Journal of Machine Learning Research 15 (2015) . - .. Submitted 04/14; Revised 11/14; Published ../..

Learning to Identify Concise Regular Expressions that Describe Email Campaigns

Paul Prasse PRASSE@CS.UNI-POTSDAM.DE
University of Potsdam, Department of Computer Science
August-Bebel-Strasse 89, 14482 Potsdam, Germany

Christoph Sawade CHRISTOPH@SOUNDCLOUD.COM
SoundCloud Ltd.
Rheinsberger Str. 76/77, 10115 Berlin, Germany

Niels Landwehr LANDWEHR@CS.UNI-POTSDAM.DE
Tobias Scheffer SCHEFFER@CS.UNI-POTSDAM.DE
University of Potsdam, Department of Computer Science
August-Bebel-Strasse 89, 14482 Potsdam, Germany

Editor: Ivan Titov

Keywords: Applications of machine learning, learning with structured output spaces, supervised learning, regular expressions, email campaigns

Abstract

This paper addresses the problem of inferring a regular expression from a given set of strings that resembles, as closely as possible, the regular expression that a human expert would have written to identify the language. This is motivated by our goal of automating the task of postmasters who use regular expressions to describe and blacklist email spam campaigns. Training data contains batches of messages and corresponding regular expressions that an expert postmaster feels confident to blacklist. We model this task as a two-stage learning problem with structured output spaces and appropriate loss functions. We derive decoders and the resulting optimization problems which can be solved using standard cutting plane methods. We report on a case study conducted with an email service provider.

1. Introduction

The problem setting introduced in this paper is motivated by the intuition of *automatically reverse engineering* email spam campaigns. Email-spam generation tools allow users to implement mailing campaigns by specifying simple grammars that serve as message templates. A grammar is disseminated to nodes of a bot net; the nodes create messages by instantiating the grammar at random. Email service providers can easily sample elements of new mailing campaigns by collecting messages in spam traps or by tapping into known bot nets. When messages from multiple campaigns are collected in a joint spam trap, clustering tools can separate the campaigns reliably (Haider and Scheffer, 2009). However, probabilistic cluster descriptions that use a bag-of-words representation incur the risk of false positives, and it is difficult for a human to decide whether they in fact characterize the correct set of messages.

©2015 Paul Prasse, Christoph Sawade, Niels Landwehr and Tobias Scheffer.

Typically, mailing campaigns are quite specific. A specific, comprehensible regular expression written by an expert postmaster can be used to blacklist the bulk of emails of that campaign at virtually no risk of covering any other messages. This, however, requires the continuous involvement of a human postmaster.

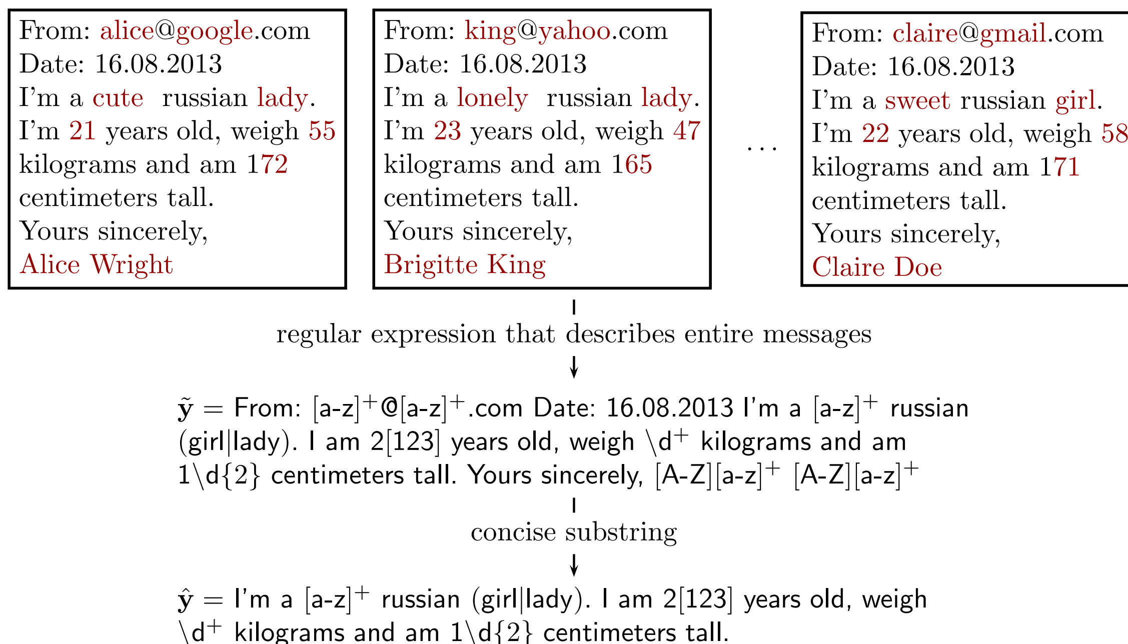


Figure 1: Elements of a message spam campaign, a regular expression that describes the entirety of the messages, and a concise regular expression that describes a characteristic substring of the messages.

Regular expressions are a standard tool for specifying simple grammars. Widely available tools match strings against regular expressions efficiently and can be used conveniently from scripting languages. A regular expression can be translated into a deterministic finite automaton that accepts the language and has an execution time linear in the length of the input string.

Language identification has a rich history in the algorithmic learning theory community, see Section 6 for a brief review. Our problem setting reflects the process that we seek to automate; it differs from the classical problem of language identification in the learner's exact goal, and in the available training data. Batches of strings and corresponding regular expressions are observable in the training data. These regular expressions have been written by postmasters to blacklist mailing campaigns. The learner's goal is to produce a predictive model that maps batches of strings to regular expressions that resemble, as closely as possible, the regular expressions which the postmaster would have written and feels confident to blacklist. As an illustration of this problem, Figure 1 shows three messages of a mailing campaign, a regular expression that describes the entirety of the messages, and

a more concise regular expression that characterizes a characteristic substring, and that a postmaster has selected to blacklist the corresponding email campaign.

This paper extends a conference publication (Prasse et al., 2012) that addresses this problem setting with linear models and structured output spaces. In the decoding step, a set of strings is given and the space of all regular expressions has to be searched for an element that maximizes the decision function. Since this space is very large and difficult to search, the approach of Prasse et al. (2012) is constrained to finding specializations of an approximate maximal alignment of all strings. The maximal alignment is a regular expression that contains all character sequences which occur in each of the strings, and uses wildcards wherever there are differences between the strings.

The maximal alignment is extremely specific. By constraining the output to specializations of the alignment, the method keeps the risk that any message which is not part of the same campaign is accidentally matched at a minimum. However, since all specializations of this alignment describe the entire length of the strings, the method produces regular expressions that tend to be much longer than the more concise expressions that postmasters prefer. Also, as a consequence of their greater length, the finite state automata which correspond to these expressions tend to have more states, which limits the number of regular expressions that can be matched in parallel against incoming new messages. This paper therefore extends the method by including a mechanism which learns to select expressions that describe only the most characteristic part of the mailing campaign, using regular expressions written by an expert postmaster as training data.

The rest of this paper is structured as follows. Section 2 reviews regular expressions before Section 3 states the problem setting. Section 4 introduces the feature representations and derives the decoders and the optimization problems. In Section 5, we discuss our findings from a case study with an email service. Section 6 discusses related work and Section 7 concludes.

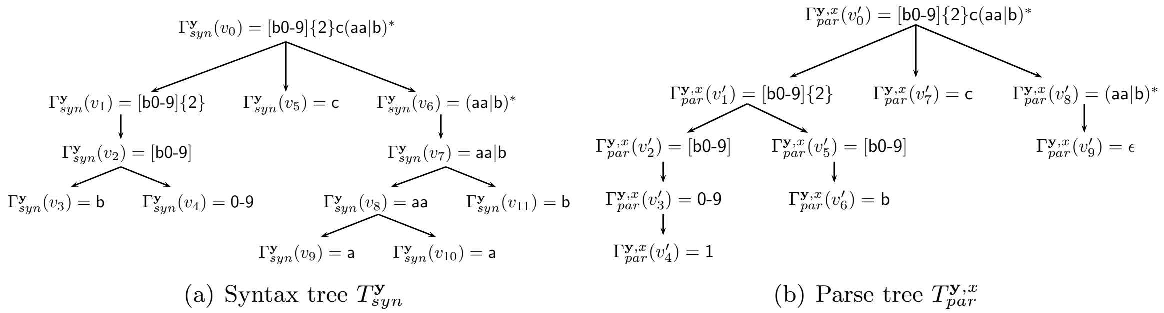
2. Regular Expressions

Before we formulate the problem setting, let us briefly revisit the syntax and semantics of regular expressions. Regular expressions are a popular syntactic convention for the definition of regular languages. Syntactically, a regular expression $\mathbf{y} \in \mathcal{Y}_\Sigma$ is either a character from an alphabet Σ, or it is an expression in which an operator is applied to one or several argument expressions. Basic operators are the concatenation (*e.g.,* "abc"), disjunction (*e.g.,* "a|b"), and the *Kleene* star ("*"), written in postfix notation ("(abc)*"), that accepts any number of repetitions of its preceding argument expression. Parentheses define the syntactic structure of the expression. For better readability, several shorthands are used, which can be defined in terms of the basic operators. For instance, the *any character* symbol (".") abbreviates the disjunction of all characters in Σ, square brackets accept the disjunction of all characters (*e.g.,* "[abc]") or ranges (*e.g.,* "[a-z0-9]") that are included. For instance, the regular expression [a-z0-9] accepts all lower-case letters and digits. The postfix operator "+" accepts an arbitrary, positive number of reiterations of the preceding expression, while "$\{l, u\}$" accepts between l and u reiterations, where $l \leq u$. We include a set of popular macros—for instance "\d" for *any digit* or the macro "\e" for all

characters, which can occur in a URL. A formal definition of the set of regular expressions can be found in Definition 3 in the appendix.

The set of all regular expressions can be described by a context-free language. The syntactic structure of a regular expression $\mathbf{y}$ is typically represented by its *syntax tree* $T^{\mathbf{y}}_{syn} = (V^{\mathbf{y}}_{syn}, E^{\mathbf{y}}_{syn}, \Gamma^{\mathbf{y}}_{syn}, \leq^{\mathbf{y}}_{syn})$. Definition 4 in the appendix assigns one such tree to each regular expression. Each node $v \in V^{\mathbf{y}}_{syn}$ of this syntax tree is tagged by a labeling function $\Gamma^{\mathbf{y}}_{syn} : V^{\mathbf{y}}_{syn} \to \mathcal{Y}_\Sigma$ with a subexpression $\Gamma^{\mathbf{y}}_{syn}(v) = \mathbf{y}_j$. The edges $(v, v') \in E^{\mathbf{y}}_{syn}$ indicate that node v' represents an argument expression of v. Relation $\leq^{\mathbf{y}}_{syn} \subseteq V^{\mathbf{y}}_{syn} \times V^{\mathbf{y}}_{syn}$ defines an ordering on the nodes and identifies the root node. Note that the root node is labeled with the entire regular expression $\mathbf{y}$.

A regular expression $\mathbf{y}$ defines a regular language $L(\mathbf{y})$. Given the regular expression, a deterministic finite state machine can decide whether a string x is in $L(\mathbf{y})$ in time linear in $|x|$ (Dubé and Feeley, 2000). The trace of verification is typically represented as a *parse tree* $T^{\mathbf{y},x}_{par} = (V^{\mathbf{y},x}_{par}, E^{\mathbf{y},x}_{par}, \Gamma^{\mathbf{y},x}_{par}, \leq^{\mathbf{y},x}_{par})$, describing how the string x can be derived from the regular expression $\mathbf{y}$. At least one parse tree exists if and only if the string is an element of the language $L(\mathbf{y})$; in this case, $\mathbf{y}$ is said to generate x. Multiple parse trees can exist for one regular expression $\mathbf{y}$ and a string x. Nodes $v \in V^{\mathbf{y}}_{syn}$ of the syntax tree generate the nodes of the parse tree $v' \in V^{\mathbf{y},x}_{par}$, where nodes of the syntax tree may spawn none (alternatives which are not used to generate a string), one, or several ("loopy" syntactic elements such as "*" or "+") nodes in the parse tree. In analogy to the syntax trees, the labeling function $\Gamma^{\mathbf{y},x}_{par} : V^{\mathbf{y},x}_{par} \to \mathcal{Y}_\Sigma$ assigns a subexpression to each node, and the relation $\leq^{\mathbf{y},x}_{par} \subseteq V^{\mathbf{y},x}_{par} \times V^{\mathbf{y},x}_{par}$ defines the ordering of sibling nodes. The set of all parse trees for a regular expression $\mathbf{y}$ and a string x is denoted by $\mathcal{T}^{\mathbf{y},x}_{par}$. When multiple parse trees exist for a regular expression and a string, a canonical parse tree can be selected by choosing the *left-most parse*. Standard tools for regular expressions typically follow this convention and generate the left-most parse tree. Definition 5 in the appendix gives a formal definition.



(a) Syntax tree $T^{\mathbf{y}}_{syn}$

(b) Parse tree $T^{\mathbf{y},x}_{par}$

Figure 2: Syntax tree (a) and a parse tree (b) for the regular expression $\mathbf{y}$ = [b0-9]{2}c(aa|b)* and the string x = 1bc.

Leaf nodes of a parse tree $T^{\mathbf{y},x}_{par}$ are labeled with elements of $\Sigma \cup \{\epsilon\}$, where ϵ denotes the empty symbol; reading them from left to right gives the generated string x. Non-terminal nodes correspond to subexpressions $\mathbf{y}_j$ of $\mathbf{y}$ which generate substrings of x. To compare different regular expressions with respect to a given string x, we define the set $T^{\mathbf{y},x}_{par\,|i}$ of

labels of nodes which are visited on the path from the root to the the i-th character of x in the parse tree $T_{par}^{\mathbf{y},x}$.

Figure 2 (left) shows an example of a syntax tree $T_{syn}^{\mathbf{y}}$ for the regular expression $\mathbf{y} =$ $\mathsf{[b0\text{-}9]\{2\}c(aa|b)^*}$. One corresponding parse tree $T_{par}^{\mathbf{y},x}$ for the string $x = \mathsf{1bc}$ is illustrated in Figure 2 (right). The set $T_{par}^{\mathbf{y},x}{}_{|2}$ contains nodes v'_0, v'_1, v'_5, and v'_6.

Finally, we introduce the concept of a matching list. When a regular expression $\mathbf{y}$ generates a set $\mathbf{x}$ of strings, and $v \in V_{syn}^{\mathbf{y}}$ is an arbitrary node of the syntax tree of $\mathbf{y}$, then the matching list $M^{\mathbf{y},\mathbf{x}}(v)$ characterizes which substrings of the strings in $\mathbf{x}$ are generated by the node v of the syntax tree, and thus generated by the subexpression $\Gamma_{syn}^{\mathbf{y}}(v)$. A node v of the syntax tree generates a substring x' of $x \in \mathbf{x}$, if v generates a node v' in the parse tree $T_{par}^{\mathbf{y},x}$ of x, and there is a path from v' in that parse tree to every character in the substring x'. In the above example, for the set of strings $\mathbf{x} = \{\mathsf{12c}, \mathsf{b4ca}\}$, the matching list for node v_1 that represents subexpression $\Gamma_{syn}^{\mathbf{y}}(v_1) = \mathsf{[b0\text{-}9]\{2\}}$ is $M^{\mathbf{y},\mathbf{x}}(v_2) = \{\mathsf{12}, \mathsf{b4}\}$. Definition 5 in the appendix introduces matching lists more formally.

3. Problem Setting

Having established the syntax and semantics of regular expressions, we now define our problem setting. An unknown distribution $p(\mathbf{x},\mathbf{y})$ generates regular expressions $\mathbf{y} \in \mathcal{Y}_\Sigma$ from the alphabet Σ and batches $\mathbf{x}$ of strings $x \in \mathbf{x}$ that are elements of the language $L(\mathbf{y})$. In our motivating application, the strings x are messages that belong to one particular mailing campaign and have been sampled from a bot net, and the $\mathbf{y}$ are regular expressions which an expert postmaster believes to identify the campaign template, and feels highly confident to blacklist.

A $\mathbf{w}$-parameterized predictive model $f_{\mathbf{w}} : \mathbf{x} \times \hat{\mathbf{y}} \mapsto \mathbb{R}$ maps a batch of strings and a regular expression $\hat{\mathbf{y}}$ to a value of the decision function. We refer to the process of inferring the $\hat{\mathbf{y}}$ that attains the highest score $f_{\mathbf{w}}(\mathbf{x},\hat{\mathbf{y}})$ for a given batch of strings $\mathbf{x}$ as *decoding*; in this step, a decision function is maximized over $\hat{\mathbf{y}}$ which generally involves a search over the space of all regular expressions.

A loss function $\Delta(\mathbf{y},\hat{\mathbf{y}},\mathbf{x})$ quantifies the difference between the true and predicted expressions. While it would, in principle, be possible to use the zero-one loss $\Delta_{0/1}(\mathbf{y},\hat{\mathbf{y}},\mathbf{x}) =$ $[\![\mathbf{y} = \hat{\mathbf{y}}]\!]$, this loss function would treat nearly-identical expressions and very dissimilar expressions alike. We will later engineer a loss function whose gradient will guide the learner towards expressions $\hat{\mathbf{y}}$ that are more similar to the correct expression $\mathbf{y}$.

In the learning step, the ultimate goal is to identify parameters that minimize the risk—the expected loss—under the unknown distribution $p(\mathbf{x},\mathbf{y})$:

$$R[f_{\mathbf{w}}] = \iint \Delta\left(\mathbf{y}, \arg\max_{\hat{\mathbf{y}} \in \mathcal{Y}_\Sigma} f_{\mathbf{w}}(\mathbf{x},\hat{\mathbf{y}}), \mathbf{x}\right) p(\mathbf{x},\mathbf{y}) \mathrm{d}\mathbf{x}\, \mathrm{d}\mathbf{y}.$$

The underlying distribution $p(\mathbf{x},\mathbf{y})$ is not known, and therefore this goal is unattainable. We resort to training data $D = \{(\mathbf{x}_i,\mathbf{y}_i)\}_{i=1}^m$ that consists of pairs of batches $\mathbf{x}_i$ and corresponding regular expressions $\mathbf{y}_i$, drawn according to $p(\mathbf{x},\mathbf{y})$. In order to obtain a convex optimization problem that can be evaluated using the training data, we approximate the risk by the hinged upper bound of its maximum-likelihood estimate, following

the margin-rescaling approach (Tsochantaridis et al., 2005), with added regularization term $\Omega(\mathbf{w})$:

$$\hat{R}[f_{\mathbf{w}}] = \frac{1}{m}\sum_{i=1}^{m} \max_{\bar{y}} \{f_{\mathbf{w}}(\mathbf{x}_i, \bar{\mathbf{y}}) - f_{\mathbf{w}}(\mathbf{x}_i, \mathbf{y}_i) + \Delta(\mathbf{y}, \bar{\mathbf{y}}, \mathbf{x}_i), 0\} + \Omega(\mathbf{w}). \tag{1}$$

This problem setting differs fundamentally from traditional language identification settings. In our setting, the actual identification of a language from example strings takes place in the decoding step. In this step, the decoder searches the space of regular expressions. But instead of retrieving an expression that generates all strings in $\mathbf{x}$, it searches for an expression that maximizes the value of a $\mathbf{w}$-parameterized decision function that receives the strings and the candidate expression as arguments. In a separate learning step, the parameters $\mathbf{w}$ are optimized using batches of strings and corresponding regular expressions. The training process has to optimize the model parameters $\mathbf{w}$ such that the expected deviation between the decoder's output and a regular expression written by a human postmaster is minimized. Training data of this form, and an optimization criterion that measures the expected discrepancy between the conjectured regular expressions and regular expressions written by a human labeler, are not part of traditional language identification settings.

4. Identifying Regular Expressions

This section details our approach to identifying regular expressions based on generalized linear models and structured output spaces.

4.1 Problem Decomposition

Without any approximations, the decoding problem—the problem of identifying the regular expression $\mathbf{y}$ that maximizes the parametric decision function—is insurmountable. For any string, an exponential number of matching regular expressions of up to the same length can be constructed by substituting constant symbols for wildcards. In addition, constant symbols can be replaced by disjunctions and "loopy" syntactic elements can be added to create infinitely many longer regular expressions that also match the original string. Because the space of regular expressions is discrete, it also does not lend itself well to approaches based on gradient descent.

We decompose the problem into two more strongly constrained learning problems. We decompose the parameters $\mathbf{w} = (\mathbf{u}\ \mathbf{v})^{\mathsf{T}}$ and the loss function $\Delta = \Delta_{\mathbf{u}} + \Delta_{\mathbf{v}}$ into parts that are minimized sequentially. In the first step, $\mathbf{u}$-parameterized model $f_{\mathbf{u}}$ produces a regular expression $\tilde{\mathbf{y}}$ that is constrained to being a specialization of the maximal alignment of the strings in $\mathbf{x}$. Specializations of maximal alignments of the strings in $\mathbf{x}$ tend to be long regular expressions that characterize the entirety of the strings in $\mathbf{x}$. In a second step, $\mathbf{v}$-parameterized model $f_{\mathbf{v}}$ therefore produces a concise substring $\hat{\mathbf{y}}$ of $\tilde{\mathbf{y}}$.

Definition 1 (Alignment, Maximal Alignment) *The set of* alignments $A_{\mathbf{x}}$ *of a batch of strings* $\mathbf{x}$ *contains all concatenations in which strings from* Σ^+ *and the wildcard symbol "(.*)" alternate, and that generates all elements of* $\mathbf{x}$*.The set of* maximal alignments $A^*_{\mathbf{x}} \subseteq A_{\mathbf{x}}$ *contains all alignments of the strings in* $\mathbf{x}$ *which share the property that no other alignment in* $A_{\mathbf{x}}$ *has more constant symbols.*

A specialization of an alignment is a string that has been derived from an alignment by replacing one or several wildcard symbols by another regular expression. Figure 3 illustrates the process of generating a maximal alignment, and the subsequent step of specializing it.

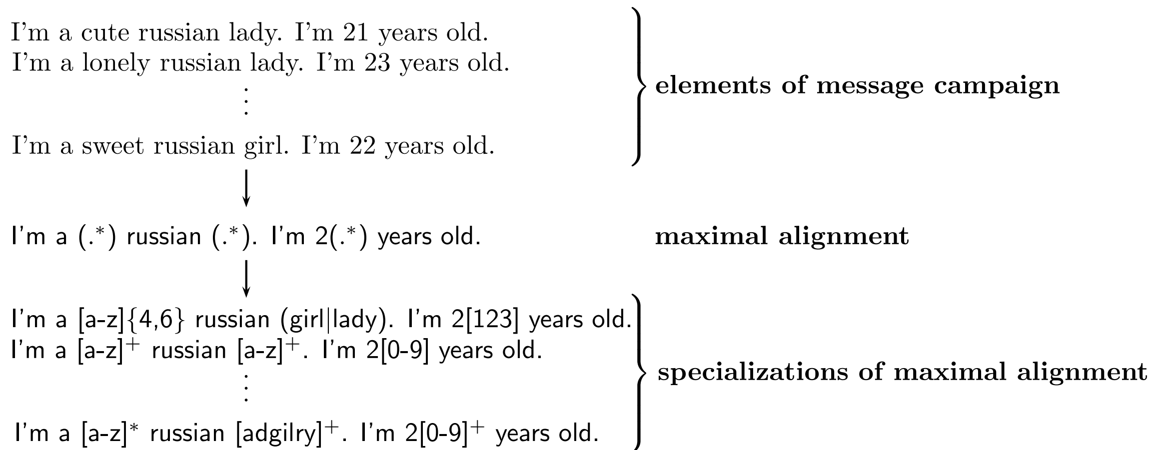


Figure 3: Examples of regular expressions, which are specializations of a maximal alignment of strings.

The loss function for this step should measure the semantic and syntactic deviation between the conjecture $\tilde{\mathbf{y}}$ and the manually written $\mathbf{y}$ for batch $\mathbf{x}$. We define a loss function $\Delta_{\mathbf{u}}(\mathbf{y}, \tilde{\mathbf{y}}, \mathbf{x})$ that compares the set of parse trees in $\mathcal{T}_{par}^{\mathbf{y},x}$, for each string $x \in \mathbf{x}$ to the most similar tree in $\mathcal{T}_{par}^{\tilde{\mathbf{y}},x}$; if no such parse tree exists, the summand is defined as $\frac{1}{|\mathbf{x}|}$ (Equation 2). Similarly to a loss function for hierarchical classification (Cesa-Bianchi et al., 2006), the difference of two parse trees for a given string x is quantified by a comparison of the paths that lead to the characters of the string. Two paths are compared by means of the intersection of their nodes (Equation 3). This loss is bounded between zero and one; it is zero if and only if the two regular expressions $\tilde{\mathbf{y}}$ and $\mathbf{y}$ are equal.

$$\Delta_{\mathbf{u}}(\mathbf{y}, \tilde{\mathbf{y}}, \mathbf{x}) = \frac{1}{|\mathbf{x}|} \sum_{x \in \mathbf{x}} \begin{cases} \Delta_{\text{tree}}(\mathbf{y}, \tilde{\mathbf{y}}, x) & \text{if } x \in L(\tilde{\mathbf{y}}) \\ 1 & \text{otherwise} \end{cases} \tag{2}$$

$$\text{with } \Delta_{\text{tree}}(\mathbf{y}, \tilde{\mathbf{y}}, x) = 1 - \frac{1}{|\mathcal{T}_{par}^{\mathbf{y},x}|} \sum_{t \in \mathcal{T}_{par}^{\mathbf{y},x}} \max_{\tilde{t} \in \mathcal{T}_{par}^{\tilde{\mathbf{y}},x}} \frac{1}{|x|} \sum_{j=1}^{|x|} \frac{|t_{|j} \cap \tilde{t}_{|j}|}{\max\{|t_{|j}|, |\tilde{t}_{|j}|\}} \tag{3}$$

Figure 4 illustrates how the tree loss is calculated for a single string: for each symbol, the corresponding paths of the syntax trees spawned by $\mathbf{y}$ and $\tilde{\mathbf{y}}$ are compared. Each pair of corresponding paths incurs a loss according to the proportion of nodes that are labeled with differing subexpressions.

Because the regular expression created in this step is a specialization of a *maximal* alignment, it is not generally concise. In the second step, $\mathbf{v}$-parameterized model $f_{\mathbf{v}}$ produces

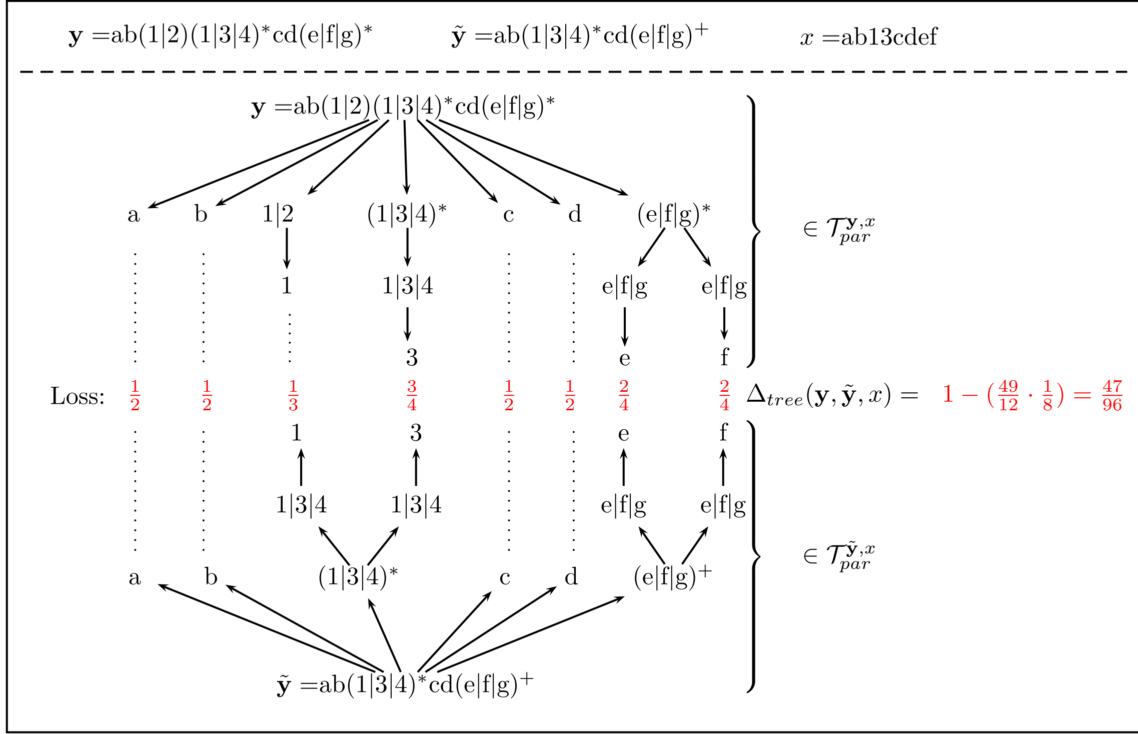


Figure 4: Calculation of the tree loss $\Delta_{tree}(\mathbf{y}, \tilde{\mathbf{y}}, x)$ for a given string x and two regular expressions $\mathbf{y}$ and $\tilde{\mathbf{y}}$.

a regular expression $\hat{\mathbf{y}} \in \mathcal{Y}_\Sigma$ that is a subexpression of $\tilde{\mathbf{y}}$; that is, $\tilde{\mathbf{y}} = \mathbf{y}_{\text{pre}}\hat{\mathbf{y}}\mathbf{y}_{\text{suf}}$ with $\mathbf{y}_{\text{pre}}, \mathbf{y}_{\text{suf}} \in \mathcal{Y}_\Sigma$. Loss function $\Delta_{\mathbf{v}}(\mathbf{y}, \hat{\mathbf{y}})$ is based on the length of the longest common substring $\text{lcs}(\mathbf{y}, \hat{\mathbf{y}})$ of $\mathbf{y}$ and $\hat{\mathbf{y}}$. The loss—defined in Equation 4—is zero, if the longest common substring of $\mathbf{y}$ and $\hat{\mathbf{y}}$ is equal to both $\mathbf{y}$ and $\hat{\mathbf{y}}$. In this case, $\mathbf{y} = \hat{\mathbf{y}}$. Otherwise, it increases as the longest common substring of $\mathbf{y}$ and $\hat{\mathbf{y}}$ decreases.

$$\Delta_{\mathbf{v}}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2}\left[\left(\frac{|\mathbf{y}| - |\text{lcs}(\mathbf{y}, \hat{\mathbf{y}})|}{|\mathbf{y}|}\right) + \left(\frac{|\hat{\mathbf{y}}| - |\text{lcs}(\mathbf{y}, \hat{\mathbf{y}})|}{|\hat{\mathbf{y}}|}\right)\right] \tag{4}$$

In the following subsections, we derive decoders and optimization problems for these two subproblems.

4.2 Learning to Generate Regular Expressions

We model $f_{\mathbf{u}}$ as a linear discriminant function $\mathbf{u}^\mathsf{T}\Psi_{\mathbf{u}}(\mathbf{x}, \mathbf{y})$ for a joint feature representation of the input $\mathbf{x}$ and output $\mathbf{y}$ (Tsochantaridis et al., 2005):

$$\tilde{\mathbf{y}} = \arg\max_{\mathbf{y} \in \mathcal{Y}_\Sigma} f_{\mathbf{u}}(\mathbf{x}, \mathbf{y}) = \arg\max_{\mathbf{y} \in \mathcal{Y}_\Sigma} \mathbf{u}^\mathsf{T}\Psi_{\mathbf{u}}(\mathbf{x}, \mathbf{y}).$$

4.2.1 Joint Feature Representation for Generating Regular Expressions

The joint feature representation $\Psi_{\mathbf{u}}(\mathbf{x}, \mathbf{y})$ captures structural properties of an expression $\mathbf{y}$ and joint properties of input batch $\mathbf{x}$ and regular expression $\mathbf{y}$.

It captures structural properties of a regular expression $\mathbf{y}$ by features that indicate a specific nesting of regular expression operators—for instance, whether a concatenation occurs within a disjunction. More formally, we first define a binary vector

$$\Lambda_{\mathbf{u}}(\mathbf{y}) = \begin{pmatrix} [\![\mathbf{y} = \mathbf{y}_1 \dots \mathbf{y}_k]\!] \\ [\![\mathbf{y} = \mathbf{y}_1 | \dots | \mathbf{y}_k]\!] \\ [\![\mathbf{y} = [\mathbf{y}_1 \dots \mathbf{y}_k]]\!] \\ [\![\mathbf{y} = \mathbf{y}_1^*]\!] \\ [\![\mathbf{y} = \mathbf{y}_1?]\!] \\ [\![\mathbf{y} = \mathbf{y}_1^+]\!] \\ [\![\mathbf{y} = \mathbf{y}_1\{l\}]\!] \\ [\![\mathbf{y} = \mathbf{y}_1\{l, u\}]\!] \\ [\![\mathbf{y} = r_1]\!] \\ \vdots \\ [\![\mathbf{y} = r_l]\!] \\ [\![\mathbf{y} \in \Sigma]\!] \\ [\![\mathbf{y} = \epsilon]\!] \end{pmatrix} \tag{5}$$

that encodes the top-level operator used in the regular expression $\mathbf{y}$, where $[\![\cdot]\!]$ is the indicator function of its Boolean argument. In Equation 5, $\mathbf{y}_1, \dots, \mathbf{y}_k \in \mathcal{Y}_\Sigma$ are regular expressions, $l, u \in \mathbb{N}$, and $\{r_1, \dots, r_l\}$ is a set of ranges and popular macros. For our application, we use the set {0-9, a-f, a-z, A-F, A-Z, \S, \e, \d, "."} (see Table 6 in the appendix) because these are frequently used by postmasters.

For any two nodes v' and v'' in the syntax tree of $\mathbf{y}$ that are connected by an edge—indicating that $\mathbf{y}'' = \Gamma^{\mathbf{y}}_{syn}(v'')$ is an argument subexpression of $\mathbf{y}' = \Gamma^{\mathbf{y}}_{syn}(v')$—the tensor product $\Lambda_{\mathbf{u}}(\mathbf{y}') \otimes \Lambda_{\mathbf{u}}(\mathbf{y}'')$ defines a binary vector that encodes the specific nesting of operators at node v'. Feature vector $\Psi_{\mathbf{u}}(\mathbf{x}, \mathbf{y})$ will aggregate these vectors over all pairs of adjacent nodes in the syntax tree of $\mathbf{y}$.

Joint properties of an input batch $\mathbf{x}$ and a regular expression $\mathbf{y}$ are encoded in a similar way as follows. Recall that for any node v' in the syntax tree, $M^{\mathbf{y},\mathbf{x}}(v')$ denotes the set of substrings in $\mathbf{x}$ that are generated by the subexpression $\mathbf{y}' = \Gamma^{\mathbf{y}}_{syn}(v')$ that v' is labeled with. We define a vector $\Phi_{\mathbf{u}}(M^{\mathbf{y},\mathbf{x}}(v'))$ of attributes of this set. Any property may be accounted for; for our application, we include the average string length, the inclusion of the empty string, the proportion of capital letters, and many other attributes. The list of attributes used in our experiments is included in the appendix in Table 3. A joint encoding of properties of the subexpression $\mathbf{y}'$ and the set of substrings generated by $\mathbf{y}'$ is given by the tensor product $\Phi_{\mathbf{u}}(M^{\mathbf{y},\mathbf{x}}(v')) \otimes \Lambda_{\mathbf{u}}(\mathbf{y}')$.

The joint feature vector $\Psi_{\mathbf{u}}(\mathbf{x}, \mathbf{y})$ is obtained by aggregating operator-nesting information over all edges in the syntax tree, and joint properties of subexpressions $\mathbf{y}'$ and the set of substrings which they generate over all nodes in the syntax tree:

$$\Psi_{\mathbf{u}}(\mathbf{x}, \mathbf{y}) = \begin{pmatrix} \sum_{(v',v'') \in E^{\mathbf{y}}_{syn}} \Lambda_{\mathbf{u}}(\Gamma^{\mathbf{y}}_{syn}(v')) \otimes \Lambda_{\mathbf{u}}(\Gamma^{\mathbf{y}}_{syn}(v'')) \\ \sum_{v' \in V^{\mathbf{y}}_{syn}} \Phi_{\mathbf{u}}(M^{\mathbf{y},\mathbf{x}}(v')) \otimes \Lambda_{\mathbf{u}}(\Gamma^{\mathbf{y}}_{syn}(v')) \end{pmatrix}. \tag{6}$$

4.2.2 Decoding Specializations of the Maximal Alignment

At application time, the highest-scoring regular expression $\tilde{\mathbf{y}}$ according to model $f_{\mathbf{u}}$ has to be decoded. Model $f_{\mathbf{u}}$ is constrained to producing specializations of the maximal alignment; however, searching the space of all possible specializations of the maximal alignment is still not feasible. The following observation illustrates that $f_{\mathbf{u}}$ may not even have a maximum, because there may always be a longer expression that attains a higher score.

Observation 1 *Given a string* $\mathbf{a}$ *that contains at least one wildcard symbol "(.*)", let* $\mathcal{Y}_{\mathbf{a}}$ *be the set of all specializations that replace wildcards in* $\mathbf{a}$ *by any regular expression in* $\mathcal{Y}$*. Then, there are parameters* $\mathbf{u}$ *such that for each* $\mathbf{y}$ *there is a* $\mathbf{y}' \in \mathcal{Y}_{\mathbf{a}}$ *with* $f_{\mathbf{u}}(\mathbf{y}') > f_{\mathbf{u}}(\mathbf{y})$*.*

Proof Joint feature vector Ψ from Equation 6 contains two parts. The first part contains operator-nesting information over all edges in the syntax tree and the second part contains joint properties of subexpressions and the set of substrings which they generate over all nodes in the syntax tree. We construct $\mathbf{u}$ as follows: Let all weights in $\mathbf{u}$ be zero, except for the entry which weights the count of alternatives within an alternative; this entry receives any positive weight. For any string $\mathbf{a}$ that contains a wildcard symbol, by substituting the wildcard for an alternative of a wildcard and arbitrarily many other subexpressions, one can create a string $\mathbf{a}'$ that contains a wildcard within an additional alternative. Repeated application of this substitution creates arbitrarily many alternatives within alternatives and the inner product of $\mathbf{u}$ and $\Psi_{\mathbf{u}}$ can therefore become arbitrarily large. ■

Observation 1 implies that exact decoding of arbitrary decision functions $f_{\mathbf{u}}$ is not possible. However, we can follow the *under-generating* principle (Finley and Joachims, 2008) and employ a decoder that maximizes $f_{\mathbf{u}}$ over a constrained subspace that has a maximum. Observation 1 implies that the decision-function value of that maximum over the constrained space may be arbitrarily much lower than the decision-function value of some elements of the unconstrained space. But when it comes to formulating the optimization problem in Subsection 4.2.3, we will require that, for each training example, the training regular expression shall have a higher decision function value (by some margin) than the highest-scoring incorrect regular expression that is actually found by the decoder. Hence, despite Observation 1, the learning problem may produce parameters which let the constrained decoder produce the desired output.

The search space is first constrained to specializations of a maximal alignment of the input set of strings $\mathbf{x}$; see Definition 1. A maximal alignment of two strings can be determined efficiently using Hirschberg's algorithm (Hirschberg, 1975) which is an instance of dynamic programming. By contrast, finding the maximal alignment of a *set of strings* is NP-hard (Wang and Jiang, 1994); known algorithms are exponential in the number $|\mathbf{x}|$ of strings in $\mathbf{x}$. However, *progressive alignment* heuristics find an alignment of a set of strings by incrementally aligning pairs of strings. Note that the set of specializations of a maximal alignment is still generally infinitely large: each wildcard symbol can be replaced by every possible regular expression $\mathcal{Y}_{\Sigma}$. Therefore, our decoding algorithm starts by finding an approximately maximal alignment using the Hirschberg algorithm, and proceeds to construct a more constrained search space in which each wildcard symbol can be replaced only by

regular expressions over constant symbols that occur in the strings in $\mathbf{x}$ at the corresponding positions.

The definition of the constrained search space is guided by an analysis of the syntactic variants and maximum nesting depth observed in expressions written by postmasters—a detailed record can be found in the appendix; see Tables 6, 7, and 8. The space contains all specializations of the maximal alignment in which the j-th wildcard is replaced by any element from $\hat{\mathcal{Y}}_D^{M_j}$, which is constructed as follows. Firstly, $\hat{\mathcal{Y}}_D^{M_j}$ contains any subexpression that occurs within any training regular expression, and that matches the substrings of input $\mathbf{x}$ which the alignment procedure has substituted for the j-th wildcard. In addition, the alternative of all substring aligned at the j-th wildcard symbol is added. For each character-alternative expression in that set—*e.g.,* [abc]—all possible iterators and range generalizations used by postmasters are added.

Given an alignment $\mathbf{a_x} = a_0(.^*)a_1 \dots (.^*)a_n$ of all strings in $\mathbf{x}$, the constrained search space

$$\hat{\mathcal{Y}}_{\mathbf{x},D} = \{a_0\mathbf{y}_1a_1 \dots \mathbf{y}_na_n | \text{for all } j : \mathbf{y}_j \in \hat{\mathcal{Y}}_D^{M_j}\} \tag{7}$$

contains all specializations of $\mathbf{a_x}$ in which the j-th wildcard symbol is replaced by any element of a set $\hat{\mathcal{Y}}_D^{M_j}$, where M_j is the matching list of the j-th node in $T_{syn}^{\mathbf{a_x}}$ that is labeled with the wildcard symbol "(.*)". The sets $\hat{\mathcal{Y}}_D^{M_j}$ are constructed using Algorithm 1. Each of the lines 7, 9, 10, 11, and 12 of Algorithm 1 adds at most one element to $\hat{\mathcal{Y}}_D^{M_j}$ and thus Algorithm 1 generates a finite set of possible regular expressions—hence, the search space of possible substitutions for each of the n wildcard symbols is linear in the number of subexpressions that occur in the training sample.

We now turn towards the problem of determining the highest-scoring regular expression $f_\mathbf{w}(\mathbf{x})$. Maximization over all regular expressions is approximated by maximization over the space defined by Equation 7:

$$\arg\max_{\mathbf{y}\in\mathcal{Y}_\Sigma} \mathbf{u}^\mathsf{T}\Psi_\mathbf{u}(\mathbf{x},\mathbf{y}) \approx \arg\max_{\mathbf{y}\in\hat{\mathcal{Y}}_{\mathbf{x},D}} \mathbf{u}^\mathsf{T}\Psi_\mathbf{u}(\mathbf{x},\mathbf{y}).$$

Due to the simple syntactic structure of the alignment and the definition of $\Psi_\mathbf{u}$ we can state the following theorem:

Theorem 2 *The maximization problem of finding the highest-scoring regular expression $f_\mathbf{u}(\mathbf{x})$ can be decomposed into independent maximization problems for each of the $\mathbf{y}_j$ that replaces the j-th wildcard in the alignment $\mathbf{a_x}$, given the alignment and the definition of $\Psi_\mathbf{u}$:*

$$\arg\max_{\mathbf{y}_1,\dots,\mathbf{y}_n} f_\mathbf{u}(\mathbf{x}, a_0\mathbf{y}_1a_1\dots\mathbf{y}_na_n) = a_0\mathbf{y}_1^*a_1\dots\mathbf{y}_n^*a_n$$
$$\text{with } \mathbf{y}_j^* = \arg\max_{\mathbf{y}_j\in\hat{\mathcal{Y}}_D^{M_j}} \mathbf{u}^\mathsf{T}\left(\Psi_\mathbf{u}(\mathbf{y}_j, M_j) + \mathbf{c}_{\mathbf{y}_j}\right).$$

Proof By its definition, $f_\mathbf{u}(\mathbf{x}, a_0\mathbf{y}_1a_1\dots\mathbf{y}_na_n) = \mathbf{u}^\mathsf{T}\Psi_\mathbf{u}(\mathbf{x}, a_0\mathbf{y}_1a_1\dots\mathbf{y}_na_n)$. Decision function Feature vector $\Psi_\mathbf{u}(\mathbf{x},\mathbf{y})$ decomposes linearly into a sum over the nodes and a

Algorithm 1 Constructing the decoding space

1: **Input:** Subexpressions $\mathcal{Y}_D$ and alignment $\mathbf{a_x} = a_0(.^*)a_1 \dots (.^*)a_n$ of the strings in $\mathbf{x}$.
2: **let** $T^{\mathbf{a_x}}_{syn}$ be the syntax tree of the alignment and $v_1, \dots, v_n$ be the nodes labeled $\Gamma^{\mathbf{a_x}}_{syn}(v_j) =$ "(.*)".
3: **for** $j = 1 \dots n$ **do**
4: **let** $M_j = M^{\mathbf{a_x},\mathbf{x}}(v_j)$.
5: Initialize $\hat{\mathcal{Y}}^{M_j}_D$ to $\{\mathbf{y} \in \mathcal{Y}_D | M_j \subseteq L(\mathbf{y})\}$
6: **let** $x_1, \dots, x_m$ be the elements of M_j; add $(x_1| \dots |x_m)$ to $\hat{\mathcal{Y}}^{M_j}_D$.
7: **let** u be the length of the longest string and l be the length of the shortest string in M_j.
8: **if** $[\beta\mathbf{y}_1 \dots \mathbf{y}_k] \in \hat{\mathcal{Y}}^{M_j}_D$, where $\beta \in \Sigma^*$ and $\mathbf{y}_1 \dots \mathbf{y}_k$ are ranges or special macros (*e.g.*, a-z, \e), then add $[\alpha\mathbf{y}_1 \dots \mathbf{y}_k]$ to $\hat{\mathcal{Y}}^{M_j}_D$, where $\alpha \in \Sigma^*$ is the longest string that satisfies $M_j \subseteq L([\alpha\mathbf{y}_1 \dots \mathbf{y}_k])$, if such an α exists.
9: **for all** $[\mathbf{y}] \in \hat{\mathcal{Y}}^{M_j}_D$ **do**
10: add $[\mathbf{y}]^*$ and $[\mathbf{y}]\{l, u\}$ to $\hat{\mathcal{Y}}^{M_j}_D$.
11: **if** $l = u$, then add $[\mathbf{y}]\{l\}$ to $\hat{\mathcal{Y}}^{M_j}_D$.
12: **if** $u \leq 1$, then add $[\mathbf{y}]?$ to $\hat{\mathcal{Y}}^{M_j}_D$.
13: **if** $l > 0$, then add $[\mathbf{y}]^+$ to $\hat{\mathcal{Y}}^{M_j}_D$.
14: **end for**
15: **end for**
16: **Output** $\hat{\mathcal{Y}}^{M_1}_D, \dots, \hat{\mathcal{Y}}^{M_n}_D$.

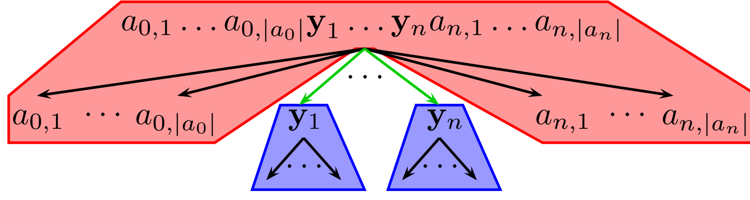


Figure 5: Structure of a syntax tree for an element of $\hat{\mathcal{Y}}_{\mathbf{x},D}$.

sum over pairs of adjacent nodes (see Equation 6). The syntax tree of an instantiation $\mathbf{y} = a_0\mathbf{y}_1a_1 \dots \mathbf{y}_na_n$ of the alignment $\mathbf{a_x}$ consists of a root node labeled as an alternating concatenation of constant strings a_j and subexpressions $\mathbf{y}_j$ (see Figure 5). This root node is connected to a layer on which constant strings $a_j = a_{j,1} \dots a_{j,|a_j|}$ and subtrees $T_{syn}^{\mathbf{y}_j}$ alternate (blue area in Figure 5). However, the terms in Equation 8 that correspond to the root node $\mathbf{y}$ and the a_j are constant for all values of the $\mathbf{y}_j$ (red area in Figure 5). Since no edges connect multiple wildcards, the feature representation of these subtrees can be decomposed into n independent summands as in Equation 9.

$$
\begin{aligned}
&\Psi_{\mathbf{u}}(\mathbf{x}, a_0\mathbf{y}_1a_1 \dots \mathbf{y}_na_n) &(8)\\
&= \begin{pmatrix} \sum_{j=1}^{n} \Lambda_{\mathbf{u}}(\mathbf{y}) \otimes \Lambda_{\mathbf{u}}(\mathbf{y}_j) + \sum_{j=0}^{n}\sum_{q=1}^{|a_j|} \Lambda_{\mathbf{u}}(\mathbf{y}) \otimes \Lambda_{\mathbf{u}}(a_{j,q}) \\ \Phi_{\mathbf{u}}(\{\mathbf{x}\}) \otimes \Lambda_{\mathbf{c}}(\mathbf{y}) + \sum_{j=0}^{n}\sum_{q=1}^{|a_j|} \Phi_{\mathbf{u}}(\{a_{j,q}\}) \otimes \Lambda_{\mathbf{u}}(a_{j,q}) \end{pmatrix} \\
&+ \begin{pmatrix} \sum_{j=1}^{n} \sum_{(v',v'') \in E_{syn}^{\mathbf{y}_j}} \Lambda_{\mathbf{u}}(\Gamma_{syn}^{\mathbf{y}_j}(v')) \otimes \Lambda_{\mathbf{u}}(\Gamma_{syn}^{\mathbf{y}_j}(v'')) \\ \sum_{j=1}^{n} \sum_{v' \in V_{syn}^{\mathbf{y}_j}} \Phi_{\mathbf{u}}(M^{\mathbf{y}_j,M_j}(v')) \otimes \Lambda_{\mathbf{u}}(\Gamma_{syn}^{\mathbf{y}_j}(v')) \end{pmatrix} \\
&= \begin{pmatrix} \mathbf{0} \\ \Phi_{\mathbf{u}}(\{\mathbf{x}\}) \otimes \Lambda_{\mathbf{u}}(\mathbf{y}) \end{pmatrix} + \sum_{j=0}^{n}\sum_{q=1}^{|a_i|} \begin{pmatrix} \Lambda_{\mathbf{u}}(\mathbf{y}) \otimes \Lambda_{\mathbf{u}}(a_{j,q}) \\ \Phi_{\mathbf{u}}(\{a_{j,q}\}) \otimes \Lambda_{\mathbf{u}}(a_{j,q}) \end{pmatrix} \\
&+ \sum_{j=1}^{n} \left(\Psi_{\mathbf{u}}(\mathbf{y}_j, M_j) + \begin{pmatrix} \Lambda_{\mathbf{u}}(\mathbf{y}) \otimes \Lambda_{\mathbf{u}}(\mathbf{y}_j) \\ \mathbf{0} \end{pmatrix} \right) &(9)
\end{aligned}
$$

Since the top-level operator of an alignment is a concatenation for any $\mathbf{y} \in \hat{\mathcal{Y}}_{\mathbf{x},D}$, we can write $\Lambda_{\mathbf{u}}(\mathbf{y})$ as a constant $\Lambda_\bullet$, defined as the output feature vector (Equation 5) of a concatenation.

Thus, the maximization over all $\mathbf{y} = a_0\mathbf{y}_1a_1 \dots \mathbf{y}_na_n$ can be decomposed into n maximization problems over

$$
\mathbf{y}_j^* = \arg\max_{\mathbf{y}_j \in \hat{\mathcal{Y}}_D^{M_j}} \mathbf{u}^{\mathsf{T}} \left(\Psi_{\mathbf{u}}(\mathbf{y}_j, M_j) + \begin{pmatrix} \Lambda_\bullet \otimes \Lambda_{\mathbf{u}}(\mathbf{y}_j) \\ \mathbf{0} \end{pmatrix} \right)
$$

which can be solved in $\mathcal{O}(n \times |\mathcal{Y}_D|)$. ■

4.2.3 Optimization Problem for Specializations of a Maximal Alignment

We will now address the process of minimizing the portion of the regularized empirical risk $\hat{R}[f_{\mathbf{w}}]$, defined in Equation 1, that depends on $\mathbf{u}$ for the ℓ_2 regularizer $\Omega_{\mathbf{c}}(\mathbf{u}) = \frac{1}{2C}||\mathbf{u}||^2$. The decision function $f_{\mathbf{w}}$ decomposes into $f_{\mathbf{u}}$ and $f_{\mathbf{v}}$; loss function $\Delta_{\mathbf{w}}$ decomposes into $\Delta_{\mathbf{u}}$ and $\Delta_{\mathbf{v}}$. While loss function $\Delta_{\mathbf{u}}$ defined in Equation 2 is not convex itself, the hinged upper bound used in Equation 1 is. Approximating a loss function by its hinged upper bound in such a way is referred to as margin-rescaling (Tsochantaridis et al., 2005). We define slack term ξ_i as this hinged loss for instance i:

$$\xi_i = \max\left\{\max_{\bar{\mathbf{y}}\neq\mathbf{y}_i}\{\mathbf{u}^\mathsf{T}(\Psi_{\mathbf{u}}(\mathbf{x}_i,\bar{\mathbf{y}}) - \Psi_{\mathbf{u}}(\mathbf{x}_i,\mathbf{y}_i)) + \Delta_{\mathbf{u}}(\mathbf{y}_i,\bar{\mathbf{y}},\mathbf{x})\}, 0\right\}. \tag{10}$$

The maximum in Equation 10 is over all $\bar{\mathbf{y}} \in \mathcal{Y}_\Sigma \setminus \{\mathbf{y}_i\}$. When the risk is rephrased as a constrained optimization problem, the maximum produces one constraint per element of $\bar{\mathbf{y}} \in \mathcal{Y}_\Sigma \setminus \{\mathbf{y}_i\}$. However, since the decoder searches only the set $\hat{\mathcal{Y}}_{\mathbf{x}_i,D}$, it is sufficient to enforce the constraints on this subset which leads to a finite search space.

When the loss is replaced by its upper bound—the slack variable ξ—and for $\Omega_{\mathbf{u}}(\mathbf{u}) = \frac{1}{2C_{\mathbf{u}}}||\mathbf{u}||^2$, the minimization of the regularized empirical risk (Equation 1) is reduced to Optimization Problem 1.

Optimization Problem 1 *Over parameters* $\mathbf{u}$*, find*

$$\mathbf{u}^* = \arg\min_{\mathbf{u},\xi} \frac{1}{2}||\mathbf{u}||^2 + \frac{C_{\mathbf{u}}}{m}\sum_{i=1}^{m}\xi_i, \text{ such that} \tag{11}$$

$$\forall i, \forall \bar{\mathbf{y}} \in \hat{\mathcal{Y}}_{\mathbf{x}_i,D}\setminus\{\mathbf{y}_i\} : \mathbf{u}^\mathsf{T}(\Psi_{\mathbf{u}}(\mathbf{x}_i,\mathbf{y}_i) - \Psi_{\mathbf{u}}(\mathbf{x}_i,\bar{\mathbf{y}})) \geq \Delta_{\mathbf{u}}(\mathbf{y}_i,\bar{\mathbf{y}},\mathbf{x}) - \xi_i, \tag{12}$$

This optimization problem is convex, since the objective (Equation 11) is convex and the constraints (Equation 12) are affine in $\mathbf{u}$. Hence, the solution is unique and can be found efficiently by cutting plane methods as Pegasos (Shalev-Shwartz et al., 2011) or SVM^{struct} (Tsochantaridis et al., 2005).

These algorithms require to identify the constraint with highest slack variable ξ_i for a given $\mathbf{x}_i$,

$$\bar{\mathbf{y}} = \arg\max_{\mathbf{y}\in\hat{\mathcal{Y}}_{\mathbf{x}_i,D}\setminus\{\mathbf{y}_i\}} \mathbf{u}^\mathsf{T}\Psi_{\mathbf{u}}(\mathbf{x}_i,\mathbf{y}) + \Delta_{\mathbf{u}}(\mathbf{y}_i,\mathbf{y},\mathbf{x}),$$

in the optimization procedure, repeatedly.

Algorithm 1 constructs the constrained search space $\hat{\mathcal{Y}}_{\mathbf{x}_i,D}$ such that $x \in L(\mathbf{y})$ for each $x \in \mathbf{x}_i$ and $\mathbf{y} \in \hat{\mathcal{Y}}_{\mathbf{x}_i,D}$. Hence, the "otherwise"-case in Equation 2 never applies within our search space. Without this case, Equations 2 and 3 decompose linearly over the nodes of the parse tree, and therefore the wildcards. Hence, $\bar{\mathbf{y}}$ can be identified by maximizing over the variables $\bar{\mathbf{y}}_j$ independently in Step 5 of Algorithm 2. Algorithm 2 finds the constraint that is violated most strongly within the constrained search space in $\mathcal{O}(n \times |\mathcal{Y}_D|)$. This ensures a polynomial execution time of the optimization algorithm. We refer to this learning procedure as *REx-SVM*.

Algorithm 2 Most strongly violated constraint

1: **Inout:** batch $\mathbf{x}$, model $f_{\mathbf{u}}$, correct output $\mathbf{y}$.
2: Infer alignment $\mathbf{a_x} = a_0(.^*)a_1 \dots (.^*)a_n$ for $\mathbf{x}$.
3: Let $T^{\mathbf{a_x}}_{syn}$ be the syntax tree of $\mathbf{a_x}$ and let $v_1, \dots, v_n$ be the nodes labeled $\Gamma^{\mathbf{a_x}}_{syn}(v_j) =$ "(.*)".
4: **for all** $j = 1 \dots n$ **do**
5: Let $M_j = M^{\mathbf{a_x},\mathbf{x}}(v_j)$ and calculate the $\hat{\mathcal{Y}}^{M_j}_D$ using Algorithm 1.
6:
$$\bar{\mathbf{y}}_j = \arg\max_{\mathbf{y}'_j \in \hat{\mathcal{Y}}^{M_j}_D} \mathbf{u}^\mathsf{T} \left(\Psi_{\mathbf{u}}(\mathbf{y}'_j, M_j) + \begin{pmatrix} \Lambda_\bullet \otimes \Lambda_{\mathbf{u}}(\mathbf{y}'_j) \\ \mathbf{0} \end{pmatrix} \right) +$$
$$\Delta_{\mathbf{u}}(\mathbf{y}, a_0(.^*)a_1 \dots (.^*)a_{j-1}\mathbf{y}'_j a_j(.^*)a_{j+1} \dots (.^*)a_n, \mathbf{x})$$
7: **end for**
8: Let $\bar{\mathbf{y}}$ abbreviate $a_0\bar{\mathbf{y}}_1 a_1 \dots \bar{\mathbf{y}}_n a_n$
9: **if** $\bar{\mathbf{y}} = \mathbf{y}$ **then**
10: Assign a value of $\bar{\mathbf{y}}'_j \in \hat{\mathcal{Y}}^{M_j}_D$ to one of the variables $\bar{\mathbf{y}}_j$ such that the smallest decrease of $f_{\mathbf{u}}(\mathbf{x}, \bar{\mathbf{y}}) + \Delta_{\text{tree}}(\mathbf{y}, \bar{\mathbf{y}})$ is obtained but the constraint $\bar{\mathbf{y}} \neq \mathbf{y}$ is enforced.
11: **end if**
12: **Output:** $\bar{\mathbf{y}}$

4.3 Learning to Extract Concise Substrings

Model $f_{\mathbf{u}}$ generates regular expressions that tend to be very specific because they are specializations of a maximal alignment of all strings in the input set $\mathbf{x}$. Human postmasters, by contrast, prefer to focus on only a characteristic part of the message for which they write a specific regular expression. In order to allow the overall model $f_{\mathbf{w}}$ to produce expressions that characterize only a part of the strings, this section focuses on a second model, $f_{\mathbf{v}}$, that selects a substring from its input string $\tilde{\mathbf{y}}$. We model $f_{\mathbf{v}}$ as a linear discriminant function with a joint feature representation $\Psi_{\mathbf{v}}$ of the input regular expression $\tilde{\mathbf{y}}$ and the output regular expression $\mathbf{y}$; decision function $f_{\mathbf{v}}$ is maximized over the set $\Pi(\tilde{\mathbf{y}})$ of all substrings of $\tilde{\mathbf{y}}$ that are themselves regular expressions:

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y} \in \Pi(\tilde{\mathbf{y}})} f_{\mathbf{v}}(\tilde{\mathbf{y}}, \mathbf{y}) = \arg\max_{\mathbf{y} \in \Pi(\tilde{\mathbf{y}})} \mathbf{v}^\mathsf{T} \Psi_{\mathbf{v}}(\tilde{\mathbf{y}}, \mathbf{y}), \tag{13}$$
$$\text{with } \Pi(\tilde{\mathbf{y}}) = \{\mathbf{y}_{\text{in}} \in \mathcal{Y}_\Sigma | \tilde{\mathbf{y}} = \mathbf{y}_{\text{pre}}\mathbf{y}_{\text{in}}\mathbf{y}_{\text{suf}} \text{ and } \mathbf{y}_{\text{pre}}, \mathbf{y}_{\text{suf}} \in \mathcal{Y}_\Sigma\}.$$

4.3.1 Joint Feature Representation for Concise Substrings

The joint feature representation $\Psi_{\mathbf{v}}(\tilde{\mathbf{y}}, \mathbf{y})$ captures structural and semantic features $\Phi_{\text{input}}(\tilde{\mathbf{y}})$ of the input regular expression $\tilde{\mathbf{y}}$, features $\Phi_{\text{output}}(\mathbf{y})$ of the output regular expression $\mathbf{y}$ and all combinations of properties of the input and output expression.

Vector $\Phi_{\text{input}}(\tilde{\mathbf{y}})$ of features of the input regular expression $\tilde{\mathbf{y}}$ includes features that indicates whether $\tilde{\mathbf{y}}$ special mail specific content like a a subject line, a "From" line, or a "Reply-To" line. A range of features test whether particular special characters are included in $\tilde{\mathbf{y}}$; other features refer to the number of subexpressions that are entailed in $\tilde{\mathbf{y}}$. The list of used features in our experiments is shown in Table 4 in the appendix.

Feature vector $\Phi_{\text{output}}(\mathbf{y})$ of the output regular expression $\mathbf{y}$ stacks up features which indicate how many subexpressions and how many words are included in the regular expression. In addition it contains features that test for special phrases that frequently occur in email batches and features that test whether words with a high spam score are included in the subject line. We identify this list of suspicious words by training a linear classifier that separates spam from non-spam emails; the list contains the 150 words which have the highest weights for the spam class. The list of features that we used in the experiments can be found in Table 5 in the appendix.

The final joint feature representation $\Psi_{\mathbf{v}}(\tilde{\mathbf{y}}, \mathbf{y})$ is defined as vector that includes the input features $\Phi_{\text{input}}(\tilde{\mathbf{y}})$, the output features $\Phi_{\text{output}}(\mathbf{y})$, and all products of an input and an output feature:

$$\Psi_{\mathbf{v}}(\tilde{\mathbf{y}}, \mathbf{y}) = \begin{pmatrix} \Phi_{\text{input}}(\tilde{\mathbf{y}}) \\ \Phi_{\text{output}}(\mathbf{y}) \\ \Phi_{\text{input}}(\tilde{\mathbf{y}}) \otimes \Phi_{\text{output}}(\mathbf{y}) \end{pmatrix}. \tag{14}$$

4.3.2 Decoding a Concise Regular Expression

At application time, the highest-scoring regular expression $\hat{\mathbf{y}}$ according to Equation 13 has to be identified. The search space $\Pi(\tilde{\mathbf{y}})$ contains all substrings of $\tilde{\mathbf{y}}$; since $\tilde{\mathbf{y}}$ is typically a very long string and calculating all features is an expensive operation, evaluating the decision function for all substrings is infeasible. Again, we follow the under-generating principle (Finley and Joachims, 2008) and constrain the search to the space $\Pi_s(\tilde{\mathbf{y}})$ contains regular expressions whose string length is at most s. Within this set, the decoder conducts an exhaustive search. One can easily observe that when the highest-scoring regular expression's string length exceeds s, then the highest-scoring regular expression of size at most s can have an arbitrarily much lower decision function value.

Observation 2 *Let* $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}\in\Pi(\tilde{\mathbf{y}})} f_{\mathbf{v}}(\tilde{\mathbf{y}}, \mathbf{y})$ *and* $\hat{\mathbf{y}}_s = \arg\max_{\mathbf{y}\in\Pi(\tilde{\mathbf{y}}),|\mathbf{y}|\leq s} f_{\mathbf{v}}(\tilde{\mathbf{y}}, \mathbf{y})$. *If* $|\hat{\mathbf{y}}| > s$, *then for each number* d, *there is a parameter vector* $\mathbf{v}$ *such that* $f_{\mathbf{v}}(\tilde{\mathbf{y}}, \hat{\mathbf{y}}) > f_{\mathbf{v}}(\tilde{\mathbf{y}}, \hat{\mathbf{y}}_s) + d$.

Proof The output features of vector $\Psi_{\mathbf{v}}$ (Equation 14) include the number of constant symbols and the number of non-constant subexpressions in output expression $\mathbf{y}$. Let $\mathbf{v}$ be all zero except for these two weights which we set to $d + 1$. Then $f_{\mathbf{v}}(\tilde{\mathbf{y}}, \mathbf{y})$ is maximized by output $\hat{\mathbf{y}} = \tilde{\mathbf{y}}$. If $|\hat{\mathbf{y}}_s| < |\hat{\mathbf{y}}|$, then $\hat{\mathbf{y}}_s$ is missing at least one initial or trailing constant or non-constant symbol. By the definition of $\mathbf{v}$, decision function $f_{\mathbf{v}}(\tilde{\mathbf{y}}, \hat{\mathbf{y}}) = \mathbf{v}^{\mathsf{T}}\Psi_{\mathbf{v}}(\tilde{\mathbf{y}}, \mathbf{y}) > \mathbf{v}^{\mathsf{T}}\Psi_{\mathbf{v}}(\tilde{\mathbf{y}}, \mathbf{y}_s) + d = f_{\mathbf{v}}(\tilde{\mathbf{y}}, \hat{\mathbf{y}}_s) + d$. ■

Choosing too small a constant s can therefore lead to poor decoding results. In our experiments, we choose s to be greater than the longest regular expressions seen in the training data.

4.3.3 Optimization Problem for Concise Expressions

Training data $D = \{((\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^m$ for the overall learning problem consist of pairs of sets $\mathbf{x}_i$ of strings and corresponding regular expressions $\mathbf{y}_i$. Model $f_{\mathbf{u}}$—discussed in Section 4.2—produces intermediate expressions $\tilde{\mathbf{y}}_i$ that are specializations of a maximal alignment, before

model $f_{\mathbf{v}}(\tilde{\mathbf{y}}_i)$ gives the final predictions $\hat{\mathbf{y}}_i$. Hence, training data for model $f_{\mathbf{v}}$ naturally consists of the pairs $\{(\tilde{\mathbf{y}}_i, \mathbf{y}_i)\}_{i=1}^m$.

We will now derive an optimization problem from the portions of Equation 1 that depend on $\mathbf{v}$. Decision function $f_{\mathbf{w}}$ decomposes into $f_{\mathbf{u}} + f_{\mathbf{v}}$; loss function $\Delta_{\mathbf{w}}$ into $\Delta_{\mathbf{u}}$ and $\Delta_{\mathbf{v}}$. The regularizer decomposes, and we use the ℓ_2 regularizer for $\mathbf{v}$ as well, $\Omega_{\mathbf{s}}(\mathbf{v}) = \frac{1}{2C}||\mathbf{v}||^2$. This leads to Optimization Problem 2.

Optimization Problem 2 *Over parameters* $\mathbf{v}$*, find*

$$\begin{aligned}
&\mathbf{v}^* = \arg\min_{\mathbf{v},\xi} \frac{1}{2}||\mathbf{v}||^2 + \frac{C_{\mathbf{v}}}{m}\sum_{i=1}^m \xi_i, \text{ such that} \\
&\forall i, \forall \bar{\mathbf{y}} \in \Pi_s(\mathbf{y}_i)\backslash\{\mathbf{y}_i\} : \mathbf{v}^\mathsf{T}(\Psi_{\mathbf{v}}(\tilde{\mathbf{y}}_i, \mathbf{y}_i) - \Psi_{\mathbf{s}}(\tilde{\mathbf{y}}_i, \bar{\mathbf{y}})) \\
&\qquad\qquad\qquad\qquad \geq \Delta_{\mathbf{v}}(\bar{\mathbf{y}}, \mathbf{y}_i) - \xi_i, \\
&\forall i : \xi_i \geq 0.
\end{aligned}$$

Optimization Problem 2 minimizes the regularized empirical risk under the assumption that the decoder uses the restricted search space $\Pi_s(\tilde{\mathbf{y}})$ for some fixed value of the maximal string length s. We refer to the complete model

$$\begin{aligned}
\hat{\mathbf{y}} &= \arg\max_{\mathbf{y}\in\Pi(\tilde{\mathbf{y}})} \mathbf{v}^\mathsf{T}\Psi_{\mathbf{v}}(\tilde{\mathbf{y}}, \mathbf{y}), \\
\text{with } \tilde{\mathbf{y}} &= a_0\mathbf{y}_1^* a_1 \dots \mathbf{y}_n^* a_n \\
\text{and } \mathbf{y}_j^* &= \arg\max_{\mathbf{y}_j \in \hat{\mathcal{Y}}_D^{M_j}} \mathbf{u}^\mathsf{T}\left(\Psi_{\mathbf{u}}(\mathbf{y}_j, M_j) + \left(\Lambda_\bullet \otimes \Lambda_{\mathbf{u}}(\mathbf{y}_j)\right)\right)
\end{aligned}$$

for predicting concise regular expressions as *REx-SVM*^{short}.

5. Case Study

We investigate whether postmasters accept the output of *REx-SVM* and *REx-SVM*^{short} for blacklisting mailing campaigns during regular operations of a commercial email service. We also evaluate how accurately *REx-SVM* and *REx-SVM*^{short} and their reference methods identify the extensions of mailing campaigns.

In order to obtain training data for the model $f_{\mathbf{u}}$ that generates a regular expression from an input batch of strings, we apply the Bayesian clustering technique of Haider and Scheffer (2009) to the stream of messages that arrive at an email service during its regular operations; the method identifies 158 mailing campaigns with a total of 12,763 messages. Postmasters of the email service write regular expressions for each batch in order to blacklist the mailing campaign; these expressions serve as labels. We will refer to this data collection as the *ESP data set*.

In order to obtain additional training data for the model $f_{\mathbf{v}}$ that selects a concise substring of a regular expression that is a specialization of the maximal alignment, we observe another 478 pairs of regular expressions with their concise subexpressions that postmasters write in order to blacklist mailing campaigns. We collected this data by using the predicted regular expression $\tilde{\mathbf{y}} = f_{\mathbf{u}}(\mathbf{x})$ for each batch of emails $\mathbf{x}$ as training observation

and the postmaster-written expression $\mathbf{y}$ as the label. We train a first-stage model $f_{\mathbf{u}}$ on the 158 labeled batches after tuning regularization parameter $C_{\mathbf{u}}$ with 10-fold cross validation. We tune the regularization parameter $C_{\mathbf{v}}$ using leave-one-out cross validation and train a global model $f_{\mathbf{v}}$ that is used in the following experiments.

5.1 Evaluation by Postmasters

<table>
<tr><th></th><th>Campaign 1</th><th>Campaign 2</th><th>Campaign 3</th></tr>
<tr><td>Postmaster</td><td>Please send the request to my email (simon|george)@(gmail|yahoo).com</td><td>Email:wester_(payin|pay)@yahoo.com
Yours sincerely,
Mr [A-Z][a-z]$^+$ [A-Z][a-z]$^+$</td><td>(Reply-To|From):(mosk@aven|sevid@donald).com
Subject: GET YOUR MONEY</td></tr>
<tr><td>REx-SVM</td><td>... This work takes [0-9-]$^+$ hours per week and requires absolutely no investment. The essence of this work for incoming client requests in your city. The starting salary is about [0-9]$^+$ EUR per month + bonuses.
...
Please send the request to my email [a-z]$^+$@(gmail|yahoo).com and I will answer you personally as soon as possible ...</td><td>... agreed that the sum of US$[0-9,]$^+$ should be transferred to you out of the funds that Federal Government of Nigeria has set aside as a compensation to everyone who have by one way or the other sent money to fraudsters in Nigeria.
...
Email:wester_(payin|pay)@yahoo.com
Yours sincerely,
Mr [A-Za-z]$^+$ [A-Za-z]$^+$ ...</td><td>...
(Reply-To|From):(mosk@aven|sevid@donald).com
Subject: GET YOUR MONEY
...
I am Mr. Sopha Chum, An Auditing and accounting section staff in National Bank of Cambodia.
...</td></tr>
<tr><td>REx-SVM^{short}</td><td>Please send the request to my email [a-z]$^+$@(gmail|yahoo).com and I will answer you personally as soon as possible</td><td>Email:wester_(payin|pay)@yahoo.com
Yours sincerely, Mr [A-Za-z]$^+$ [A-Za-z]$^+$</td><td>(Reply-To|From):(mosk@aven|sevid@donald).com
Subject: GET YOUR MONEY</td></tr>
</table>

Figure 6: Regular expressions created by a postmaster and corresponding output of *REx-SVM* and *REx-SVM*^{short}.

The trained model $f_{\mathbf{u}}$ is deployed; the user interface presents newly detected batches together with the regular expressions $f_{\mathbf{u}}(\mathbf{x})$ generated by *REx-SVM* and expressions $f_{\mathbf{w}}(\mathbf{x})$ generated by *REx-SVM*^{short} to the postmasters during regular operations of the email service. The postmasters are charged with blacklisting the campaigns with a suitable regular expression. We measure how frequently the postmasters copy the output of *REx-SVM*^{short}, copy a substring from the output of *REx-SVM*, copy but edit an output, and how frequently they choose to write an expression from scratch.

Over the course of this study, the postmasters write 153 regular expressions. They copy the exact regular expressions generated by *REx-SVM*^{short} in 64.7% of the cases. Another 14.4% of the time, they copy a substring from the output of *REx-SVM* and use it without changes. In 7.8% of the cases, the postmasters copy and edit a substring from *REx-SVM*, and in 13.1% of the cases they write an expression from scratch. Hence, tasked with producing a regular expression that will block the mailing campaign during live operations, the postmasters prefer working with the automatically generated output to writing an expression from scratch 86.9% of the time.

To illustrate different cases, Figure 6 compares regular expressions selected by a postmaster to excerpts of regular expressions generated by *REx-SVM*, and regular expressions generated by *REx-SVM*^{short}, respectively. In the first example, *REx-SVM* over-generalizes the contact email address, and *REx-SVM*^{short} predicts a slightly longer expression than

the postmaster prefers to select. Nevertheless, all three regular expressions characterize the extension of the mailing campaign accurately. In the second example, *REx-SVM* finds a slightly shorter but slightly more general expression for the closing signature (the term "[A-Za-z]$^{+}$" would allow for capital letters within the name while the term "[A-Z][a-z]$^{+}$" does not). The second-stage model $f_{\mathbf{v}}$ has selected the same substring that the postmaster prefers. In the third example, postmaster and *REx-SVM^{short}* agree perfectly.

The fact that postmasters are content to accept generated regular expression does not imply that they would have written the exact same rules. We now want to explore how frequently *REx-SVM^{short}* is able to produce the same regular expression that postmasters would have written. We execute leave-one-out cross validation over the regular expressions in the ESP data set. In each iteration, a new model $f_{\mathbf{u}}$ is trained on all but one regular expressions (model $f_{\mathbf{v}}$ is only trained once on different data).

We compare the output of *REx-SVM^{short}* to the held-out expression. We find that in 59.49% (94) of the cases, *REx-SVM^{short}* generates the exact regular expression written by the postmaster; in 11.39% (18) of the cases the held-out expression is a substring of the regular expression created by *REx-SVM* but distinct to the extracted expression found by *REx-SVM^{short}*. In 8.86% (14) of the cases the held-out regular expression can be obtained by modifying a substring of the string created by *REx-SVM*. In 20.25% (32) of the cases, generated and manually-written regular expression are distinct. These rates are consistent with the acceptance rates of the postmasters. When manually written and automatically generated regular expressions differ from each other, both expressions may still serve their purpose of filtering a particular batch of emails. We will explore to which extent this is the case in the next subsection.

5.2 Spam Filtering Performance

We evaluate the ability of *REx-SVM*, *REx-SVM^{short}*, and reference methods to identify the exact extension of email spam campaigns. We use the approximately maximal *alignment* of the strings determined by sequential alignment in a batch $\mathbf{x}$ as a reference method. Here, the *ReLIE* method (Li et al., 2008) serves as an additional reference. *ReLIE* takes the *alignment* as starting point of its search for a regular expression that matches the emails in the input batch and does not match any of the additional negative examples by applying a set of transformation rules. *ReLIE* receives an additional 10,000 emails that are not part of any batch as negative data, which gives it a small data advantage over *REx-SVM* and *REx-SVM^{short}*. *REx$_{0/1}$-SVM* is a variant of the *REx-SVM* that uses the zero-one loss instead of the loss function $\Delta_{\mathbf{u}}$ defined in Equation 2. An additional *content*-based filter employed by the provider has been trained on several million spam and non-spam emails.

Our experiments are based on two evaluation data sets. The *ESP data set* consists of the 158 batches of 12,763 emails and postmaster-written regular expressions; it is described in Section 5. In addition, we collect another 42 large spam batches with a total of 17,197 emails for which we do not have postmaster-written regular expressions. In order to be able to measure false-positive rates (the rate at which emails that are not part of a campaign are erroneously included), we use an additional 135,000 non-spam emails, also from the provider.

Additionally, we use a *public* data set that consists of 100 batches of emails extracted from the *Bruce Guenther archive*[1], containing a total of 63,512 emails. To measure false-positive rates on this public data set, we use 17,419 non-spam emails from the *Enron corpus*[2] and 76,466 non-spam emails of the *TREC corpus*[3]. The public data set is available to other researchers.

Experiments on the ESP data set are conducted as follows. We employ a constant model of $f_{\mathbf{v}}$, trained on 478 pairs of predicted expressions $\tilde{\mathbf{y}}$ and postmaster-written expressions $\mathbf{y}$. We first carry out a "leave-one-batch-out" cross-validation loop over the 158 labeled batches of the ESP data set. In each iteration, 157 batches are reserved for training of $f_{\mathbf{u}}$. On this training portion of the data, regularization parameter $C_{\mathbf{u}}$ is tuned in a nested 10-fold cross validation loop, then a model is trained on all 157 training batches. An inner loop then iterates over the size of the input batch. For each size $|\mathbf{x}|$, messages from the held-out batch are drawn into $\mathbf{x}$ at random and a regular expression $\hat{\mathbf{y}} = f_{\mathbf{w}}(\mathbf{x})$ is generated. The remaining elements of the held-out batch are used to to measure the true-positive rate of $\hat{\mathbf{y}}$, and the 135,000 non-spam emails are used to determine its false-positive rate. After that, a model is trained on all 158 labeled batches, and the evaluation iterates over the remaining 42 batches that are not labeled with a postmaster-written regular expression. For each value of $|\mathbf{x}|$, an input $\mathbf{x}$ is drawn, a prediction $\hat{\mathbf{y}}$ is generated, its true-positive rate is measured on the remaining elements of the current batch and its false-positive rate on the 135,000 non-spam messages. Standard errors are computed based on all 200 observations.

For evaluation on the public data set, parameter $C_{\mathbf{u}}$ is tuned with 10-fold cross validation and then a model is trained on all 158 labeled batches of the ESP data set. The evaluation iterates over all 100 batches of the public data set and, in an inner loop, over values of $|\mathbf{x}|$. An input set $\mathbf{x}$ is drawn at random from the current batch, the true-positive rate of $\hat{\mathbf{y}} = f_{\mathbf{w}}(\mathbf{x})$ is measured on the remaining elements of the current batch and the false-positive rate of $\hat{\mathbf{y}}$ is measured on the Enron and TREC emails.

Figure 7 shows the true- and false-positive rates for all methods on both data sets. The horizontal axis displays the number of emails in the input batch $\mathbf{x}$. Error bars indicate the standard error. The true-positive rate measures the proportion of a batch that is recognized while the false-positive rate counts emails that match a regular expression although they are not an element of the corresponding campaign. The *alignment* has the highest true-positive rate and a high false-positive rate because it is the most general bound of the decoder's search space. *ReLIE* only has to carry out very few transformation steps until no negative examples are covered—in some cases none at all. Consequently, it has similarly high true- and false-positive rates. *REx-SVM* and *REx-SVM*^{short} attain a slightly lower true-positive rate, and a substantially lower false-positive rate. The false-positive rates of *REx-SVM*, *REx*$_{0/1}$*-SVM*, and *REx-SVM*^{short} lie more than an order of magnitude below the rate of the commercial *content*-based spam filter employed by the email service provider. The zero-one loss leads to comparable false-positive but lower true-positive rates, rendering the loss function $\Delta_{\mathbf{u}}$ preferable to the zero-one loss. The true-positive rate of *REx-SVM*^{short} is significantly higher than the true-positive rate of *REx-SVM* for small sizes of the input batch; it requires only very few input strings in order to generate regular expressions which

1. http://untroubled.org/spam/
2. http://www.cs.cmu.edu/~enron/
3. http://trec.nist.gov/data/spam.html

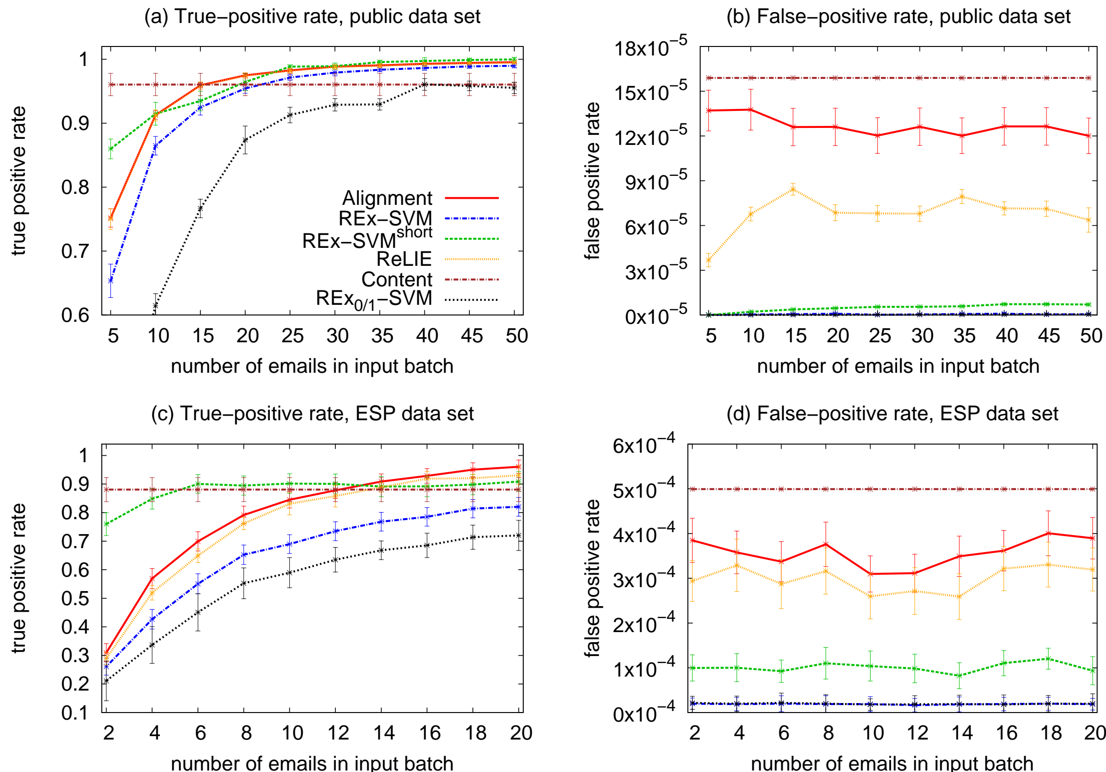


Figure 7: True-positive and false-positive rates over the number of used emails in the input batch $\mathbf{x}$ for the public and ESP data sets.

can be used to describe nearly the entire extension of a batch at a very low false-positive rate.

Finally, we determine the risk of the studied methods producing a regular expression that causes at least one false-positive match of an email which does not belong to the batch. *REx-SVM*'s risk of producing a regular expression that incurs at least one false-positive match is 2.5%; for *REx-SVM*^{short}, this risk is 3.7%; for *alignment*, the risk is 6.3%, and for *ReLIE*, it is 5.1%.

5.3 Learning Curves, Execution Time

We study learning curves of the loss functions of *REx-SVM* and *REx-SVM*^{short}. Figure 8 (a) shows the average loss $\Delta_{\mathbf{u}}$ based on cross validation with one batch held out, as a function of the number of training batches. The "minimum loss" baseline shows the smallest possible loss within the constrained search space; it visualizes how much constraining the search space contributes to the overall loss. This value is obtained by an altered search procedure that minimizes the loss function between prediction and the postmaster-written regular expression instead of the decision function. This loss-minimizing expression has a

lower decision function value than the predicted regular expression; the difference between minimum loss and the loss of *REx-SVM* and $REx_{0/1}$-*SVM*, respectively, can be attributed to imperfections of the model. Figure 8 (a) also shows the loss of the *alignment*. This loss serves as an upper bound and visualizes how much the parameterized models contribute to minimizing the error. For completeness, Figure 10 in the appendix shows the learning curves on the training data.

Figure 8 (b) shows the average loss $\Delta_{\mathbf{v}}$ based on 10 fold cross validation and the average loss on the training data. The impact of the regularization parameters $C_{\mathbf{u}}$ and $C_{\mathbf{v}}$ is shown in Figure 11 in the appendix.

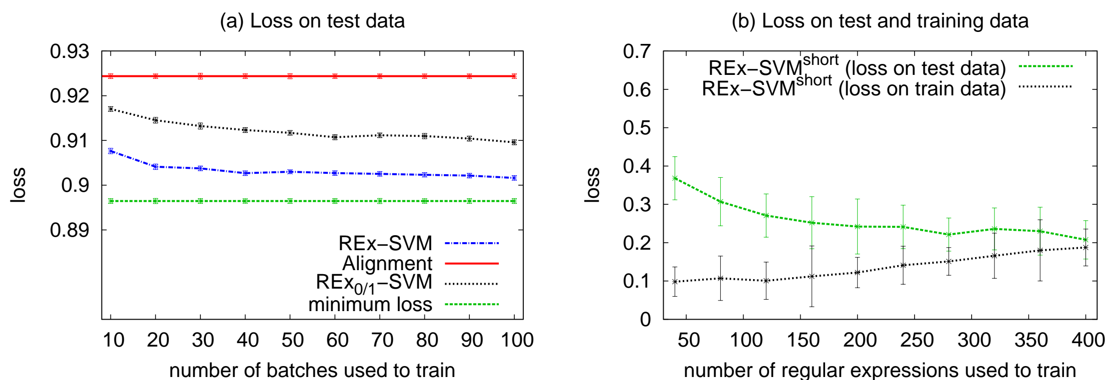


Figure 8: (a) Loss $\Delta_{\mathbf{u}}$ of model $f_{\mathbf{u}}$ on the test data (left Figure). (b) Loss $\Delta_{\mathbf{v}}$ of model $f_{\mathbf{v}}$ on the training and test data. Error bars indicate standard errors.

Table 5.3 measures how much *REx-SVM*^{short} reduces the length of the expressions produced by *REx-SVM*. We can conclude that *REx-SVM*^{short} reduces the length of the output of *REx-SVM* by an average of 92%.

Method	mean	standard error
REx-SVM	2141	2063
REx-SVM^{short}	95	92

Table 1: Number of characters in automatically-generated regular expressions.

The execution time for learning is consistent with prior findings of between linear and quadratic for the SVM optimization process—see Figure 9(a). Figure 9 (b) shows the execution time of the decoder that generates a regular expression for input batch $\mathbf{x}$ at application time. *ReLIE* does not require training.

In order to use regular expressions to blacklist email spam, the email service provider's infrastructure has to continuously match all active regular expressions against the stream of incoming emails. This acceptor is implemented as a deterministic finite-state automaton.

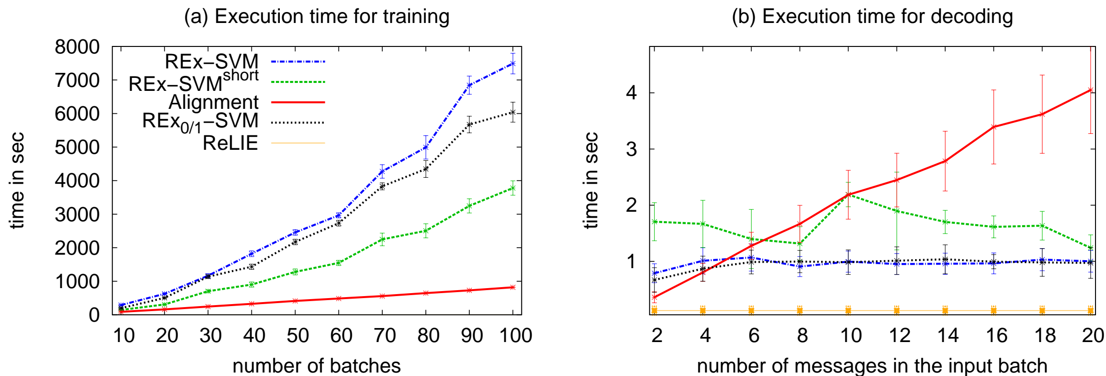


Figure 9: Execution time for training a model (a) and decoding a regular expression at application time (b).

The automaton has to be kept in main memory, and therefore the number of states determines the number of regular expressions that can be searched for in parallel. Table 2 shows the average number of states of an acceptor, generated by the method of Dubé and Feeley (2000) from the regular expressions of *REx-SVM* and *REx-SVM*^{short}. The average number of states of regular expressions by *REx-SVM*^{short} is close to the average number of states of expressions written by a human postmaster, while *alignment*, *ReLIE*, and *REx-SVM* require impractically large accepting automata.

Method	mean	median	standard error
alignment	5709	4059	389.1
REx-SVM	5473	2995	520.8
ReLIE	5632	3587	465.9
REx-SVM^{short}	72	69	1.8
postmaster	68	48	4.6

Table 2: Number of states of an accepting finite-state automaton.

6. Related Work

Gold (1967) shows that it is impossible to exactly identify any regular language from finitely many positive examples. In his framework, a learner makes a conjecture after each new positive example; only finitely many initial conjectures may be incorrect. Our notion of minimizing an expected difference between conjecture and target language over a distribution of input strings reflects a more statistically-inspired notion of learning. Also, in our problem setting the learner has access to pairs of sets of strings and corresponding regular expressions.

Most work of identification of regular languages focuses on learning automata (Denis, 2001; Parekh and Honavar, 2001; Clark and Thollard, 2004). Since regular languages are

accepted by finite automata, the problems of learning regular languages and learning finite automata are tightly coupled. However, a compact regular language may have an accepting automaton with a large number of states and, analogously, transforming compact automata into regular expressions can lead to lengthy terms that do not lend themselves to human comprehension (Fernau, 2009).

Positive learnability results can be obtained for restricted classes of deterministic finite automata with positive examples (Angluin, 1978; Abe and Warmuth, 1990); for instance expressions in which each symbol occurs at most k times (Bex et al., 2008), disjunction-free expressions (Brāzma, 1993), and disjunctions of left-aligned disjunction-free expressions (Fernau, 2009) have been studied. These approaches aim only at the identification of a target language. By contrast, here the structural resemblance of the conjecture to a target regular expression is integral part of the problem setting. This also makes it necessary to account for the broader syntactic spectrum of regular expressions.

Xie et al. (2008) use regular expressions to detect URLs in spam batches and develop a spam filter with low false-positive rate. The *ReLIE*-algorithm (Li et al., 2008) (used as a reference method in our experiments) learns regular expressions from positive and negative examples given an initial expression by applying a set of transformation rules as long as this improves the separation of positive and negative examples. Brauer et al. (2011) develop an algorithm that builds a data structure of commonalities of several aligned strings and transforms these strings into a specific regular expression. Because of a high data overhead, their algorithm works best for short strings, such as telephone numbers and names of software products.

Structured output spaces are a flexible tool for a wide array of problem settings, including sequence labeling, sequence alignment, and natural language parsing (Tsochantaridis et al., 2005). In our problem setting we are interested in predicting a structured object, i.e. a regular expression. To solve problems with structured output spaces an extension of the support vector mashines (SVMs, Vapnik, 1998) can be used. Such structural SVMs were used to solve a several number of prediction tasks ranging from classification with taxonomies, label sequence learning, sequence alignment to natural language parsing (Tsochantaridis et al., 2005). The problem of detecting message campaigns in the stream of emails has been addressed with structured output spaces based on manually grouped training messages (Haider et al., 2007), and with graphical models without the need for labeled training data (Haider and Scheffer, 2009).

Our problem setting and method differ from all prior work on learning regular expressions in their objective criterion and training data. Unlike in prior work, the learner in our setting has access to additional labeled data in the form of pairs of a set of strings and a corresponding regular expressions. At the same time, the learner's goal is not just to find an expression that identifies an extension of strings, but to find *the* expression which the process that has labeled the training data would most likely generate. This implies that the learner has to model the labeler's preference of using specific syntactic constructs in a specific syntactic context and for specific matching substrings.

7. Conclusions

Complementing the language-identification paradigm, we address the problem of learning to map a set of strings to a concise regular expression that resembles an expression which a human would have written. Training data consists of batches of strings and corresponding regular expressions. We phrase this problem as a two-staged learning problem with structured output spaces and engineer appropriate loss functions. We devise a first-stage decoder that searches a space of specializations of a maximal alignment of the input strings. We devise a second-stage decoder that searches for a substring of the first-stage result. We derive optimization problems for both stages.

From our case study, we conclude that *REx-SVM*^{short} frequently predicts the exact regular expression that a postmaster would have written. In other cases, it generates an expression that postmasters accept without or with small modifications. Regarding their accuracy for the problem of filtering email spam, we conclude that *REx-SVM* and *REx-SVM*^{short} give a high true-positive rate at a false-positive rate that is an order of magnitude lower than that of a commercial content-based filter. *REx-SVM*^{short} attains a higher true-positive rate, in particular for small input batches. *REx-SVM*^{short} generates regular expressions that can be accepted by a finite-state automaton that has just slightly more states than an accepting automaton for regular expressions written by a human postmaster. *REx-SVM* and all reference methods, by contrast, can only be accepted by impractically large finite-state automata. *REx-SVM*^{short} is being used by a commercial email service provider and complements content-based and IP-address based filtering.

Acknowledgments

This work was funded by a grant from STRATO AG. We would like to thank the anonymous reviewers for their helpful comments.

References

N. Abe and M. K. Warmuth. On the computational complexity of approximating distributions by probabilistic automata. In *Proceedings of the Conference on Learning Theory*, pages 52–66, 1990.

D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, 1978.

G. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. In *Proceeding of the International World Wide Web Conference*, pages 825–834, 2008.

F. Brauer, R. Rieger, A. Mocan, and W.M. Barczynski. Enabling information extraction by inference of regular expressions from sample entities. In *Proceedings of the Conference on Information and Knowledge Management*, pages 1285–1294. ACM, 2011. ISBN 978-1-4503-0717-8.

A. Brāzma. Efficient identification of regular expressions from representative examples. In *Proceedings of the Annual Conference on Computational Learning Theory*, pages 236–242, 1993.

N. Cesa-Bianchi, C. Gentile, and L. Zaniboni. Incremental algorithms for hierarchical classification. *Machine Learning*, 7:31–54, 2006.

A. Clark and F. Thollard. PAC-learnability of probabilistic deterministic finite state automata. *Machine Learning Research*, 5:473–497, 2004.

F. Denis. Learning regular languages from simple positive examples. *Machine Learning*, 44:27–66, 2001.

D. Dubé and M. Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, 2000.

H. Fernau. Algorithms for learning regular expressions from positive data. *Information and Computation*, 207(4):521–541, 2009.

T. Finley and T. Joachims. Training structural SVMs when exact inference is intractable. In *Proceedings of the International Conference on Machine Learning*, 2008.

E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

P. Haider and T. Scheffer. Bayesian clustering for email campaign detection. In *Proceedings of the International Conference on Machine Learning*, 2009.

P. Haider, U. Brefeld, and T. Scheffer. Supervised clustering of streaming data for email batch detection. In *Proceedings of the International Conference on Machine Learning*, 2007.

D. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 21–30, 2008.

R. Parekh and V. Honavar. Learning DFA from simple examples. *Machine Learning*, 44: 9–35, 2001.

P. Prasse, C. Sawade, N. Landwehr, and T. Scheffer. Learning to identify regular expressions that describe email campaigns. In *Proceedings of the International Conference on Machine Learning*, 2012.

S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter. Pegasos: primal estimated sub-gradient solver for svm. *Mathematical Programming*, 127(1):1–28, 2011.

I. Tsochantaridis, T. Joachims, T. Hofmann, and Y. Altun. Large margin methods for structured and interdependent output variables. *Journal of Machine Learning Research*, 6:1453–1484, 2005.

V. Vapnik. *Statistical Learning Theory*. Wiley, 1998.

L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.

Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: signatures and characteristics. In *Proceedings of the ACM SIGCOMM Conference*, pages 171–182, 2008.

Appendix A

Syntax and Semantics of Regular Expressions

Definition 3 (Regular Expressions) The set $\mathcal{Y}_\Sigma$ of regular expressions over an ordered alphabet Σ is recursively defined as follows.

- Every $\mathbf{y}_j \in \Sigma \cup \{\epsilon, ., \backslash\mathsf{S}, \backslash\mathsf{e}, \backslash\mathsf{w}, \backslash\mathsf{d}\}$, every range $\mathbf{y}_j = l_{min}\text{–}l_{max}$, where $l_{min}, l_{max} \in \Sigma$ and $l_{min} < l_{max}$, and their disjunction $[\mathbf{y}_1 \dots \mathbf{y}_k]$ are regular expressions.

- If $\mathbf{y}_1, \dots, \mathbf{y}_k \in \mathcal{Y}_\Sigma$ are regular expressions, so are the concatenation $\mathbf{y} = \mathbf{y}_1 \dots \mathbf{y}_k$, the disjunction $\mathbf{y} = \mathbf{y}_1 | \dots | \mathbf{y}_k$, $\mathbf{y} = \mathbf{y}_1?$, $\mathbf{y} = (\mathbf{y}_1)$, and the repetitions $\mathbf{y} = \mathbf{y}_1^*$, $\mathbf{y} = \mathbf{y}_1^+$, $\mathbf{y} = \mathbf{y}_1\{l\}$, and $\mathbf{y} = \mathbf{y}_1\{l, u\}$, where $l, u \in \mathbb{N}$ and $l \le u$.

We now define the syntax tree, the parse tree, and the matching lists for a regular expression $\mathbf{y}$ and a string $x \in \Sigma^*$. The shorthand $(\mathbf{y} \to T_1, \dots, T_k)$ denotes the tree $T = (V, E, \Gamma, \le)$ with root node $v_0 \in V$ labeled with $\Gamma(v_0) = \mathbf{y}$ and subtrees $T_1, \dots, T_k$. The order $\le$ maintains the subtree orderings $\le_i$ and defines the root node as the minimum over the set V and $v' \le v''$ for all $v' \in V_i$ and $v'' \in V_j$, where $i < j$.

Definition 4 (Syntax Tree) The abstract syntax tree $T^{\mathbf{y}}_{syn}$ for a regular expression $\mathbf{y}$ is recursively defined as follows. Let $T^{\mathbf{y}_j}_{syn} = (V^{\mathbf{y}_j}_{syn}, E^{\mathbf{y}_j}_{syn}, \Gamma^{\mathbf{y}_j}_{syn}, \le^{\mathbf{y}_j}_{syn})$ be the syntax tree of the subexpression $\mathbf{y}_j$.

- If $\mathbf{y} \in \Sigma \cup \{\epsilon, ., \backslash\mathsf{S}, \backslash\mathsf{e}, \backslash\mathsf{w}, \backslash\mathsf{d}\}$, or if $\mathbf{y} = l_{min}\text{–}l_{max}$, where $l_{min}, l_{max} \in \Sigma$, we define $T^{\mathbf{y}}_{syn} = (\mathbf{y} \to \emptyset)$.

- If $\mathbf{y} = (\mathbf{y}_1)$, where $\mathbf{y}_1 \in \mathcal{Y}_\Sigma$, we define $T^{\mathbf{y}}_{syn} = T^{\mathbf{y}_1}_{syn}$.

- If $\mathbf{y} = \mathbf{y}_1^*$, $\mathbf{y} = \mathbf{y}_1^+$, $\mathbf{y} = \mathbf{y}_1\{l, u\}$, or if $\mathbf{y} = \mathbf{y}_1\{l\}$, where $\mathbf{y}_1 \in \mathcal{Y}_\Sigma$, $l, u \in \mathbb{N}$, and there exist no $\mathbf{y}', \mathbf{y}'' \in \mathcal{Y}_\Sigma$ such that $\mathbf{y}_1 = \mathbf{y}'|\mathbf{y}''$ or $\mathbf{y}_1 = \mathbf{y}'\mathbf{y}''$, we define $T^{\mathbf{y}}_{syn} = (\mathbf{y} \to T^{\mathbf{y}_1}_{syn})$.

- If $\mathbf{y} = \mathbf{y}_1 \dots \mathbf{y}_k$, where $\mathbf{y}_j \in \mathcal{Y}_\Sigma$, and there exist no $\mathbf{y}', \mathbf{y}'' \in \mathcal{Y}_\Sigma$ such that $\mathbf{y}_j = \mathbf{y}'|\mathbf{y}''$ or $\mathbf{y}_j = \mathbf{y}'\mathbf{y}''$, we define $T^{\mathbf{y}}_{syn} = (\mathbf{y} \to T^{\mathbf{y}_1}_{syn}, \dots, T^{\mathbf{y}_k}_{syn})$.

- If $\mathbf{y} = \mathbf{y}_1 | \dots | \mathbf{y}_k$, where $\mathbf{y}_j \in \mathcal{Y}_\Sigma$, and there exist no $\mathbf{y}', \mathbf{y}'' \in \mathcal{Y}_\Sigma$ such that $\mathbf{y}_j = \mathbf{y}'|\mathbf{y}''$, or if $\mathbf{y} = [\mathbf{y}_1 \dots \mathbf{y}_k]$ and there exist no $\mathbf{y}', \mathbf{y}'' \in \mathcal{Y}_\Sigma$ such that $\mathbf{y}_j = \mathbf{y}'\mathbf{y}''$, we define $T^{\mathbf{y}}_{syn} = (\mathbf{y} \to T^{\mathbf{y}_1}_{syn}, \dots, T^{\mathbf{y}_k}_{syn})$.

Definition 5 (Parse Tree and Matching List) Given a syntax tree $T^{\mathbf{y}}_{syn} = (V^{\mathbf{y}}_{syn}, E^{\mathbf{y}}_{syn}, \Gamma^{\mathbf{y}}_{syn}, \leq^{\mathbf{y}}_{syn})$ of a regular expression $\mathbf{y}$ with nodes $v \in V^{\mathbf{y}}_{syn}$ and a string $x \in L(\mathbf{y})$, a parse tree $T^{\mathbf{y},x}_{par}$ and the matching lists $M^{\mathbf{y},x}(v)$ for each $v \in V^{\mathbf{y}}_{syn}$ are recursively defined as follows. Let $T^{\mathbf{y}_j,x}_{par} = (V^{\mathbf{y}_j,x}_{par}, E^{\mathbf{y}_j,x}_{par}, \Gamma^{\mathbf{y}_j,x}_{par}, \leq^{\mathbf{y}_j,x}_{par})$ be the parse tree and $T^{\mathbf{y}_j}_{syn} = (V^{\mathbf{y}_j}_{syn}, E^{\mathbf{y}_j}_{syn}, \Gamma^{\mathbf{y}_j}_{syn}, \leq^{\mathbf{y}_j}_{syn})$ the syntax tree of the subexpression $\mathbf{y}_j$.

- If $\mathbf{y} = x$ and $x \in \Sigma \cup \{\epsilon\}$, we define
  $M^{\mathbf{y},x}(v_0) = \{x\}$ and
  $T^{\mathbf{y},x}_{par} = (\mathbf{y} \to \emptyset)$.

- If $\mathbf{y} = .$ and $x \in \Sigma$,
  $\mathbf{y} = l_{min}\text{–}l_{max}$ and $l_{min} \leq x \leq l_{max}$, or if
  $\mathbf{y} \in \{\backslash\mathsf{S}, \backslash\mathsf{w}, \backslash\mathsf{e}, \backslash\mathsf{d}\}$ and x is either a non-whitespace character (everything but spaces, tabs, and line breaks), a word character (letters, digits, and underscores), a character in $\{., -, \#, +\}$ or a word character, or a digit, respectively, we define
  $M^{\mathbf{y},x}(v) = \{x\}$ for all $v \in V^{\mathbf{y}}_{syn}$ and
  $T^{\mathbf{y},x}_{par} = (\mathbf{y} \to T^{x,x}_{par})$.

- If $\mathbf{y} = (\mathbf{y}_1)$ and $x \in \Sigma^*$, we define
  $M^{\mathbf{y},x}(v) = M^{\mathbf{y}_1,x}(v)$ for all $v \in V^{\mathbf{y}}_{syn}$ and
  $T^{\mathbf{y},x}_{par} = T^{\mathbf{y}_1,x}_{par}$

- If $\mathbf{y} = \mathbf{y}_1^*$, $x = x_1 \ldots x_k$, and $k \geq 0$, or if
  $\mathbf{y} = \mathbf{y}_1^+$, and $k > 0$, or if
  $\mathbf{y} = \mathbf{y}_1\{l, u\}$, and $l \leq k \leq u$, or if
  $\mathbf{y} = \mathbf{y}_1\{l\}$, and $k = l$,
  where $x_i \in \Sigma^+$, and there exist no $\mathbf{y}', \mathbf{y}'' \in \mathcal{Y}_\Sigma$ such that $\mathbf{y}_1 = \mathbf{y}'|\mathbf{y}''$ or $\mathbf{y}_1 = \mathbf{y}'\mathbf{y}''$, we define
  $$M^{\mathbf{y},x}(v) = \begin{cases} \{x\} & \text{, if } v = v_0 \\ \bigcup_{i=1}^{k} M^{\mathbf{y}_1,x_i}(v) & \text{, if } v \in V^{\mathbf{y}_1}_{syn} \end{cases}, \text{ and}$$
  $T^{\mathbf{y},x}_{par} = (\mathbf{y} \to T^{\mathbf{y}_1,x_1}_{par}, \ldots, T^{\mathbf{y}_1,x_k}_{par})$.

- If $\mathbf{y} = \mathbf{y}_1 \ldots \mathbf{y}_k$, $x = x_1 \ldots x_k$,
  where $x_i \in \Sigma^*$, and there exist no $\mathbf{y}', \mathbf{y}'' \in \mathcal{Y}_\Sigma$ such that $\mathbf{y}_j = \mathbf{y}'|\mathbf{y}''$ or $\mathbf{y}_j = \mathbf{y}'\mathbf{y}''$, we define
  $$M^{\mathbf{y},x}(v) = \begin{cases} \{x\} & \text{, if } v = v_0 \\ M^{\mathbf{y}_j,x_i}(v) & \text{, if } v \in V^{\mathbf{y}_j}_{syn} \end{cases}, \text{ and}$$
  $T^{\mathbf{y},x}_{par} = (\mathbf{y} \to T^{\mathbf{y}_1,x_1}_{par}, \ldots, T^{\mathbf{y}_k,x_k}_{par})$.

- If $\mathbf{y} = \mathbf{y}_1| \ldots |\mathbf{y}_k$, $x \in \Sigma^*$
  and there exist no $\mathbf{y}', \mathbf{y}'' \in \mathcal{Y}_\Sigma$ such that $\mathbf{y}_j = \mathbf{y}' \mid \mathbf{y}''$, or if
  $\mathbf{y} = [\mathbf{y}_1 \ldots \mathbf{y}_k]$, $x \in \Sigma^+$
  and there exist no $\mathbf{y}', \mathbf{y}'' \in \mathcal{Y}_\Sigma$ such that $\mathbf{y}_j = \mathbf{y}'\ \mathbf{y}''$, we define
  $$M^{\mathbf{y},x}(v) = \begin{cases} \{x\} & \text{, if } v = v_0 \\ M^{\mathbf{y}_j,x}(v) & \text{, if } v \in V^{\mathbf{y}_j}_{syn} \text{, and} \\ \emptyset & \text{, otherwise} \end{cases}$$
  $T^{\mathbf{y},x}_{par} = (\mathbf{y} \to T^{\mathbf{y}_j,x}_{par})$.

If $x \notin L(\mathbf{y})$, that is, no parse tree can be derived by the specification above, the empty sets $M^{\mathbf{y},x}(v) = \emptyset$ for all $v \in V^{\mathbf{y}}_{syn}$ and $T^{\mathbf{y},x}_{par} = \emptyset$ are returned. Otherwise, we denote the set of all parse trees and the unions of all matching lists for each $v \in V^{\mathbf{y}}_{syn}$ satisfying Definition 5 by $\mathcal{T}^{\mathbf{y},x}_{par}$ and $\mathcal{M}^{\mathbf{y},x}(v)$, respectively. Finally, the matching list $M^{\mathbf{y},\mathbf{x}}(v)$ for a set of strings $\mathbf{x}$ for node $v \in V^{\mathbf{y}}_{syn}$ is defined as $M^{\mathbf{y},\mathbf{x}}(v) = \bigcup_{x \in \mathbf{x}} \mathcal{M}^{\mathbf{y},x}(v)$.

Joint Feature Representations

The list of binary and continuous features $\Psi_{\mathbf{u}}$ used to train model $f_{\mathbf{u}}$ is shown in Table 3. The input and output features $\Psi_{\mathbf{v}}$ for model $f_{\mathbf{v}}$ are shown in Table 4 and 5, respectively. The set S_{spam} is defined as follows: We train a linear classifier that separates spam emails from non-spam emails on the ESP dataset, using a bag of words representation. We construct the set S_{spam} as the 150 words that have the highest weights for the class spam.

Feature	Description
$[\![\varepsilon \in M]\!]$	Matching list contains the empty string?
$[\![\forall \mathbf{x} \in M : \lvert\mathbf{x}\rvert = 1]\!]$	All elements of the matching list have the length one?
$[\![\exists i \in \mathbb{N}, \forall \mathbf{x} \in M : \lvert\mathbf{x}\rvert = i]\!]$	All elements of the matching list have the same length?
$\frac{\lvert\Sigma_M \cap \{A,\ldots,Z\}\rvert}{26}$	Portion of characters A–Z in the matching list
$\frac{\lvert\Sigma_M \cap \{a,\ldots,z\}\rvert}{26}$	Portion of characters a–z in the matching list
$\frac{\lvert\Sigma_M \cap \{0,\ldots,9\}\rvert}{10}$	Portion of characters 0–9 in the matching list
$\frac{\lvert\Sigma_M \cap \{A,\ldots,F\}\rvert}{6}$	Portion of characters A–F in the matching list
$\frac{\lvert\Sigma_M \cap \{a,\ldots,f\}\rvert}{6}$	Portion of characters a–f in the matching list
$\frac{\lvert\Sigma_M \cap \{G,\ldots,Z\}\rvert}{20}$	Portion of characters G–Z in the matching list
$\frac{\lvert\Sigma_M \cap \{g,\ldots,z\}\rvert}{20}$	Portion of characters g–z in the matching list
$[\![\forall x \in \Sigma_M : x \notin \{A,\ldots,Z\}\ ]\!]$	No characters of A–Z in the matching list?
$[\![\forall x \in \Sigma_M : x \notin \{a,\ldots,z\}\ ]\!]$	No characters of a–z in the matching list?
$[\![\forall x \in \Sigma_M : x \notin \{0,\ldots,9\}\ ]\!]$	No characters of 0–9 in the matching list?
$[\![\forall x \in \Sigma_M : x \notin \{a,\ldots,f\}\ ]\!]$	No characters of a–f in the matching list?
$[\![\forall x \in \Sigma_M : x \notin \{A,\ldots,F\}\ ]\!]$	No characters of A–F in the matching list?
$[\![\lvert\Sigma_M \cap \{-,/,?,=,.,@,:\}\rvert > 0]\!]$	Matching list contains URL/Email characters?
$[\![\forall \mathbf{x} \in M : \lvert\mathbf{x}\rvert \geq 1 \wedge \lvert\mathbf{x}\rvert \leq 5]\!]$	Length of strings in the matching list is between 1 and 5?
$[\![\forall \mathbf{x} \in M : \lvert\mathbf{x}\rvert \geq 6 \wedge \lvert\mathbf{x}\rvert \leq 10]\!]$	Length of strings in the matching list is between 5 and 10?
$[\![\forall \mathbf{x} \in M : \lvert\mathbf{x}\rvert \geq 11 \wedge \lvert\mathbf{x}\rvert \leq 20]\!]$	Length of strings in the matching list is between 10 and 20?
$[\![\forall \mathbf{x} \in M : \lvert\mathbf{x}\rvert > 20]\!]$	Length of strings in the matching list is higher than 20?
$[\![\lvert M\rvert = 0]\!]$	Matching list is empty?

Table 3: Features for model $f_{\mathbf{u}}$.

Additional Experimental Results

Figure 10 shows the average loss $\Delta_{\mathbf{u}}$ on the training data as a function of the sample size. The corresponding loss on the test data can be seen in Figure 8 (a).

Feature	**Description**
$[\![0 \leq \text{constant symbols in } \tilde{\mathbf{y}} < 568]\!]$	Number of constant symbols that are arguments of the top-most concatenation is less than 568
$[\![568 \leq \text{constant symbols in } \tilde{\mathbf{y}} < 1032]\!]$	... between 568 and 1031
$[\![1032 \leq \text{constant symbols in } \tilde{\mathbf{y}} < 1724]\!]$	... between 1032 and 1723
$[\![1724 \leq \text{constant symbols in } \tilde{\mathbf{y}} < 2748]\!]$	... between 1724 and 2747
$[\![2748 \leq \text{constant symbols in } \tilde{\mathbf{y}}]\!]$	... 2748 or higher
$[\![0 \leq \text{non-constant arguments in } \tilde{\mathbf{y}} < 48]\!]$	Number of non-constant arguments of the top-level concatenation is less than 48
$[\![48 \leq \text{non-constant arguments in } \tilde{\mathbf{y}} < 77]\!]$	... between 48 and 76
$[\![77 \leq \text{non-constant arguments in } \tilde{\mathbf{y}} < 133]\!]$	... between 77 and 132
$[\![133 \leq \text{non-constant arguments in } \tilde{\mathbf{y}} < 246]\!]$	... between 133 and 245
$[\![246 \leq \text{non-constant arguments in } \tilde{\mathbf{y}}]\!]$	... 246 or higher
$[\![\tilde{\mathbf{y}} \text{ contains Latin characters}]\!]$	
$[\![\tilde{\mathbf{y}} \text{ contains Greek characters}]\!]$	
$[\![\tilde{\mathbf{y}} \text{ contains Russian characters}]\!]$	
$[\![\tilde{\mathbf{y}} \text{ contains Asian characters}]\!]$	
$[\![\tilde{\mathbf{y}} \text{ contains "subject:"}]\!]$	Expression refers to a subject line
$[\![\tilde{\mathbf{y}} \text{ contains "from:"}]\!]$	Refers to a sender address
$[\![\tilde{\mathbf{y}} \text{ contains "to:"}]\!]$	Refers to recipient address
$[\![\tilde{\mathbf{y}} \text{ contains "reply-to:"}]\!]$	Refers to a reply-to address
$[\![\tilde{\mathbf{y}} \text{ contains attachment}]\!]$	Expression refers to an attachment

Table 4: Input features that refer to properties of $\tilde{\mathbf{y}}$ for model $f_{\mathbf{v}}$.

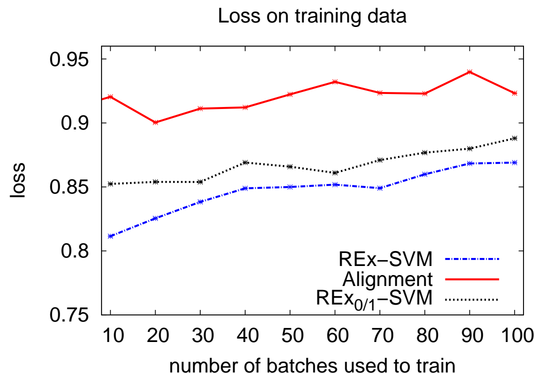


Figure 10: Average loss $\Delta_{\mathbf{u}}$ on training data for a varying number of training batches. Error bars indicate standard errors.

Figure 11 shows how the loss on the test dataset changes when we varying the regularization parameters $C_{\mathbf{u}}$ and $C_{\mathbf{v}}$.

Feature	Description
Constant symbols in $\hat{\mathbf{y}}$	Number of constant symbols in the top-most concatenation
Non-constant subexpressions in $\hat{\mathbf{y}}$	Number of non-constant arguments of the top-most concatenation
$[\![\hat{\mathbf{y}}$ contains Latin characters$]\!]$	
$[\![\hat{\mathbf{y}}$ contains Greek characters$]\!]$	
$[\![\hat{\mathbf{y}}$ contains Russian characters$]\!]$	
$[\![\hat{\mathbf{y}}$ contains Asian characters$]\!]$	
$[\![\hat{\mathbf{y}}$ contains "subject:"$]\!]$	Expression refers to subject line
$[\![\hat{\mathbf{y}}$ contains "from:"$]\!]$	Expression contains a sender address
$[\![\hat{\mathbf{y}}$ contains "to:"$]\!]$	Contains a recipient address
$[\![\hat{\mathbf{y}}$ contains "reply-to:"$]\!]$	Contains a reply-to address
$[\![\hat{\mathbf{y}}$ contains attachment$]\!]$	Expression refers to attachment
$[\![\hat{\mathbf{y}}$ starts with "subject:" and ends with \n$]\!]$	Expression only refers to subject line
$[\![\hat{\mathbf{y}}$ starts with "from:" and ends with \n$]\!]$	Expression only refers to sender address
$[\![\hat{\mathbf{y}}$ starts with "to:" and ends with \n$]\!]$	Expression only refers to recipient address
$[\![\hat{\mathbf{y}}$ starts with "reply-to:" and ends with \n$]\!]$	Only refers to reply-to address
$[\![\hat{\mathbf{y}}$ starts with "attachment:" and ends with \n$]\!]$	Contains only refers to attachment
$[\![\hat{\mathbf{y}}$ starts with "subject:"$]\!]$	Expression starts with a subject line
$[\![\hat{\mathbf{y}}$ starts with "from:"$]\!]$	Starts with a sender address
$[\![\hat{\mathbf{y}}$ starts with "to:"$]\!]$	Starts with a recipient address
$[\![\hat{\mathbf{y}}$ starts with "reply-to:"$]\!]$	Starts with a reply-to address
$[\![\hat{\mathbf{y}}$ starts with "attachment:"$]\!]$	Starts with a subject line
$[\![\hat{\mathbf{y}}$ ends with "subject:"$]\!]$	Ends with a subject line
$[\![\hat{\mathbf{y}}$ ends with "from:"$]\!]$	Ends with a sender address
$[\![\hat{\mathbf{y}}$ ends with "to:"$]\!]$	Ends with a recipient address
$[\![\hat{\mathbf{y}}$ ends with "reply-to:"$]\!]$	Ends with a reply-to address
$[\![\hat{\mathbf{y}}$ ends with "attachment:"$]\!]$	Ends with reference to attachment
number of newlines in $\hat{\mathbf{y}}$	Number of line breaks in the expression
$[\![\hat{\mathbf{y}}$ contains a URL$]\!]$	
$[\![\hat{\mathbf{y}}$ is only a URL$]\!]$	
$[\![\hat{\mathbf{y}}$ contains an email address$]\!]$	
$[\![\hat{\mathbf{y}}$ is only an email address$]\!]$	
$[\![\hat{\mathbf{y}}$ contains a phone number$]\!]$	
$[\![\hat{\mathbf{y}}$ is only a phone number$]\!]$	
$[\![\hat{\mathbf{y}}$ contains an IP address$]\!]$	
$[\![\hat{\mathbf{y}}$ contains an attachment of type .exe$]\!]$	
$[\![\hat{\mathbf{y}}$ contains an attachment of type .jpg$]\!]$	
$[\![\hat{\mathbf{y}}$ contains an attachment of type .zip$]\!]$	
$[\![\hat{\mathbf{y}}$ contains an attachment of type .html$]\!]$	
$[\![\hat{\mathbf{y}}$ contains an attachment of type .doc$]\!]$	
$[\![\hat{\mathbf{y}}$ contains substring $\in S_{spam}]\!]$	Contains terms from the highest-scoring bag-of-words features for spam

Table 5: Output features that refer to properties of $\hat{\mathbf{y}}$ for model $f_{\mathbf{v}}$.

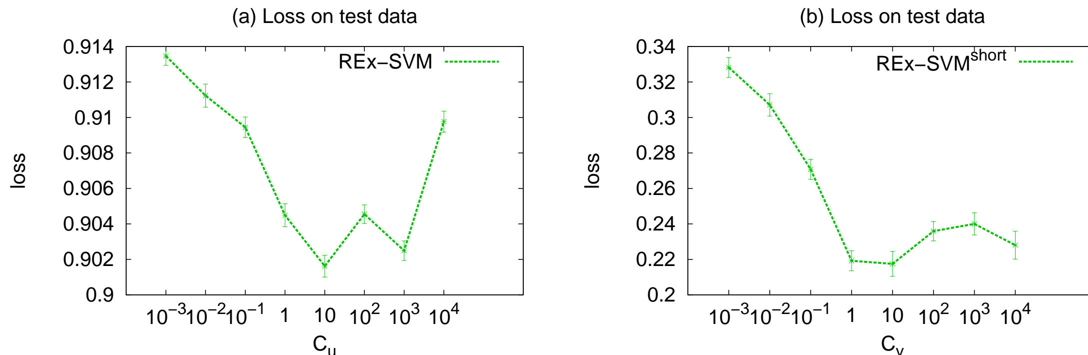


Figure 11: Average loss on test data for a varying regularization parameters $C_\mathbf{u}$ and $C_\mathbf{v}$ to train a model $\mathbf{f_u}$ (a) and a model $\mathbf{f_v}$, respectively. Error bars indicate standard errors.

Syntax of Postmasters' Regular Expressions

This section summarizes the syntactic constructs used by postmasters and their frequency. These observations provide the rationale behind the definition of the constrained search space of Algorithm 1. Table 6 shows the frequency at which macros occur in the ESP data set. Table 7 shows which iterators (*, $^+$, ?, $\{x\}$, $\{x,y\}$ for $x, y \in \mathbb{N}$) postmasters use as a suffix of the disjunction of characters (*e.g.,* [abc]* or [0-9]$^+$). Table 8 counts the frequency of iterators in conjunction with an alternative of regular expressions (*e.g.,* (girl|woman)?).

Macro	Frequency
\d	97
\S	71
\e	16
A-Z	25
a-z	86
A-F	28
a-f	17
0-9	65

Table 6: Macros used in the postmasters' expressions.

We measure the maximum nesting depth of alternatives of regular expression in the ESP data set: We find that 95.6% have a nesting depth of at most one—that is, they contain no layer of alternatives within the top-most alternative, such as a[a-z]$^+$. Only 4.4% have a greatrer nesting depth (e.g. a([a-z]$^+$|01), having a nesting depth of two). Algorithm 1 constructs the set of possible specializations of the j-th wildcard, starting with all subexpressions of all expressions in the training data. Hence, the nesting depth

Iterator	Frequency
$[\dots]$	21
$[\dots]^*$	2
$[\dots]^+$	73
$[\dots]?$	0
$[\dots]\{x\}$	49
$[\dots]\{x,y\}$	39

Table 7: Iterators used in conjunction with a character disjunction—*e.g.,* [abc0-9]*.

Iterator	Frequency
$(\dots\|\dots)$	166
$(\dots\|\dots)^*$	0
$(\dots\|\dots)^+$	0
$(\dots\|\dots)?$	2
$(\dots\|\dots)\{x\}$	0
$(\dots\|\dots)\{x,y\}$	0

Table 8: Iterators used in conjunction with alternatives—*e.g.,* (viagra|cialis)$^+$.

of alternatives in the constrained search space is at least the nesting depth of the training data. In line 6, the alternative of constant strings aligned at the j-th wildcard symbol is added; hence, the constrained search space has a nesting depth of at least one, even if the training data have a nesting depth of zero. For all character alternatives in the set of possible specializations, all macros from Table 6 and all iterators shown in Tables 7 and 8 are added.