_0$: user-provided regular expression
// $\mathcal{T}$: set of transformations
begin
 1. $R_{new} = R_0$
 2. do {
 3.     for each transformation $t_i \in \mathcal{T}$
 4.         Candidate$_i$=ApplyTransformations($R_{new}, t_i$)
 5.     let Candidates = $\bigcup_i$ Candidate$_i$
 6.     let $R' = \mathrm{argmax}_{R \in \text{Candidates}} \mathcal{F}(R, \mathcal{M}_{tr})$
 7.     if ($\mathcal{F}(R', \mathcal{M}_{tr}) <= \mathcal{F}(R_{new}, \mathcal{M}_{tr})$) return $R_{new}$
 8.     if ($\mathcal{F}(R', \mathcal{M}_{val}) < \mathcal{F}(R_{new}, \mathcal{M}_{val})$) return $R_{new}$
 9.     $R_{new} = R'$
10. } while(true)
end
```

Figure 2: ReLIE Search Algorithm

4 ReLIE Search Algorithm

Figure 2 describes the ReLIE algorithm for the Regex Learning Problem (Definition 3) based on the transformations described in Section 3. ReLIE is a greedy hill climbing search procedure that chooses, at every iteration, the regex with the highest F-measure. An iteration in ReLIE consists of:

- Applying every transformation on the current regex R_{new} to obtain a set of candidate regexes
- From the candidates, choosing the regex R' whose F-measure over the training dataset is maximum

To avoid overfitting, ReLIE terminates when either of the following conditions is true: (i) there is no improvement in F-measure over the training set; (ii) there is a drop in F-measure when applying R' on the validation set.

The following proposition provides an upper bound for the running time of the ReLIE algorithm.

Proposition 2. *Given any valid set of inputs $\mathcal{M}_{tr}$, $\mathcal{M}_{val}$, R_0, and $\mathcal{T}$, the ReLIE algorithm terminates in at most $|\mathrm{M_n}(R_0, \mathcal{M}_{tr})|$ iterations. The running time of the algorithm $T_{Total}(R_0, \mathcal{M}_{tr}, \mathcal{M}_{val}) \leq |\mathrm{M_n}(R_0, \mathcal{M}_{tr})| * t_0$, where t_0 is the time taken for the first iteration of the algorithm.*

Proof. With reference to Figure 2, in each iteration, the F-measure of the "best" regex R' is strictly better than R_{new}. Since $L(R') \subseteq L(R_{new})$, R' eliminates at least one additional negative match compared to R_{new}. Hence, the maximum number of iterations is $|\mathrm{M_n}(R_0, \mathcal{M}_{tr})|$.

For a regular expression R, let $n_{cc}(R)$ and $n_q(R)$ denote, respectively, the number of character classes and quantifiers in R. The maximum number of possible sub-expressions in R is $|R|^2$, where $|R|$ is the length of R. Let $MaxQ(R)$ denote the maximum number of ways in which a single quantifier appearing in R can be restricted to a smaller range. Let F_{cc} denote the maximum fanout[5] of the character class hierarchy. Let $T_{ReEval}(\mathcal{D})$ denote the average time taken to evaluate a regex over dataset $\mathcal{D}$.

Let R_i denote the regex at the beginning of iteration i. The number of candidate regexes obtained by applying the three transformations is

$$\mathit{NumRE}(R_i, \mathcal{M}_{tr}) \leq n_{cc}(R_i) * F_{cc} + n_q(R_i) * \mathit{MaxQ}(R_i) + |R_i|^2$$

The time taken to enumerate the character class and quantifier restriction transformations is proportional to the resulting number of candidate regexes. The time taken for the negative dictionaries transformation is given by the running time of the greedy heuristic (Section 3). The total time taken to enumerate all candidate regexes is given by (for some constant c)

$$T_{Enum}(R_i, \mathcal{M}_{tr}) \leq c * (n_{cc}(R_i) * F_{cc} + n_q(R_i) * \mathit{MaxQ}(R_i) + |R_i|^2 * \mathrm{M_n}(R_i, \mathcal{M}_{tr}) * T_{ReEval}(\mathcal{M}_{tr}))$$

Choosing the best transformation involves evaluating each candidate regex over the training and validation corpus and the time taken for this step is

$$T_{PickBest}(R_i, \mathcal{M}_{tr}, \mathcal{M}_{val}) = \mathit{NumRE}(R_i, \mathcal{M}_{tr}) * (T_{ReEval}(\mathcal{M}_{tr}) + T_{ReEval}(\mathcal{M}_{val}))$$

The total time taken for an iteration can be written as

$$T_I(R_i, \mathcal{M}_{tr}, \mathcal{M}_{val}) = T_{Enum}(R_i, \mathcal{M}_{tr}) + T_{PickBest}(R_i, \mathcal{M}_{tr}, \mathcal{M}_{val})$$

It can be shown that the time taken in each iteration decreases monotonically (details omitted in the interest of space). Therefore, the total running time of the algorithm is given by

$$T_{Total}(R_0, \mathcal{M}_{tr}, \mathcal{M}_{val}) = \sum T_I(R_i, \mathcal{M}_{tr}, \mathcal{M}_{val}) \leq |\mathrm{M_n}(R_0, \mathcal{M}_{tr})| * t_0.$$

where $t_0 = T_I(R_0, \mathcal{M}_{tr}, \mathcal{M}_{val})$ is the running time of the first iteration of the algorithm. □

5 Experiments

In this section, we present an empirical study of the ReLIE algorithm using four extraction tasks over three real-life data sets. The goal of this study is to evaluate the effectiveness of ReLIE in learning complex regexes and to investigate how it compares with standard machine learning algorithms.

5.1 Experimental Setup

Data Set The datasets used in our experiments are:

- **`EWeb`:** A collection of 50,000 web pages crawled from a corporate intranet.
- **`AWeb`:** A set of 50,000 web pages obtained from the publicly available University of Michigan Web page collection (Li et al., 2006), including a subcollection of 10,000 pages (**`AWeb-S`**).
- **`Email`:** A collection of 10,000 emails obtained from the publicly available Enron email collection (Minkov et al., 2005).

Extraction Tasks *SoftwareNameTask*, *CourseNumberTask* and *PhoneNumberTask* were evaluated on `EWeb`, `AWeb` and `Email`, respectively. Since web pages have large number of URLs, to keep the labeling task manageable, *URLTask* was evaluated on `AWeb-S`.

Gold Standard For each task, the gold standard was created by manually labeling all matches for the initial regex. Note that only exact matches with the gold standard are considered correct in our evaluations. [6]

Comparison Study To evaluate ReLIE for entity extraction vis-a-vis existing algorithms, we used the popular conditional random field (CRF). Specifically, we used the MinorThird (Cohen, 2004) implementation of CRF to train models for all four extraction tasks. For training the CRF we provided it with the set of positive and negative matches from the initial regex with a context of 200 characters on either side of each match[7]. Since it is unlikely that useful features are located far away from the entity, we believe that 200 characters on either side is sufficient context. The CRF used the base features described in (Cohen et al., 2005). To ensure fair comparison with ReLIE, we also included the matches corresponding to the input regex as a feature to the CRF. In practice, more complex features (e.g., dictionaries, simple regexes) derived by domain experts are often provided to CRFs. However, such features can also be used to refine the initial regex given to ReLIE. Hence, with a view to investigating the "raw" learning capability of the two approaches, we chose to run all our experiments without any additional manually derived features. In fact, the patterns learned by ReLIE through transformations are often similar

[5]Fanout is the number of ways in which a character class may be restricted as defined by the hierarchy (e.g. Figure 1).

[6]The labeled data will be made publicly available at `http://www.eecs.umich.edu/db/regexLearning/`.

[7]Ideally, we would have preferred to let MinorThird extract appropriate features from complete documents in the training-set but could not get it to load our large datasets.

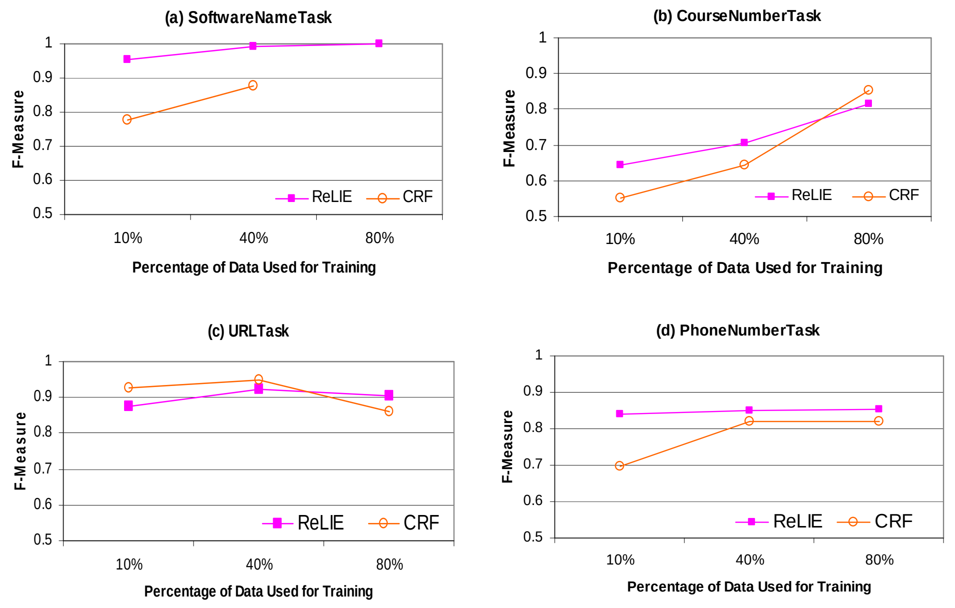

Figure 3: Extraction Quality[a]

[a]For *SoftwareNameTask*, with 80% training data we could not obtain results for CRF as the program failed repeatedly during the training phase.

to the features that domain experts may provide to CRF. We will revisit this issue in Section 5.4.

Evaluation We used the standard F-measure to evaluate the effectiveness of ReLIE and CRF. We divided each dataset into 10 equal parts and used X% of the dataset for training (X=10, 40 and 80), 10% for validation, and remaining (90-X)% for testing. All results are reported on the test set.

5.2 Results

Four extraction tasks were chosen to reflect the entities commonly present in the three datasets.

- *SoftwareNameTask*: Extracting software names such as Lotus Notes 8.0, Open Office Suite 2007.
- *CourseNumberTask*: Extracting university course numbers such as EECS 584, Pharm 101.
- *PhoneNumberTask*: Extracting phone numbers such as 1-800-COMCAST, (425)123 5678.
- *URLTask*: Extracting URLs such as http:\\www.abc.com and lsa.umich.edu/ foo/.[8]

This section summarizes the results of our empirical evaluation comparing ReLIE and CRF.

[8]*URLTask* may appear to be simplistic. However, extracting URLs without the leading protocol definitions (e.g. `http`) can be challenging.

Raw Extraction Quality The cross-validated results across all four tasks are presented in Figure 3.

- With 10% training data, ReLIE outperforms CRF on three out of four tasks with a difference in F-measure ranging from 0.1 to 0.2.
- As training data increases, both algorithms perform better with the gap between the two reducing for all the four tasks. For *CourseNumberTask* and *URLTask*, CRF does slightly better than ReLIE for larger training dataset. For the other two tasks, ReLIE retains its advantage over CRF.[9]

The above results indicate that ReLIE performs comparably with CRF with a slight edge in conditions of limited training data. Indeed, the capability to learn high-quality extractors using a small training set is important because labeled data is often expensive to obtain. For precisely this same reason, we would ideally like to learn the extractors once and then apply them to other datasets as needed. Since these other datasets may be from a different domain, we next performed a cross-domain test (i.e., training

[9]For *SoftwareNameTask*, with 80% training data we could not obtain results for CRF as the program failed repeatedly during the training phase.

Task(Training, Testing) \ Data for Training	10%		40%		80%	
	ReLIE	CRF	ReLIE	CRF	ReLIE	CRF
SoftwareNameTask(`EWeb`,`AWeb`)	**0.920**	0.297	**0.977**	0.503	**0.971**	N/A
URLTask(`AWeb-S`,`Email`)	**0.690**	0.209	**0.784**	0.380	**0.801**	0.507
PhoneNumberTask(`Email`,`AWeb`)	**0.357**	0.130	**0.475**	0.125	**0.513**	0.120

Table 1: Cross Domain Test (F-measure).

Technique	*SoftwareNameTask*		*CourseNumberTask*		*URLTask*		*PhoneNumberTask*	
	training	testing	training	testing	training	testing	training	testing
ReLIE	511.7	20.6	69.3	18.4	73.8	7.7	39.4	1.1
CRF	7597.0	2315.8	482.5	75.4	438.7	53.8	434.8	57.7
$\frac{t(\text{ReLIE})}{t(\text{CRF})}$	0.067	0.009	0.144	0.244	0.168	0.143	0.091	0.019

Table 2: Average Training/Testing Time (sec)(with 40% data for training)

Task(Extra Feature) \ Data for Training	10%		40%		80%	
	CRF	C+RL	CRF	C+RL	CRF	C+RL
CourseNumberTask(Negative Dictionary)	0.553	**0.624**	0.644	**0.764**	**0.854**	0.845
PhoneNumberTask(Quantifier)	0.695	**0.893**	0.820	**0.937**	0.821	**0.964**

Table 3: ReLIE as Feature Extractor (C+RL is CRF enhanced with features learned by ReLIE).

and testing on different domains).

Cross-domain Evaluation Table 1 summarizes the results of training the algorithms on one data set and testing on another. The scenarios chosen are: (i) *SoftwareNameTask* trained on `EWeb` and tested on `AWeb`, (ii) *URLTask* trained on `AWeb` and tested on `Email`, and (iii) *PhoneNumberTask* trained on `Email` and tested on `AWeb`.[10] We can see that ReLIE significantly outperforms CRF for all three tasks, even when provided with a large training dataset. Compared to testing on the same dataset, there is a reduction in F-measure (less than 0.1 in many cases) when the regex learned by ReLIE is applied to a different dataset, while the drop for CRF is much more significant (over 0.5 in many cases).[11]

Training Time Another issue of practical consideration is the efficiency of the learning algorithm. Table 2 reports the average training and testing time for both algorithms on the four tasks. On average ReLIE is *an order of magnitude faster* than CRF in both building the model and applying the learnt model.

Robustness to Variations in Input Regexes The transformations done by ReLIE are based on the structure of the input regex. Therefore given different input regexes, the final regexes learned by ReLIE will be different. To evaluate the impact of the structure of the input regex on the quality of the regex learned by ReLIE, we started with different regexes[12]

for the same task. We found that ReLIE is robust to variations in input regexes. For instance, on *SoftwareNameTask*, the standard deviation in F-measure of the final regexes generated from six different input regexes was less than 0.05. Further details of this experiment are omitted in the interest of space.

5.3 Discussion

The results of our comparison study (Figure 3) indicates that for raw extraction quality ReLIE has a slight edge over CRF for small training data. However, in cross-domain performance (Table 1) ReLIE is significantly better than CRF (by 0.41 on average) . To understand this discrepancy, we examined the final regex learned by ReLIE and compared that with the features learned by CRF. Examples of initial regexes with corresponding final regexes learnt by ReLIE with 10% training data are listed in Table 4. Recall, from Section 3, that ReLIE transformations include character class restrictions, quantifier restrictions and addition of negative dictionaries. For instance, in the *SoftwareNameTask*, the final regex listed was obtained by restricting `[a-zA-Z]` to `[a-z]`, `\w` to `[a-zA-Z]`, and adding the negative dictionary `(Copyright|Fall|···|Issue)`. Similarly, for the *PhoneNumberTask*, the final regex involved two negative dictionaries (expressed as `(?![,])` and `(?![,:])`) [13] and quantifier restrictions (e.g. the first `[A-Z\d]{2,4}` was transformed

[10]We do not report results for *CourseNumberTask* as course numbers are specific to academic webpages and do not appear in the other two domains

[11]Similar cross-domain performance deterioration for a machine learning approach has been observed by (Guo et al., 2006).

[12]Recall that the search space of ReLIE is limited by $L(R_0)$ (Assumption 1). Thus to ensure meaningful comparison, for the same task any two given input regexes R_0 and R_0' are chosen in such a way that although their structures are different, $\mathrm{M_p}(R_0, \mathcal{D}) = \mathrm{M_p}(R_0', \mathcal{D})$ and $\mathrm{M_n}(R_0, \mathcal{D}) = \mathrm{M_n}(R_0', \mathcal{D})$.

[13]To obtain these negative dictionaries, ReLIE not only needs to correctly identify the dictionary entries from negative matches but also has to place the corresponding negative lookahead expression at the appropriate place in the regex.

SoftwareNameTask	R_0	`\b([A-Z][a-zA-Z]{1,10}\s){1,5}\s*(\w{0,2}\d[\.]?){1,4}\b`
	R_{final}	`\b((?!(Copyright|Page|Physics|Question|···|Article|Issue))[A-Z][a-z]{1,10}\s){1,5}\s*([a-zA-Z]{0,2}\d[\.]?){1,4}\b`
$PhoneNumberTask$	R_0	`\b(1\W+)?\W?\d{3,3}\W*\s*\W?[A-Z\d]{2,4}\s*\W?[A-Z\d]{2,4}\b`
	R_{final}	`\b(1\W+)?\W?\d{3,3}((?![,])\W*)\s*\W?[A-Z\d]{3,3}\s*((?![,:])\W?)[A-Z\d]{3,4}\b`
CourseNumberTask	R_0	`\b([A-Z][a-zA-Z]+)\s+\d{3,3}\b`
	R_{final}	`\b(((?!(At|Between|···Contact|Some|Suite|Volume))[A-Z][a-zA-Z]+))\s+\d{3,3}\b`
URLTask	R_0	`\b(\w+://)?(\w+\.){0,2}\w+\.\w+(/[^\s]+){0,20}\b`
	R_{final}	`\b((?!(Response_20010702_1607.csv|···))((\w+://)?(\w+\.){0,2}\w+\.(?!(ppt|···doc))[a-zA-Z]{2,3}))(/[^\s]+){0,20}\b`

Table 4: Sample Regular Expressions Learned by ReLIE(R_0: input regex; R_{final}: final regex learned; the parts of R_0 modified by ReLIE and the corresponding parts in R_{final} are highlighted.)

into `[A-Z\d]{3,3}`).

After examining the features learnt by CRF, it was clear that while CRF could learn features such as the negative dictionary it is unable to learn character-level features. This should not be surprising since our CRF was trained with primarily tokens as features (cf. Section 5.1). While this limitation was less of a factor in experiments involving data from the same domain (some effects were seen with smaller training data), it does explain the significant difference between the two algorithms in cross-domain tasks where the vocabulary can be significantly different. Indeed, in practical usage of CRF, the main challenge is to come up with additional complex features (often in the form of dictionary and regex patterns) that need to be given to the CRF (Minkov et al., 2005). Such complex features are largely hand-crafted and thus expensive to obtain. Since the ReLIE transformations are operations over characters, a natural question to ask is: "Can the regex learned by ReLIE be used to provide features to CRF?" We answer this question below.

5.4 ReLIE as Feature Extractor for CRF

To understand the effect of incorporating ReLIE-identified features into CRF, we chose the two tasks (*CourseNumberTask* and *PhoneNumberTask*) with the least F-measure in our experiments to determine raw extraction quality. We examined the final regex produced by ReLIE and manually extracted portions to serve as features. For example, the negative dictionary learned by ReLIE for the *CourseNumberTask* `(At|Between|···|Volume)` was incorporated as a feature into CRF. To help isolate the effects, for each task, we only incorporated features corresponding to a single transformation: negative dictionaries for *CourseNumberTask* and quantifier restrictions for *PhoneNumberTask*. The results of these experiments are shown in Table 3. The first point worthy of note is that performance has improved in all but one case. Second, despite the F-measure on *CourseNumberTask* being lower than *PhoneNumberTask* (presumably more potential for improvement), the improvements on *PhoneNumberTask* are significantly higher. This observation is consistent with our conjecture in Section 5.1 that CRF learns token-level features; therefore incorporating negative dictionaries as extra feature provides only limited improvement. Admittedly more experiments are needed to understand the full impact of incorporating ReLIE-identified features into CRF. However, we do believe that this is an exciting direction of future research.

6 Summary and Future Work

We proposed a novel formulation of the problem of learning complex character-level regexes for entity extraction tasks. We introduced the concept of regex transformations and described how these could be realized using the syntactic constructs of modern regex languages. We presented ReLIE, a powerful regex learning algorithm that exploits these ideas. Our experiments demonstrate that ReLIE is very effective for certain classes of entity extraction, particularly under conditions of cross-domain and limited training data. Our preliminary results also indicate the possibility of using ReLIE as a powerful feature extractor for CRF and other machine learning algorithms. Further investigation of this aspect of ReLIE presents an interesting avenue of future work.

Acknowledgments

We thank the anonymous reviewers for their insightful and constructive comments and suggestions. We are also grateful for comments from David Gondek and Sebastian Blohm.

References

R. Alquezar and A. Sanfeliu. 1994. Incremental grammatical inference from positive and negative data using unbiased finite state automata. In *SSPR*.

Douglas E. Appelt and Boyan Onyshkevych. 1998. The common pattern specification language. In *TIPSTER TEXT PROGRAM*.

Geert Jan Bex et al. 2006. Inference of concise DTDs from XML data. In *VLDB*.

Eric Brill. 2000. Pattern-based disambiguation for natural language processing. In *SIGDAT*.

William W. Cohen and Andrew McCallum. 2003. Information Extraction from the World Wide Web. in *KDD*

William W. Cohen. 2004. Minorthird: Methods for identifying names and ontological relations in text using heuristics for inducing regularities from data. http://minorthird.sourceforge.net.

William W. Cohen et al. 2005. Learning to Understand Web Site Update Requests. In *IJCAI*.

Fabio Ciravegna. 2001. Adaptive information extraction from text by rule induction and generalization. In *IJCAI*.

H. Cunningham. 1999. JAPE – a java annotation patterns engine.

Francois Denis et al. 2004. Learning regular languages using RFSAs. *Theor. Comput. Sci.*, 313(2):267–294.

Francois Denis. 2001. Learning regular languages from simple positive examples. *Machine Learning*, 44(1/2):37–66.

Pedro DeRose et al. 2007. DBLife: A Community Information Management Platform for the Database Research Community In *CIDR*

Pierre Dupont. 1996. Incremental regular inference. In *ICGI*.

Ronen Feldman et all. 2006. Self-supervised Relation Extraction from the Web. In *ISMIS*.

Henning Fernau. 2005. Algorithms for learning regular expressions. In *ALT*.

Laura Firoiu et al. 1998. Learning regular languages from positive evidence. In *CogSci*.

K. Fukuda et al. 1998. Toward information extraction: identifying protein names from biological papers. *Pac Symp Biocomput.*, 1998:707–718

Ugo Galassi and Attilio Giordana. 2005. Learning regular expressions from noisy sequences. In *SARA*.

Minos Garofalakis et al. 2000. XTRACT: a system for extracting document type descriptors from XML documents. In *SIGMOD*.

Hong Lei Guo et al. 2006. Empirical Study on the Performance Stability of Named Entity Recognition Model across Domains In *EMNLP*.

Java Regular Expressions. 2008. http://java.sun.com /javase/6/docs/api/java/util/regex/package-summary.html.

Dan Klein et al. 2003. Named Entity Recognition with Character-Level Models. In *HLT-NAACL*.

Vijay Krishnan and Christopher D. Manning. 2006. An Effective Two-Stage Model for Exploiting Non-Local Dependencies in Named Entity Recognition. In *ACL*.

Yunyao Li et al. 2006. Getting work done on the web: Supporting transactional queries. In *SIGIR*.

Andrew McCallum et al. 2000. Maximum Entropy Markov Models for Information Extraction and Segmentation. In *ICML*.

Einat Minkov et al. 2005. Extracting personal names from emails: Applying named entity recognition to informal text. In *HLT/EMNLP*.

Stephen Soderland. 1999. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34:233–272.

Lorraine Tanabe and W. John Wilbur 2002. Tagging gene and protein names in biomedical text. *Bioinformatics*, 18:1124–1132.

Tianhao Wu and William M. Pottenger. 2005. A semi-supervised active learning algorithm for information extraction from textual data. *JASIST*, 56(3):258–271.

Huaiyu Zhu, Alexander Loeser, Sriram Raghavan, Shivakumar Vaithyanathan 2007. Navigating the intranet with high precision. In *WWW*.