Institut für Informatik
Professur Maschinelles Lernen

Pattern Recognition for Computer Security: Discriminative Models for Email Spam Campaign and Malware Detection

Kumulative Dissertation

zur Erlangung des akademischen Grades
"doctor rerum naturalium"
(Dr. rer. nat.)
in der Wissenschaftsdisziplin Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Paul Prasse

Potsdam, den 27.05.2016

Abstract

Computer Security deals with the detection and mitigation of threats to computer networks, data, and computing hardware. This thesis addresses the following two computer security problems: email spam campaign and malware detection. Email spam campaigns can easily be generated using popular dissemination tools 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 spam campaigns can encompass huge data volumes and therefore pose a threat to the stability of the infrastructure of email service providers that have to store them. Malware —software that serves a malicious purpose— is affecting web servers, client computers via active content, and client computers through executable files. Without the help of malware detection systems it would be easy for malware creators to collect sensitive information or to infiltrate computers.

The detection of threats—such as email-spam messages, phishing messages, or malware—is an adversarial and therefore intrinsically difficult problem. Threats vary greatly and evolve over time. The detection of threats based on manually-designed rules is therefore difficult and requires a constant engineering effort. Machine-learning is a research area that revolves around the analysis of data and the discovery of patterns that describe aspects of the data. Discriminative learning methods extract prediction models from data that are optimized to predict a target attribute as accurately as possible. Machine-learning methods hold the promise of automatically identifying patterns that robustly and accurately detect threats. This thesis focuses on the design and analysis of discriminative learning methods for the two computer-security problems under investigation: email-campaign and malware detection.

The first part of this thesis addresses email-campaign detection. We focus on regular expressions as a syntactic framework, because regular expressions are intuitively comprehensible by security engineers and administrators, and they can be applied as a detection mechanism in an extremely efficient manner. In this setting, a prediction model is provided with exemplary messages from an email-spam campaign. The prediction model has to generate a regular expression that reveals the syntactic pattern that underlies the entire campaign, and that a security engineers finds comprehensible and feels confident enough to use the expression to blacklist further messages at the email server. We model this problem as two-stage learning problem with structured input and output spaces which can be solved using standard cutting plane methods. Therefore we develop an appropriate loss function, and derive a decoder for the resulting optimization problem.

The second part of this thesis deals with the problem of predicting whether a given JavaScript or PHP file is malicious or benign. Recent malware analysis techniques use static or dynamic features, or both. In fully dynamic analysis, the software or script is executed and observed for malicious behavior in a sandbox environment. By contrast, static analysis is based on features that can be extracted directly from the program file. In order to bypass static detection mechanisms, code obfuscation techniques are used to spread a malicious program file in many different syntactic variants. Deobfuscating the code before applying a static classifier can be subjected to mostly static code analysis and can overcome the problem of obfuscated malicious code, but on the other hand increases the computational costs of malware detection by an order of magnitude. In this thesis we present a cascaded architecture in which a classifier first performs a static analysis of the original code and—based on the outcome of this first classification step—the code may be deobfuscated and classified again. We explore several types of features including token n-grams, orthogonal sparse bigrams, subroutine-hashings, and syntax-tree features and study the robustness of detection methods and feature types against the evolution of malware over time. The developed tool scans very large file collections quickly and accurately.

Each model is evaluated on real-world data and compared to reference methods. Our approach of inferring regular expressions to filter emails belonging to an email spam campaigns leads to models with a high true-positive rate at a very low false-positive rate that is an order of magnitude lower than that of a commercial content-based filter. Our presented system —REx-SVM$^{\texttt{short}}$— is being used by a commercial email service provider and complements content-based and IP-address based filtering. Our cascaded malware detection system is evaluated on a high-quality data set of almost 400,000 conspicuous PHP files and a collection of more than 1,00,000 JavaScript files. From our case study we can conclude that our system can quickly and accurately process large data collections at a low false-positive rate.

Zusammenfassung

Computer-Sicherheit beschäftigt sich mit der Erkennung und der Abwehr von Bedrohungen für Computer-Netze, Daten und Computer-Hardware. In dieser Dissertation wird die Leistungsfähigkeit von Modellen des maschinellen Lernens zur Erkennung von Bedrohungen anhand von zwei konkreten Fallstudien analysiert. Im ersten Szenario wird die Leistungsfähigkeit von Modellen zur Erkennung von Email Spam-Kampagnen untersucht. E-Mail Spam-Kampagnen werden häufig von leicht zu bedienenden Tools erzeugt. Diese Tools erlauben es dem Benutzer, mit Hilfe eines Templates (z.B. einer regulären Grammatik) eine Emailvorlage zu definieren. Ein solches Template kann z.B. auf die Knoten eines Botnetzes verteilt werden. Dort werden Nachrichten mit diesem Template generiert und an verschiedene Absender verschickt. Die damit entstandenen E-Mail Spam-Kampagnen können riesige Datenmengen produzieren und somit zu einer Gefahr für die Stabilität der Infrastruktur von E-Mail-Service-Providern werden. Im zweiten Szenario wird die Leistungsfähigkeit von Modellen zur Erkennung von Malware untersucht. Malware bzw. Software, die schadhaften Programmcode enthält, kann Web-Server und Client-Computer über aktive Inhalte und Client-Computer über ausführbare Dateien beeinflussen. Somit kann die die reguläre und legitime Nutzung von Diensten verhindert werden. Des Weiteren kann Malware genutzt werden, um sensible Informationen zu sammeln oder Computer zu infiltrieren.

Die Erkennung von Bedrohungen, die von E-Mail-Spam-Mails, Phishing-E-Mails oder Malware ausgehen, gestaltet sich schwierig. Zum einen verändern sich Bedrohungen von Zeit zu Zeit, zum anderen werden E-Mail-Spam-Mails oder Malware so modifiziert, dass sie von aktuellen Erkennungssystemen nicht oder nur schwer zu erkennen sind. Erkennungssysteme, die auf manuell erstellten Regeln basieren, sind deshalb wenig effektiv, da sie ständig administriert werden müssen. Sie müssen kontinuierlich gewartet werden, um neue Regeln (für veränderte oder neu auftretende Bedrohungen) zu erstellen und alte Regeln anzupassen bzw. zu löschen. Maschinelles Lernen ist ein Forschungsgebiet, das sich mit der Analyse von Daten und der Erkennung von Mustern beschäftigt, um bestimmte Aspekte in Daten, wie beispielsweise die Charakteristika von Malware, zu beschreiben. Mit Hilfe der Methoden des Maschinellen Lernens ist es möglich, automatisiert Muster in Daten zu erkennen. Diese Muster können genutzt werden, um Bedrohung gezielt und genau zu erkennen.

Im ersten Teil wird ein Modell zur automatischen Erkennung von E-Mail-Spam-Kampagnen vorgestellt. Wir verwenden reguläre Ausdrücke als syntaktischen Rahmen, um E-Mail-Spam-Kampagnen zu beschreiben und E-Mails die zu einer E-Mail-Spam-Kampagne gehören zu identifizieren. Reguläre Ausdrücke sind intuitiv verständlich und können einfach von Administratoren genutzt werden, um E-Mail-Spam-Kampagnen zu beschreiben. Diese Arbeit stellt ein Modell vor, das für eine gegebene E-Mail-Spam-Kampagne einen regulären Ausdruck vorhersagt. In dieser Arbeit stellen wir ein Verfahren vor, um ein Modell zu bestimmen, das reguläre Ausdrücke vorhersagt, die zum Einen die Gesamtheit aller E-Mails in einer Spam-Kampagne abbilden und zum Anderen so verständlich aufgebaut sind, dass ein Systemadministrator eines E-Mail Servers diesen verwendet. Diese Problemstellung wird als ein zweistufiges Lernproblem mit strukturierten Ein- und Ausgaberäumen modelliert, welches mit Standardmethoden des Maschinellen Lernens gelöst werden kann. Hierzu werden eine geeignete Verlustfunktion, sowie ein Dekodierer für das resultierende Optimierungsproblem hergeleitet.

Der zweite Teil behandelt die Analyse von Modellen zur Erkennung von Java-Script oder PHP-Dateien mit schadhaften Code. Viele neu entwickelte Malwareanalyse-Tools nutzen statische, dynamische oder eine Mischung beider Merkmalsarten als Eingabe, um Modelle zur Erkennung von Malware zu bilden. Um dynamische Merkmale zu extrahieren, wird eine Software oder ein Teil des Programmcodes in einer gesicherten Umgebung ausgeführt und das Verhalten (z.B. Speicherzugriffe oder Funktionsaufrufe) analysiert. Bei der statischen Analyse von Skripten und Software werden Merkmale direkt aus dem Programcode extrahiert. Um Erkennungsmechanismen, die nur auf statischen Merkmalen basieren, zu umgehen, wird der Programmcode oft maskiert. Die Maskierung von Programmcode wird genutzt, um einen bestimmten schadhaften Programmcode in vielen syntaktisch unterschiedlichen Varianten zu erzeugen. Der originale schadhafte Programmcode wird dabei erst zur Laufzeit generiert. Wird der Programmcode vor dem Anwenden eines Vorhersagemodells demaskiert, spricht man von einer vorwiegend statischen Programmcodeanalyse. Diese hat den Vorteil, dass enthaltener Schadcode einfacher zu erkennen ist. Großer Nachteil dieses Ansatzes ist die erhöhte Laufzeit durch das Demaskieren der einzelnen Dateien vor der Anwendung des Vorhersagemodells. In dieser Arbeit wird eine mehrstufige Architektur präsentiert, in der ein Klassifikator zunächst eine Vorhersage auf Grundlage einer statischen Analyse auf dem originalen Programmcode trifft. Basierend auf dieser Vorhersage wird der Programcode in einem zweiten Schritt demaskiert und erneut ein Vorhersagemodell angewendet. Wir betrachten dabei eine Vielzahl von möglichen Merkmalstypen, wie n-gram Merkmale, orthogonal sparse bigrams, Funktions-Hashes und Syntaxbaum Merkmale. Zudem wird in dieser Dissertation untersucht, wie robust die entwickelten Erkennungsmodelle gegenüber Veränderungen von Malware über die Zeit sind. Das vorgestellte Verfahren ermöglicht es, große Datenmengen mit hoher Treffergenauigkeit nach Malware zu durchsuchen.

Alle in dieser Dissertation vorgestellten Modelle wurden auf echten Daten evaluiert und mit Referenzmethoden verglichen. Das vorgestellte Modell zur Erkennung von E-Mail-Spam-Kampagnen hat eine hohe richtig-positive Rate und eine sehr kleine falsch-positiv Rate die niedriger ist, als die eines kommerziellen E-Mail-Filters. Das Modell wird von einem kommerziellen E-Mail Service Provider während des operativen Geschäfts genutzt, um eingehende und ausgehende E-Mails eines E-Mails-Servers zu überprüfen. Der Ansatz zur Malwareerkennung wurde auf einem Datensatz mit rund 400.000 verdächtigen PHP Dateien und einer Sammlung von mehr als 1.000.000 Java-Script Dateien evaluiert. Die Fallstudie auf diesen Daten zeigt, dass das vorgestellte System schnell und mit hoher Genauigkeit riesige Datenmengen mit wenigen Falsch-Alarmen nach Malware durchsuchen kann.

Contents

Chapter 1

Introduction

The manual design of patterns that detect computer security threats requires a constant and substantial engineering effort. Machine-learning methods offer the potential to discover such patterns automatically, based on the analysis of data (Chan and Lippmann, 2006). Example computer-security problems for which machine learning has been applied include email spam classification that classifies ingoing emails as spam or non-spam or malware detection that classifies executable or program files as malicious or benign. In this thesis we investigate the predictive power of discriminative models for two sample applications: email spam campaign and malware detection.

1.1 Pattern Recognition for Computer Security

Recently, machine-learning techniques are used to develop computer security systems for many different applications ranging from intrusion and network anomaly detection, email spam or malware detection to detecting of Denial-of-Service (DOS) or Distributed DOS attacks (DDoS).

Intrusion detection systems using machine-learning techniques can be divided into two categories: supervised (classification) and unsupervised (anomaly detection and clustering) methods (Laskov et al., 2005). Intrusion detection systems try to detect unauthorized use, misuse, and abuse of computer systems. In Laskov et al. (2005) common supervised and unsupervised machine-learning algorithms are compared to each other to detect network intrusions based on a fixed set of connection-based features. In Lane and Brodley (1997) a user profile based on sequences of actions (UNIX commands) is learned to detect the presence of an intruder masquerading as the valid user. In Zanero and Savaresi (2004) a two-tier architecture to detect network intrusions is proposed: the first tier is an unsupervised clustering algorithm which reduces the network packets payload to a tractable size and the second tier is a traditional anomaly detection algorithm, whose efficiency is improved by the availability of data on the packet payload content. In Lee and Stolfo (1998) machine-learning techniques are used to discover consistent and useful patterns of system features that describe program and user behavior. The set of relevant system features is being used to learn classifiers that can recognize anomalies and known intrusions. In Lee and Stolfo (2000) a data mining framework for adaptively building intrusion detection models is proposed.

Denial-of-Service (DoS) or Distributed Denial-of-Service (DDoS) attacks are a serious problem for many web services. DDoS attacks aim that users are prevented from using a web-based service by exhausting server or network resources. Known defense mechanisms against DDoS attacks use features that aggregate behavioral information about individual clients over time (Ranjan et al., 2006; Xie and Yu, 2009; Liu and Chang, 2011). In Xie and Yu (2009) the HTTP request sequences of users are analyzed. Fitting a sequence of a user to the model of normal browsing behavior, it is possible to detect malicious network patterns (users).

Email spam messages impose a substantial burden on network and storage capacities, and are frequently used to distribute viruses, obtain personal information, and promote illegal businesses. Using textual features such as n-grams or orthogonal sparse bigrams a statistical model can be learned to distinguish between legitimate and spam emails at a high true-positive and a low false-positive rate. Email spam campaigns can be observed using spam traps containing many spam emails. Haider and Scheffer (2009) presented a Bayesian clustering approach that can be used to cluster emails into sets of email spam campaigns. This method transforms emails into a feature space of independent binary feature vectors that can reflect arbitrary dependencies within emails in the input space. Using this clustering approach, entire spam and phishing campaigns can be detected reliable. To detect future emails belonging to a known email spam campaign a binary indicator vector (bag-of-words) containing all words shared among all emails in the cluster can be used. This feature vector has a component for each word in the vocabulary and serves as cluster description.

Malware is affecting web servers, client computers via active web content, and client computers through executable files. Malware can be used to steal information or spy on computer users for an extended period without their knowledge. Existing malware detection algorithms use features extracted directly from the program code of a software or script (static classifier) or features obtained in a sandbox environment (dynamic classifier) (Kolter and Maloof, 2006; Canali et al., 2012; Curtsinger et al., 2011; Anderson et al., 2012; Christodorescu et al., 2005; Kong and Yan, 2013). Where the dynamic classifiers suffer from a high execution time the purely static classifiers suffer from a lower predictive power compared to dynamic classification approaches. Malicious software can also be detected using algorithms that analyze the network traffic a software causes (Franc et al., 2015; Bartos and Sofka, 2015). This approach addresses the problem of learning a detector for malicious network traffic. The malicious behavior is detected based on the analysis of network proxy logs that capture malware communication between client and server computers.

All computer security applications mentioned above use training data to model the relationship between an instance and the target label. Most models assume that the given training data consists of representative samples which do not differ from the test data which the model is applied to during application time. For some applications this assumption is not fulfilled which can be the case when the test data depends on the predictive model trained on prior training data, which is referred to as adversarial prediction problem. Brückner et al. (2012) formulated this problem as a prediction game with two players: the learner and the data generator. The learner infers a predictive model from the training data and the data generator controls the data generation process. The goal is to find the optimal model of the learner which is most robust to all expected actions of the data generator. The distribution of the training and test data can also differ from each other when they are sampled from different time periods. So typically malware and email spam evolve over time and sampling the training and test data according to the time they firstly appeared would lead to a difference between the training and test distribution. This is especially the case, when a predictive model is trained in the past and evaluated into the future.

1.2 Discriminative Models for Email Spam Campaign Detection

Methods for email spam detection typically model the problem setting as follows: Given the input x which is the content of an email, we want to find a mapping for all possible emails $x \in \mathcal{X}$ to the binary target label $y \in \{+1, -1\}$, representing spam and non-spam. Most previously presented works use a statistical model learned on a feature representation of emails x. Textual patterns are commonly extracted using n-grams or orthogonal sparse bigrams which are transformed into a vector which contains the frequency of each pattern for the resulting vocabulary of known patterns. In the application of email spam filtering a very low false-positive rate must be achieved so that a model can be deployed.

Popular spam dissemination tools are extensively used to spread mailing campaigns by specifying simple grammars that serve as message templates. The grammar is disseminated to nodes of a bot net, where the nodes create messages by instantiating the grammar at random. Samples of such mailing campaigns can be collected by an email service provider using a previously trained spam filter on their ingoing emails. Typically messages from multiple campaigns are collected jointly using a spam filter applied to all ingoing emails. Clustering tools can be used to separate the campaigns reliably (Haider and Scheffer, 2009). Using a probabilistic cluster description it is possible to assign new incoming emails to a known email campaign. Such a probabilistic cluster description can consist of a bag-of-word representation of tokens shared among all emails of a campaign and a threshold determining how many words an email assigned to this cluster have to share with the cluster. Unfortunately, such a probabilistic cluster description can cause false-positives and the bag-of-word representation makes it sometimes very hard to figure out what kind of emails would be assigned to a specific cluster. To tackle this problem postmasters of an email service provider (ESP) write specific, comprehensible regular expressions which cover the observed instances and are used to blacklist the bulk of emails of that campaign at virtually no risk of covering other messages. 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. Every regular expression $\mathbf{y}$ defines a regular language $L(\mathbf{y})$ containing all strings $x \in \Sigma^*$ which can be generated using expression $\mathbf{y}$.

In this thesis a two-staged learning problem with structured output spaces and an appropriat loss functions is presented. In a first step we learn a mapping from a given set of strings (an email campaign) to a regular expression. In a second step we infer a mapping from a regular expression (inferred by the first stage) to a substring of the initially given regular expression which is most informative.

1.3 Predicting Regular Expressions for Message Campaigns

The first paper presented in this thesis addresses the problem of inferring a regular expression from a given set of strings that resemble the regular expression that a human expert would have written to identify the language, as closely as possible. We are given a set of emails $x \in \mathbf{x}$ belonging to the same spam campaign $\mathbf{x}$. The goal is to infer a regular expression $\mathbf{y}$ that identifies the campaign template and is similar to a regular expression a human postmaster would use to blacklist this specific campaign $\mathbf{x}$. More formally, we want to generate a regular expression $\mathbf{y}$ so that all emails $x \in \mathbf{x}$ are elements of the language $L(\mathbf{y})$. To model this task we assume that an unknown distribution $p(\mathbf{x}, \mathbf{y})$ generates regular expressions $\mathbf{y}$ for sets of strings $x \in \mathbf{x}$ that are elements of the language $L(\mathbf{y})$. We formulate this problem as a learning problem with structured output spaces, a joint feature representation, and an appropriate loss function. The goal is to learn a $\mathbf{w}$-parametrized discriminative predictive model $f_{\mathbf{w} \mapsto \hat{\mathbf{y}}}$ that accepts a set of strings $\mathbf{x}$ and infers a regular expression $\hat{\mathbf{y}}$. To measure the quality of an inferred regular expression we defined a loss function $\Delta(\mathbf{y}, \hat{\mathbf{y}}, \mathbf{x})$ that captures the deviation of the predicted expression $\hat{\mathbf{y}}$ from $\mathbf{y}$ for campaign $\mathbf{x}$ by comparing each of the accepting parse trees for $\mathbf{y}$ and $\mathbf{x}$ when generating the strings from $\mathbf{x}$. This loss function is formulated in a way that syntactically more similar regular expressions would gain a lower loss than syntactically more dissimilar regular expressions. This is motivated by the use-case of a human postmaster that would only use regular expressions that are written comprehensibly and neatly.

Formally, given the loss function, our goal is to find the model $f_{\mathbf{w}}$ with minimal risk:

$$R[f_{\mathbf{w}}] = \iint \Delta(\mathbf{y}, f_{\mathbf{w}}(\mathbf{x}), \mathbf{x}) p(\mathbf{x}, \mathbf{y}) \mathrm{d}\mathbf{x}\, \mathrm{d}\mathbf{y},$$

where the training data $D = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{m}$ consists of pairs of batches $\mathbf{x}_i$ and generating regular expressions $\mathbf{y}_i$, drawn according to $p(\mathbf{x}, \mathbf{y})$.

This approach is first presented and discussed by Prasse, Sawade, Landwehr, and Scheffer (Prasse et al., 2012). That paper presents a learning framework which can be used to infer regular expressions to describe batches of strings by learning a model which captures the preferences in writing regular expressions from a training data set consisting of pairs of batches and their corresponding regular expressions.

1.4 Extracting Concise Regular Expressions

The second paper presented in the thesis addresses the problem of inferring a concise substring of a given regular expression which is most informative and can be used to blacklist email spam campaigns (Prasse et al., 2015). This results in shorter, easier to interpret, and more general regular expressions which are more similar to regular expressions a human postmaster would write. The model presented in Prasse et al. (2012) generates regular expressions that tend to be very specific because they match all the parts of the strings in a campaign $\mathbf{x}$. Human postmasters, by contrast, prefer to focus on only a characteristic part of the messages for which they would write a concise regular expression. So we are interested in a model $f_{\mathbf{v}}$ that selects a substring from its input regular expression. In Prasse et al. (2015) we model $f_{\mathbf{v}}$ as a linear discriminant function with a joint feature representation of the input regular expression and the output regular expression. Training data to train this model $f_{\mathbf{v}}$ consists of pairs $(\tilde{\mathbf{y}}_i, \mathbf{y}_i)$, whereas $\mathbf{y}_i$ is a concise substring of the regular expression $\tilde{\mathbf{y}}_i$.

This approach is first presented and discussed by Prasse et al. (2015). The paper Prasse et al. (2015) extends the conference paper Prasse et al. (2012) that addresses the problem of inferring a regular expression for a given set of strings. In this paper an approach to infer the most informative and concise subexpression for a previously mostly long regular expression is presented.

1.5 Detecting Malware

In the application of malware detection the inputs x are JavaScript, PHP, PDF, or executable files. The label that is to be inferred is a binary target variable, $y \in \{+1, -1\}$, representing malicious or benign content. Most malware analysis techniques are using static or dynamic features or both to decide whether a given file contains malicious content or not.

In a fully dynamic analysis the software or the script is executed and analyzed for malicious patterns in a sandbox environment. This thesis addresses the problem of malware detection that have to scan large file collections in a quick and accurate way. Setting up a virtual operating environment and running the software or script for every instance would be impractical because this requires minutes of CPU time and computer resources for each file to inspect. By contrast, static analysis is based on features that can be extracted directly from the program file. Regarding binary files, byte n-gram features, DLLs, and system calls can be extracted. Furthermore, a Gabor-filter representation of a gray scale visualization can be used to represent a binary file (Nataraj et al., 2011). Regarding script files all kinds of textual patterns like n-gram features (Kolter and Maloof, 2006), bag of tuples (Canali et al., 2012), syntax-tree features (Curtsinger et al., 2011), control-flow graph features (Anderson et al., 2012; Christodorescu et al., 2005), and function-call graph features (Kong and Yan, 2013) can be used. Such analysis tools can be real-time capable and be trained on collections of malware and benign files. Static and dynamic program analysis can also be combined to use all kinds of features to improve the predictive power.

In order to bypass static detection mechanisms code obfuscation techniques are used heavily. Code obfuscation techniques allow malware engineers to package malicious content into patterns that may not have previously occurred in malware. To overcome code obfuscation, deobfuscation tools were designed to allow partial code execution to the point where all dynamically generated code have been generated. Deobfuscating all files before performing the static malware analysis may still be impractical when billions of files should be managed in a quickly way. To deal with large file collections we study a cascaded malware detection framework that bases its malware detections on static code features whenever this decision can be made with near certainty. Otherwise the code is being deobfuscated and a decision is being made based on static features of the unpacked code.

This approach is first detailed and discussed by Prasse and Scheffer (2016). This paper presents a fast and robust cascaded malware detection framework.

1.6 Contributions

In this thesis, we present new machine-learning techniques for two computer security applications: email spam campaign and malware detection. In this section, a list of all presented papers together with general and own contributions is provided.

Prasse et al. (2012)	Prasse, P., Sawade, C., Landwehr, N., Scheffer, T.: Learning to Identify Regular Expressions that Describe Email Campaigns. In: Proceedings of the 29th International Conference on Machine Learning (2012)

That paper contributes a learning framework with structured output spaces to learn the mapping from a set of strings to a target regular expression. Furthermore, it presents a loss function to syntactically compare different regular expressions and a joint feature representation for sets of strings and regular expressions. To evaluate the performance of the proposed model and its baselines we conducted a case study with training and test data from an email service provider.
For that paper, I formulated the problem setting and the formal definitions, designed the joint feature representation and the loss function, derived the decoder, implemented the algorithm and its baselines, and conducted the experiments.

Prasse et al. (2015)	Prasse, P., Sawade, C., Landwehr, N., Scheffer, T.: Learning to Identify Concise Regular Expressions that Describe Email Campaigns. In: Journal of Machine Learning Research 16 (2015)

That paper extends the conference paper Prasse et al. (2012), includes a more detailed analysis and contributes a two-staged learning problem for inferring concise regular expressions with structured output spaces and appropriate loss functions. In this paper we derive decoders, joint feature representations and loss functions for the resulting optimization problems which can be solved using standard cutting plane methods. To evaluate the performance of the proposed model we conducted a case study on real world data collected by an email service provider.
For this paper I formulated the problem setting and the formal definitions, proved the observation and the theorem, designed the loss function, derived the decoders, implemented the algorithm and its baselines, and conducted the experiments.

Prasse and Scheffer (2016)	Prasse, P., Scheffer, T.: Cascaded Malware Detection at Scale. In: unpublished manuscript (2016)

This paper addresses the problem of detecting malicious JavaScript and PHP scripts and makes the following contributions: Firstly, we presented a cascaded malware detection framework in which most decisions are made based on fully static analysis and only a limited fraction of files are singled out for deobfuscation. Secondly, we explored this approach in a large-scale empirical study including about 400,000 PHP files and 1,000,000 JavaScript files. Furthermore, we made our JavaScript data set publicly available to other researches so that they could compare their methods with our approach. In our empirical study we investigated the effectiveness of several feature types used for malware classification. Feature types we analyzed included n-grams, orthogonal sparse bigrams, syntax-tree features, and function-hashing features. To the best of our knowledge, we presented the first empirical study of the robustness of malware detection models over time. Finally, we presented a fast and robust cascaded malware-filtering system that attains highest accuracies for JavaScript and PHP files at a very low false-positive rate.
For the paper I collected the training data, designed the cascaded malware detection model, defined the set of features used in the model, implemented the algorithms and its baselines, and conducted the experiments on the data sets.

Learning to Identify Regular Expressions that Describe Email Campaigns

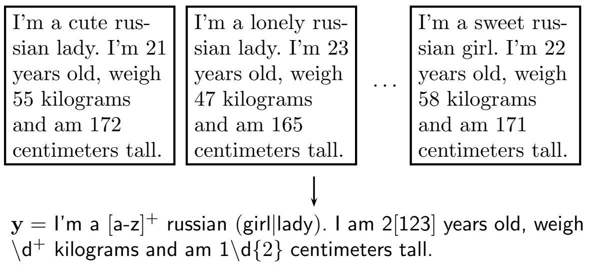
Paul Prasse PRASSE@CS.UNI-POTSDAM.DE
Christoph Sawade SAWADE@CS.UNI-POTSDAM.DE
Niels Landwehr LANDWEHR@CS.UNI-POTSDAM.DE
Tobias Scheffer SCHEFFER@CS.UNI-POTSDAM.DE
University of Potsdam, Department of Computer Science, August-Bebel-Strasse 89, 14482 Potsdam, Germany

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 of an email service 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 learning problem with structured output spaces and an appropriate loss function, derive a decoder and the resulting optimization problem, and a report on a case study conducted with an email service.

1. Introduction

Popular spam dissemination 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 & 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.


I'm a cute russian lady. I'm 21 years old, weigh 55 kilograms and am 172 centimeters tall.

I'm a lonely russian lady. I'm 23 years old, weigh 47 kilograms and am 165 centimeters tall.

...

I'm a sweet russian girl. I'm 22 years old, weigh 58 kilograms and am 171 centimeters tall.

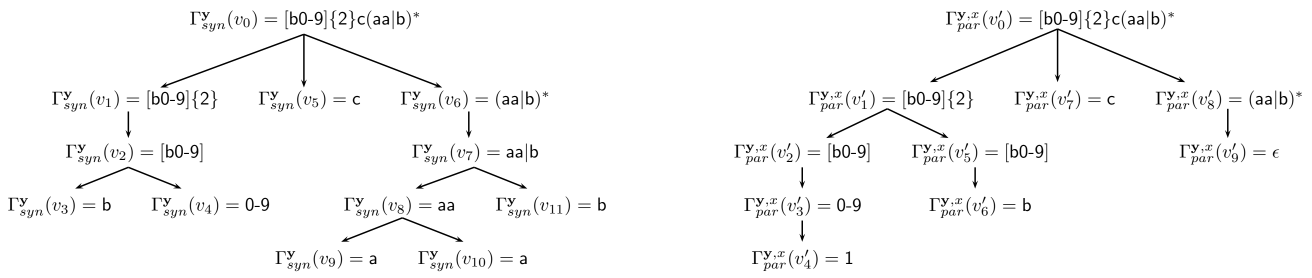
$\mathbf{y}$ = I'm a [a-z]$^+$ russian (girl|lady). I am 2[123] years old, weigh \d$^+$ kilograms and am 1\d{2} centimeters tall.

Figure 1. Elements of a message campaign and a regular expression created by a postmaster.

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 finite state machine that accepts the language and has an execution time linear in the length of the input string. A specific, comprehensible regular expression which covers the observed instances and has been 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.

Language identification has a rich history in the algorithmic learning theory community (see Section 6). Our problem setting differs from the 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. 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 (see Figure 1).

Appearing in *Proceedings of the 29th International Conference on Machine Learning*, Edinburgh, Scotland, UK, 2012. Copyright 2012 by the author(s)/owner(s).


$\Gamma^{\mathbf{y}}_{syn}(v_0) = \text{[b0-9]\{2\}c(aa|b)*}$
$\Gamma^{\mathbf{y}}_{syn}(v_1) = \text{[b0-9]\{2\}}$
$\Gamma^{\mathbf{y}}_{syn}(v_5) = \text{c}$
$\Gamma^{\mathbf{y}}_{syn}(v_6) = \text{(aa|b)*}$
$\Gamma^{\mathbf{y}}_{syn}(v_2) = \text{[b0-9]}$
$\Gamma^{\mathbf{y}}_{syn}(v_7) = \text{aa|b}$
$\Gamma^{\mathbf{y}}_{syn}(v_3) = \text{b}$
$\Gamma^{\mathbf{y}}_{syn}(v_4) = \text{0-9}$
$\Gamma^{\mathbf{y}}_{syn}(v_8) = \text{aa}$
$\Gamma^{\mathbf{y}}_{syn}(v_{11}) = \text{b}$
$\Gamma^{\mathbf{y}}_{syn}(v_9) = \text{a}$
$\Gamma^{\mathbf{y}}_{syn}(v_{10}) = \text{a}$
$\Gamma^{\mathbf{y},x}_{par}(v'_0) = \text{[b0-9]\{2\}c(aa|b)*}$
$\Gamma^{\mathbf{y},x}_{par}(v'_1) = \text{[b0-9]\{2\}}$
$\Gamma^{\mathbf{y},x}_{par}(v'_7) = \text{c}$
$\Gamma^{\mathbf{y},x}_{par}(v'_8) = \text{(aa|b)*}$
$\Gamma^{\mathbf{y},x}_{par}(v'_2) = \text{[b0-9]}$
$\Gamma^{\mathbf{y},x}_{par}(v'_5) = \text{[b0-9]}$
$\Gamma^{\mathbf{y},x}_{par}(v'_9) = \epsilon$
$\Gamma^{\mathbf{y},x}_{par}(v'_3) = \text{0-9}$
$\Gamma^{\mathbf{y},x}_{par}(v'_6) = \text{b}$
$\Gamma^{\mathbf{y},x}_{par}(v'_4) = 1$

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

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

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 representation and derives the decoder and the optimization problem. In Section 5, we discuss our findings from a case study with an email service. Section 6 discusses related work; Section 7 concludes.

2. Regular Expressions

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, that accepts any number of repetitions of its preceding argument expression. Parentheses define the syntactic structure of the expression. Several shorthands improve the readability of regular expressions and 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. 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.* A formal definition of the set of regular expressions can be found in the online appendix.

The syntactic structure of a regular expression is 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 3 in the online appendix assigns one such tree to each regular expression. A node $v \in V^{\mathbf{y}}_{syn}$ of this tree is tagged by labeling function $\Gamma^{\mathbf{y}}_{syn} : V^{\mathbf{y}}_{syn} \to \mathcal{Y}_\Sigma$ with a subexpression $\Gamma^{\mathbf{y}}_{syn}(v) = \mathbf{y}_j$. 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.

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 & 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. Nodes $v \in V^{\mathbf{y}}_{syn}$ of the syntax tree generate the nodes of the parse tree $v' \in V^{\mathbf{y},x}_{par}$; a node 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 $\mathcal{T}^{\mathbf{y},x}_{par}$. A formal definition can be found in the online appendix.

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 $T^{\mathbf{y},x}_{par}$.

Figure 2 shows an example of a syntax tree $T^{\mathbf{y}}_{syn}$ and a parse tree $T^{\mathbf{y},x}_{par}$ for the regular expression $\mathbf{y} = \text{[b0-9]\{2\}c(aa|b)}^*$ and the string $x = \text{1bc}$.

Finally, we introduce the concept of a matching list. When a regular expression $\mathbf{y}$ generates a set $\mathbf{x}$ of strings, and $v \in V^{\mathbf{y}}_{syn}$ 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. 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 $\mathcal{T}_{par}^{\mathbf{y},x}$ 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 [b0-9]{2} is $M^{\mathbf{y},\mathbf{x}}(v_2) = \{\mathsf{12}, \mathsf{b4}\}$. Definition 4 in the online appendix introduces matching lists formally.

3. Problem Setting

Having established the syntax and semantics of regular expressions, we now turn towards the problem setting. An unknown distribution $p(\mathbf{x}, \mathbf{y})$ generates regular expressions $\mathbf{y} \in \mathcal{Y}_\Sigma$ 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 emails sampled from a bot net, and the $\mathbf{y}$ are regular expressions which an expert postmaster believes to identify the campaign template, and feels confident to blacklist.

A $\mathbf{w}$-parameterized predictive model $f_\mathbf{w} : \mathbf{x} \mapsto \hat{\mathbf{y}}$ accepts a batch of strings and conjectures a regular expression $\hat{\mathbf{y}}$. We now define the loss $\Delta(\mathbf{y}, \hat{\mathbf{y}}, \mathbf{x})$ that captures the deviation of the conjecture $\hat{\mathbf{y}}$ from $\mathbf{y}$ for batch $\mathbf{x}$. In our application, postmasters will not use an expression to blacklist the campaign unless they consider it to be comprehensibly and neatly written, and believe it to accurately identify the campaign.

Loss function $\Delta(\mathbf{y}, \hat{\mathbf{y}}, \mathbf{x})$ compares each of the accepting parse trees in $\mathcal{T}_{par}^{\mathbf{y},x}$, for each string $x \in \mathbf{x}$, with the most similar tree in $\mathcal{T}_{par}^{\hat{\mathbf{y}},x}$; if no such parse tree exists, the summand is defined as $\frac{1}{|\mathbf{x}|}$ (Equation 1). Similarly to a loss function for hierarchical classification (Cesa-Bianchi et al., 2006), the difference of two parse trees for string x is quantified by a comparison of the paths that lead to the characters of the string; paths are compared by means of the intersection of their nodes (Equation 2). By its definition, this loss function is bounded between zero and one; it attains zero if and only if the expressions $\mathbf{y}$ and $\hat{\mathbf{y}}$ are equal.

$$\Delta(\mathbf{y}, \hat{\mathbf{y}}, \mathbf{x}) = \frac{1}{|\mathbf{x}|} \sum_{x \in \mathbf{x}} \begin{cases} \Delta_{\text{tree}}(\mathbf{y}, \hat{\mathbf{y}}, x) & \text{if } x \in L(\hat{\mathbf{y}}) \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

$$\text{with } \Delta_{\text{tree}}(\mathbf{y}, \hat{\mathbf{y}}, x) \quad (2)$$

$$= 1 - \frac{1}{|\mathcal{T}_{par}^{\mathbf{y},x}|} \sum_{t \in \mathcal{T}_{par}^{\mathbf{y},x}} \max_{t' \in \mathcal{T}_{par}^{\hat{\mathbf{y}},x}} \frac{1}{|x|} \sum_{j=1}^{|x|} \frac{|t_{|j} \cap t'_{|j}|}{\max\{|t_{|j}|, |t'_{|j}|\}}$$

We will also explore the zero-one loss, $\Delta_{0/1}(\mathbf{y}, \hat{\mathbf{y}}, \mathbf{x}) = [\![\mathbf{y} \neq \hat{\mathbf{y}}]\!]$, where $[\![.]\!]$ is the indicator function of its boolean argument. The zero-one loss serves as an alternative, conceptually simpler reference model.

Our goal is to find the model $f_\mathbf{w}$ with minimal risk

$$R[f_\mathbf{w}] = \iint \Delta(\mathbf{y}, f_\mathbf{w}(\mathbf{x}), \mathbf{x}) p(\mathbf{x}, \mathbf{y}) \mathrm{d}\mathbf{x}\, \mathrm{d}\mathbf{y}. \quad (3)$$

Training data $D = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^m$ consists of pairs of batches $\mathbf{x}_i$ and generating regular expressions $\mathbf{y}_i$, drawn according to $p(\mathbf{x}, \mathbf{y})$.

Since the true distribution $p(\mathbf{x}, \mathbf{y})$ is unknown, the risk $R[f_\mathbf{w}]$ cannot be calculated. We state the learning problem as the problem of minimizing the regularized empirical counterpart of the risk over the parameters $\mathbf{w}$ and the regularizer $\Omega(\mathbf{w})$:

$$\hat{R}[f_\mathbf{w}] = \frac{1}{m} \sum_{(\mathbf{x},\mathbf{y}) \in D} \Delta(\mathbf{y}, f_\mathbf{w}(\mathbf{x}), \mathbf{x}) + \Omega(\mathbf{w}). \quad (4)$$

4. Identifying Regular Expressions

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

$$f_\mathbf{w}(\mathbf{x}) = \arg\max_{\mathbf{y} \in \mathcal{Y}_\Sigma} \mathbf{w}^\mathsf{T}\Psi(\mathbf{x}, \mathbf{y}). \quad (5)$$

4.1. Joint Feature Representation

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

Structural properties of a regular expression $\mathbf{y}$ are captured 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

$$\Lambda(\mathbf{y}) = \begin{pmatrix} [\![\mathbf{y} = \mathbf{y}_1 \ldots \mathbf{y}_k]\!] \\ [\![\mathbf{y} = \mathbf{y}_1 | \ldots | \mathbf{y}_k]\!] \\ [\![\mathbf{y} = [\mathbf{y}_1 \ldots \mathbf{y}_k]]\!] \\ [\![\mathbf{y} = \mathbf{y}_1^*]\!] \\ [\![\mathbf{y} = \mathbf{y}_1?]\!] \\ [\![\mathbf{y} = \mathbf{y}_1^+]\!] \\ [\![\mathbf{y} = \mathbf{y}_1\{l\}]\!] \\ [\![\mathbf{y} = \mathbf{y}_1\{l, u\}]\!] \\ [\![\mathbf{y} = r_1]\!] \\ \vdots \\ [\![\mathbf{y} = r_l]\!] \\ [\![\mathbf{y} \in \Sigma]\!] \\ [\![\mathbf{y} = \epsilon]\!] \end{pmatrix} \quad (6)$$

encoding the top-level operator used in the regular expression $\mathbf{y}$. In Equation 6, $\mathbf{y}_1, \ldots, \mathbf{y}_k \in \mathcal{Y}_\Sigma$ are regular expressions, $l, u \in \mathbb{N}$, and $\{r_1, \ldots, r_l\}$ 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, \w, \d, "."}. 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 $\Lambda(\mathbf{y}') \otimes \Lambda(\mathbf{y}'')$ defines a binary vector that encodes the specific nesting of operators at node v'. Feature vector $\Psi(\mathbf{x}, \mathbf{y})$ 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 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 $\Phi(M^{\mathbf{y},\mathbf{x}}(v'))$ 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 full list of attributes used in our experiments is included in the online appendix. 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 $\Phi(M^{\mathbf{y},\mathbf{x}}(v')) \otimes \Lambda(\mathbf{y}')$.

The joint feature vector $\Psi(\mathbf{x}, \mathbf{y})$ 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 they generate over all nodes in the syntax tree:

$$\Psi(\mathbf{x},\mathbf{y}) \quad (7)$$
$$= \begin{pmatrix} \sum_{(v',v'')\in E^{\mathbf{y}}_{syn}} \Lambda(\Gamma^{\mathbf{y}}_{syn}(v')) \otimes \Lambda(\Gamma^{\mathbf{y}}_{syn}(v'')) \\ \sum_{v'\in V^{\mathbf{y}}_{syn}} \Phi(M^{\mathbf{y},\mathbf{x}}(v')) \otimes \Lambda(\Gamma^{\mathbf{y}}_{syn}(v')) \end{pmatrix}.$$

4.2. Decoding

At application time, the highest-scoring regular expression $f_{\mathbf{w}}(\mathbf{x}) = \arg\max_{\mathbf{y}\in\mathcal{Y}_\Sigma} \mathbf{w}^\mathsf{T}\Psi(\mathbf{x},\mathbf{y})$ has to be identified. This maximization is over the infinite space of all regular expressions $\mathcal{Y}_\Sigma$. To alleviate the intractability of this problem, we approximate this maximum by the maximum over a constrained, finite search space which can be found efficiently.

The constrained search space initially contains an alignment of all strings in $\mathbf{x}$. An alignment is a regular expression that contains only constants—which have to occur in all strings of the batch—and the wildcard symbol "(.*)". The initial alignment $\mathbf{a}_\mathbf{x}$ of $\mathbf{x}$ can be thought of as the most-general bound of this space.

Definition 1 (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 generate all elements of $\mathbf{x}$.*

An alignment is maximal if no other alignment in $A_\mathbf{x}$ contains more constant symbols. 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 & Jiang, 1994); known algorithms are exponential in the number $|\mathbf{x}|$ of strings in $\mathbf{x}$. *Progressive alignment* heuristics find an alignment of a set of strings by incrementally aligning pairs of strings.

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

$$\hat{\mathcal{Y}}_{\mathbf{x},D} = \{a_0\mathbf{y}_1a_1 \ldots \mathbf{y}_na_n | \mathbf{y}_j \in \hat{\mathcal{Y}}^{M_j}_D\} \qquad (8)$$

contains all specializations of $\mathbf{a}_\mathbf{x}$ in which the j-th wildcard symbol is replaced by any element of a set $\hat{\mathcal{Y}}^{M_j}_D$. The sets $\hat{\mathcal{Y}}^{M_j}_D$ are constructed by Algorithm 1. The algorithm starts with $\mathcal{Y}_D$ which we define to be the set of all subexpressions that occur anywhere in the training data D. From this set, it takes a subset such that each regular expression in $\hat{\mathcal{Y}}_{\mathbf{x},D}$ generates all strings in $\mathbf{x}$, and adds a number of syntactic variants and subexpressions in which constants have been replaced to match the elements of M_j, where M_j is the matching list of the node which belongs to the j-th wildcard symbol. Each of the lines 7, 9, 10, 11, and 12 of Algorithm 1 adds at most one element to $\hat{\mathcal{Y}}^{M_j}_D$—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 8:

$$\arg\max_{\mathbf{y}\in\mathcal{Y}_\Sigma} \mathbf{w}^\mathsf{T}\Psi(\mathbf{x},\mathbf{y}) \approx \arg\max_{\mathbf{y}\in\hat{\mathcal{Y}}_{\mathbf{x},D}} \mathbf{w}^\mathsf{T}\Psi(\mathbf{x},\mathbf{y}). \qquad (9)$$

We will now argue that this maximization problem can be decomposed into independent maximization problems for each of the $\mathbf{y}_j$ that replaces the j-th wildcard in the alignment $\mathbf{a}_\mathbf{x}$ due to the simple syntactic structure of the alignment and the definition of Ψ.

Feature vector $\Psi(\mathbf{x},\mathbf{y})$ decomposes linearly into a sum over the nodes and a sum over pairs of adjacent nodes (see Equation 7). The syntax tree of an instantiation $\mathbf{y} = a_0\mathbf{y}_1a_1 \ldots \mathbf{y}_na_n$ of the alignment $\mathbf{a}_\mathbf{x}$ consists of a root node labeled as an alternating concatenation of constant strings a_j and subexpressions $\mathbf{y}_j$ (see Figure 3). This root node is connected to a layer on which constant strings $a_j = a_{j,1} \ldots a_{j,|a_j|}$ and subtrees $T^{\mathbf{y}_j}_{syn}$ alternate (blue area in Figure 3). However, the terms in Equation 10 that correspond to the root node $\mathbf{y}$ and

Algorithm 1 Constructing the decoding space

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

the a_j are constant for all values of the $\mathbf{y}_j$ (red area in Figure 3). Since no edges connect multiple wildcards, the feature representation of these subtrees can be decomposed into n independent summands as in Equation 11.

$$\Psi(\mathbf{x}, a_0\mathbf{y}_1 a_1 \ldots \mathbf{y}_n a_n) \tag{10}$$

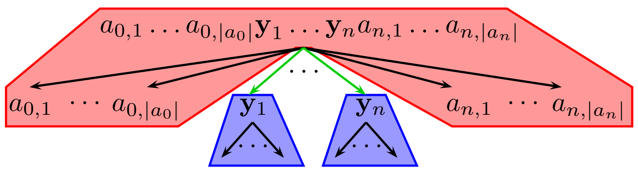
$$= \begin{pmatrix} \sum_{j=1}^{n} \Lambda(\mathbf{y}) \otimes \Lambda(\mathbf{y}_j) + \sum_{j=0}^{n}\sum_{q=1}^{|a_j|} \Lambda(\mathbf{y}) \otimes \Lambda(a_{j,q}) \\ \Phi(\{\mathbf{x}\}) \otimes \Lambda(\mathbf{y}) + \sum_{j=0}^{n}\sum_{q=1}^{|a_j|} \Phi(\{a_{j,q}\}) \otimes \Lambda(a_{j,q}) \end{pmatrix}$$

$$+ \begin{pmatrix} \sum_{j=1}^{n} \sum_{(v',v'') \in E^{\mathbf{y}_j}_{syn}} \Lambda(\Gamma^{\mathbf{y}_j}_{syn}(v')) \otimes \Lambda(\Gamma^{\mathbf{y}_j}_{syn}(v'')) \\ \sum_{j=1}^{n} \sum_{v' \in V^{\mathbf{y}_j}_{syn}} \Phi(M^{\mathbf{y}_j, M_j}(v')) \otimes \Lambda(\Gamma^{\mathbf{y}_j}_{syn}(v')) \end{pmatrix}$$

$$= \begin{pmatrix} \mathbf{0} \\ \Phi(\{\mathbf{x}\}) \otimes \Lambda(\mathbf{y}) \end{pmatrix} + \sum_{j=0}^{n}\sum_{q=1}^{|a_i|} \begin{pmatrix} \Lambda(\mathbf{y}) \otimes \Lambda(a_{j,q}) \\ \Phi(\{a_{j,q}\}) \otimes \Lambda(a_{j,q}) \end{pmatrix}$$

$$+ \sum_{j=1}^{n} \left(\Psi(\mathbf{y}_j, M_j) + \begin{pmatrix} \Lambda(\mathbf{y}) \otimes \Lambda(\mathbf{y}_j) \\ \mathbf{0} \end{pmatrix} \right) \tag{11}$$

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 $\Lambda(\mathbf{y})$ as



$a_{0,1} \cdots a_{0,|a_0|}\mathbf{y}_1 \cdots \mathbf{y}_n a_{n,1} \cdots a_{n,|a_n|}$
$a_{0,1} \cdots a_{0,|a_0|}$
$\mathbf{y}_1$
$\mathbf{y}_n$
$a_{n,1} \cdots a_{n,|a_n|}$

Figure 3. Structure of a syntax tree for an element of $\hat{\mathcal{Y}}_{\mathbf{x},D}$.

a constant $\Lambda_\bullet$, defined as the output feature vector (Equation 6) of a concatenation.

Thus, the maximization over all $\mathbf{y} = a_0\mathbf{y}_1 a_1 \ldots \mathbf{y}_n a_n$ can be decomposed into n maximization problems over

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

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

4.3. Optimization Problem

We will now address the process of minimizing the regularized empirical risk $\hat{R}$, defined in Equation 4, for the ℓ_2 regularizer $\Omega(\mathbf{w}) = \frac{1}{2C}||\mathbf{w}||^2$. Loss function Δ, defined in Equation 1, is not convex. To obtain a convex optimization problem, we upper-bound the loss by its hinged version, following the margin-rescaling approach (Tsochantaridis et al., 2005):

$$\xi_i = \max_{\mathbf{y} \neq \mathbf{y}_i} \{\mathbf{w}^\mathsf{T}(\Psi(\mathbf{x}_i, \mathbf{y}_i) - \Psi(\mathbf{x}_i, \mathbf{y})) + \Delta(\mathbf{y}_i, \mathbf{y}, \mathbf{x})\}. \tag{12}$$

The maximum in Equation 12 is over all $\mathbf{y} \in \mathcal{Y}_\Sigma \setminus \{\mathbf{y}_i\}$. When the risk is rephrased as a constrained optimization problem, the maximum produces one constraint per element of $\mathbf{y} \in \mathcal{Y}_\Sigma \setminus \{\mathbf{y}_i\}$. 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.

When the loss is replaced by its upper bound—the slack variable ξ—and for $\Omega(\mathbf{w}) = \frac{1}{2C}||\mathbf{w}||^2$, the minimization of the regularized empirical risk (Equation 4) is reduced to Optimization Problem 1.

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

$$\mathbf{w}^* = \arg\min_{\mathbf{w},\xi} \frac{1}{2}||\mathbf{w}||^2 + \frac{C}{m}\sum_{i=1}^{m} \xi_i, \text{ such that} \tag{13}$$

$$\forall i, \forall \bar{\mathbf{y}} \in \hat{\mathcal{Y}}_{\mathbf{x}_i,D} \setminus \{\mathbf{y}_i\} : \mathbf{w}^\mathsf{T}(\Psi(\mathbf{x}_i, \mathbf{y}_i) - \Psi(\mathbf{x}_i, \bar{\mathbf{y}})) \geq \Delta(\mathbf{y}_i, \bar{\mathbf{y}}, \mathbf{x}) - \xi_i, \tag{14}$$

$$\forall i : \xi_i \geq 0. \tag{15}$$

This optimization problem is convex, since the objective (Equation 13) is convex and the constraints

(Equation 14 and 15) are affine in $\mathbf{w}$. 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).

Algorithm 2 Most strongly violated constraint

```
Input: batch $\mathbf{x}$, model $f_{\mathbf{w}}$, correct output $\mathbf{y}$.
 1: Infer alignment $\mathbf{a_x} = a_0(.^*)a_1 \dots (.^*)a_n$ for $\mathbf{x}$.
 2: 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 $\Gamma^{\mathbf{a_x}}_{syn}(v_j) =$ "(.*)".
 3: for all $j = 1 \dots n$ do
 4:     Let $M_j = M^{\mathbf{a_x},\mathbf{x}}(v_j)$ and calculate the $\hat{\mathcal{Y}}^{M_j}_D$ using Algorithm 1.
 5:     $\bar{\mathbf{y}}_j = \arg\max_{\mathbf{y}'_j \in \hat{\mathcal{Y}}^{M_j}_D} \mathbf{w}^{\mathsf{T}} \left( \Psi(\mathbf{y}'_j, M_j) + \begin{pmatrix} \Lambda_{\bullet} \otimes \Lambda(\mathbf{y}'_j) \\ \mathbf{0} \end{pmatrix} \right) + \Delta(\mathbf{y}, a_0(.^*)a_1 \dots (.^*)a_{j-1}\mathbf{y}'_j a_j(.^*)a_{j+1} \dots (.^*)a_n, \mathbf{x})$
 6: end for
 7: Let $\bar{\mathbf{y}}$ abbreviate $a_0\bar{\mathbf{y}}_1 a_1 \dots \bar{\mathbf{y}}_n a_n$
 8: if $\bar{\mathbf{y}} = \mathbf{y}$ then
 9:     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
        $f_{\mathbf{w}}(\mathbf{x}, \bar{\mathbf{y}}) + \Delta_{\text{tree}}(\mathbf{y}, \bar{\mathbf{y}})$ is obtained but the constraint $\bar{\mathbf{y}} \neq \mathbf{y}$ is enforced.
10: end if
Return: $\bar{\mathbf{y}}$
```

During the optimization procedure, the regular expression that incurs the highest slack ξ_i for a given $\mathbf{x}_i$,

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

has to be identified 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 1 never applies within our search space. Without this case, Equations 1 and 2 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*.

5. Case Study

We investigate whether postmasters accept the output of *REx-SVM* to blacklist mailing campaigns during regular operations of a commercial email service. We also evaluate how accurately *REx-SVM* and reference methods identify the extensions of mailing campaigns.

	Campaign 1	Campaign 2	Campaign 3
Postmaster	First name: [\S]$^{+}$ Surname: \S^{+} Height: 1\d^{+} cm. Weights: \d{2} kg.	The trans(fer|action) ID: \d^{+}... ID:( )*\d^{+}( )* ... report_\d^{+}.doc	http://(LOVEGAME [S0-9]*|lovegame [s0-9]*).(com|net)
REx-SVM	First name: [\S]$^{+}$ Surname: \S^{+} Height: 1\d^{+} cm. Weights: \d{2} kg.	The trans(fer|action) ID: \d^{+}... ID:[0-9]$^{+}$ ... report_\d^{+}.doc	http://\e^{+}.(com|net)
REx$_{0/1}$-SVM	First name: [\S]$^{+}$ Surname: [a-zA-Z]$^{+}$ Height: 1\d^{+} cm. Weights: [1467]$^{+}$ kg.	The trans[a-z]$^{+}$ ID: \d^{+}... ID:[a-z0-9]{2,6} ... report_(2| ... |73).doc	http://\e^{+}.[a-z]$^{+}$

Figure 4. Regular expressions created by a postmaster and corresponding output of *REx-SVM* and *REx$_{0/1}$-SVM*.

5.1. Evaluation by Postmasters

REx-SVM is trained on the *ESP* data set that contains 158 batches with a total of 12,763 emails and corresponding regular expressions, collected from the email service provider. The model is deployed; the user interface presents newly detected batches of spam emails together with the regular expression conjectured by *REx-SVM* to a postmaster during regular operations of the service. The postmaster is charged with blacklisting the campaigns by suitable regular expressions. Over the study, the postmasters created 188 regular expressions. Of these, they created 169 expressions (89%) by copying a substring of the automatically generated expression. We observe that postmasters prefer to describe only a part of the message which they feel is characteristic for the campaign whereas *REx-SVM* describes the entirety of the messages. In 12 cases, the postmasters edited the string, and in 7 cases they wrote an expression from scratch.

To illustrate different cases, Figure 4 compares excerpts of expressions created by *REx*- and *REx$_{0/1}$-SVM* (a variant of *REx-SVM* that uses the zero-one loss instead of Δ defined in Equation 1) to expressions of a postmaster. The first example shows a perfect agreement between *REx-SVM* and postmaster. In the second example, the expressions are close but distinct. In the third example, the SVMs produce expressions that generate an overly general set of URLs and lead to false positives ("\e" stands for characters that can occur in a URL). In all three cases, *REx-SVM* is more similar to the postmaster than *REx$_{0/1}$*.

The top right diagram of Figure 5 shows the average loss Δ of *REx*- and *REx$_{0/1}$-SVM*, measured by cross validation with one batch held out. While postmasters show the tendency to write expressions that only characterize about 10% of the message, the *REx-SVM*

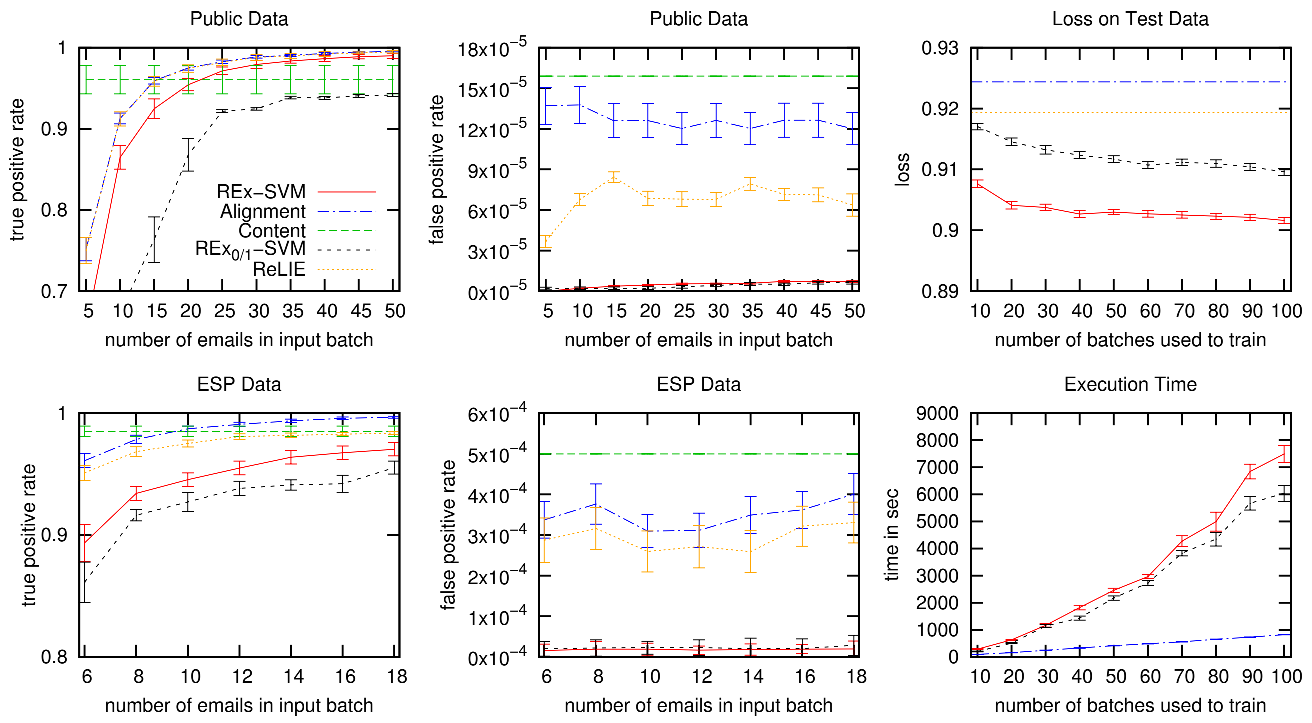


Figure 5. Empirical results on public and ESP data sets. Error bars indicate standard errors.

variants describe the entirety of the message. This leads to relatively high values of the loss function.

5.2. Spam Filtering Performance

We evaluate the ability of *REx-SVM* and baselines to identify the exact extension of email campaigns. We use the *alignment* of the strings in $\mathbf{x}$ as a baseline. In addition, *ReLIE* (Li et al., 2008) searches 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; we use the alignment of the input batch as starting point. *ReLIE* receives an additional 10,000 emails that are not part of any batch as negative data. An additional *content*-based filter employed by the provider has been trained on several million spam and non-spam emails.

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 combine the ESP data set with 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, we combine this collection of spam batches with 17,419 emails from the *Enron corpus*[2] of non-spam emails and 76,466 non-spam emails of the *TREC corpus*[3]. The public data set is available to researchers.

In an outer loop of leave-one-out cross validation, one batch is held back to evaluate the true-positive rate (the proportion of the campaign that is correctly recognized). In an inner loop of 10-fold cross validation, regularization parameter C is tuned.

Figure 5 shows the true and false positive rates for all methods and both data sets. The horizontal axis displays the number of emails in the input batch $\mathbf{x}$. Error bars indicate the standard error. The *alignment* exhibits 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* needs only very few or zero replacement steps until no negative examples are covered. Consequently, it has similarly high true- and false-positive rates. *REx-SVM* attains a slightly lower true positive rate, and a substantially lower false-positive rate. The false-positive rates of *REx* and *REx*$_{0/1}$ 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 func-

[1]http://untroubled.org/spam/

[2]http://www.cs.cmu.edu/~enron/

[3]http://trec.nist.gov/data/spam.html

tion of Equation 1 preferable to the zero-one loss.

The execution time to learn a model (bottom right) is consistent with prior findings of between linear and quadratic for the SVM optimization process.

6. Related Work

Gold (1967) shows that it is impossible to exactly identify any regular language from finitely many positive examples. 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; Clark & Thollard, 2004). While these problems are identical in theory, transforming generated automata into regular expressions can lead to lengthy terms that do not lend themselves to human comprehension (Fernau, 2009). Some work focuses on restricted classes, such as 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).

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.

7. Conclusions

Complementing the language-identification paradigm, we pose the problem of learning to map a set of strings to a target regular expression. Training data consists of batches of strings and corresponding expressions. We phrase this problem as a learning problem with structured output spaces and engineer an appropriately loss function. We derive the resulting optimization problem, and devise a decoder that searches a space of specializations of a maximal alignment.

From our case study we conclude that *REx-SVM* gives a high true positive rate at a false positive rate that is more than an order of magnitude lower than that of a commercial *content*-based filter. The system 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.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Learning to Identify Regular Expressions that Describe Email Campaigns (Online Appendix)

Paul Prasse PRASSE@CS.UNI-POTSDAM.DE
Christoph Sawade SAWADE@CS.UNI-POTSDAM.DE
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

A. Definitions

Definition 2 (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 $l \leq u$.

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 $(\mathbf{y} \rightarrow T_1, \dots, T_k)$ denotes the tree $T = (V, E, \Gamma, \leq)$ with root node $v_0 \in V$ labeled with $\Gamma(v_0) = \mathbf{y}$ and subtrees $T_1, \dots, T_k$. The order $\leq$ maintains the subtree orderings $\leq_i$ and defines the root node as the minimum over the set V and $v' \leq v''$ for all $v' \in V_i$ and $v'' \in V_j$, where $i < j$.

Definition 3 (Syntax Tree)**.** The abstract syntax tree $T^{\mathbf{y}}_{syn}$ for a regular expression $\mathbf{y}$ is recursively defined as follows. Let $T^{\mathbf{y}_j}_{syn} = (V^{\mathbf{y}_j}_{syn}, E^{\mathbf{y}_j}_{syn}, \Gamma^{\mathbf{y}_j}_{syn}, \leq^{\mathbf{y}_j}_{syn})$ 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
  $T^{\mathbf{y}}_{syn} = (\mathbf{y} \rightarrow \emptyset)$.
- 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
  $T^{\mathbf{y}}_{syn} = (\mathbf{y} \rightarrow T^{\mathbf{y}_1}_{syn})$.
- 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
  $T^{\mathbf{y}}_{syn} = (\mathbf{y} \rightarrow T^{\mathbf{y}_1}_{syn}, \dots, T^{\mathbf{y}_k}_{syn})$.
- 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
  $T^{\mathbf{y}}_{syn} = (\mathbf{y} \rightarrow T^{\mathbf{y}_1}_{syn}, \dots, T^{\mathbf{y}_k}_{syn})$.

Definition 4 (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
  $T^{\mathbf{y},x}_{par} = (\mathbf{y} \rightarrow \emptyset)$.
- 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 $\mathbf{y} = \mathbf{y}_1^*$, $x = x_1 \dots x_k$, 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
$M^{\mathbf{y},x}(v) = \begin{cases} \{x\} & \text{, if } v = v_0 \\ \bigcup_{i=1}^{k} M^{\mathbf{y}_1,x_i}(v) & \text{, if } v \in V^{\mathbf{y}_1}_{syn} \end{cases}$, and
$T^{\mathbf{y},x}_{par} = (\mathbf{y} \to T^{\mathbf{y}_1,x_1}_{par}, \dots, T^{\mathbf{y}_1,x_k}_{par})$.
- If $\mathbf{y} = \mathbf{y}_1 \dots \mathbf{y}_k$, $x = x_1 \dots x_k$,
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
$M^{\mathbf{y},x}(v) = \begin{cases} \{x\} & \text{, if } v = v_0 \\ M^{\mathbf{y}_j,x_i}(v) & \text{, if } v \in V^{\mathbf{y}_j}_{syn} \end{cases}$, and
$T^{\mathbf{y},x}_{par} = (\mathbf{y} \to T^{\mathbf{y}_1,x_1}_{par}, \dots, T^{\mathbf{y}_k,x_k}_{par})$.
- If $\mathbf{y} = \mathbf{y}_1 | \dots | \mathbf{y}_k$, $x \in \Sigma^*$
and there exist no $\mathbf{y}', \mathbf{y}'' \in \mathcal{Y}_\Sigma$ such that $\mathbf{y}_j = \mathbf{y}' \mid \mathbf{y}''$, or if
$\mathbf{y} = [\mathbf{y}_1 \dots \mathbf{y}_k]$, $x \in \Sigma^+$
and there exist no $\mathbf{y}', \mathbf{y}'' \in \mathcal{Y}_\Sigma$ such that $\mathbf{y}_j = \mathbf{y}' \ \mathbf{y}''$, we define
$M^{\mathbf{y},x}(v) = \begin{cases} \{x\} & \text{, if } v = v_0 \\ M^{\mathbf{y}_j,x}(v) & \text{, if } v \in V^{\mathbf{y}_j}_{syn} \\ \emptyset & \text{, otherwise} \end{cases}$, and
$T^{\mathbf{y},x}_{par} = (\mathbf{y} \to T^{\mathbf{y}_j,x}_{par})$.

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 4 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 $M^{\mathbf{y},\mathbf{x}}(v) = \bigcup_{x \in \mathbf{x}} \mathcal{M}^{\mathbf{y},x}(v)$.

B. Used Features

Let M be a matching list and M_Σ be the set of characters in Σ which appear in the matching list M. The list of binary and continuous features, used to train *REx-SVM* is shown in Table 1.

Table 1. Important features used to train *REx-SVM*.

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?
$\frac{\lvert M_\Sigma \cap \{A,\ldots,Z\}\rvert}{26}$	Portion of characters A–Z in the matching list
$\frac{\lvert M_\Sigma \cap \{a,\ldots,z\}\rvert}{26}$	Portion of characters a–z in the matching list
$\frac{\lvert M_\Sigma \cap \{0,\ldots,9\}\rvert}{10}$	Portion of characters 0–9 in the matching list
$\frac{\lvert M_\Sigma \cap \{A,\ldots,F\}\rvert}{6}$	Portion of characters A–F in the matching list
$\frac{\lvert M_\Sigma \cap \{a,\ldots,f\}\rvert}{6}$	Portion of characters a–f in the matching list
$\frac{\lvert M_\Sigma \cap \{G,\ldots,Z\}\rvert}{20}$	Portion of characters G–Z in the matching list
$\frac{\lvert M_\Sigma \cap \{g,\ldots,z\}\rvert}{20}$	Portion of characters g–z in the matching list
$[\![\forall x \in M_\Sigma x \notin \{A,\ldots,Z\}\ ]\!]$	No characters of A–Z in the matching list?
$[\![\forall x \in M_\Sigma x \notin \{a,\ldots,z\}\ ]\!]$	No characters of a–z in the matching list?
$[\![\forall x \in M_\Sigma x \notin \{0,\ldots,9\}\ ]\!]$	No characters of 0–9 in the matching list?
$[\![\forall x \in M_\Sigma x \notin \{a,\ldots,f\}\ ]\!]$	No characters of a–f in the matching list?
$[\![\forall x \in M_\Sigma x \notin \{A,\ldots,F\}\ ]\!]$	No characters of A–F in the matching list?
$[\![\lvert M_\Sigma \cap \{-,/,?,=,.,@,:\}\rvert > 0]\!]$	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?

Journal of Machine Learning Research 16 (2015) 3687-3720 Submitted 4/14; Revised 2/15; Published 12/15

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

Abstract

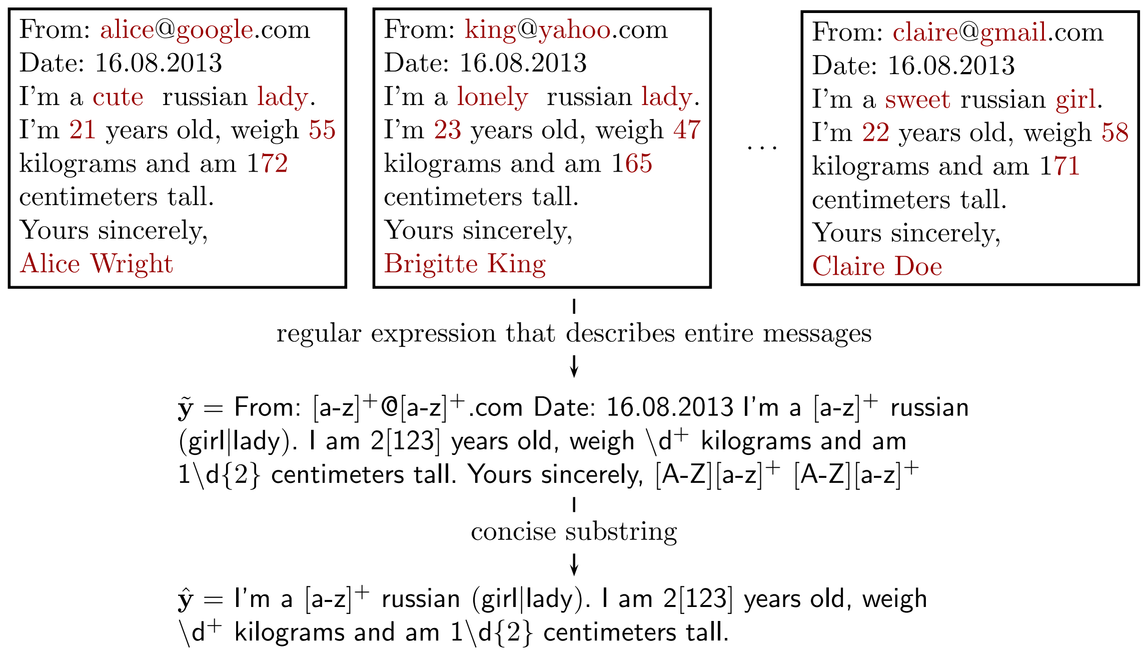
This paper addresses the problem of inferring a regular expression from a given set of strings that resembles, as closely as possible, the regular expression that a human expert would have written to identify the language. This is motivated by our goal of automating the task of postmasters who use regular 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.

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

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.

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.



From: alice@google.com
Date: 16.08.2013
I'm a cute russian lady.
I'm 21 years old, weigh 55 kilograms and am 172 centimeters tall.
Yours sincerely,
Alice Wright

From: king@yahoo.com
Date: 16.08.2013
I'm a lonely russian lady.
I'm 23 years old, weigh 47 kilograms and am 165 centimeters tall.
Yours sincerely,
Brigitte King

...

From: claire@gmail.com
Date: 16.08.2013
I'm a sweet russian girl.
I'm 22 years old, weigh 58 kilograms and am 171 centimeters tall.
Yours sincerely,
Claire Doe

regular expression that describes entire messages

$\tilde{\mathbf{y}}$ = From: [a-z]$^+$@[a-z]$^+$.com Date: 16.08.2013 I'm a [a-z]$^+$ russian (girl|lady). I am 2[123] years old, weigh \d$^+$ kilograms and am 1\d{2} centimeters tall. Yours sincerely, [A-Z][a-z]$^+$ [A-Z][a-z]$^+$

concise substring

$\hat{\mathbf{y}}$ = I'm a [a-z]$^+$ russian (girl|lady). I am 2[123] years old, weigh \d$^+$ kilograms and am 1\d{2} centimeters tall.

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.

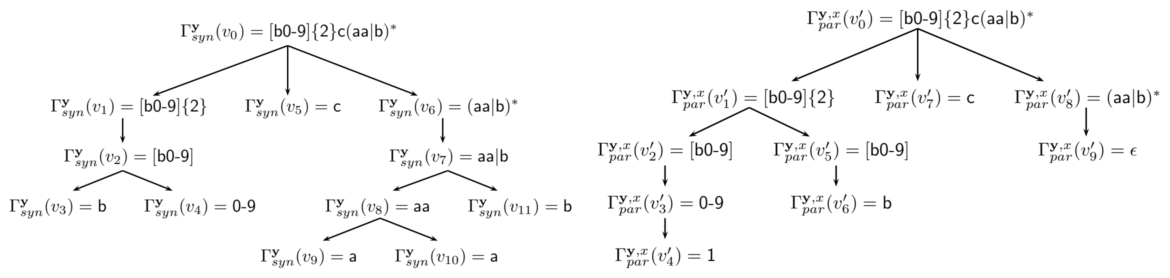
The rest of this paper is structured as follows. Section 2 reviews regular expressions before Section 3 states the problem setting. Section 4 introduces the feature representations and derives the decoders and the optimization problems. In Section 5, we discuss our findings from a case study with an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


$\Gamma^{\mathbf{y}}_{syn}(v_0)$ = [b0-9]{2}c(aa|b)*
$\Gamma^{\mathbf{y}}_{syn}(v_1)$ = [b0-9]{2}
$\Gamma^{\mathbf{y}}_{syn}(v_5)$ = c
$\Gamma^{\mathbf{y}}_{syn}(v_6)$ = (aa|b)*
$\Gamma^{\mathbf{y}}_{syn}(v_2)$ = [b0-9]
$\Gamma^{\mathbf{y}}_{syn}(v_7)$ = aa|b
$\Gamma^{\mathbf{y}}_{syn}(v_3)$ = b
$\Gamma^{\mathbf{y}}_{syn}(v_4)$ = 0-9
$\Gamma^{\mathbf{y}}_{syn}(v_8)$ = aa
$\Gamma^{\mathbf{y}}_{syn}(v_{11})$ = b
$\Gamma^{\mathbf{y}}_{syn}(v_9)$ = a
$\Gamma^{\mathbf{y}}_{syn}(v_{10})$ = a
$\Gamma^{\mathbf{y},x}_{par}(v'_0)$ = [b0-9]{2}c(aa|b)*
$\Gamma^{\mathbf{y},x}_{par}(v'_1)$ = [b0-9]{2}
$\Gamma^{\mathbf{y},x}_{par}(v'_7)$ = c
$\Gamma^{\mathbf{y},x}_{par}(v'_8)$ = (aa|b)*
$\Gamma^{\mathbf{y},x}_{par}(v'_2)$ = [b0-9]
$\Gamma^{\mathbf{y},x}_{par}(v'_5)$ = [b0-9]
$\Gamma^{\mathbf{y},x}_{par}(v'_9)$ = ϵ
$\Gamma^{\mathbf{y},x}_{par}(v'_3)$ = 0-9
$\Gamma^{\mathbf{y},x}_{par}(v'_6)$ = b
$\Gamma^{\mathbf{y},x}_{par}(v'_4)$ = 1

(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 $\mathbf{y} =$ [b0-9]{2}c(aa|b)*. One corresponding parse tree $T_{par}^{\mathbf{y},x}$ for the string $x =$ 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} = \{$12c, b4ca$\}$, the matching list for node v_1 that represents subexpression $\Gamma_{syn}^{\mathbf{y}}(v_1) =$ [b0-9]{2} is $M^{\mathbf{y},\mathbf{x}}(v_2) = \{$12, b4$\}$. 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 $p(\mathbf{x}, \mathbf{y})$:

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

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.

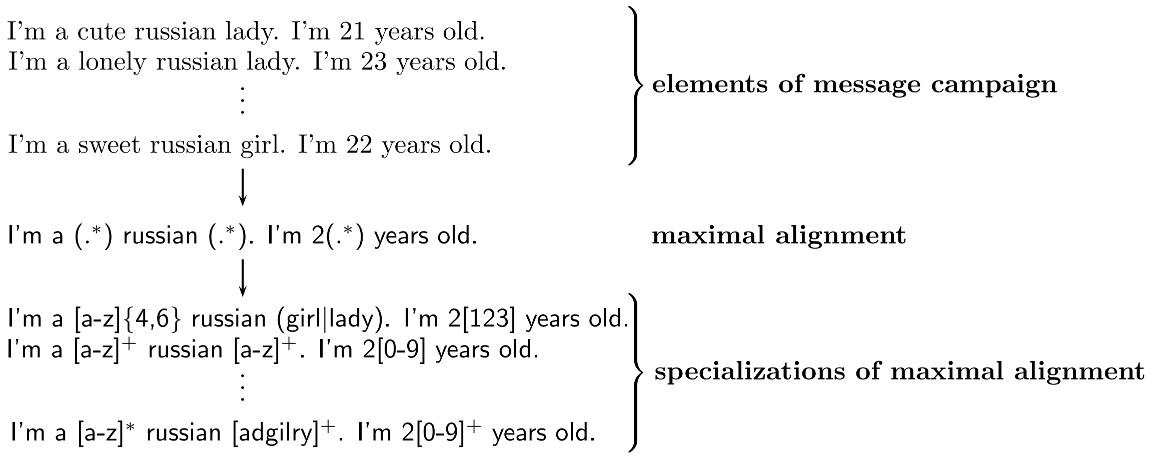
4.1 Problem Decomposition

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.


I'm a cute russian lady. I'm 21 years old.
I'm a lonely russian lady. I'm 23 years old.
⋮
I'm a sweet russian girl. I'm 22 years old.

elements of message campaign

↓

I'm a (.*) russian (.*). I'm 2(.*) years old.

maximal alignment

↓

I'm a [a-z]{4,6} russian (girl|lady). I'm 2[123] years old.
I'm a [a-z]$^+$ russian [a-z]$^+$. I'm 2[0-9] years old.
⋮
I'm a [a-z]* russian [adgilry]$^+$. I'm 2[0-9]$^+$ years old.

specializations of maximal alignment

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

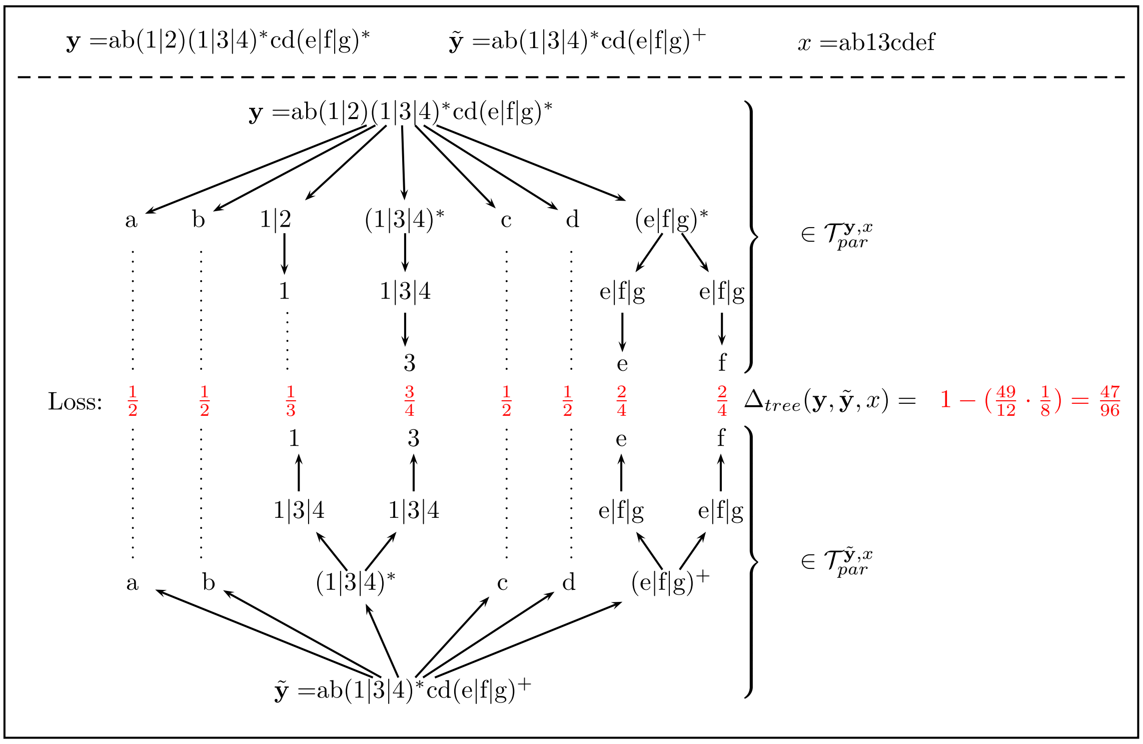


Figure 4: Calculation of the tree loss $\Delta_{tree}(\mathbf{y}, \tilde{\mathbf{y}}, x)$ for a given string x and two regular expressions $\mathbf{y}$ and $\tilde{\mathbf{y}}$.

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

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

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, $\mathbf{y}_1, \ldots, \mathbf{y}_k \in \mathcal{Y}_\Sigma$ are regular expressions, $l, u \in \mathbb{N}$, and $\{r_1, \ldots, r_l\}$ 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:

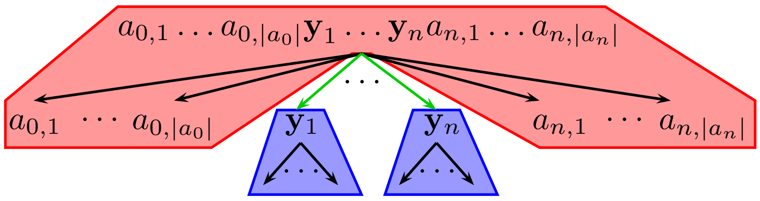
Theorem 2 *The maximization problem of finding the highest-scoring regular expression $f_{\mathbf{u}}(\mathbf{x})$ can be decomposed into independent maximization problems for each of the $\mathbf{y}_j$ that replaces the j-th wildcard in the alignment $\mathbf{a_x}$, given the alignment and the definition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

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

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 $\mathbf{a_x} = a_0(.^*)a_1 \ldots (.^*)a_n$ of the strings in $\mathbf{x}$.
2: let $T^{\mathbf{a_x}}_{syn}$ be the syntax tree of the alignment and $v_1, \ldots, v_n$ be the nodes labeled
   $\Gamma^{\mathbf{a_x}}_{syn}(v_j) =$ "(.*)".
3: for $j = 1 \ldots n$ 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 $x_1, \ldots, x_m$ be the elements of $M_j$; add $(x_1| \ldots |x_m)$ 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 $[\beta \mathbf{y}_1 \ldots \mathbf{y}_k] \in \hat{\mathcal{Y}}^{M_j}_D$, where $\beta \in \Sigma^*$ and $\mathbf{y}_1 \ldots \mathbf{y}_k$ are ranges or special macros (e.g.,
      a-z, \e), then add $[\alpha \mathbf{y}_1 \ldots \mathbf{y}_k]$ to $\hat{\mathcal{Y}}^{M_j}_D$, where $\alpha \in \Sigma^*$ is the longest string that satisfies
      $M_j \subseteq L([\alpha \mathbf{y}_1 \ldots \mathbf{y}_k])$, if such an $\alpha$ 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 $\hat{\mathcal{Y}}^{M_1}_D, \ldots, \hat{\mathcal{Y}}^{M_n}_D$.
```



$a_{0,1} \dots a_{0,|a_0|}\mathbf{y}_1 \dots \mathbf{y}_n a_{n,1} \dots a_{n,|a_n|}$
$a_{0,1}$ $\cdots$ $a_{0,|a_0|}$
$\mathbf{y}_1$
$\mathbf{y}_n$
$a_{n,1}$ $\cdots$ $a_{n,|a_n|}$

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 $\mathbf{y} = a_0\mathbf{y}_1 a_1 \dots \mathbf{y}_n a_n$ of the alignment $\mathbf{a}_\mathbf{x}$ 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.

$$\begin{aligned}
&\Psi_\mathbf{u}(\mathbf{x}, a_0\mathbf{y}_1 a_1 \dots \mathbf{y}_n a_n) \qquad (8)\\
&= \begin{pmatrix} \sum_{j=1}^{n} \Lambda_\mathbf{u}(\mathbf{y}) \otimes \Lambda_\mathbf{u}(\mathbf{y}_j) + \sum_{j=0}^{n}\sum_{q=1}^{|a_j|} \Lambda_\mathbf{u}(\mathbf{y}) \otimes \Lambda_\mathbf{u}(a_{j,q}) \\ \Phi_\mathbf{u}(\{\mathbf{x}\}) \otimes \Lambda_\mathbf{c}(\mathbf{y}) + \sum_{j=0}^{n}\sum_{q=1}^{|a_j|} \Phi_\mathbf{u}(\{a_{j,q}\}) \otimes \Lambda_\mathbf{u}(a_{j,q}) \end{pmatrix} \\
&+ \begin{pmatrix} \sum_{j=1}^{n} \sum_{(v',v'') \in E_{syn}^{\mathbf{y}_j}} \Lambda_\mathbf{u}(\Gamma_{syn}^{\mathbf{y}_j}(v')) \otimes \Lambda_\mathbf{u}(\Gamma_{syn}^{\mathbf{y}_j}(v'')) \\ \sum_{j=1}^{n} \sum_{v' \in V_{syn}^{\mathbf{y}_j}} \Phi_\mathbf{u}(M^{\mathbf{y}_j, M_j}(v')) \otimes \Lambda_\mathbf{u}(\Gamma_{syn}^{\mathbf{y}_j}(v')) \end{pmatrix} \\
&= \begin{pmatrix} \mathbf{0} \\ \Phi_\mathbf{u}(\{\mathbf{x}\}) \otimes \Lambda_\mathbf{u}(\mathbf{y}) \end{pmatrix} + \sum_{j=0}^{n}\sum_{q=1}^{|a_i|} \begin{pmatrix} \Lambda_\mathbf{u}(\mathbf{y}) \otimes \Lambda_\mathbf{u}(a_{j,q}) \\ \Phi_\mathbf{u}(\{a_{j,q}\}) \otimes \Lambda_\mathbf{u}(a_{j,q}) \end{pmatrix} \\
&+ \sum_{j=1}^{n} \left(\Psi_\mathbf{u}(\mathbf{y}_j, M_j) + \begin{pmatrix} \Lambda_\mathbf{u}(\mathbf{y}) \otimes \Lambda_\mathbf{u}(\mathbf{y}_j) \\ \mathbf{0} \end{pmatrix} \right) \qquad (9)
\end{aligned}$$

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 $\mathbf{y} = a_0\mathbf{y}_1 a_1 \dots \mathbf{y}_n a_n$ 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*

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

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

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 $\mathbf{x}_i$,

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

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*

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

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

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

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

	Campaign 1	Campaign 2	Campaign 3
Postmaster	Please send the request to my email (simon|george)@(gmail|yahoo).com	Email:wester_(payin|pay)@yahoo.com Yours sincerely, Mr [A-Z][a-z]$^+$ [A-Z][a-z]$^+$	(Reply-To|From):(mosk@aven|sevid@donald).com Subject: GET YOUR MONEY
REx-SVM	... 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 ...	... 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]$^+$ ...	... (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. ...
REx-SVM^{short}	Please send the request to my email [a-z]$^+$@(gmail|yahoo).com and I will answer you personally as soon as possible	Email:wester_(payin|pay)@yahoo.com Yours sincerely, Mr [A-Za-z]$^+$ [A-Za-z]$^+$	(Reply-To|From):(mosk@aven|sevid@donald).com Subject: GET YOUR MONEY

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`

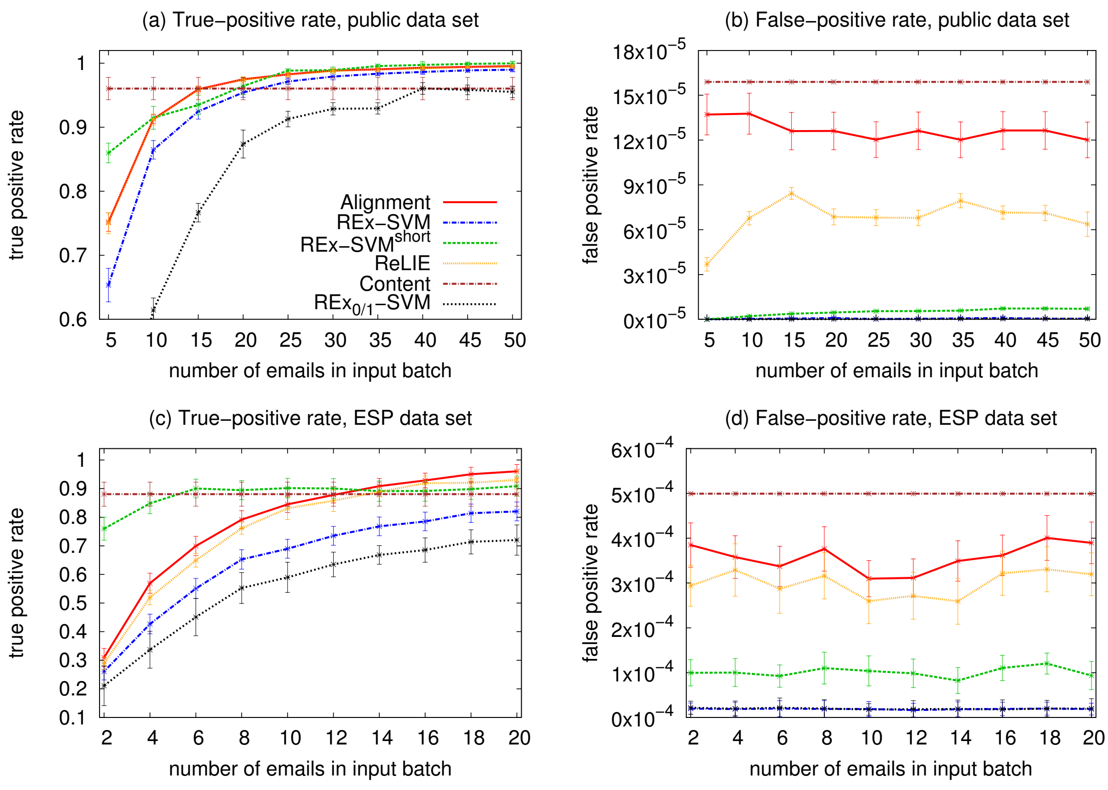


Figure 7: True-positive and false-positive rates over the number of used emails in the input batch $\mathbf{x}$ for the public and ESP data sets.

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.

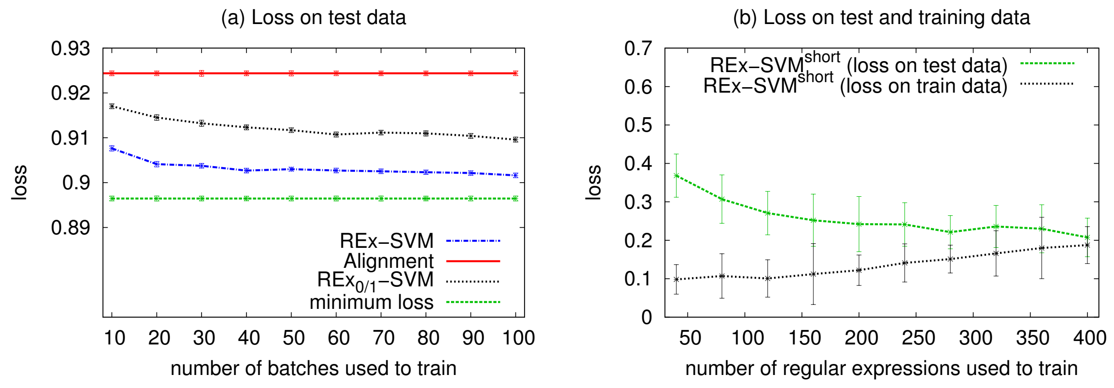


Figure 8: (a) Loss $\Delta_{\mathbf{u}}$ of model $f_{\mathbf{u}}$ on the test data (left Figure). (b) Loss $\Delta_{\mathbf{v}}$ of model $f_{\mathbf{v}}$ on the training and test data. Error bars indicate standard errors.

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.

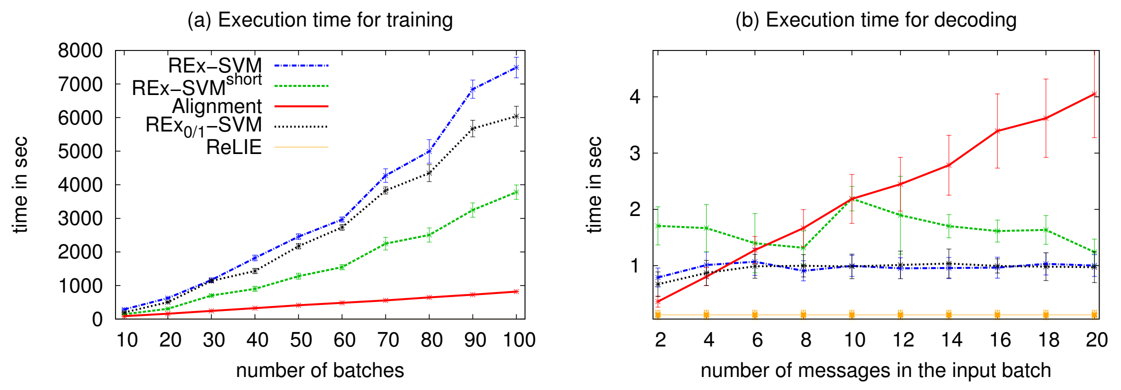


Figure 9: Execution time for training a model (a) and decoding a regular expression at application time (b).

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 machines (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. *Journal of 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 subgradient 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

A.1 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 $l \leq u$.

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 $T = (V, E, \Gamma, \leq)$ with root node $v_0 \in V$ labeled with $\Gamma(v_0) = \mathbf{y}$ and subtrees $T_1, \dots, T_k$. The order $\leq$ maintains the subtree orderings $\leq_i$ and defines the root node as the minimum over the set V and $v' \leq v''$ 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 $T^{\mathbf{y}_j}_{syn} = (V^{\mathbf{y}_j}_{syn}, E^{\mathbf{y}_j}_{syn}, \Gamma^{\mathbf{y}_j}_{syn}, \leq^{\mathbf{y}_j}_{syn})$ 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
  $$M^{\mathbf{y},x}(v) = \begin{cases} \{x\} & \text{, if } v = v_0 \\ M^{\mathbf{y}_j,x}(v) & \text{, if } v \in V^{\mathbf{y}_j}_{syn} \\ \emptyset & \text{, otherwise} \end{cases}\text{, and}$$
  $T^{\mathbf{y},x}_{par} = (\mathbf{y} \to T^{\mathbf{y}_j,x}_{par})$.

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 $M^{\mathbf{y},\mathbf{x}}(v) = \bigcup_{x\in\mathbf{x}} \mathcal{M}^{\mathbf{y},x}(v)$.

A.2 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 data set, 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 $f_{\mathbf{u}}$.

A.3 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 $f_{\mathbf{v}}$.

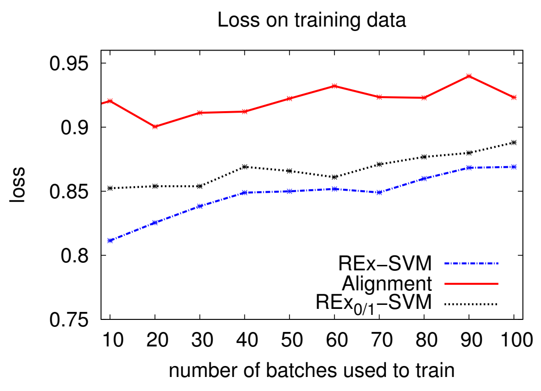
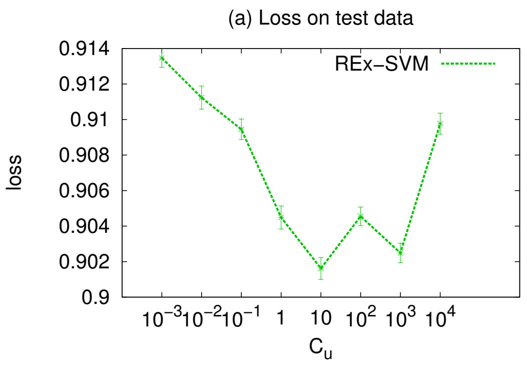


Figure 10: Average loss $\Delta_{\mathbf{u}}$ on training data for a varying number of training batches. Error bars indicate standard errors.

Figure 11 shows how the loss on the test data set 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 $f_{\mathbf{v}}$.



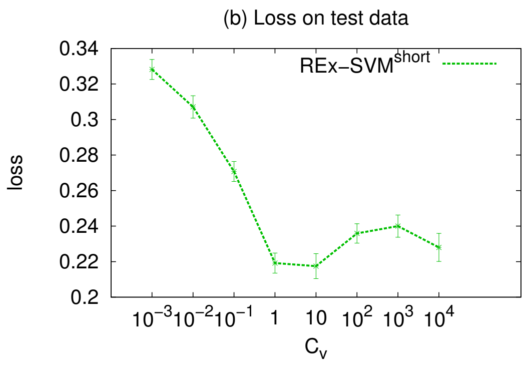


Figure 11: Average loss on test data for a varying regularization parameters $C_\mathbf{u}$ and $C_\mathbf{v}$ to train a model $\mathbf{f_u}$ (a) and a model $\mathbf{f_v}$, respectively. Error bars indicate standard errors.

A.4 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 greater 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 of alternatives

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

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

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

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

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.

Cascaded Malware Detection at Scale

Paul Prasse
University of Potsdam
Department of Computer Science
Potsdam, Germany
prasse@cs.uni-potsdam.de

Tobias Scheffer
University of Potsdam
Department of Computer Science
Potsdam, Germany
scheffer@cs.uni-potsdam.de

ABSTRACT

Code obfuscation techniques are an obstacle for static malware detection by statistical classifiers. Deobfuscation by partial code execution can overcome this problem, but increases the computational costs of malware detection by an order of magnitude. We study a cascaded architecture in which a classifier first performs a static analysis of the original code and—based on the outcome of this first classification step—the code may be deobfuscated and classified again. We conduct a large-scale empirical study of PHP and JavaScript malware detection in which we compare the cascaded method to static detection and to deobfuscation followed by classification. The study is based on a high-quality data set of almost 400,000 conspicuous PHP files and a collection of more than 1,000,000 JavaScript files. We explore several types of features and study the robustness of detection methods and feature types against the evolution of malware over time and develop a tool that scans very large file collections quickly and accurately.

1. INTRODUCTION

Malware—software that serves a malicious purpose—is affecting web servers, client computers via active web content, and client computers through executable files. Decisions whether files are deemed safe and allowed to be executed have to be made under somewhat differing requirements. Our primary motivation is the problem that web hosting services face: when a user creates a new file on a server, the hosting service has to decide whether the software is allowed to be executed on its server or distributed to connecting clients in the form of active content. Because of this main motivation, we focus on the two types of malware that are prevalent in hosting environments: PHP and JavaScript.

The execution of PHP malware on web servers is not only an abuse of valuable computational resources. The dissemination of malware via drive-by downloads as well as email spam campaigns, DDoS attacks, and click fraud initiated by server-based malware causes servers to become blacklisted and legitimate services to become unavailable. When a hosting service manages billions of files, the false-positive rate is naturally required to be extremely low. After each false-positive decision, a user account is blocked unnecessarily; the account then has to be scrutinized and unlocked again. Decisions have to be made in real time and, because of the high throughput of files, efficiently. In an environment with a high throughput of files, the CPU time spent on each decision is a crucial economic factor.

Browser-based malware detection in active content (such as JavaScript or JavaScript-bearing PDF files [14]) is a second application scenario in which JavaScript plays a dominant role and the efficiency of decision making is crucial.

Malware analysis techniques use static or dynamic features, or both. In fully dynamic analysis, the software is executed and observed for malicious behavior in a sandbox environment. Setting up a virtual operating environment, running, and observing the software typically requires minutes of CPU time. This would be impractical for our focused application environment as well as for our experimental study; we therefore exclude fully dynamic analysis from our investigation. By contrast, static analysis is based on features that can be extracted from the program file. Parameters of the decision criteria are often optimized using machine-learning techniques on collections of malware and non-malware files [12, 8]. Static analysis tools can be real-time capable.

In order to bypass detection mechanisms, polymorphic malware comes in an abundance of minor variations [10]. Consider that on average around *300,000 unique files per day* that are uploaded to the virustotal.com malware scanning service, classify as malware by at least one of the scanners that the service employs[1]. Code obfuscation techniques that revolve around the evaluation of dynamically generated strings allow malware engineers to package malicious content into patterns that may not have previously occurred in known malware. Such patterns may therefore be absent in data collections that are used to train statistical classifiers. Code obfuscation itself is not generally evidence of malicious intent—we find that over 35% of benign JavaScript and over 40% of benign PHP programs employ obfuscation, often as makeshift protection from code piracy. By evaluating the arguments of "eval", "document.write", "unescape" and similar statements until no such statements remain, the code can be deobfuscated; the deobfuscated code can be subjected to static code analysis [7]. However, malware engineers can make the deobfuscation arbitrarily complex by

© 2016 ACM. ISBN .
DOI: 10.1145/1235

[1]https://www.virustotal.com/en/statistics/

nesting “eval” expressions and performing expensive calculations within these expressions. We find that even the fastest available deobfuscation tools still require an order of magnitude more CPU time than static code analysis. Deobfuscation followed by static analysis covers a middle ground between fully dynamic and fully static code analysis.

Our paper makes the following contributions. (a) We develop a fast and robust cascaded malware-filtering method that uses fully static analysis for most decision and singles out a limited fraction of files for deobfuscation. However, whether this approach has any merit depends on whether fully static analysis will be able to make most decisions with near certainty. (b) We explore this method in a large-scale empirical study with hundreds of thousands of PHP and more than a million JavaScript files. We describe our methodology for obtaining a large collection of files with known malware status, survey the types of malware that are included, and make our data set available. We present an in-depth analysis of the most informative features and of the limitations of static and deobfuscated code analysis. (c) We investigate the effectiveness of linear classification models, of nonlinear random forest classifiers, and of several types of features for cascaded malware detection. Besides n-gram features [12], our study includes orthogonal-sparse-bigrams [19], syntax-tree features [7, 20], and subroutine-hashing features. We find that the combination of all feature types gives the best detection performance. (d) We present the first empirical study of the robustness of malware detection models over time. We study the rate at which various models deteriorate as malware evolves over time.

The rest of this paper is organized as follows. Section 2 describes the background of our work. Section 3 elaborates on the detection mechanisms and types of features that we study. Our empirical study is described in Section 4. Section 5 provides an in-depth analysis of the models, their abilities and limitations. Section 6 concludes.

2. BACKGROUND

This section briefly reviews the background; Gandotra et. al [8] give a more complete survey of malware detection.

Static analysis techniques extract features from the file under investigation. For script files—such as PHP and JavaScript—a disassembling step is not necessary for feature extracation. Given the script files, n-gram features [12], bag of tuples [4], syntax-tree features [7], or features of the control-flow graph [1, 5] or function-call graph [13] can be extracted. For active web content, URL and host features are additionally available [3].

Code obfuscation techniques pose a challenge for static analysis [15]; dynamic analysis techniques [2, 11] offer a potential remedy because they observe the software as it is evaluated in a controlled environment. In scripting languages, *eval unfolding*—executing code that is generated at runtime—is a common obfuscation technique and even a limited dynamic analysis can reveal informative features [7]. In order to perform a full dynamic analysis, a sandbox virtual environment has to be set up, system and registry changes, and networking behavior have to be monitored. An intrinsic limitation of dynamic analysis lies in the high costs of preparing the sandbox environment and observing the software’s behavior; the analysis cannot be carried out in real time, and is less easily scalable. The adversarial nature of the problem poses challenges for dynamic analysis, too. Malicious behavior may only be triggered at a specific time, under specific conditions, or via a remote command, and may therefore not be observed in the controlled environment.

Static and dynamic program analysis can be combined [1]. A wide range of supervised machine learning techniques has been studied for the task of combining the extracted features into a decision rule [18, 12]. Semi-supervised learning methods that exploit unlabeled data in addition to labeled training data [17] and unsupervised data analysis techniques that discover relationships between malware files have been studied. For instance, BitShred employs a feature hashing scheme and co-clustering based on the Jacard similarity metric to uncover relationships between malware samples [9].

Empirical investigations of malware detection have so far been made on a small to medium-sized scale, using hundreds [3, 7, 1] or thousands [18, 16, 12, 7] of malware files, often of a specific type of malware. The largest empirical malware study that we are aware of was carried out on 65,000 PDF files, some 16,000 of which have JavaScript embedded [14]. Most empirical investigations report accuracies of between 95 and 98%.

3. MALWARE DETECTION

This section introduces the malware detection cascade as well as the baselines that perform fully static analysis and deobfuscation followed by static analysis. This section also describes the types of features that we will investigate in the following study.

3.1 Malware Detection Cascade

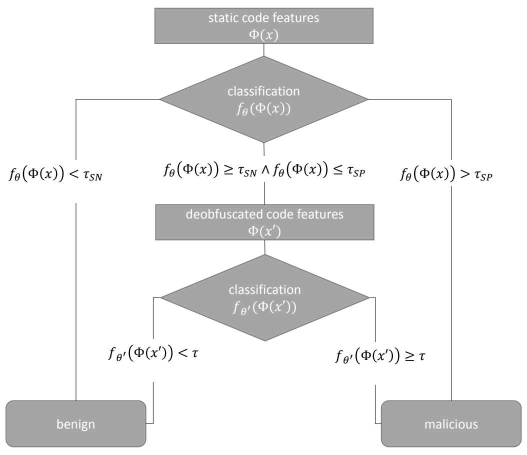
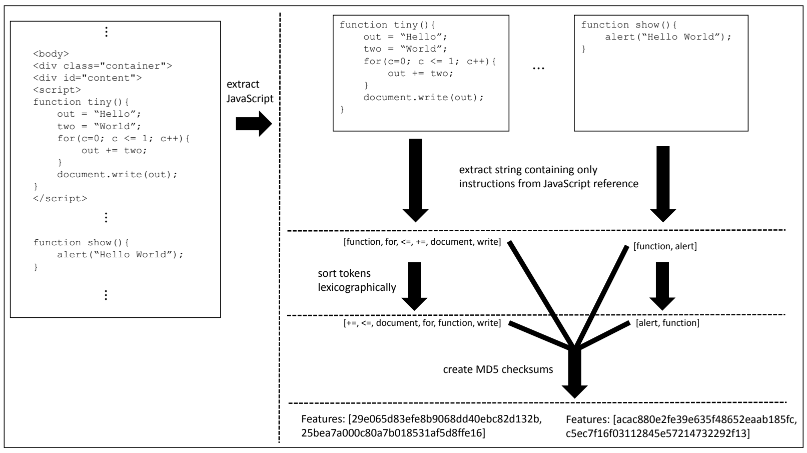

static code features $\Phi(x)$
classification $f_\theta(\Phi(x))$
$f_\theta(\Phi(x)) < \tau_{SN}$
$f_\theta(\Phi(x)) \geq \tau_{SN} \wedge f_\theta(\Phi(x)) \leq \tau_{SP}$
$f_\theta(\Phi(x)) > \tau_{SP}$
deobfuscated code features $\Phi(x')$
classification $f_{\theta'}(\Phi(x'))$
$f_{\theta'}(\Phi(x')) < \tau$
$f_{\theta'}(\Phi(x')) \geq \tau$
benign
malicious

Figure 3: Cascaded malware detection architecture

Figure 3 shows an abstract overview of the detection cascade. In a first step, a high-dimensional vector of static code features $\Phi(x)$ is extracted from file x—the following sections will elaborate on these features. A statistical classifier f_θ determines a malware detection score $f_\theta(\Phi(x))$ based on this vector. This score is compared against two thresholds: if $f_\theta(\Phi(x)) < \tau_{SN}$, then the file is considered to be a “safe negative” and receives the label *benign*. Threshold τ_{SN} has to be adjusted on a test data collection such that the fraction of malware for which the detection score of the static detection model $f_\theta(\Phi(x))$ does not exceed τ_{SN} stays below a *false-negative rate* that is deemed acceptable. If



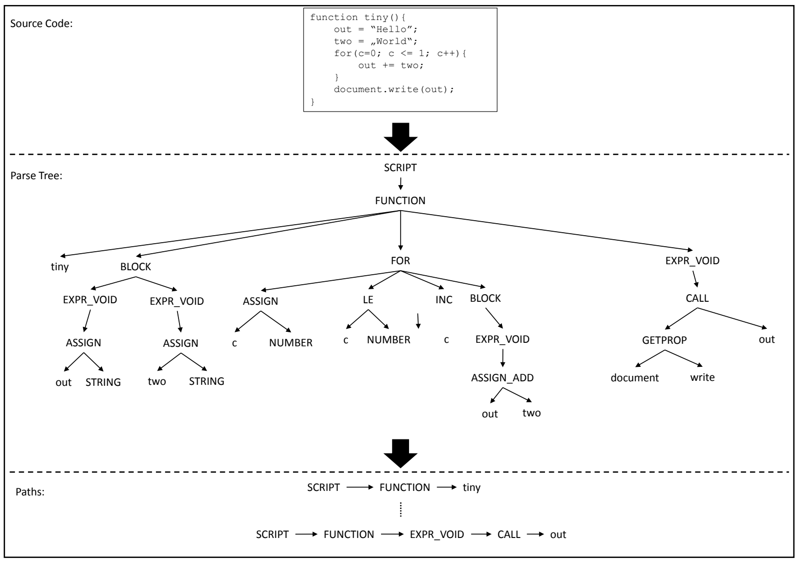
```
⋮
<body>
<div class="container">
<div id="content">
<script>
function tiny(){
    out = "Hello";
    two = "World";
    for(c=0; c <= 1; c++){
        out += two;
    }
    document.write(out);
}
</script>
⋮
function show(){
    alert("Hello World");
}
⋮
```

```
function tiny(){
    out = "Hello";
    two = "World";
    for(c=0; c <= 1; c++){
        out += two;
    }
    document.write(out);
}
```

```
function show(){
    alert("Hello World");
}
```

extract JavaScript
...
extract string containing only instructions from JavaScript reference
[function, for, <=, +=, document, write]
[function, alert]
sort tokens lexicographically
[+=, <=, document, for, function, write]
[alert, function]
create MD5 checksums
Features: [29e065d83efe8b9068dd40ebc82d132b, 25bea7a000c80a7b018531af5d8ffe16]
Features: [acac880e2fe39e635f48652eaab185fc, c5ec7f16f03112845e57214732292f13]

Figure 1: Extraction of function-hashing features



```
function tiny(){
    out = "Hello";
    two = „World";
    for(c=0; c <= 1; c++){
        out += two;
    }
    document.write(out);
}
```

Figure 2: Extraction of parse-tree features

$f_\theta(\Phi(x)) > \tau_{SP}$, the file is immediately classified as *malicious.* Threshold τ_{SP} has to be adjusted such that the rate of benign files for which $f_\theta(\Phi(x))$ exceeds τ_{SP} stays below an acceptable *false-positive rate.*

If the score falls into the interval of τ_{SN} and τ_{SP}, the code is deobfuscated. The unfolded code—the code that results from evaluating the arguments of "eval" and "unescape" statements—is appended to the original code which results in a deobfuscated file x'. The partial code evaluation can be arbitrarily complex or time consuming. Therefore, we terminate the unfolding process when a hard limit of 10 CPU seconds is reached. The same set of features $\Phi(x')$ is extracted from the deobfuscated file; a second classification model $f_{\theta'}$ processes the feature vector, and the final classification result is obtained by comparing $f_{\theta'}(\Phi(x'))$ against a decision threshold τ. We define the *aggregate score* of the cascaded model as $f_\theta(\Phi(x))$ if x is classified on the first level, and $f_{\theta'}(\Phi(x'))$ if it is classified on the second level.

The malware detection cascade can be employed as an automatic blacklisting mechanism, as an automatic whitelisting mechanism, or as an interactive tool. A blacklisting mechanism identifies malware that can be disabled automatically; here, the false-positive rate has to be extremely low, and therefore the aggregate score has to be compared against a high threshold. A whitelisting mechanism identifies benign files that require no further attention. Here, the false-negative rate has to be very low and therefore the aggregate score has to be compared against a low threshold. When the model is applied as an interactive tool, it automatically disables files whose aggregate score exceeds a blacklisting threshold, marks files whose aggregate score lies below a whitelisting threshold as safe, and tags files whose score lies between the whitelisting and blacklisting thresholds as requiring manual inspection.

For PHP, we use a deobfuscation tool that has been created and is used by an engineer who is in charge of malware detection for a web hosting service. In order to deobfuscate JavaScript files, we use the SpiderMonkey package by Didier

Table 1: Number of features

Feature type	PHP	JavaScript
n-grams	815,866	1,174,378
OSB	2,786,283	1,751,993
Function hashes	34,743	125,810
Parse-tree features	125,436	724,557

Stevens[2] which is a modified version of Mozilla's C implementation of JavaScript[3]. We use this version because it is the fastest implementation that also exhibits the best ability to unpack and deobfuscate JavaScript on our data set. Other tools are for example Creme Brulee[4], JSDebug[5] and SpiderMonkey + V8[6].

3.2 Reference Methods

In addition to the *cascaded* model, our experimental study will include a *static* model $f_\theta(\Phi(x))$ in which the decision is always based on static features as well as a *deobfuscation*-based model in which the code is always deobfuscated first and the decision function $f_{\theta'}(\Phi(x'))$ is always based on static features of the deobfuscated file x'.

3.3 Classification Methods

In our study, we explore a logistic regression classifier and a random forest classifier.

Logistic regression is a linear classifier that computes the product of an input feature vector and its parameter vector θ; it uses a logistic function to squash the result of the vector product into a normalized output probability $f_\theta(\Phi(x)) = \frac{1}{1+e^{\theta^\top \Phi(x)}}$.

We use the Liblinear[7] implementation of logistic regression that conducts a stochastic gradient search for model parameters θ that minimize the ℓ_2-regularized logistic loss on the training set. We train the static model f_θ on a training set that contains the static feature vectors $\Phi(x)$ and the class label (*benign* or *malware*) for all files in the PHP or JavaScript collection, respectively. Deobfuscated model $f_{\theta'}$ is trained using features $\Phi(x')$ extracted from the deobfuscated versions x' of all files from the PHP or Javascript collections.

The random forest classifier is a nonlinear model that uses an ensemble of decision trees to determine the class label. Each of the trees is trained on a bootstrapped sample of the training data using only a randomly drawn subset of the static features in $\Phi(x)$ or the features $\Phi(x')$ of the deobfuscated file, respectively. The proportion of trees that vote for the positive class serves as decision-function value $f_\theta(\Phi(x))$. We use the random forest implementation of the scikit-learn library with an ensemble of 1,000 trees.

3.4 N-Gram Features

Token n-grams are overlapping sequences of n subsequent tokens. The n-gram feature representation of a file is a sparse, high-dimensional vector that has an entry for any 1- to n-gram that occurs at least twice in the training set of files. In order to extract n-gram features, the file is first tokenized; all characters that are not a letter or a digit act as separators. Then, sliding windows of size 1 to n pass over the token sequence, and the vector element that corresponds to the n-grams under the windows is incremented. In our experiments, we limit n to 5; this results in 815,866 non-unique features on the *PHP* data set and 1,174,378 features on the *JavaScript* data set that occur in the data in at least two different instances (see Table 1). An excerpt of features extracted from the *PHP* and *JavaScript* data sets is shown in Table 8, respectively.

3.5 Orthogonal Sparse Bigrams

Orthogonal sparse bigrams (OSB) can be thought of as n-grams with a wildcard symbol; they have proven to be effective for email spam classification [19]. A generalization of this approach has been investigated for malware detection [4]. An orthogonal sparse bigram is an n-gram in which the first and the n-th token are fixed, and $n-2$ tokens in between are replaced by wildcards. In order to extract these features, windows of sizes 1 through n slide over the tokenized file. For each window position, the vector entry that corresponds to the 1- to n-grams in which the $n-2$ central tokens are replaced by a wildcard symbol are incremented. Again, we limit n to 5; this results in 2,786,283 non-unique features for the *PHP* and 1,751,993 non-unique features for the *JavaScript* data sets (see Table 1). An excerpt of features extracted from the *PHP* and *JavaScript* data sets is shown in Table 8, respectively.

3.6 Function Hashes

Function and subroutine-hashing features can be thought of as robust signatures of individual functions. Figure 1 shows the feature extraction process. All PHP functions are directly extracted from the file; JavaScript functions are extracted from the HTML code. A sequence of tokens is extracted from each function, but only those tokens are kept that occur in a language reference. We use a PHP reference[8] and the a JavaScript reference[9] to extract the sequence of tokens for PHP and JavaScript files, respectively. The resulting token sequence contains reserved words and library functions, but since it does not contain any identifiers, the resulting function-hashing features are invariant with respect to the naming of variables. An MD5 hash is calculated for this token sequence; in addition, the sequence is sorted alphabetically and a second MD5 hash is calculated. Each function is therefore represented by two hashing features.

The function-hashing feature vector contains an entry for each MD5 hash that occurs at least twice in the training data. Each file is represented as a sparse vector that is summed over all containing functions and their associated MD5 hashs. This procedure results in 125,810 non-unique features for the *JavaScript* data set and 34,743 non-unique features for the *PHP* data set (see Table 1).

3.7 Parse-Tree Features

Parse-tree features are calculated as illustrated in Figure 2. We use the PHP deobfuscation tool to parse PHP

[2]https://didierstevens.com/files/software/js-1.7.0-mod-b.zip
[3]http://www.mozilla.org/js/spidermonkey/
[4]http://code.google.com/p/cremebrulee/
[5]http://codeproject.com/KB/scripting/hostilejsdebug.aspx
[6]http://code.google.com/p/v8/
[7]http://www.csie.ntu.edu.tw/~cjlin/liblinear/
[8]http://www.phpbox.de/php_befehle/
[9]http://wiki.selfhtml.org/wiki/Referenz:JavaScript

Table 2: Prevalence of types of malware in the *JavaScript* collection

Malware type	occurrences	percentage
HideLink	188,104	37%
Iframe	88,835	17%
Includer	54,237	10%
Dropper	40,657	8%
Redirector	35,129	7%
Clickjack	30,124	6%
Agent	21,733	4%
Autolike	15,668	3%
Decode	12,737	2%
Script	5,317	1%

and the Spidermonkey[10] parser to create the parse tree for JavaScript. We treat all paths in the syntax tree to any leaf node as a sequence of tokens, and extract all n-gram features of these sequences. Each n-gram of non-terminal and terminal symbols in a branch of the parse tree that occurs at least twice in the training data is represented by an element of the feature vector. Again, we limit n to 5; this results in 125,436 non-unique features for the *PHP* and 724,557 non-unique features for the *JavaScript* data set data set (see Table 1).

4. EMPIRICAL STUDY

This section describes the data sets that we collect, performance metrics, and the feature subset selection method that we employ. We then report on the malware detection performance of all methods and types of features under investigation, and explore the robustness of all models against the evolution of malware over time.

4.1 Data Collections

The *PHP* data set contains 397,395 conspicuous PHP files that have come under scrutiny of administrators of a web hosting service. These difficult cases are found by handcrafted rules and signatures that were created to detect files which shows suspicious behavior (such as a high volume of outgoing CGI emails or HTTP requests), or because the script has some detectable connection to known software vulnerabilities. Of these conspicuous cases, 392,276 are in fact malware and 5,119 turn out to be benign after detailed analysis. PHP malware includes, for instance, email spam dissemination tools, and software that dynamically generates JavaScript malware which is then used for drive-by download attacks. The *PHP* collection contains relatively few benign files, and since only suspicious files are scrutinzied, all negative examples are difficult cases. Therefore, it is difficult to infer a meaningful and accurate false-positive rate on this collection. For this reason, we additionally collect an *auxiliary PHP collection* of 23,735 benign PHP files by downloading 17 popular content-management systems implemented in PHP.

We determine the proportion of obfuscated files among benign and malicious PHP scripts. Obfuscation is even more prevalent for PHP; we find that 42% of all benign and 99% of all malicious PHP files execute code which is constructed at runtime (see Table 3).

[10]https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

Table 3: Number of obfuscated files

Class label	PHP	JavaScript
benign	12,219 (42%)	161,392 (35%)
malicious	388,353 (99%)	435,833 (80%)

We create a *JavaScript* data set as follows. The virustotal.com website provides a malware scanning service; uploaded files are scanned with 57 virus and malware tools. We collect 544,792 JavaScript files that have been classified as malware by at least 5 of the 57 scanning tools. In order to collect benign JavaScript files, we identify the one million most popular websites using traffic data from alexa.com. We crawl these websites and use virustotal.com to check all files that contain JavaScript for possible malware. This results in 457,333 files that are not recognized as malware by any of 57 virus scanners. A Python script that recreates this *JavaScript* data set is included in the supplementary material and will be uploaded to GitHub.

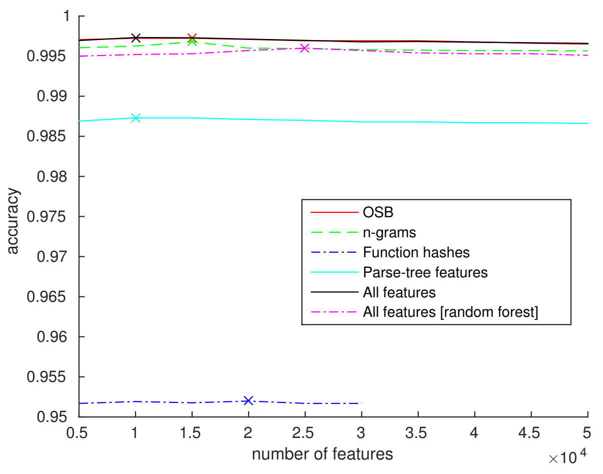
We determine the proportion of obfuscated files among benign an malicious JavaScript files. We find that 35% of all benign and 80% of all malicious files execute code which is constructed at runtime (see Table 3). Table 2 shows the prevalence of different types of malware. The types correspond to the malware classification of antivirus tools; malwarefixes.com provides a detailed explanation of these types. We can conclude that the *JavaScript* collection reflects the full range of malware types that are in circulation on the internet.

4.2 Performance Metrics

In the following experiments, we will quantify the accuracy of various models with a number of performance metrics. First, we measure the classification *accuracy* of the classifiers with the default classification threshold of $\tau = \frac{1}{2}$. The *true-positive rate*—also called *recall*—is the proportion of malware that is recognized as malicious. The *false-positive rate* is the proportion of benign files among those files that are classified as malware. Conversely, the *true-negative rate* is the proportion of benign software that is recognized as benign, and the *false-negative rate* is the proportion of malware that is not detected by a classifier.

In order to evaluate the performance of a model as a blacklisting mechanism, we will determine the *recall at a given false-positive rate*. To this end, we adjust the classification threshold τ such that the desired false-positive rate is obtained on the test data, and measure the recall (true-positive rate, rate of malicious files that are blacklisted) of the classifier with this classification threshold. In order to evaluate the performance of a model as a whitelisting mechanism we measure the *true-negative rate at false-negative rate zero*. Here, we set the classification threshold below the lowest score of any malware example (*i.e.,* no malware is whitelisted) and measure the resulting true-negative rate (the rate of benign files that are whitelisted).

We will determine ROC curves and the area under the ROC curve. ROC curves detail the possible trade-offs between the false-positive rate and the true positive rate that can be attained by varying the classification threshold τ. Each point on a ROC curve corresponds to a classifier with a fixed classification threshold. By increasing the threshold,



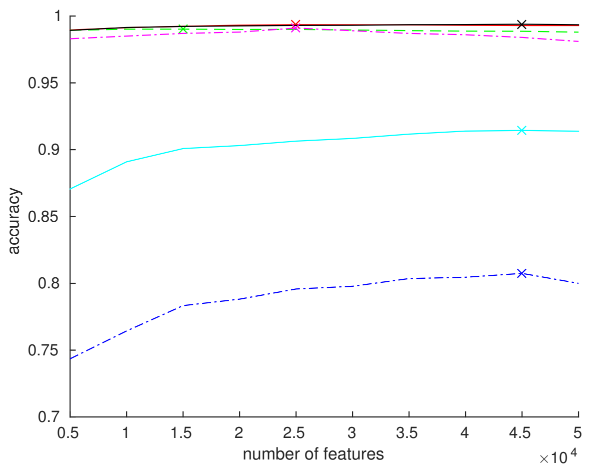


Figure 4: Accuracy over the number of features of each type. Left: *PHP*; right: *JavaScript*. Points with maximal accuracy are marked with a cross

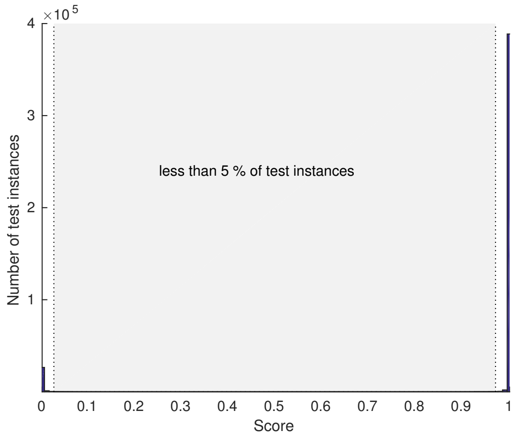


Figure 5: Histogram of scores of the static detection model for PHP

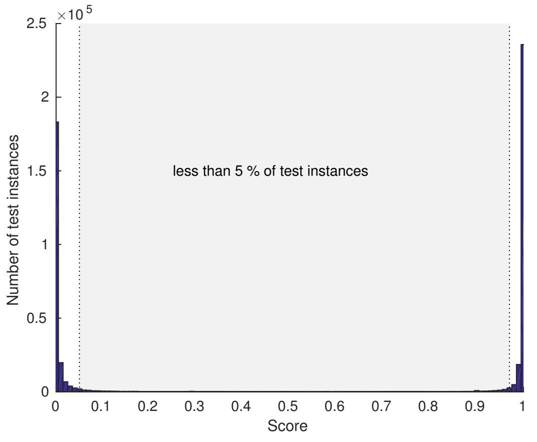


Figure 6: Histogram of scores of the static detection model for JavaScript

one moves down and to the left (lower false-positive and true-positive rates), by decreasing it one moves up and to the right (higher false-positive and true-positive rates). The area under the ROC curve (AUC) is an aggregate performance measure of the decision function, independent of the classification threshold. It is equal to the probability that a randomly drawn positive instance (malware) will have a higher decision-function value than a randomly drawn negative instance (benign file).

4.3 Learning Methods and Feature Selection

Table 1 show the total number of features that the extraction process creates for the two data collections. These features are most likely not equally discriminative. Since the sample size is still relativerly small compared to the number of features, the overall accuracy of the final classifier may be higher when only a subset of the features is used. This section reports on experiments that aim at determining the optimal subset of features of each type, and the optimal subset of the joint set of all features.

We increment the number of features k in steps of 5,000. For each number k, we run an outer 2-fold cross-validation loop. In each iteration, first a logistic regression model is trained on the training part of the data; the model coefficients provide a ranking of all features. We then use the k features with highest model coefficients. In order to train a model using these k features, the regularization parameter C of the logistic regression has to be tuned. To this end, we run an inner loop of 2-fold cross validation on the training data of the outer cross-validation loop. We set C to 10^i for i from -4 to 4, train a model on the inner training set, evaluate the model on the inner test set, then set C to the optimal value and train a model on the entire training portion of the outer loop. This model is evaluated on the test portion of the outer cross-validation loop. We repeat this experiment for each type of features, and for the entirety of all features.

Figure 4 shows that the optimal number of features lies between 10,000 and 45,000, depending on the data set and

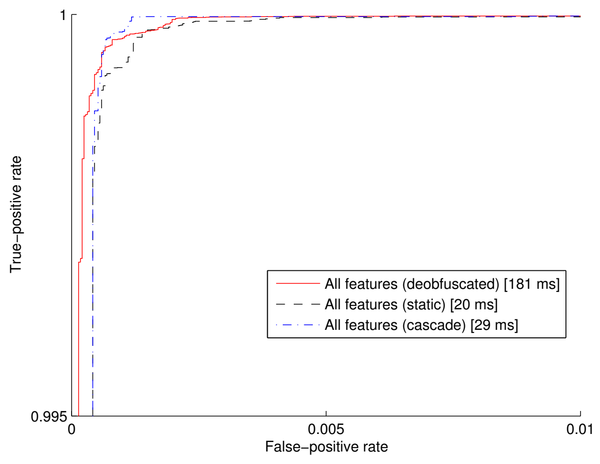


Figure 7: ROC curves for PHP malware

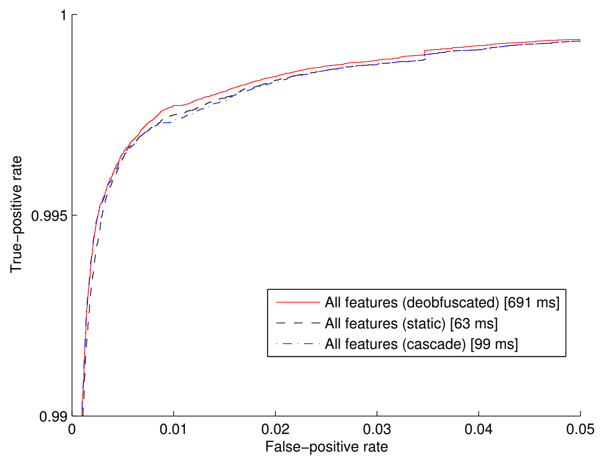


Figure 9: ROC curves for JavaScript malware

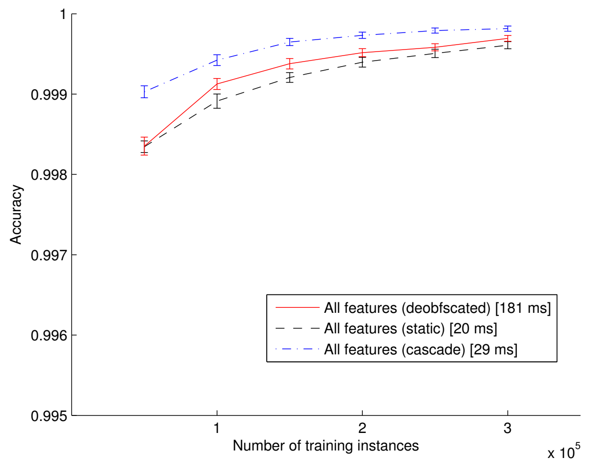


Figure 8: Accuracy over varying number of instances for *PHP* data set. Error bars show the standard error

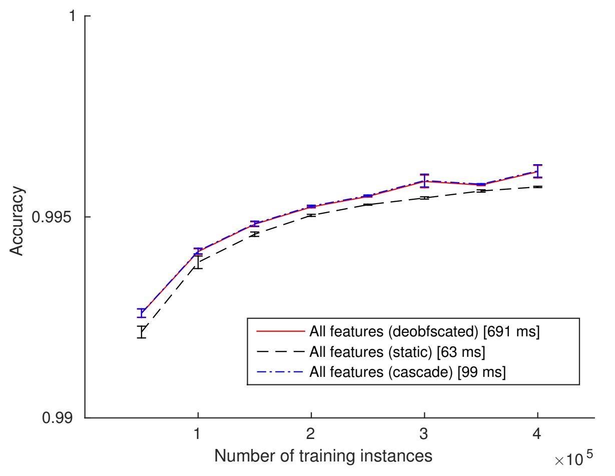


Figure 10: Accuracy over varying number of training instances for JavaScript. Error bars show the standard error

feature type. For each feature type and data set, we use these numbers of features for our subsequent experiments. Increasing the number of features further deteriorates the classification performance slowly.

Figure 4 also compares the performance of the logistic regression to the performance of the random forest classifier. The hyperparameters of the learning algorithm are tuned using the same nested 2-fold cross-validation protocol. The random forest performs slightly worse than logistic regression. Here, the feature space is sufficiently large, so that a non-linear random forest has no advantage over a linear classification model. We therefore exclude the random forest from subsequent experiments.

4.4 Thresholds for Cascaded Detector

This section discusses the choice of thresholds τ_{SN} and τ_{SP} which determine whether an instance x, after classification based on static features $\Phi(x)$, is either immediately classified as benign or malicious, or else is classified again after deobfuscation.

For PHP, we have a relatively small number of benign files in the collection of conspicuous files that have come under scrutiny of an administrator, which makes it more difficult to estimate small false-positive rates accurately. Therefore, we adjust τ_{SN} such that the false-positive rate on the difficult cases is 10^{-3}, and the false-positive rate on the auxiliary PHP collection that contains files of standard content-management systems is zero. We adjust τ_{SP} to a false-negative rate of 10^{-4}. Figure 5 shows the histogram of scores of the static classifier for PHP in which the resulting threshold values are highlighted. We observe a strongly bimodal distribution for the static classifier; less than 5% of all files have to be passed on the the deobfuscation and subsequent analysis stage.

For JavaScript, we adjust τ_{SN} such that the static model $f_\theta(\Phi(x))$ attains a false-positive rate of 10^{-4}, and adjust τ_{SP} such that it attains a false-negative rate of 10^{-4}. Figure 6 shows the histogram of scores of the static classifier

Table 4: *PHP*: Accuracy, AUC with 95% confidence intervals; malware recall at a false-positive rate of 0.1%, false-positive rate on the *auxiliary PHP data*, and true-negative rate at a false-negative rate of 0

Feature type	Accuracy	AUC	Recall@FPR 10^{-3}	FPR (aux)	TNR@FNR0
n-grams (static)	0.9949 ± 0.07198	0.9935 ± 0.00276	0.8794 ± 0.01567	0.00007	0.8359
OSB (static)	0.9957 ± 0.00045	0.9955 ± 0.00173	0.8967 ± 0.01678	0.00003	0.8605
Function hashes (static)	0.9665 ± 0.00103	0.9913 ± 0.00090	0.8590 ± 0.03633	0.00011	0.7435
All features (static) [20 ms]	0.9996 ± 0.00004	0.9999 ± 0.00004	0.9825 ± 0.0337	**0**	0.9833
All (deobfuscated) [181 ms]	**0.9998 ± 0.00003**	**0.9999 ± 0.00001**	**0.9996 ± 0.0002**	**0**	0.9895
All (cascade) [29 ms]	**0.9998 ± 0.00003**	0.99997 ± 0.00001	0.9983 ± 0.0033	**0**	**0.9986**

Table 5: *JavaScript*: Accuracy, AUC, malware recall rate at a false-positive rate of 10^{-4}

Feature type	Accuracy	AUC	Recall @ FPR 10^{-4}
n-grams	0.9923	0.9981	0.8734
OSB	0.99596	0.9989	0.9023
Function hashes	0.88852	0.8968	0.4579
Parse-tree features	0.94068	0.9852	0.4949
All features (static) [63 ms]	0.99592	**0.9998**	0.9423
All features (deobfuscated) [691 ms]	0.99607	**0.9998**	**0.9525**
All features (cascade) [99 ms]	**0.9961**	**0.9998**	0.95

for JavaScript in which the resulting threshold values are highlighted. As for PHP we observe a strongly bimodal distribution for the static classifier; less than 5% of all files have to be passed on the the deobfuscation and subsequent analysis stage.

We keep τ_{SN} and τ_{SP} fixed and vary the classification threshold τ of the second stage in the following experiments. The baselines have only one threshold which we vary accordingly in the following experiments.

4.5 Malware Detection Performance

This section explores the accuracy, false-positive, and false-negative rates that can be attained using different types of features.

4.5.1 PHP Collection

For this data set, we run an outer loop of 20-fold cross validation. In each iteration, we run an inner loop of 2-fold cross validation on the training portion of the outer loop in order to tune the regularization parameter C. When C is set, we train a model on the training portion of the outer cross-validation loop, evaluate the model on the test portion of the cross-validation loop, and reiterate.

Figure 7 shows a zoomed-in view of the ROC curve of all models; the second and third columns of Table 4 show the accuracy (for a classification threshold of $\tau = \frac{1}{2}$) and the area under the ROC curve (AUC) for for all models and feature types. The columns also specify 95%-confidence intervals based on a two-sided t-test. Here, the cascaded model and the baseline that deobfuscates all files perform equally well in terms of accuracy. The deobfuscated model has a marginally (but statistically insignificantly) higher AUC. The static model performs (insignificantly) worse both in terms of accuracy and AUC. Notably, the cascaded model has an average execution time of 29 ms, compared to 20 ms for the static model and 181 ms for the model that deobfuscates all files.

In terms of the feature sets, OSB features are the single best type, followed by n-gram features; the combination of all features leads to a significantly higher accuracy, AUC, and recall at a false-positive rate of 10^{-3}, based on a paired t test with $p < 0.05$.

We measure the performance of the models as an automatic blacklisting mechanism. We set the classification threshold value such that the false-positive rate for the *PHP* collection of hard, conspicuous cases is 10^{-3}. We observe that the cascaded model has a significantly ($p < 0.05$) higher recall than the static model; the deobfuscated model has a marginally (statistically insignificantly) higher recall than the cascaded model. Column 5 of Table 4 measures the false-positive rate on the *auxiliary PHP* collection that contains PHP files of standard content-management systems, using the same classification threshold. For the combined feature set, we observe a false-positive rate of zero for the static, cascaded, and deobfuscated models.

We now measure the performance of the model as a whitelisting mechanism. We set the classification threshold such that all malware files are recognized as malicious. Column 6 of Table 4 measures the resulting true-negative rate. The cascaded model with combined feature set has a significantly ($p < 0.05$) higher true-negative rate than the static and the deobfuscated models; it can whitelist 99.86% of the conspicuous PHP files without missing any malware files. When the cascaded model is applied as both a blacklisting and a whitelisting mechanism, it blacklists 99.83% of all malware in real-time, whitelists 99.86% of all benign files in real-time, marks the remaining files for manual inspection.

4.5.2 JavaScript Collection

Here, we use a single training-and-test split with half of the data a training and the other half as test data. We run a 10-fold cross-validation loop on the training part of the data to tune regularization parameter C. We then train a model on the entire training data and evaluate it on the test data.

Figure 9 shows a zoomed-in view of the ROC curve for small false-positive rates; we see that the deobfuscation model performs marginally better for small false-positive values; however, Column 3 of Table 5 shows that the cascade

Table 6: Execution time of feature extraction and classification with 95% confidence intervals in ms

Feature Type	PHP	JavaScript
n-grams	8 ± 2.15	26 ± 1.26
OSB	9 ± 2.31	32 ± 1.15
Function hashes	6 ± 3.650	34 ± 2.64
Parse-tree features	5 ± 0.98	22 ± 1.11
Deobfuscation	166 ± 16.00	639 ± 40.65
All features (static)	20 ± 4.392	63 ± 7.32
All features (deobfuscated)	181 ± 18.45	691 ± 37.17
All features (cascade)	29 ± 5.07	99 ± 9.10

and the deobfuscation model have identical AUC values and the cascaded model has an insignificantly higher accuracy. The static model has a lower accuracy than the other models but an identical AUC. Notably, the cascaded model has an execution time of 99 ms compared to 63 ms for the static and 691 ms for the deobfuscated model.

For JavaScript, orthogonal sparse bigrams are the best single feature type. They perform strictly better than n-gram features. However, the combination of all features offers a significantly ($p < 0.05$) higher AUC and recall at a false-positive rate of 10^{-4}, and an insignificantly higher accuracy. We evaluate the malware detector as a blacklisting mechanism. Table 5 shows that the deobfuscaded model has an insignificantly higher recall at a false-positive rate of 10^{-4} than the cascaded model; the static model has a significantly ($p < 0.05$) lower recall.

The classification accuracy of 0.9961 exceeds values that have been reported in prior work [8]. For Zozzle, a false-negative rate of 91% at a false-positive rate of 3×10^{-6} has been reported. However, two factors should be noted. Firstly, for this data set, we classify all files as benign that are not recognized as malware by any of 57 commercial malware detection tools. The set of false-positives contains an unknown proportion of actual malware that is not yet recognized by any malware tool. Secondly, it should be noted that Zozzle has been trained and evaluated on a focused set of 919 malicious JavaScript files that attempt a heap-spraying attack, whether our JavaScript collection contains hundreds of thousands of malware files that cover a wider range of attack mechanisms (see table 2). Other related systems have been trained and evaluated on 823 [6] malicious JavaScript files or on 15,331 malicious and 908 benign JavaScript files (which makes it impossible to measure false-positive rates below 0.1%) [14].

4.6 Number of Training Instances

This section studies the impact that the number of training instances has on the performance of the static, cascaded, and deobfuscated models. Figure 8 shows the learning curve for *PHP*; Figure 10 shows the learning curves for *JavaScript*. We conclude that all models can benefit from additional training data. With growing sample size, the difference between static, cascaded, and deobfuscated detection decreases.

4.7 Comparison to Antivirus Products

Classifiers f_θ and $f_{\theta'}$ has been trained to recognize files that at least 5 of 57 antivirus products recognize as malware, respectively. Figure 11 compares the recall of the fully static,

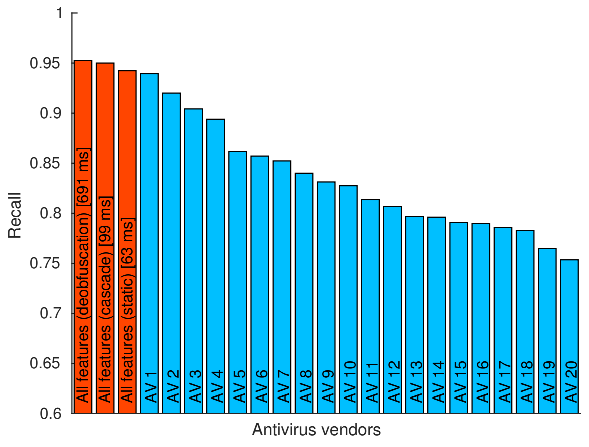


Figure 11: Comparison with antivirus products for JavaScript: recall at a false-positve-rate of 0.01%

cascaded, and deobfuscation model at a false-positive rate of 0.01% to the recall of the 20 best (in terms of recall) of these antivirus products. We cannot draw definitive conclusions from this comparison because we cannot precisely determine the false-positive rate of these products. For PHP, the highest recall of our tested antivirus tools is 17%.

4.8 Robustness against Malware Evolution

This section explores the robustness of the various models and types of features against the evolution of malware over time. In order to visualize the evolution of malware, we perform a principal component analysis on the static features $\Phi(x)$ of files x and map the JavaScript data into a 10-dimensional space. We train a malware classifier in this space, and take the two dimensions that have the highest model coefficients trained on all 10 principal components. This gives us a two-dimensional space that is highly relevant for the classification of JavaScript files as malware. Figure 12 visualizes malware that has been submitted in two different time periods in different colors. The nonstationarity of the distribution is clearly visible; this observation calls into question whether a classifier that has been trained at some point can maintain a high level of accuracy for a long time.

To investigate this question, we use 167,870 malware JavaScript files that have been submitted to virustotal.com before February 1, 2014, as positive examples. We use equally many randomly drawn benign files as negative examples—we assume that the distribution of benign files does not change substantially. We stratify 376,922 malware files that have been submitted from February 2015 to August 2015 into intervals of one month. Figure 13 shows the development of the recall rate of the static, cascaded, and deobfuscated model from February 2015 to August 2015. Table 7 summarizes the decay rates. Most notably, the static classifier deteriorates sharply in February and March, caused by a wave of new malware which it fails to recognize. Overall, the recall of the static classifier deteriorates to 99% of the initial value over the course of seven months, whereas the cascaded classifier deteriorates only to 99.71% of its initial recall and the model that deobfuscates all files deteriorates to 99.75%. We can conclude that the deterioration of the

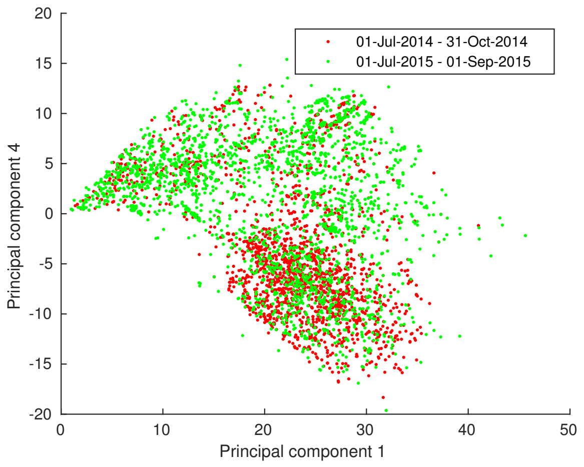


Figure 12: Evolution of JavaScript malware over time

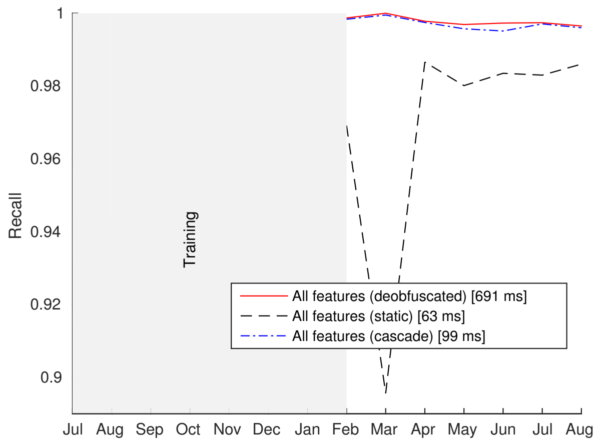


Figure 13: JavaScript malware recall evaluated into the future

cascaded and deobfuscation models over time is very mild whereas the fully static model behaves less robustly.

4.9 Execution Time

This section studies the computational costs of extracting the various types of features, the costs of deobfuscation, and the resulting execution time of all studied models.

Table 6 shows the detailed execution times on an i7 CPU at 2 GHz. The logistic regression classifier only calculates an inner product of feature and parameter vector; the execution time of the static classifier (20 ms for PHP and 63 ms for JavaScript) is therefore dominated by the costs of feature extraction. Note that the cost of calculating all features is lower than the sum of costs for individual feature types because multiple executions of some calculations can be saved when all features are calculated at the same time. Deobfuscation takes 166 ms for PHP and 639 ms for JavaScript and on average. The execution time of the deobfuscated model (166 ms for PHP and 691 ms for JavaScript) is therefore dominated by the costs of partial code execution. For the cascaded model, the execution time of 29 ms for PHP and 99 ms for JavaScript is 45% higher for PHP and 57% higher for JavaScript than the execution time of the fully static model, whereas the execution time of the model that deobfuscates all files is 9 times and 11 times higher, respectively.

5. DETAILED ANALYSIS OF MODELS

This section provides a detailed analysis of the features that have the highest influence on the classification result, and of the limitations of the models.

5.1 Most Influential Features

We analyze which types of features (n-gram, OSB, parse-tree, function hashes) are present in the 1% of features that have the highest coefficients θ_i and θ'_i in the static and deobfuscated models. We observe that for the static model and the *JavaScript* data set, text features are most important; the top 1% consists of 45% OSB-, 43% n-gram-, 9% parse-tree- and 3% function hash features. We observe a similar result for the deobfuscated model θ'; here, the top 1% features include 47% OSB-, 44% n-gram-, 8% parse-tree- and 1% function hash features.

For the *PHP* data set we observe that the proportion of OSB-features within the 1% most influential features is much higher. In the static model, 92% of the top features are OSB features; only 6% of the features are n-grams and both parse-tree and function hash features are insignificant (1%). For the deobfuscated model, the most influential features consist of OSB-features (91%) followed by n-gram features (7%). Once more, the function hash and parse-tree features have the smallest impact (1% each).

Table 8 enumerates the features that have the highest model coefficients for the static and deobfuscated models and for JavaScript and PHP data set. We observe that the tokens “fromcharcode”, “eval”, and “unescape” provide the strongest evidence for malware, and yet they occur remarkably frequentlly in benign files—code obfuscation is no evidence for malicious intent. We also see that iframes—in particular “iframe” tokens that become visible only after deobfuscation—are evidence for malicious intent. Most malicious tokens fall into the categories *masking* (“base64”, “decode”, “valueexporter”, “htmlspecialchars”), *email* (“mailboxes”), and *system* (“directories”, “sql_refresh”). Tokens with the most negative coefficients provide evidence that a file is benign. The features with the lowest coefficients do in fact look harmless but heterogeneous.

5.2 Limitations of the Models

This section analyzes the models’ malware-detection abilities and their limitations.

The static model is primarily constrained by its inability to observe token sequences that are generated dynamically and executed. Given the high prevalence of obfuscated code, it is rather surprising that the static model recognizes 98% of all PHP and 94% of all JavaScript malware. We investigate the model parameters and feature vectors of a number of obfuscated malware files to understand how the static model can recognize obfuscated files as malware. We find that n-grams of hexadecimal numbers that encode known malicious URLs have a (low) positive weight in the model parameter vector. For instance, the n-gram “%70%68%70” that en-

Table 7: Recall over time and recall decay rates

Feature type	Jul - Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	decay rate
All features (static) [63 ms]	0.9958	0.9690	0.8956	0.9864	0.9800	0.9832	0.9830	0.9858	0.9900
All (deobfuscated) [692 ms]	0.9990	0.9987	1	0.9978	0.9969	0.9973	0.9974	0.9965	**0.9975**
All (cascade) [99 ms]	**0.9990**	0.9984	0.9995	0.9975	0.9957	0.9952	0.9971	0.9961	0.9971

Table 8: Most significant OSB and n-gram features of the static and deobfuscated models. P_B is the proportion of benign files, P_M the proportion of malware files that contain each token

		PHP data set			JavaScript data set		
		token	P_B	P_M	token	P_B	P_M
static	malicious	OSB array <> mailboxes	4%	36%	n-gram eval	4%	43%
		OSB collection <> <> <> <> public	1%	18%	n-gram string	9%	49%
		OSB valueexporter <> <> get_class	1%	9%	n-gram fromcharcode	2%	47%
		OSB get_site_name <> <> <> 0	1%	7%	n-gram unescape	15%	15%
		OSB count <> emptyfields	1%	11%	n-gram iframe	%37	%41
	benign	OSB cookiejar <> <> <> <> getcookiejar	72%	6%	n-gram html	97%	95%
		OSB please <> <> <> <> zip	72%	6%	OSB img <> <> http	54%	51%
		OSB module <> <> <> jump	43%	1%	n-gram google	75%	44%
		OSB echo <> <> onmousedown	11%	1%	n-gram png	77%	58%
		OSB valuedump <> <> return	9%	1%	n-gram support	21%	10%
deobfuscated	malicious	OSB ev_m	4%	34%	OSB id <> <> script	2%	7%
		OSB return <> <> <> directories	1%	14%	n-gram php id	9%	13%
		OSB visits	4%	14%	OSB body <> <> script	1%	13%
		OSB p4 <> htmlspecialchars	3%	71%	n-gram fromcharcode	2%	47%
		OSB developers <> <> <> function	1%	68%	OSB javascript <> <> http	43%	49%
	benign	OSB cookiejar <> <> <> <> getcookiejar	72%	7%	OSB img <> <> http	54%	51%
		OSB array <> associations	43%	1%	OSB href <> <> php	16%	21%
		OSB regex <> <> pathvariables	8%	1%	n-gram google	75%	44%
		OSB convertto <> array	69%	1%	n-gram png	77%	58%
		OSB checkformat <> <> return	3%	1%	n-gram support	21%	10%

codes the file ending "php" has a low positive score. However, it should be noted that each of these individual features is not robust by itself, because malware scripts frequently perform arithmetic operations on strings before "unescaping" them. We also find that function hashes of standard wordpress functions have a negative weight (they indicate benign code) whereas variations of these functions that include an additional "unescape" or "eval" token have positive weights.

Static analysis of deobfuscated code suffers from two main limitations. First, the deobfuscation can be nested and made arbitrarily computationally expensive. This may result in the deobfuscation tool being unable to complete unfold all "eval" statements, which in our data happens in a number of cases. When malware succeeds at this, the subsequent feature extraction will be constrained by the same limitations as static code analysis. Secondly, content that is loaded from the web at runtime and then executed as code is not analyzed. This limitation could in principle be overcome by executing statements that load new content without executing the subsequent statements that execute that content as code. This technique is used frequently, and we therefore analyze how the model can still detect 99.96% of all PHP and 95% of all JavaScript malware. We find positive weights for several OSB and parse-tree features that match statements in which an invisible iframe is created, its content is generated dynamically on a server by a PHP script, and the loaded content is then evaluated as JavaScript code. This combination is such strong evidence of malicious intent that an analysis of the actual loaded content appears to be unnecessary. Even just loading content that is generated by a PHP script into an invisible iframe is strong evidence of malware.

Zero-day exploits that are directly embedded in an HTML page (without being loaded dynamically into a new iframe) pose a challenge to both the static model and the deobfuscated model—and in fact to all malware detection mechanisms—because they cannot be present in the training data. The models that we investigate can only detect such malware if it shares some syntactic resemblance to known malware.

6. CONCLUSIONS

The costs of fully dynamic code analysis lie in the order of CPU minutes, because an execution environment has to be set up for each file. In an application environment with a high throuput of files—such as a web hosting environment—it is impractical to subject all files to such an analysis. Code deobfuscation—partial execution to the point where all dynamically generated code has been generated—followed by a static analysis analysis of the resulting code can in most cases be executed in under a CPU second. While this is a great improvement, it is still a critical economic factor or may be impractical when billions of files are managed. The costs of static code analysis, on the other hand, are dominated by a step of feature extraction from the file which lies in the order of milliseconds which is feasible even in high-throuput environments. However, code obfuscation makes it relatively easy to conceal malicious code. Therefore, we studied cascaded malware detection; here, malware-detection decisions are made based on static code features whenever this decision can be made with near certainty. Otherwise, the code is deobfuscated and a decision is made based on static features of the deobfuscated code.

From our empirical study that involved around 400,000 difficult PHP files and 1,000,000 JavaScript files we can conclude that (a) cascaded malware detection is about or nearly as accurate as static analysis of the deobfuscated code for PHP and JavaScript, (b) cascaded detection incurs around 50% higher computational costs than static code analysis whereas deobfuscation followed by static analysis incurs roughly an order of magnitude higher costs, (c) fully static code analysis is consistently less accurate than cascaded detection, (d) OSB features perform better than n-gram and all other features, (e) the combination of OSB, n-gram, parse-tree and function-hash features performs better still, (f) logistic regression is more accurate than a random forest classifier for this problem, and (g) both cascaded detection and deobfuscation followed by detection are more robust against the evolution of malware over time than static code analysis. Cascaded malware detection is computationally feasible in high-throuput environments and allows to make detection decisions with low false-positive rates (10^{-4} for JavaScript, 10^{-3} for hard, conspicuous PHP files, and 0 for standard PHP files) and high true-positive rates (such as 95% for JavaScript and 99.83% for PHP). The cascaded model deobfuscates less than 5% of all files, which implies that many obfuscated files can safely be classified as benign based on static features only.

7. REFERENCES

[1] B. Anderson, C. Storlie, and T. Lane. Improving malware classification: Bridging the static/dynamic gap. In *Proceedings of 5th ACM Workshop on Security and Artificial Intelligence*, 2012.

[2] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the Network and Distributed System Security Symposium*, 2009.

[3] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th International World Wide Web Conference*, 2011.

[4] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 122–132, New York, NY, USA, 2012. ACM.

[5] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, 2005.

[6] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International World Wide Web Conference*, pages 281–290, 2010.

[7] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proceedings of the 20th USENIX Conference on Security*, 2011.

[8] E. Gandotra, D. Bansal, and S. Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 5:56–64, 2014.

[9] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 309–320, 2011.

[10] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1-2):13–23, 2005.

[11] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[12] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2006, 2006.

[13] D. Kong and G. Yan. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2013.

[14] P. Laskov and N. Šrndić. Static detection of malicious javascript-bearing PDF documents. In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[15] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference*, pages 421–430, 2007.

[16] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, page 4, 2011.

[17] I. Santos, J. Nieves, and P. Bringas. Semi-supervised learning for unknown malware detection. In *International Symposium on Distributed Computing and Artificial Intelligence Advances in Intelligent and Soft Computing*, 2011.

[18] M. Schultz, E. Eskin, F. Zadok, and S. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

[19] C. Siefkes, F. Assis, S. Chhabra, and W. S. Yerazunis. Combining winnow and orthogonal sparse bigrams for incremental spam filtering. In *Knowledge Discovery in Databases: PKDD 2004*, pages 410–421, 2004.

[20] N. Šrndić and P. Laskov. Detection of malicious PDF files based on hierarchical document structure. In *20th Annual Network and Distributed System Security Symposium*, 2013.

Chapter 5

Discussion

All presented papers improve the state of the art in their specific domain of computer security applications. The algorithm presented in Prasse et al. (2012, 2015) to infer regular expressions for given sets of emails is fast and accurate enough that it is used during regular operation of an email service to blacklist ingoing spam campaigns reliable. Furthermore the presented learning framework leads to models that attain higher recalls for recognizing emails within an email spam campaigns compared to previous work. Furthermore, our approach attains an extremely low false-positive rate of detecting non-spam emails. In Prasse and Scheffer (2016) a fast and accurate cascaded malware detection mechanism which attains state of the art classification results, but scans files more quickly compared to previous publications is presented.

The following chapter discusses the overall improvements of the presented papers to previous work. This chapter is organized as follows: Firstly, we discuss the existing approaches for detecting email spam campaigns, spam emails, and malicious malware and highlight the differences to our presented approaches. In Section 5.2 we describe how we modeled the different computer security applications. In Section 5.3 we discuss the improvements of the presented methods in terms of the performance to classify new unseen instances (e.g., emails, JavaScript or PHP files). Section 5.4 discusses the general improvements over previous works and the limitations of the presented methods.

5.1 Prior Work

The problem of detecting message campaigns in a stream of emails can be modeled as a clustering of the emails, whereas all emails belonging to one cluster are similar to each other. Such clusters can be found using standard clustering mechanisms on a bag-of-word representation of all emails like K-Means. Other more sophisticated approaches include agglomerative methods (Griffiths et al., 1984) and Bayesian models as proposed by Haider and Scheffer (2009).

Using a probabilistic cluster description, we can classify emails as being spam if they belong to a cluster of spam emails. Such a cluster description could use a bag-of-word representation of all tokens shared by the emails within an email campaign. This representation can be used to identify emails belonging to a specific cluster by checking if they share a certain amount of tokens with each other. However, this representation could lead to false positives and it is difficult for a human to decide whether a bag-of-word representation correctly characterizes the set of messages within a spam campaign. That's why postmasters decided to use regular expressions to describe email campaigns reliable. Regular expressions have the advantage of being easily interpretable and the risk of covering email outside a specific spam campaign can be reduced.

In Prasse et al. (2012) we presented a model that maps a set of strings $\mathbf{x}$ —an email campaign— to a regular expression $\mathbf{y}$ —a campaign description— such that every string $x \in \mathbf{x}$ belongs to the regular language $L(\mathbf{y})$ defined by the regular expression $\mathbf{y}$. Identification of regular languages has a rich history in the learning theory community. Gold (1967) shows that it is impossible to exactly identify any regular language from finitely many positive examples.

Our method differ from all prior work on learning regular expressions in their specific objective criterion and the training data. Unlike in prior work, in our problem setting the learner has access to pairs of strings and their corresponding regular expressions. At the same time, the learners goals is not just to find an expression that can generate all strings from a given batch, but to find *the* expression a human postmaster would write. Our idea of minimizing the expected difference between the predicted regular expression and target regular expression over a distribution of input strings reflects a statistically-inspired notion of learning.

Existing work to identify regular languages ranges from learning automata that accept a regular language (Denis, 2001; Clark and Thollard, 2004; Parekh and Honavar, 2001) or restricted classes of deterministic finite automata (Angluin, 1978; Angluin, 1980; Abe and Warmuth, 1990). Other papers address the task of inferring a regular 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).

Regular expressions are used in computer security applications to specify patterns. Xie et al. (2008) use regular expressions to detect and describe malicious URLs obtained in spam batches. The *ReLIE* algorithm 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 (Li et al., 2008). The problem setting of this paper is closest related to our task of infering a regular expression from a set of strings and is used as baseline for our presented approach (see Chapter 2 Section 2.5 and Chapter 3.1 Section 3.5). Another method introduced in Brauer et al. (2011) builds a data structure of commonalities of aligned strings and transforms these strings into a specific regular expression and is best used for short strings, such as telephone numbers and names of software products. Because of the high complexity of this algorithm we did not use this approach as a baseline.

We modeled the problem of learning a mapping from a set of strings to a regular expression as a learning problem with structured output spaces (see Chapter 2 Section 2.3). Structured output spaces are used in many machine-learning applications, including sequence labeling (Tsochantaridis et al., 2004), and natural language parsing (Tsochantaridis et al., 2005). Problems with structured output spaces can be solved by an extension of the support vector machines (Vapnik, 1998). Tsochantaridis et al. (2005) gives an overview how problems ranging from classification with taxonomies, label sequence learning, sequence alignment to natural language parsing can be solved using structural support vector machines. In the papers Prasse et al. (2012, 2015) we use the concept of structured output spaces to solve the problem of inferring regular expressions for a given set of strings.

The third paper Prasse and Scheffer (2016) of this thesis addresses the problem of accurately and quickly detecting malware that is contained in JavaScript or PHP files. A complete overview of malware detection mechanisms is given by Gandotra et al. (2014). Prior work mostly uses either static or dynamic analysis techniques to extract features and patterns that can be used to recognize malware.

Several feature types were analyzed in prior work, including n-gram features (Kolter and Maloof, 2006), bag of tuples (Canali et al., 2012), syntax-tree features (Curtsinger et al., 2011), control-flow graph features (Anderson et al., 2012; Christodorescu et al., 2005), and function-call graph features (Kong and Yan, 2013). Some approaches also use URL and host features (Canali et al., 2011). In Prasse and Scheffer (2016) we use a similar set of features including n-gram and syntax-tree features which can be extracted from the program code. Furthermore, we use orthogonal sparse bigrams which are a specialization of the bag of tuples method and function-hashing features that can be seen as a signature over a small fraction of the program code.

The classification performance of fully static malware detection methods typically drops with the presents of code obfuscation (see Chapter 4 Section 4.4). Dynamic analysis techniques offer a potential remedy because they observe the program code as it is evaluated in a controlled environment (Bayer et al., 2009; Kolbitsch et al., 2009). In JavaScript and PHP files code that is executed and unpacked at runtime can be observed using deobfuscation tools. Sample tools for deobfuscating JavaScript Code are the SpiderMonkey package by Didier Stevens[1] which is a modified version of Mozilla's C implementation of JavaScript [2] and is used in our paper. Other tools are Creme Brulee[3], JSDebug[4] and SpiderMonkey + V8[5]. With the help of such tools it is possible to reveal informative features (Curtsinger et al., 2011). Static and dynamic analysis can be combined to improve the classification performance (Anderson et al., 2012).

Many supervised machine-learning methods has been studied to combine the extracted features into a detection mechanism (Schultz et al., 2001; Kolter and Maloof, 2006). Most empirical investigations of malware detection have so far been made on relatively small sized data sets, using hundreds (Canali et al., 2011; Curtsinger et al., 2011; Anderson et al., 2012) or thousands (Schultz et al., 2001; Nataraj et al., 2011; Kolter and Maloof, 2006) of malware files. These investigations are done on completely different data sets containing malware files from different malware families, which makes it difficult to compare the results to each other. We contribute a new publicly available data set containing more than one million JavaScript files. This data set can be used by researches to compare different methods with each other.

[1]https://didierstevens.com/files/software/js-1.7.0-mod-b.zip

[2]http://www.mozilla.org/js/spidermonkey/

[3]http://code.google.com/p/cremebrulee/

[4]http://www.codeproject.com/KB/scripting/hostilejsdebug.aspx

[5]http://code.google.com/p/v8/

5.2 Modeling

In our applications we are faced with the goal of learning a model to find the relationship between given input data and a target label. This section describes our new approaches of modeling the two sample applications covered in this thesis: detecting email spam campaigns and malware.

In Chapter 2 (Prasse et al., 2012) and Chapter 3.1 (Prasse et al., 2015) we extend the structured output prediction framework to handle sets of strings as input and regular expressions as output. Therefore, we define a joint feature representation and a appropriate specific loss function to compare different regular expressions with respect to a set of strings. In the learning phase of our model we have to identify the most violated constrained —a regular expression— and during classification we have to find the highest scoring regular expression for a given model and a set of strings which are emails of a spam campaign in our sample application.

To find such a regular expression in an efficient way we follow the *under-generating* approach presented by Finley and Joachims (2008) by restricting the class of possible regular expressions to a subset defined in Chapter 2 in Sectin 2.4.2 in Algorithm 1. Furthermore, we present an efficient way to identify the highest scoring regular expression in this restricted search space. Therefore, we decompose the maximization over all regular expressions within the search space into independent maximization problems (see Theorem 2 in Chapter 3.1 in Section 3.4.2) which can be solved easily. Thus the paper Prasse et al. (2012) extends the application scope of structured output prediction by presenting a proper formulation for set of strings as input and regular expressions as output and an efficient inference algorithm.

The paper Prasse et al. (2015) extends the approach of Chapter 2 (Prasse et al., 2015) to infer a regular sub-expressions for a given input regular expression that only describe the characteristic part of a given email spam campaign. We model this by extending the structured output prediction framework. We define a joint feature representation for input regular expressions and output regular expressions and a joint feature representation consisting of features extracted for the input and output regular expression and all products of an input and an output feature (see Chapter 3.1 Section 3.4.3).

To find the most violated constrained and the highest scoring regular expression we restrict the search to the space of all regular sub-expressions of the input regular expression with a maximal length of s. Within this set, the decoder performs an exhaustive search. Combining the two models of firstly inferring a regular expression from a set of strings and secondly extracting a concise sub-expression we present a system that can efficiently and accurately recognize emails belonging to a specific email campaign.

In Chapter 4 (Prasse and Scheffer, 2016) we address the problem of detecting malicious JavaScript and PHP files. We present a cascaded model which combines the execution time advantages of purely static analysis with the advantages of higher accuracies for a partially dynamic analysis using deobfuscation techniques. We present a framework were most classification is done only on statically obtained features from the program code. Only a very small fraction has to be deobfuscated and classified with a second model. Thus this paper contributes a new model for malware classification with state-of-the-art classification results and execution times that makes it possible to scan huge file collection in a quick way.

5.3 Classification Performance

In this section we discuss the classification performances of the presented methods and compare them to prior work.

In Chapter 2 and Chapter 3.1 (Prasse et al., 2012, 2015) we evaluate the spam filtering performance of the regular expressions inferred for a set of strings belonging to one email spam campaign. We found that the ability of detecting emails that belong to a specific spam campaign improves over prior work tested on two real world data sets. Furthermore we evaluate the false-positive rates of our models compared to the false-positive rates of baselines. From this experiment, we conclude that our models obtain a significant lower false-positive rates compared to baselines. This false-positive rates are such good that our approach is being used by an email service during regular operation to complement content-based IP-address based filtering. Thus, the presented papers Prasse et al. (2012, 2015) improve the state-of-the-art in terms of email spam campaign classification performance.

In Chapter 4 (Prasse and Scheffer, 2016) we presented a method to detect malicious JavaScript and PHP files. Because of the absence of publicly available data sets for this application a fair comparison to baselines is not possible. Instead, we compared several performance metrics such as the area under a ROC curve, accuracy, recall at a specific false-positive rate, and true-negative rate at a specific false-negative rate with reported values in prior work on different data sets.

From our experimental results we can conclude that our method gains state-of-the-art results by saving execution time compared to methods using dynamic analysis techniques. Furthermore, our paper contributes a evaluation of the malware detection methods over time that is more natural than the evaluation on training and test sets with overlapping times of the first occurrences. Hence, we want to apply a trained model on new unknown future input files and thus the evaluation into the future better reflects the application case. We can conclude, that the presented cascaded model is much more robust than the model using only features extracted from the original program code known as purely static malware detection. Furthermore their is no big difference in terms of recall comparing the fully mostly static —using deobfuscation techniques— and the cascaded approach.

In an other experimental setup we compared the recall of the 20 best commercial antivirus vendors with the recall of our presented models at a false-positive rate of 0.01%. From this experiment we can conclude, that the recall of our models is slightly higher than the recall for commercial antivirus vendors.

5.4 Contributions and Comparison to Prior Work

In this section we discuss the overall contributions and improvements of the presented papers compared to prior work. Furthermore, this section discusses the limitations of the presented methods.

The main advantage of the approach presented in Chapter 2 (Prasse et al., 2012) and Chapter 3.1 (Prasse et al., 2015) is the ability of learning the preferences of a human postmaster in writing regular expressions for email spam campaigns from training samples. Our experimental evaluation shows that this approach leads to models that infer well understandable regular expressions which a postmaster could easily check and use to blacklist email campaigns at virtually no risk of matching emails outside a spam campaign. Furthermore, the models are good enough that they are used during regular operations of an email service.

Like every learning method our model can only capture preferences of writing regular expressions which are reflected in the training data and thus new or unseen patterns of an human postmaster can not be learned. Because the regular expressions are specializations of an alignment of all strings within a spam campaign the method only conjectures useful regular expressions if the emails within a campaign have words together and ideally at similar positions within each email. If this is not the case, it could happen, that to general regular expressions causing false-positives would be inferred (see Figure 4 in Chapter 2 Section 2.5). To obtain a fast inference algorithm we restricted the search space of regular expressions which can be inferred in such a way that we limit the maximum nesting depth within a regular expression. Therefore our approach is not able to infer regular expressions which exceeds this maximum nesting depth.

In Chapter 4 (Prasse and Scheffer, 2016) we presented a cascade malware detection framework for JavaScript and PHP files. This approach improves over prior work by achieving state-of-the-art detection results by saving processing time using a cascaded architecture. This approach uses the advantages of the low time to classify of fully static malware detection algorithms on the one hand and the high classification accuracy of a system based on static features extracted from the deobfuscated program code on the other hand.

This results in a system that is able to quickly scan huge file collection with state-of-the-art classification performance. Our paper is the first paper reporting an a case study based on two very large data collections where the JavaScript collection is being published for comparison.

A limitation of our presented model is the inability of detecting zero-day exploits never seen in the training data, because the presented models can only detect malware which shares syntactic resemblance to known malware. In some cases malicious files are marked as malicious only because a link contained in the program file leads to a server listed on an IP blacklist.

Our model is only able to label such files as malicious if some other features suggests this. To detect such malicious links, it would be necessary to visit all links in a safe environment and try to detect malicious behavior from system calls, and upstream or downstream information. Such an analysis is known as dynamic malware detection and is very time consuming and therefore not suitable for our use case of quickly scanning large file collections.

Bibliography

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

B. Anderson, C. Storlie, and T. Lane, "Improving malware classification: Bridging the static/dynamic gap," in *Proceedings of 5th ACM Workshop on Security and Artificial Intelligence*, 2012.

D. Angluin, "Inductive inference of formal languages from positive data," *Information and Control*, vol. 45, no. 2, pp. 117–135, 1980.

D. Angluin, "On the complexity of minimum inference of regular sets," *Information and Control*, vol. 39, no. 3, pp. 337–350, 1978.

K. Bartos and M. Sofka, "Robust representation for domain adaptation in network security," in *Machine Learning and Knowledge Discovery in Databases.* Springer International Publishing, 2015, vol. 9286, pp. 116–132.

U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proceedings of the Network and Distributed System Security Symposium*, 2009.

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*, 2008, pp. 825–834.

F. Brauer, R. Rieger, A. Mocan, and W. Barczynski, "Enabling information extraction by inference of regular expressions from sample entities," in *Proceedings of the Conference on Information and Knowledge Management.* ACM, 2011, pp. 1285–1294.

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

M. Brückner, C. Kanzow, and T. Scheffer, "Static prediction games for adversarial learning problems," *Journal of Machine Learning Research*, vol. 13, no. 1, pp. 2617–2654, 2012.

D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: A fast filter for the large-scale detection of malicious web pages," in *Proceedings of the 20th International World Wide Web Conference*, 2011.

D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012, 2012, pp. 122–132.

P. K. Chan and R. P. Lippmann, "Machine learning for computer security," *Journal of Machine Learning Research*, vol. 7, pp. 2669–2672, 2006.

M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, 2005.

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

C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Fast and precise in-browser javascript malware detection," in *Proceedings of the 20th USENIX Conference on Security*, 2011.

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

H. Fernau, "Algorithms for learning regular expressions from positive data," *Information and Computation*, vol. 207, no. 4, pp. 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.

V. Franc, M. Sofka, and K. Bartos, "Learning detector of malicious network traffic from weak labels," in *Machine Learning and Knowledge Discovery in Databases.* Springer International Publishing, 2015, vol. 9286, pp. 85–99.

E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," *Journal of Information Security*, vol. 5, pp. 56–64, 2014.

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

A. Griffiths, L. A. Robinson, and P. Willett, "Hierarchic agglomerative clustering methods for automatic document classification," *Journal of Documentation*, vol. 40, no. 3, pp. 175–205, 1984.

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

C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *Journal of Machine Learning Research*, vol. 7, p. 2006, 2006.

D. Kong and G. Yan, “Discriminant malware distance learning on structural information for automated malware classification,” in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2013.

T. Lane and C. E. Brodley, “Detecting the abnormal: Machine learning in computer security,” Tech. Rep., 1997.

P. Laskov, P. Düssel, C. Schäfer, and K. Rieck, “Learning intrusion detection: Supervised or unsupervised?” *Image Analysis and Processing ICIAP*, vol. 3617, pp. 50—-57, 2005.

W. Lee and S. J. Stolfo, “A framework for constructing features and models for intrusion detection systems,” *ACM Transactions on Information and System Security*, vol. 3, no. 4, pp. 227–261, Nov. 2000.

W. Lee and S. J. Stolfo, “Data mining approaches for intrusion detection,” in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. USENIX Association, 1998, pp. 6–6.

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*, 2008, pp. 21–30.

H. Liu and K. Chang, “Defending systems against tilt DDoS attacks,” in *Proceedings of the International Conference on Telecommunication Systems, Services, and Applications*, 2011.

L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath, “Malware images: visualization and automatic classification,” in *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, 2011, p. 4.

R. Parekh and V. Honavar, “Learning DFA from simple examples,” *Machine Learning*, vol. 44, pp. 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.

P. Prasse and T. Scheffer, “Cascaded malware detection at scale,” vol. xx, no. xx, 2016, pp. xx–xx.

P. Prasse, C. Sawade, N. Landwehr, and T. Scheffer, “Learning to identify concise regular expressions that describe email campaigns,” *Journal of Machine Learning Research*, vol. 16, pp. 3687–3720, 2015.

S. Ranjan, R. Swaminathan, M. Uysal, and E. Knightley, “DDoS-resilient scheduling to counter application layer attacks under imperfect detection,” in *Proceedings of IEEE INFOCOM*, 2006.

M. Schultz, E. Eskin, F. Zadok, and S. Stolfo, “Data mining methods for detection of new malicious executables,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

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

I. Tsochantaridis, T. Hofmann, T. Joachims, and Y. Altun, “Support vector machine learning for interdependent and structured output spaces,” in *Proceedings of the International Conference on Machine Learning.* ACM, 2004, p. 104.

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

Y. Xie and S. Z. Yu, “A large-scale hidden semi-markov model for anomaly detection on user browsing behaviors,” *IEEE/ACM Transactions on Networking*, vol. 17, no. 1, pp. 54–65, 2009.

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*, 2008, pp. 171–182.

S. Zanero and S. M. Savaresi, “Unsupervised learning techniques for an intrusion detection system,” in *Proceedings of the 2004 ACM Symposium on Applied Computing*, ser. SAC ’04, 2004, pp. 412–419.