IRISH: A Hidden Markov Model to Detect Coded Information Islands in Free Text

Luigi Cerulo\textsuperscript{a,c,∗}, Massimiliano Di Penta\textsuperscript{b}, Alberto Bacchelli\textsuperscript{d}, Michele Ceccarelli\textsuperscript{a,e}, Gerardo Canfora\textsuperscript{b}

\textsuperscript{a}Dep. of Science and Technology, University of Sannio, Benevento, Italy
\textsuperscript{b}Dep. of Engineering, University of Sannio, Benevento, Italy
\textsuperscript{c}BioGeM, Institute of Genetic Research “Gaetano Salvatore”, Ariano Irpino (AV), Italy
\textsuperscript{d}Dep. of Software Technology, Delft University of Technology, The Netherlands
\textsuperscript{e}QCRI - Qatar Computing Research Institute, Doha, Qatar

Abstract

Developers’ communication, as contained in emails, issue trackers, and forums, is a precious source of information to support the development process. For example, it can be used to capture knowledge about development practice or about a software project itself. Thus, extracting the content of developers’ communication can be useful to support several software engineering tasks, such as program comprehension, source code analysis, and software analytics. However, automating the extraction process is challenging, due to the unstructured nature of free text, which mixes different coding languages (e.g., source code, stack dumps, and log traces) with natural language parts.

We conduct an extensive evaluation of IRISH (InfoRmation ISlands Hmm), an approach we proposed to extract islands of coded information from free text at token granularity, with respect to the state of art approaches based on island parsing or island parsing combined with machine learners. The evaluation considers a wide set of natural language documents (e.g., textbooks, forum discussions, and development emails) taken from different contexts and encompassing different coding languages. Results indicate an F-measure of IRISH between 74\% and 99\%; this is in line with existing approaches which,
differently from Irish, require specific expertise for the definition of regular expressions or grammars.

*Keywords:* Hidden Markov Models, Mining Unstructured Data, Developers’ communication

1. Introduction

Mailing lists and issue tracking systems are communication tools widely adopted by developers to exchange information about implementation details, high-level design, bug reports, code fragments, patch proposals, erroneous behavior, etc. Some software projects, such as Linux Kernel, adopt mailing lists as the main tool for managing and storing software documentation [1]. Further communication means, very popular nowadays, are web forums, such as StackOverflow.¹

Communication regularly used to support software development, such free text development emails, are a very attractive source of knowledge to support program comprehension and development, because they contain discussions, for example, on how some parts of the system work or should not be used. Communication data has been exploited to develop recommender systems, for example aimed at supporting bug triaging [2], at providing examples of how some APIs should work [3, 4], and at aiding program comprehension when documentation is scarce [5].

Nevertheless, extracting relevant information from developers’ communication is challenging, due to the mix of different languages adopted by developers, e.g., to describe system failures or code changes. Many development emails, for example, include natural language text interleaved with a detailed stack trace to describe a failure and source code to propose a solution. The email text can appear mixed up with code when it is part of a thread of discussion, in which more developers participate and consider alternative solutions. Also in web forums, such as Stack Overflow, natural language is interleaved with code snippets and stack traces; to improve readability, users are invited to use tags to separate communication from code snippets and stack traces, but this does not happen consistently in practice.²

¹[http://stackoverflow.com](http://stackoverflow.com)
²Performing manual analysis of the Stack Overflow posts used in this paper, we found approximately 5–10% of code tags to not enclose a piece of code or named code entity.
Separating different pieces of information contained in developers’ communication improve the quality of data extraction. This is because different parts of the communication require appropriate ways of being analyzed. Just to make an example, sentences expressed in natural language would benefit of analysis performed using Information Retrieval (IR) techniques or maybe natural language parsing, whereas source code fragments should be analyzed using parsers. In some cases—see for example the duplicate bug report detection approach proposed by Wang et al. [6]—different kinds of information such as natural language text and stack traces contribute to the approach accuracy. In other cases, some elements contained in the discussion should be isolated. This could be for instance the case when one wants to mine source code snippets contained in developers’ communication.

On the one hand, many software engineering approaches [7, 8] have treated free text using traditional IR models such as Vector Space Models (VSM) [9] or Latent Semantic Indexing (LSI) [10]. Although this simplification works well for pure natural language documents, it may easily fail for software engineering artefacts [11], where free text is interleaved with source code, stack traces and other elements. On the other hand, in recent years authors have proposed alternative approaches to treat text based on island parsers, regular expressions, and supervised learning [12, 13]. Recently, Bacchelli et al. [11] proposed a hybrid approach, combining island parsers and machine learning. Such an approach outperforms the use of the two techniques in isolation, when not well formed languages (e.g., noise and random characters) appear together with more structured ones.

This paper describes IRISH (InfoRmation ISlands Hmm), an approach that we initially proposed in our previous work [14], based on Hidden Markov Models (HMM) to extract islands of coded information from free text at token granularity. Tokens are particles of the text, such as natural language words, programming language keywords, digits, and punctuation. In IRISH, we consider the sequence of tokens of a textual document (e.g., a development email) as the emission generated by the hidden states of a HMM. Hidden states are adopted to model a specific coded information content, e.g., source code and natural language text. We adopt the Viterbi algorithm [15] to search for the path that maximizes the probability of switching among hidden states. Such a path allows us to classify each observed token in the corresponding coded information category. If appropriately modeled with hidden states and given a proper set of training examples, the approach can, in principle, include an arbitrary number of different text interleaved
languages, for example stack traces, patches, and markup languages.

The specific goal of this paper is to provide an extensive evaluation of IRISH and to highlight its points of strength and weaknesses, by comparing it to the current state of the art, i.e., two methods (PETITISLAND and MUCCA) proposed by Bacchelli et al. [11], based respectively on island parsing and island parsing and machine learners combined. The contributions of this paper are:

- The evaluation of IRISH on two new datasets: (1) the mailing list dataset built and used by Bacchelli et al. [11], and (2) the Stack Overflow dataset used to build a traceability recovery approach [16].

- A direct comparison with the approaches by Bacchelli et al. [11]. To this aim we evaluate the methods proposed by Bacchelli et al. on datasets previously used to evaluate IRISH [14].

- A qualitative evaluation of the learning curve necessary to set up both approaches.

Results of the evaluation indicate that, overall, IRISH exhibits performance in line with the state-of-the-art approach. However, differently from these approaches, it only requires a manual classification of a training set, rather than writing island grammars.

Structure of the paper. Section 2 summarizes backgrounds and Section 3 introduces IRISH. Section 4 details the empirical evaluation procedure. Section 5 reports and discusses the obtained results. Section 6 discusses threats to the validity of the evaluation. Section 7 discusses related work about approaches for extracting encoded information from unstructured sources. Section 8 concludes the paper and outlines directions for future work.

2. Background Notions about HMM

A Markov model (or Markov chain) is a stochastic process in which the state of a system is modelled as a random variable that changes through time, and the distribution of this variable only depends on the distribution of the previous states (Markov property). A Hidden Markov Model (HMM) is a Markov model for which the state is partially, or entirely, not observable [17]; in other words, the state is not directly visible, while the output, dependent on the state, is visible. Each state has a probability distribution over the
possible output symbols, thus the sequence of symbols generated by an HMM gives some information about the sequence of hidden states.

HMMs have been applied to temporal pattern recognition (e.g., speech, handwriting, and gesture recognition [18, 19]), part-of-speech tagging [20], musical score following [21], and bioinformatics analyses (e.g., CpG island detection and splice site recognition [22]).

Formally, a HMM is a quadruple \((S, Q, T, E)\), where:

- \(S\) is an alphabet of output symbols;
- \(Q\) is a finite set of states capable of emitting output symbols from alphabet \(S\);
- \(T\) a set of transition probabilities denoted by \(t_{kl}\) for each \(k, l \in Q\), such that for each \(k \in Q\), \(\sum_{i \in Q} t_{ki} = 1\);
- \(E\) a set of emission probabilities denoted by \(e_{kb}\) for each \(k \in Q\) and \(b \in S\), such that for each \(k \in Q\), \(\sum_{i \in S} e_{ki} = 1\).

Given a sequence of observable symbols \((X = \{x_1, x_2, \ldots, x_L\})\) emitted by following a path \((\Pi = \{\pi_1, \pi_2, \ldots, \pi_L\})\) among the states, the transition probability \((t_{kl})\) is defined as \(t_{kl} = P(\pi_i = l | \pi_{i-1} = k)\), and the emission probability \((e_{kb})\) is defined as \(e_{kb} = P(x_i = b | \pi_i = k)\). Therefore, assuming the Markov property and given the path \(\Pi\), the probability that the sequence \(X\) was generated by the HMM is determined by:

\[
P(X|\Pi) = t_{\pi_0\pi_1} \prod_{i=1}^{L-1} e_{\pi_ix_{i+1}} t_{\pi_i\pi_{i+1}}
\]

where \(\pi_0\) and \(\pi_{L+1}\) are dummy states assumed to be respectively the initial and final states. The objective of the decoding problem is to find an optimal generating path \(\Pi^*\) for a given sequence of symbols \(X\), i.e., a path such that \(P(X|\Pi)\) is maximized over the space of possible paths:

\[
\Pi^* = \arg \max_{\Pi} P(X|\Pi)
\]

The Viterbi algorithm finds such a path in \(O(L \cdot |Q|^2)\) time by using a dynamic programming searching strategy [15]. The algorithm works basically as follows. For an observed symbol \(x_{i+1}\), for \(1 \leq i \leq L - 1\), the optimal hidden state \(\pi_{i+1}^*\) reached at the \((i + 1)\)th transition is computed recursively as:
\[
\pi_{i+1}^* = \arg \max_{k \in Q} P(X|\Pi_i^*) e_{\pi_{i+1}^* k} t_{\pi_i^* k}
\]

where \(\pi_i^*\) is the optimal hidden state reached at the previous \(i\)th transition and \(\Pi_i^*\) is the optimal path to reach the optimal hidden state \(\pi_i^*\). The overall optimal generating path is given by: \(\Pi^* = \{\pi_1^*, \pi_2^*, \ldots, \pi_L^*\}\), where the initial state \(\pi_1^*\) is chosen among \(Q\).

As a clarification example, let us consider the classical unfair-casino problem [22]. To improve the chances of winning, a dishonest casino uses loaded dice occasionally, while most of the time it uses a fair dice. The loaded dice has a higher probability of landing on a 6, with respect to the fair dice (where the probability of each outcome is always \(1/6\)). Suppose the loaded dice has the following probability distribution: \(P(1) = P(2) = P(3) = P(4) = P(5) = 1/10\) and \(P(6) = 1/2\). The goal is to unmask the casino: Given an output sequence, we would like to recognize when the casino uses a fair dice and when a loaded one.

![Diagram of the unfair-casino HMM](image)

Figure 1: The unfair-casino HMM.

Figure 1 shows the HMM representing the rolling dice game, where the alphabet is \(S = \{1, 2, 3, 4, 5, 6\}\), and the state space is \(Q = \{\text{FairDice, LoadedDice}\}\). Suppose the dishonest casino switches between the two hidden states (i.e., fair dice and loaded dice) with the transition probabilities shown in Figure 1. When the casino uses the fair dice the emission probabilities are those of the fair dice, otherwise those of the loaded dice. Let us assume, for example, that we observe the following roll of dice sequence:

1, 2, 6, 4, 3, 6, 5, 2, 6, 6, 4, 1, 3, 6, 6, 6, 6, 6, 6, 6, 5, 4, 6, 1, 6.
From such a sequence, we cannot tell which state each rolling is in. For example, the subsequence:

\[ 6, 6, 6, 6, 6, 6 \]

may occur using the loaded dice, or it can occur using the fair dice, even though the latter is less likely to occur. The state is hidden from the sequence, *i.e.*, we cannot determine the sequence of states from the given sequence. The Viterbi algorithm can decode the most probable sequence of states emitting the observed sequence of symbols [15], which is the following:

\[ \text{FairDice}, \text{FairDice}, \text{FairDice}, \text{FairDice}, \text{FairDice}, \text{FairDice}, \text{FairDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}, \text{LoadedDice}. \]

### Table 1: The token alphabet \( S \).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Token</th>
<th>Regexp</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD</td>
<td>any alphanumeric character</td>
<td>([a-zA-Z0-9]+)</td>
</tr>
<tr>
<td>KEY</td>
<td>a WORD token that is also a language keyword (<em>e.g.</em>, C/C++, Java, Perl, SQL)</td>
<td></td>
</tr>
<tr>
<td>UNDSC</td>
<td>the underscore character</td>
<td>(_+)</td>
</tr>
<tr>
<td>NUM</td>
<td>a WORD token that is a sequence of digits</td>
<td>(\d+)</td>
</tr>
<tr>
<td>the char itself</td>
<td>any character not matching the previous patterns</td>
<td>([-s\w])</td>
</tr>
</tbody>
</table>

#### 3. Irish: The Proposed Approach

To model our problem we adopt an HMM defined as follows. We model a text as a sequence of tokens. Each token belongs to a symbol class that constitutes the observable alphabet of our HMM. Table 1 shows the alphabet and the regular expression we use to detect those symbols in free text.

Table 2 reports some textual examples, and their representation as a sequence of tokens. The goal is similar to the unfair casino problem described in Section 2: We want to detect whether a symbol encountered in a text
Table 2: Some token sequence examples.

<table>
<thead>
<tr>
<th>Text</th>
<th>Token sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;My dear Frankenstein,&quot; exclaimed he, &quot;how glad I am to see you!&quot;</td>
<td>&quot; WORD WORD WORD , &quot; WORD WORD , &quot; WORD WORD WORD WORD WORD WORD ! &quot;</td>
</tr>
<tr>
<td>for(int i=0;i&lt;10;i++) s+=1;</td>
<td>KEY ( WORD WORD = NUM ; WORD &lt; NUM ; WORD + ) WORD + = NUM ;</td>
</tr>
</tbody>
</table>

sequence is part of the natural language text or part of a source code fragment. Natural language and source code are the hidden states that we aim to detect. Encountering a language keyword symbol (KEY), for example, does not guarantee that the portion of the analyzed text is a source code fragment as many keywords are also valid natural language (e.g., while, for, if, function, select): the KEY symbol could be emitted by both hidden states. In principle, each observed symbol could be emitted by both hidden states, one modeling natural text language, and one modeling source code text. The HMM is composed of two sub-HMM, one modeling the transitions among symbols belonging to natural text language sequences, and another modeling the transitions among symbols belonging to source code text. The transition probabilities between two symbols could be different if they belong to different language syntaxes. For example, after a KEY symbol in natural language text, it is more likely to find a WORD symbol, while in source code fragments it is more usual to find punctuation or special characters, such as opening parentheses, as shown by the transitions probabilities in Figure 3 and Figure 4. Formally, the HMM state space is defined as:

$$Q = \{S_{TXT}, S_{SRC}\}$$

where $S_{TXT}$ are the hidden states modeling natural text language transitions and $S_{SRC}$ are the hidden states modeling the source code transitions. We represent hidden states by subscripting observable symbols with $TXT$ or $SRC$ to indicate from which language syntax that symbol could be emitted.
Each hidden state can emit only the corresponding alphabet symbol without the subscript label $TXT$ or $SRC$. For example, the KEY symbol can be emitted only by $KEY_{TXT}$ or $KEY_{SRC}$ each with a probability equal to 1.

A transition can occur between hidden states belonging to the same language syntax category ($e.g.$, $WORD_{TXT} \rightarrow KEY_{TXT}$), or belonging to different language syntax categories ($e.g.$, $WORD_{TXT} \rightarrow KEY_{SRC}$). The latter case happens when the text switches from one language to another. If the probability of staying in natural language text is $p$ and the probability of staying in source code text is $q$, then the transition from a state in $S_{TXT}$ to a state in $S_{SRC}$ is $1 - p$ and the inverse transition is $1 - q$. Formally, the aforementioned HMM emits the sequence of symbols observed in a text by evolving through a sequence of states ($\{\pi_1, \pi_2, \ldots, \pi_i, \pi_{i+1}, \ldots\}$) with the transition probabilities ($t_{kl}$) defined as:

![Diagram of the source code – natural text island HMM.](image-url)
Figure 3: A natural text HMM trained on the Frankenstein novel (transition probabilities less than 0.1 are not shown).

Figure 4: A source code HMM trained on PosgreSQL source code (transition probabilities less than 0.2 are not shown).
\[ t_{kl} = P(\pi_i = l|\pi_{i-1} = k) \cdot p, \text{ if } k, l \in S_{TXT} \]
\[ t_{kl} = P(\pi_i = l|\pi_{i-1} = k) \cdot q, \text{ if } k, l \in S_{SRC} \]
\[ t_{kl} = \frac{1 - p}{|Q|}, \text{ if } k \in S_{TXT}, l \in S_{SRC} \]
\[ t_{kl} = \frac{1 - q}{|Q|}, \text{ if } k \in S_{SRC}, l \in S_{TXT} \]

and the emission probabilities defined as:

\[ e_{kb} = 1, \text{ if } k = b_{TXT} \text{ or } k = b_{SRC}, \text{ otherwise } 0. \]

Figure 2 shows the global HMM composed of two sub-HMM, one modeling natural language text and one modeling C source code. Figure 3 and Figure 4 show the transition probabilities within a language syntax category, estimated on the Frankenstein novel and PostgreSQL source code respectively. Section 3.2 proposes heuristics to empirically estimate such probabilities.

By observing how typical token sequences are modeled by each HMM, we see, for example, that a number (NUM) in the source code HMM is typically followed and preceded by commas (probability respectively equal to 0.77 and 0.47), thus modeling arguments separation in a function call; and the underscore character follows and is followed by a WORD (probability more than 0.9), thus modeling typical variable naming convention. Instead, in the natural language HMM, numbers (NUM) are preceded just by the dollar symbol ($) (probability equal to 1) indicating currency, and likely followed by a dot (probability equal to 0.45), indicating text item enumerations. In the source code, numbers are part of an arithmetic/logic expressions, array indexing, or function argument enumeration, while in the natural language text they belong to currency representations.

### 3.1. An extension of the basic model

The basic HMM can be extended to include an arbitrary number of language syntaxes. In development email we find patches, log messages, configuration parameters, steps to reproduce program failures, XML dialects, and so on. To include \( n \) language syntaxes we introduce the language transition
probability matrix \( W = w_{ij} \), for \( i, j \in \{1, 2, \ldots, n\} \), which defines the probabilities of staying into a particular language syntax and of switching from one syntax to another. Formally, if \( w_{ii} \) is the probability of staying into a language syntax \( i \), then the probability of switching from \( i \) to \( j \neq i \) is given by: \( w_{ij} = (1 - w_{ii})/(n - 1) \), supposing a uniform distribution among language syntaxes and assuring that \( \sum_{j=1}^n w_{ij} = 1 \). For two language syntaxes (e.g., natural language and source code), the language transition probability matrix becomes as in the following formula (in which \( w_{11} = p \) and \( w_{22} = q \)):

\[
W = \begin{bmatrix}
  w_{11} & (1 - w_{11}) \\
  1 - w_{22} & w_{22}
\end{bmatrix}
\]

### 3.2. Model parameter estimation

The model parameters to be estimated are the transition probability matrix \( T = \{ t_{ki} \} \) and the language transition probability matrix \( W = \{ w_{ij} \} \). Within language transitions, probabilities can be empirically estimated by counting the number of transitions between two subsequent token symbols in examples in that language syntax. For example, Figure 3 shows the transition probabilities estimated on the Frankenstein novel,\(^3\) while Figure 4 shows the transition probabilities estimated inside a collection of C source code files.\(^4\)

Transitions from one language syntax category to another are harder to estimate empirically as they may depend on the nature of the text and the writing style. This information may not be known in advance. The aim is to estimate how many transitions could happen between two language syntaxes. For example, if in each development email there is usually no more than one source code fragment, we can assume \textit{a priori} the transition probability from natural language text to source code approximated to \( 1/N \), where \( N \) is the number of tokens in the message. However, it could happen that the transition between two language syntaxes could never occur. This may be the case of stack traces and patches. In development emails, we usually notice that after a stack trace the email author introduces the resolution patch using natural language phrases (e.g., “Here is the change that we made to fix the problem”). This means having a null transition probability from stack trace to patch code. Other heuristics, which reflect specific properties or adopted styles, could be adopted to refine the estimation of transition probabilities.

---

\(^3\)Downloaded from the “Project Gutenberg” (http://www.gutenberg.org)
\(^4\)Extracted from the PostgreSQL source code repository (http://www.postgresql.org)
probabilities [23]. In all empirical experiments we estimate the transition probabilities from one language to another empirically by dividing the number of occurrences of a transition with the number of tokens in the training set.

4. Empirical Study Design

In the following, we describe our study using the Goal Question Metrics template and guidelines [24].

Goal: Analyzing the effectiveness of IRISH.

Purpose: Comparing IRISH with the state of the art approaches for detecting information islands in unstructured sources (i.e., [11]).

Perspective: A researcher interested to choose the most suitable method for analyzing developers’ communication.

Context: Four datasets: (i) two publicly available HTML/latex textbooks with fully annotated source code fragments; (ii) a dataset with 188 Stack Overflow discussions; (iii) a corpus of 200 random text files generated by pasting together natural language text, source code fragments, and patches; and (iv) the dataset used by Bacchelli et al. [11], composed of 1,493 development emails extracted from the mailing lists of four Java projects (i.e., ArgoUML, Freenet, JMeter, and Mina).

We address the following research question:

*How does the classification effectiveness of IRISH compare with approaches based on island parsers or their combination with machine learning?*

Specifically, we are interested in comparing the performance of IRISH to that proposed by Bacchelli et al. [11]. The latter provides two classification approaches: PETITISLAND, an approach based on island parsing, which improves a previously proposed approach [25], and MUCCA (eMail Unified Content Classification Approach), which combines the output of PETITISLAND with machine learning (specifically Naive Bayes and Decision Trees [26]) in a two-step unified approach. We chose to compare IRISH with PETITISLAND
and Mucca as, to the best of the authors’ knowledge, the latter two approaches constitute the state-of-the-art for what concerns classification of unstructured source content.

To evaluate the effectiveness of Irish and conduct the comparison, we rely on two broadly used IR metrics: precision and recall. In particular, for each language syntax i, precision ($P_i$) and recall ($R_i$) are defined as:

$$P_i = \frac{TP_i}{TP_i + FP_i}; \quad R_i = \frac{TP_i}{TP_i + FN_i}$$

where $TP_i$ is the number of tokens correctly classified in the language syntax i; $FP_i$ is the number of tokens wrongly classified as i; and $FN_i$ is the number of tokens wrongly classified in a language syntax different from i. To provide an aggregate value of precision and recall, we use their harmonic mean, named F-measure and defined as:

$$F_i = \frac{2 \cdot P_i \cdot R_i}{P_i + R_i}$$

4.1. Study design and context

In this section we describe in detail the four experiments we conducted to evaluate Irish and compare it with state-of-the-art techniques. In the following, such experiments, referred as E1, ..., E4, consists of datasets with increasing noisy in data. Table 3 provides a summary of the four experiments, indicating the purpose of each study, the source of information, the granularity level of the classification, and the approach (PetitIsland or Mucca) being compared with Irish.

As our starting point, we consider edited and proof-read text books written by professionals (E1-BOOK), which are one of the cleanest and most accurate forms of text in which languages (i.e., source code and natural language) are interleaved. Subsequently we continue with Stack Overflow posts (E2-SO), which should be clearly tagged by the users, and randomly extracted text of different categories (E3-RND). Finally, we classify the extremely noisy content of development emails (E4-DEV).

While in E1-BOOK, E2-SO, and E3-RND the comparison will be performed between Irish and PetitIsland, for E4-DEV we compare Irish with Mucca. In essence, in each experiment we compare Irish with the alternative approach that according to previous work [11] works better for that particular kind of dataset. Both Irish and PetitIsland perform the
Table 3: Summary of the four experiments conducted.

<table>
<thead>
<tr>
<th>Exp</th>
<th>Purpose</th>
<th>Dataset</th>
<th>Granularity</th>
<th>Comparing Irish with</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1-BOOK</td>
<td>Assess Irish on sources with a relatively clear separation between source code and natural language.</td>
<td>Annotated textbooks</td>
<td>token</td>
<td>PetitIsland</td>
</tr>
<tr>
<td>E2-SO</td>
<td>Assess Irish where separators exist but they are not consistently used (e.g., Stack Overflow).</td>
<td>Stack Overflow</td>
<td>token</td>
<td>PetitIsland</td>
</tr>
<tr>
<td>E3-RND</td>
<td>Assess Irish on random combinations of natural language, source code, and patches.</td>
<td>Randomly generated text</td>
<td>token</td>
<td>PetitIsland</td>
</tr>
<tr>
<td>E4-DEV</td>
<td>Assess Irish on noisy, real-world content.</td>
<td>Mailing lists</td>
<td>line</td>
<td>MUCCA</td>
</tr>
</tbody>
</table>

classification at token level, whereas MUCCA works at line level [11]. To compare the performances of Irish with those of MUCCA, we convert our token-level classification into a line-level classification. This is done by considering, for each line of the file, the class to which the majority of the tokens belong. For example, if a line contains 100 tokens, 70 of which have been classified as “natural language text” and 30 as “Java source code” then the line will be classified as “natural language text”.

4.1.1. E1-BOOK: Annotated textbooks

The goal of this study is to assess Irish with the purpose of understanding to what extent it is able to classify different coded language elements when their separation is relatively clear and consistent. This, for example, happens in textbooks. Textbooks are proof-read and edited by professionals. They mostly contain Natural Language Text (NLT) interleaved with some well defined portions of Source Code (SRC). In the dataset used in experiment E1-BOOK, we consider two classic textbooks related to computer programming: Thinking in Java (about the Java language) and Programming in C (about the C language). They both contain several source code examples. To produce an oracle for our classifier, we rely on the presence HTML/latex specific tags enclosing different elements (as shown in Table 4).

For each book, we use natural language transition probabilities estimated from the Frankenstein novel. Instead, for the source code HMM, we adopt a collection of source code files drawn from software systems written in the
Table 4: Annotated textbooks used in experiment E1-BOOK.

<table>
<thead>
<tr>
<th>Text Book</th>
<th>Source code tagged with</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Thinking in Java, 3rd Edition (Bruce Eckel)</td>
<td><code>&lt;p class='code'&gt;...&lt;/p&gt;</code> <code>&lt;p class='codeInline'&gt;...&lt;/p&gt;</code></td>
</tr>
<tr>
<td>2. Programming in C: A Tutorial (Brian W. Kernighan)</td>
<td><code>&lt;pre&gt;...&lt;/pre&gt;</code></td>
</tr>
</tbody>
</table>

same programming language described by each textbook. In particular, we use JEdit (version 3) source code for Java and the Linux Kernel (version 3.9-rc6) source code for C.

In this experiment, we compare IRISH with PETITISLAND, because it is the most effective method among those presented by Bacchelli et al. [11] to distinguish source code from natural language. PETITISLAND is based on a Java island grammar manually written and derived from the official Java Language Specification [27]; the parser does not require a training set, thus we use it on the entire dataset.

4.1.2. E2-SO: Stack Overflow dataset

The goal of this study is to assess IRISH with the purpose of understanding to what extent it is able to classify different coded language elements in forums, where although there exist separators between different coded languages, these are not consistently used. Stack Overflow is a popular Question and Answer (Q&A) website. Q&A services provide developers with the infrastructure to exchange knowledge in form of questions and answers: Developers pose questions and receive answers regarding issues from people that are not part of the same project. Stack Overflow has gained popularity among developers and is becoming an important venue for sharing knowledge on software development [28]. To improve the readability of posts, specific tags are used to label source code fragments. Given this possibility, and the collaborative editing done by Stack Overflow users, the quality of the text of the posts should be similar to that of textbooks. Nevertheless, in practice this does not always happen, because authors may not do tagging properly and users may have different opinions on what should be marked as code.

As a second experiment, to assess IRISH and compare it with the state of the art, we use a dataset of 188 Stack Overflow posts (the same used by Rigby and Robillard to evaluate a traceability recovery approach [16]). The
Table 5: Stack Overflow posts used in experiment E2-SO, by project tag.

<table>
<thead>
<tr>
<th>Project tag</th>
<th>Project website</th>
<th>Posts</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Hibernate</td>
<td><a href="http://www.hibernate.org">http://www.hibernate.org</a></td>
<td>51</td>
</tr>
<tr>
<td>3. HttpClient</td>
<td><a href="http://hc.apache.org">http://hc.apache.org</a></td>
<td>74</td>
</tr>
</tbody>
</table>

dataset regards posts pertaining to three Java project tags (namely Android, Hibernate, and HttpClient), as depicted in Table 5. Before conducting our analysis, we manually reinspected all the posts and fixed any incorrect tags in the oracle.

We evaluate IRISH under two different training conditions: in-domain and out-domain. The first is the same used in E1-BOOK, i.e., natural language and source code HMMs estimated respectively from the Frankenstein novel and source code files drawn from jEdit. The second is a 10-fold internal cross validation where StackOverflow posts are partitioned in 10, almost equal sized, random sets. Each set is used for testing IRISH trained with natural language and source code examples drawn from the remaining 9 sets.

As done in E1-BOOK, in E2-SO we compare IRISH to PetitIsland, which does not require training.

4.1.3. E3-RND: Randomly generated text

The goal of this study is to assess IRISH with the purpose of understanding to what extent is it able to classify multiple coded languages, such as natural language, source code, and patches, randomly combined together. In the following experiment (E3-RND), we build an artificial corpus of textual files by combining three different kinds of language syntaxes: Natural Language text (NLT), Source Code (SRC), and Patches (PCH). Source code and patches are similar languages, because they both contain source code, but the latter presents some peculiarities. For example, consider the patch in Figure 5, which follows the broadly used unified diff format [29]. In contrast to normal source code, a patch presents a special header (first three lines) and changed lines are preceded by plus or minus sign. Moreover, many grammar productions might be incomplete, because reported only as context.

A textual file is generated by pasting together pieces of information uni-
/*
 * public void actionPerformed(ActionEvent event) {
   super.actionPerformed(event);
   - new Import(ArgoFrame.getInstance());
   + if (ImporterManager.getInstance().hasImporters()) {
   +   new Import(ArgoFrame.getInstance());
   + } else { LOG.info("Import dialog not shown"); }
   }
 */

Figure 5: Example of patch in unified diff format.

form randomly chosen from: (i) source code C files from the Linux Kernel,\(^5\) (ii) patch proposals, and (iii) natural language text from four Linux patchwork repositories\(^6\). We extract natural language from the patch textual comments after a manual purification, which consists of eliminating automatic mailman directives (e.g., from, reply, suggested by) and any embedded code and stack trace fragments. An example of random text used in this experiment is shown in Figure 6. The Linux patchwork repository is organized in different sub-projects, and patches are attached as supplementary files separated from the body of the message. For the scope of this experiment we select 50 random mailing list messages from the following Linux patchwork projects: linux-pci (Linux PCI development list), linux-pm (Linux power management), linux-nfs (Linux NFS mailing list), and LKML (Linux Kernel mailing list). Table 6 shows, for each mailing list, the distribution among random generated languages categories.

We perform a specific cross validation which leaves out three of the four considered mailing lists and uses the remaining one for testing. Writing and programming styles could affect the transition probability estimation of the corresponding HMM. For example, a source code HMM estimated on C source files coming from two different software systems may not be the same

--- src/org/argouml/ui/Explorer.java (revision 14345)
++ src/org/argouml/ui/Explorer.java (revision 15443)
@@ -64,4 +72,7 @@
        */
       public void actionPerformed(ActionEvent event) {
         super.actionPerformed(event);
-         new Import(ArgoFrame.getInstance());
+         if (ImporterManager.getInstance().hasImporters()) {
+           new Import(ArgoFrame.getInstance());
+         } else { LOG.info("Import dialog not shown"); }
       }

\(^5\)Kernel version 3.9-rc6, available at www.kernel.org
\(^6\)Available at https://patchwork.kernel.org
Now that we have the necessary drivers and Device Tree informations to support PCIe on Armada 370 and Armada XP, enable the CONFIG_PCI option. Also, since the Armada 370 Mirabox has a built-in USB XHCI controller connected on the PCIe bus, enable the corresponding options as well.

To fix this use the observation that before commit 1aeeae8 acpi_pci_slot_add() was always run after pci_bus_add_devices() and that happened to the acpiphp's .add() callback routine too. Thus it is safe to reorder acpi_pci_root_add() to make the PCI root drivers' .add() callbacks be run after pci_bus_add_devices(), so do that.

This approach was previously proposed by Myron Stowe.

The commit mentioned in the changelog above is in linux-pm.git/linux-next.

Thanks,
Rafael

Figure 6: An example of randomly-generated text file containing NLT (Natural language text), SRC (Source code), and PCH (Patches).
transition probabilities estimated on three out of the four considered mailing lists left out.

**E3-RND.3:** Source code transition probabilities estimated on a collection of Linux kernel (version 3.9-rc6) C source code files not used for random text generation; natural language transition probabilities estimated on the Frankenstein novel; and patches transition probabilities estimated on three out of the four considered mailing lists left out.

**E3-RND.4:** Source code transition probabilities estimated on a collection of Linux kernel (version 3.9-rc6) C source code files not used for random text generation; natural language text and patches transition probabilities estimated on three out of the four considered mailing lists left out.

The training condition E3-RND.4 encloses the writing and programming styles adopted in the Linux Patchwork mailing lists in both natural language text and source code HMMs, as the samples are taken from the same domain used for testing. Instead, the training conditions E3-RND.1, E3-RND.2, and E3-RND.3 use source code and natural language text taken from a completely or partially different domain. Table 7 summarizes how training samples are combined for each training condition.

| Table 6: Randomly generated text - distribution of categories across tokens. |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| Category         | Linux-nfs | PKML | Linux-pci | Linux-pm | Total |
| Natural language (NLT) | 27.0% | 30.8% | 30.8% | 19.2% | 27.6% |
| Patch (PCH)      | 35.0% | 30.7% | 34.5% | 44.6% | 35.5% |
| Source code (SRC) | 38.0% | 38.5% | 34.7% | 36.2% | 36.9% |

| Table 7: E3-RND: Training conditions. |
|-----------------|-----------------|-----------------|-----------------|
| Condition       | PostgreSQL | Linux Kernel | Frankenstein novel | Patchwork comments | Patchwork Patches |
| E3-RND.1        | √           |                |                |                | √             |
| E3-RND.2        | √           |                |                | √              | √             |
| E3-RND.3        |              |                | √              |                | √             |
| E3-RND.4        |              | √              |                |                | √             |
In this case, we evaluate a PetitIsland setup that uses two parsers (i.e., source code parser and patch parser) on the entire dataset (as it does not require training data). As tested previously [11], PetitIsland obtains the best results when the two parsers are used in chain: First the patch parser, and then the source code parser on the parts not already recognized.

4.1.4. E4-DEV: Java Development Mailing Lists

The goal of this study is to assess IRISH with the purpose of investigating how it works when analyzing the noisy content of development mailing lists of real world projects. We considered the dataset manually prepared and validated by Bacchelli et al. [11]. It consists of 1,493 emails sampled from the development mailing lists of four unrelated open-source Java projects. The considered samples (summarized in Table 8) are representative of the respective populations with a 95% confidence level and a ±5% error margin.

The rationale of such an experiment is two-fold: (i) considering a very noisy source, where elements written using different syntaxes are not consistently separated and where one of the language categories is specifically junk, or text that is not relevant to the analysis of the data (e.g., author's signatures and random characters); and (ii) evaluating IRISH on a benchmark created and previously validated by other researchers. Table 9 reports the distribution of different language syntax categories (natural language, junk, patches, source code, and stack traces) over the lines of the analyzed emails. One could notice from the second row of Table 9 that the percentage of junk is quite high (ranging between 36.5% and 52.3%). For emails, this is due to the amount of text contained in the email header.

Table 8: Java Development Mailing Lists Dataset - number of manually classified emails.

<table>
<thead>
<tr>
<th>System</th>
<th>Inception</th>
<th>Total</th>
<th>After filtering</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>Jan 2000</td>
<td>25,538</td>
<td>25,538</td>
<td>379</td>
</tr>
<tr>
<td>Freenet</td>
<td>Apr 2000</td>
<td>23,134</td>
<td>23,134</td>
<td>378</td>
</tr>
<tr>
<td>JMeter</td>
<td>Jan 2006</td>
<td>24,005</td>
<td>5,814</td>
<td>361</td>
</tr>
<tr>
<td>Mina</td>
<td>Feb 2001</td>
<td>21,384</td>
<td>14,499</td>
<td>375</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>94,061</td>
<td>68,985</td>
<td>1,493</td>
</tr>
</tbody>
</table>

In this case, due to the presence of the JNK category (which is highly irregular and difficult to express in a grammar), PetitIsland do not perform particularly well on this dataset. For this reason, we compare IRISH with Mucca, the unified (machine learning plus island parsers) approach.
Table 9: Java Development Mailing Lists Dataset - distribution of categories across lines.

<table>
<thead>
<tr>
<th>Category</th>
<th>ArGoUML</th>
<th>Freenet</th>
<th>JMeter</th>
<th>Mina</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural language</td>
<td>47.2%</td>
<td>59.6%</td>
<td>41.8%</td>
<td>51.2%</td>
<td>48.9%</td>
</tr>
<tr>
<td>Junk</td>
<td>47.9%</td>
<td>30.8%</td>
<td>52.3%</td>
<td>36.5%</td>
<td>43.6%</td>
</tr>
<tr>
<td>Patch</td>
<td>2.0%</td>
<td>7.4%</td>
<td>1.8%</td>
<td>2.3%</td>
<td>3.1%</td>
</tr>
<tr>
<td>Source code</td>
<td>1.3%</td>
<td>0.2%</td>
<td>3.2%</td>
<td>7.8%</td>
<td>2.8%</td>
</tr>
<tr>
<td>Stack trace</td>
<td>1.6%</td>
<td>1.9%</td>
<td>0.9%</td>
<td>2.3%</td>
<td>1.6%</td>
</tr>
</tbody>
</table>

designed by Bacchelli et al. [11]. In this case, for both Irish and MUCCA, we conduct cross mailing list validation: We train on three mailing lists and we test on the remaining one; we repeat this for each list and we average the results. It is important to point out that using MUCCA in not necessary for all other datasets, where specific language island parsers—part of PetitIsland—guarantee good performances already, and as shown in a previous work [11] MUCCA does not introduce any improvement: on the contrary, it could produce worse results. In summary, we always compare Irish with the approach that performs better for each specific kind of dataset.

Similarly to E2-SO we evaluate Irish under a further training condition where for natural language and source code HMMs we use the Frankenstein novel and source code files drawn from jEdit respectively, while for the other language syntaxes examples are drawn from the cross-validation training set.

5. Empirical Study Results

In the following, we report results of the study described in Section 4. Specifically, we first report results for the four experiments E1 ... E4. Then, in Section 5.5 we discuss the effort/difficulty tradeoffs between writing island parsers (when using the alternative approaches) and training Irish.

5.1. E1-BOOK: Annotated textbooks

Table 10 reports results of Irish and PetitIsland performed with annotated textbooks (E1-BOOK), as well as their differences.

For Irish, the F-measure achieved for natural language in both textbooks is higher than for source code. This means that, while the simple example of the Frankenstein novel textbook is adequate to model natural language enclosed in the considered programming textbooks, it is certainly not enough to fully capture the source code characteristics. This is especially
Table 10: E1-BOOK: Annotated textbooks - results of Irish and PetitISLAND.

<table>
<thead>
<tr>
<th>Text Book</th>
<th>Class</th>
<th>IRISH P</th>
<th>IRISH R</th>
<th>IRISH F</th>
<th>PETITISLAND P</th>
<th>PETITISLAND R</th>
<th>PETITISLAND F</th>
<th>IRISH-PETITISLAND P</th>
<th>IRISH-PETITISLAND R</th>
<th>IRISH-PETITISLAND F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thinking in Java</td>
<td>NLT</td>
<td>0.923</td>
<td>0.958</td>
<td>0.940</td>
<td>0.923</td>
<td>0.996</td>
<td>0.957</td>
<td>0.000</td>
<td>-0.038</td>
<td>-0.017</td>
</tr>
<tr>
<td></td>
<td>SRC</td>
<td>0.888</td>
<td>0.806</td>
<td>0.845</td>
<td>0.987</td>
<td>0.796</td>
<td>0.881</td>
<td>-0.099</td>
<td>0.010</td>
<td>-0.036</td>
</tr>
<tr>
<td>Programming in C</td>
<td>NLT</td>
<td>0.949</td>
<td>0.925</td>
<td>0.937</td>
<td>0.897</td>
<td>0.999</td>
<td>0.945</td>
<td>0.052</td>
<td>-0.074</td>
<td>-0.008</td>
</tr>
<tr>
<td></td>
<td>SRC</td>
<td>0.814</td>
<td>0.868</td>
<td>0.840</td>
<td>0.995</td>
<td>0.697</td>
<td>0.820</td>
<td>-0.181</td>
<td>0.171</td>
<td>0.020</td>
</tr>
</tbody>
</table>

because the source code examples are drawn from real software systems (PostgreSQL for C language and Jedit for Java language). It is likely that coding styles adopted in textbooks differ from coding practices adopted by developers of complex software systems. The former are directed to teach basic programming techniques, the latter could conduct very sophisticated coding constructs especially for C. The C language in the second textbook is predicted more accurately than the Java language in the first textbook (F-measure 0.845). This is because the first textbook contains chapters related to JavaServer Pages\(^7\) (JSP) script examples, which are syntactically different from pure Java code.

For PETITISLAND, results on the Java textbook are quite good, and this is expectable because the grammar was written for the Java language. Nevertheless, recall of source code exhibits a lower performance; this is mostly due to the JSP fragments. Concerning the C textbook, performance is lower (especially in terms of recall of source code) because we used the unmodified island parser implemented for Java. We expect higher results (similar to those for the Java book) with a proper island grammar for C.

5.2. E2-SO: Stack Overflow dataset

Table 11 reports results of experiment E2-SO performed with the StackOverflow dataset and obtained with Irish and PetitISLAND, and their differences. Results indicate high performance for both Irish and PetitISLAND, although, as expected, PetitISLAND performs slightly better (F-measure up to 5.4% better than Irish for natural language text and 11.4% better for source code). In Irish in-domain training exhibits higher performance than out-domain training where natural language and source code

\(^7\)http://www.oracle.com/technetwork/java/jsp-138432.html
models have been trained with the Frankenstein novel and JEdit, respectively. For in-domain training natural language false positives are ascribed almost all to source code comments, while a major cause cannot be found to explain the tokens wrongly classified as source code. In general, it can happen that the HMM learns that a particular sequence of tokens indicates the presence of source code, however such a sequence can also occur in other communication elements such as natural language. One reason is related to parentheses and special characters, adopted sometime in natural language text, arranged similarly to a function call that induce the HMM to switch to the source code category and after a while return to the natural language category. The out-domain training still preserves good performances if compared with in-domain training.

As expected from the results of previous work [11], the PetitIsland approach works at its full potential on this dataset. In fact, the dataset is composed only of natural language and Java fragments, for which the island grammar had been implemented specifically.

![Table 1](image)

<table>
<thead>
<tr>
<th>Class</th>
<th>Irish (in-domain training)</th>
<th>Irish (out-domain training)</th>
<th>PetitIsland</th>
<th>PetitIsland</th>
<th>PetitIsland</th>
<th>PetitIsland</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>0.946</td>
<td>0.980</td>
<td>0.914</td>
<td>0.987</td>
<td>0.937</td>
<td>0.961</td>
</tr>
<tr>
<td>R</td>
<td>0.968</td>
<td>0.954</td>
<td>0.904</td>
<td>0.999</td>
<td>0.988</td>
<td>0.994</td>
</tr>
<tr>
<td>F</td>
<td>0.995</td>
<td>0.994</td>
<td>0.994</td>
<td>0.994</td>
<td>0.994</td>
<td>0.994</td>
</tr>
</tbody>
</table>

5.3. E3-RND: Randomly generated text

Table 12 reports results of experiment E3-RND performed with random text obtained with Irish under four different training conditions (respectively E3-RND.1, E3-RND.2, E3-RND.3, and E3-RND.4). The first training condition (E3-RND.1) uses, for source code, training samples coming from a different test set domain. The second training condition (E3-RND.2) uses, for source code, training samples coming from a different test set domain (PostgreSQL development community). The third training condition (E3-RND.3) uses, for natural language, training samples coming from a different domain (Frankenstein novel). The fourth training condition (E3-RND.4) uses, for each language syntax (NLT, SRC, and PCH), training samples coming from the same domain of the test set (the Linux kernel development community).

As expected, the best performance for Irish is obtained under the training condition E3-RND.4 (F-measure ranging between 0.93 and 0.98), where
natural language, source code, and patch syntaxes are modeled with examples coming from a domain that is closely related to the domain of examples we wish to classify. When the natural language is modeled on examples of a different domain (E3-RND.2 and E3-RND.3) the corresponding NLT prediction performance decreases (F-measure around 0.82). Instead, the NLT prediction performance persists on almost the same level in E3-RND.1 and E3-RND.4 (F-measure around 0.93).

Different results can be observed for the source code. When the source code is modeled with examples of a different domain (E3-RND.1 and E3-RND.2) the decrement in prediction performance affects both source code and patches (F-measure ranges between 0.75 and 0.81). This is because the source code model obtained from PostgreSQL is not adequate to distinguish between source code and patches. Many source code snippets are in fact classified as patches causing a significant decrement of source code Recall and similarly a significant decrement of patch Precision.

Table 13 reports the results of PetitIsland for the entire E3-RND dataset, as no training is required and therefore we do not have the various training conditions in E3-RND.1 – E3-RND.4. The values underline the need for a more suitable island parser here. In fact, the dataset used for this evaluation contains source code snippets and patches of C language, while the current implementation of PetitIsland is based on a Java grammar. Although the lexicon of Java and C are similar, the differences are enough to lower the performance. The higher performance for the patch languages is due to the fact that the unified diff format is the same for both Java and C.

Table 12: E3-RND: Randomly generated text - results of IRISH at different training conditions.

<table>
<thead>
<tr>
<th>Class</th>
<th>E3-RND.1</th>
<th>E3-RND.2</th>
<th>E3-RND.3</th>
<th>E3-RND.4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>NLT</td>
<td>0.914</td>
<td>0.949</td>
<td>0.931</td>
<td>0.930</td>
</tr>
<tr>
<td>SRC</td>
<td>0.971</td>
<td>0.671</td>
<td>0.794</td>
<td>0.945</td>
</tr>
<tr>
<td>PCH</td>
<td>0.680</td>
<td>0.930</td>
<td>0.786</td>
<td>0.716</td>
</tr>
</tbody>
</table>

Table 13: E3-RND: Random generated text - results of PetitIsland.

<table>
<thead>
<tr>
<th>Class</th>
<th>P</th>
<th>R</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLT</td>
<td>0.767</td>
<td>0.713</td>
<td>0.739</td>
</tr>
<tr>
<td>SRC</td>
<td>0.749</td>
<td>0.822</td>
<td>0.784</td>
</tr>
<tr>
<td>PCH</td>
<td>0.967</td>
<td>0.993</td>
<td>0.980</td>
</tr>
</tbody>
</table>
5.4. E4-DEV: Java Development Mailing Lists

Table 14 reports results of IRISH and MUCCA (E4-DEV) performed with Bacchelli et al. dataset. IRISH was run under two different training conditions: in-domain and out-domain training. The first uses, in each cross validation run, training examples coming exclusively from three out of four mailing lists. Instead, the second adopts, just for natural language and source code, outsider training examples, i.e. Frankenstein novel for natural language and PostgreSQL for source code. Curiously in the last training condition the most affected language syntaxes are neither natural language nor source code (F-measure drops from 0.795 to 0.749 for source code and from 0.913 to 0.886 for natural language). Instead, the highest performance decrease is registered for patches and stack traces: F-measure drops from 0.749 to 0.637 for patch and from 0.861 to 0.788 for stack trace. This can be explained by observing precision and recall. Natural language performance is balance by and improvement of Precision at the cost of Recall, while source code is balanced by an increment of Recall at the cost of Precision. For patches and stack traces precision and recall are compromised, because natural language is confused with patches and stack traces.

MUCCA exhibits better performance than HMM for all language in terms of F-measure, namely between 3.2% and 21% better than IRISH for in domain training, and between 5.9% and 32.2% better than IRISH for out domain training.

Table 14: E4-DEV: Java Development Mailing Lists - results of IRISH and MUCCA.

<table>
<thead>
<tr>
<th>Class</th>
<th>IRISH (in domain training)</th>
<th>IRISH (out domain for NLT and SRC)</th>
<th>MUCCA (in domain training)</th>
<th>MUCCA (out domain for NLT and SRC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLT</td>
<td>0.877 0.952 0.913</td>
<td>0.910 0.864 0.886</td>
<td>0.937 0.953 0.945</td>
<td>-0.060 -0.001 -0.032</td>
</tr>
<tr>
<td>SRC</td>
<td>0.741 0.857 0.795</td>
<td>0.646 0.892 0.749</td>
<td>0.880 0.889 0.885</td>
<td>-0.129 -0.032 -0.090</td>
</tr>
<tr>
<td>PCH</td>
<td>0.723 0.778 0.749</td>
<td>0.555 0.747 0.637</td>
<td>0.990 0.929 0.959</td>
<td>-0.287 -0.151 -0.210</td>
</tr>
<tr>
<td>STR</td>
<td>0.806 0.923 0.861</td>
<td>0.685 0.927 0.788</td>
<td>0.997 0.972 0.984</td>
<td>-0.191 -0.049 -0.123</td>
</tr>
<tr>
<td>JNK</td>
<td>0.944 0.834 0.886</td>
<td>0.880 0.876 0.878</td>
<td>0.943 0.929 0.936</td>
<td>-0.001 -0.095 -0.050</td>
</tr>
</tbody>
</table>

5.5. Discussion: Training supervised approaches vs. writing island parsers

Figure 7 shows the learning curve of the HMM approach obtained with the Java Development Mailing List (E4-DEV) dataset. Steady state levels, corresponding to the results of experiment E4-DEV (Table 14) are reached when all items of the training set are used. Natural language and junk reach the maximum performance with 3 email examples. Stack trace, source code
and patch need more examples to reach their steady state values (between 10 and 20 email examples). Moreover Stack trace and source code exhibit a transitory behavior which is due to the noisy nature of such languages.

Although PetitIsland does not require a training set, it still requires an implementation of the subject grammar. When an accurate implementation exists (e.g., the Java grammar used in the experiments), the approach reaches better performance than Irish (i.e., on E1-BOOK, E2-SO, E4-DEV, with Java datasets), and—being a full-fledged parser—has the advantage of parsing the structured content, thus providing an abstract syntax tree from which facts can be extracted. Nevertheless, implementing a correct island grammar is not a trivial task: it requires an engineering process, which is currently not even semi-automated. Given the previous experience [11], we can estimate that implementing a new island grammar for another programming language (e.g., C) would require at least two days of work, for a trained person. Moreover, for the unified approach MUCCA, a training phase is required for the machine learning aspect.

On the contrary, the training phase of Irish can be conducted by less trained individuals. In fact, it does not require special knowledge, but only a pattern recognition process (i.e., specifying which parts belong to which language), for which humans are naturally fitted. However, Irish only conducts classification, thus it does not provide an abstract syntax tree for later fact extraction. In the case of fact extraction, it would be necessary for a second phase to parse the parts classified by Irish, thus we deem PetitIsland a better approach for this particular application.

6. Threats to Validity

This section describes threats that can affect the validity of the approach validation, namely construct, conclusion, reliability, and external validity.

6.1. Construct Validity

Threats to construct validity may be related to imprecisions in our measurements. In particular, such threats can be due to (i) how the oracles for the different experiments have been built, and (ii) the extent to which the metrics used to evaluate the performance of Irish and compare it with PetitIsland. Concerning the oracles, errors are unlikely in E1-BOOK because of the presence of clear separators between source code and natural language text. The dataset of E3-RND is correct by construction since it is artificially
generated. Finally, data from E2-SO and E4-DEV was manually validated, because the assumption that different elements of Stack Overflow discussions are kept separated by proper tags is not always valid, and it is not valid for emails either.

When—as in the case of E4-DEV i.e., of emails—it was not possible to manually validate the whole dataset (being very large), as explained in Section 4.1.4 the manual validation was performed on a statistically representative sample of the population, with 95% confidence level and a ±5% error margin.

One point that is worth discussing here is the subjectiveness in the manual classification. For example, comments may or may not be considered as natural language text. More important, the presence of references to method calls in natural language sentences—especially when such calls include a complete signature—may or may not qualify as a source code snippet.

Figure 7: Learning curve of HMM.
6.2. Conclusion Validity

Threats concerning the relationship between the treatment and the outcome may affect the statistical significance of the outcomes. We performed our experiments with representative samples letting us to obtain outcomes with an adequate level of confidence and error.

6.3. Reliability validity

Threats to reliability validity concern the capability of replicating this study and obtaining the same results. Scripts and datasets used to run the experiments are available online.\(^8\)

6.4. External Validity

Threats concerning the generalization of results may induce the approach to exhibit different performance when applied to other contexts and/or different language syntaxes. We have conducted five different studies, using datasets from a wide variety of sources (textbooks, Stack Overflow, mailing list), and considering different kinds of language categories (natural language text, source code written in different languages, patches, stack traces, and junk elements). Noticeably, we have also shown the capability of IRISH to exhibit good performances when performing a cross-source validation (see Section E3-RND, where a cross-mailing lists validation was performed). Having said that, we are aware that further empirical studies would always be beneficial.

7. Related Work

The problem of extracting useful models from textual software artifacts has been approached mainly by combining three different techniques: regular expressions, island parsing, and machine learning. To this aim, a first approach has been proposed by Murphy and Notkin [13]. They proposed a lightweight lexical approach based on regular expressions that a practitioner should follow to extract pattern of interests (e.g., source code, function calls or definitions). Approaches based on island grammars are able to extract parts encoded with a formal language of interest from generic free text [30].

\(^8\)http://www.rcost.unisannio.it/cerulo/dataset-irish.tgz
Bettenburg et al. developed infoZilla, a tool to detect and extract patches, stack traces, source code, and enumerations from bug reports and their discussions [12]. They adopted a fuzzy parser and regular expressions to detect well defined formats of each coded information category obtaining, on Eclipse bug reports, an accuracy of 100% for patches, and 98.5% for stack traces and source code. A different, simple yet accurate approach developed by Bettenburg et al. [31] is based on the use of available spell-checkers to distinguish different pieces of information contained in unstructured sources. Specifically, the use of a spell-checker is followed by further processing, in which elements that are not recognized by the spell-checker are analysed to check the presence of camel case separators, programming language keywords and special characters. This is because these three elements characterize source code rather than free text. Results of an empirical evaluation conducted by the authors indicated a precision between 84% and 88% and a recall between 64% and 68%. While extremely simple to apply, this approach relies on the availability of a specific spell-checker and on specific sets of keywords. Also, especially when natural language is interleaved with code elements (e.g., method or class names) the adopted heuristics may fail.

Tang et al. proposed an approach to clean email data for subsequent text mining [32]. They used an approach based on Support Vector Machines to detect source code fragments in emails obtaining a precision of 93% and a recall of 72%. Bacchelli et al. [11] introduced a supervised method that classify lines into five classes: natural language text, source code, stack traces, patches and junk text. The method combines term based classification and parsing techniques, obtaining a total accuracy ranging between 89% and 94%.

Although such approaches are lightweight and exhibit promising levels of performance they may be affected by the following drawbacks:

- **Granularity level.** Most of the methods, in particular those based on machine learning techniques, classify lines. Our method classifies tokens, thus reaching a finer level of granularity useful for high interspersed language constructs.

- **Training effort.** Methods based on island parsers and regular expressions require expertise for the parser or the regular expression construction. Furthermore, they work well on the corpus adopted for the construction of the parser, however are not generalizable. Our technique learns directly from data—and a small dataset, e.g., less than 20
mails, is generally sufficient for a good training—and does not require particular skills.

- **Parser limitations.** Context free parsers or regular expression parsers rely on deterministic finite state automata designed on pre-defined patterns. For example, in modeling the patch language syntax, Bacchelli et al. search for lines surrounding two `@@`s. This may be a limitation if such a pattern is not consistently used or exhibits some variations. Sometimes developers may report only the modified lines by copying the output of a differencing tool, and such output is slightly different from source code. Our method is based on Markov models, which rely on a nondeterministic finite state automaton making the detection of noisy languages, such as stack traces, more robust.

- **Extension.** Since using island parsers and/or regular expressions requires a significant expertise, introducing a new language syntax can be problematic. We propose a method that learns directly from data, thus requiring an adequate number of training samples to model the language syntax of interest.

An application of a HMM for extracting structured information from unstructured free text has been proposed by Skounakis et al. [33]. They represent the grammatical structure of sentences with a hierarchical hidden Markov model and adopt a shallow parser to construct a multilevel representation of each sentence to capture text regularities. The approach has been validated in a biomedical domain for extracting relevant biological concepts.

8. Conclusions and Future Work

This paper described IRISH (InfoRmation ISlands Hmms), an approach based on Hidden Markov Models to identify coded information—such as source code fragments, stack traces, or logs—and typically included in development mailing lists, issue reports, or discussion forums. The paper reported an evaluation of IRISH over four different datasets i.e., (i) textbooks, (ii) Stack Overflow discussions, (iii) datasets artificially composed combining source code and text from issue trackers, and (iv) mailing lists. In this evaluation, we compared IRISH with alternative, state-of-the-art approaches previously proposed by Bacchelli et al. [11], and based on the use of island
parsers (PETITISLAND) or on a combination of island parsers and machine learners (MUCCA).

Results of the study indicate that, in general, IRISH exhibits performances comparable to PETITISLAND and MUCCA. On the one hand, approaches like PETITISLAND are unsupervised, i.e., they do not require to be trained (however its extension, MUCCA, which exhibits better performance dealing with irregular text such as junk, still requires a training set). On the other hand, IRISH does not require the creation of any kind of island parser, but rather the creation of a training set. Although the latter still means some manual work needs to be done, we believe that in many circumstances this can have noticeable advantages over the construction of island parsers:

- When dealing with unstructured sources—say emails—containing several kinds of coded information, multiple island parsers would be required. Instead, training IRISH would just require a manual labeling of a set of emails.

- While writing an island parser (or a parser in general) requires some specific skills that not all developers may have, training IRISH requires to simply manually tagging different elements in the unstructured source. This is clearly an easier task than writing an island parser, therefore it does not require any particular skill. This makes it easier to adopt IRISH in a development context where people might not have specific source code analysis skills. Moreover, as shown in Figure 7, IRISH can achieve good performance with a training set of about 30 items.

- IRISH does not necessarily require training on exactly the same dataset on which it has to be applied, as demonstrated by E3-RND with only minimal loss of performances in terms of precision, recall, and F-measure.

- In presence of pattern variations in the encoded data to be extracted from the unstructured source, adapting IRISH just means re-training it. Instead, for approaches based on island parsers this means modifying the existing grammar or writing a new one, which requires significant expert knowledge.

- The classification granularity of IRISH, whatever the language category, is at the token level. Instead, for very informal and noisy language
categories, the classification of current machine learning methods, such as MUCCA, work at the line level, even if they could be adapted to work also at the token level.

Work-in-progress aims at improving IRISH, and in particular using more sophisticated HMMs in its implementation, specifically:

- **HMM alphabet.** The token alphabet has been designed for general purposes. It can be improved by exploiting the language syntax to be detected. To this aim island parsers could be adopted to identify token patterns that may be meaningful and effective for a particular language syntax category. This will increase the HMM alphabet but could improve also the language detection capability.

- **High order HMM.** A HMM is also known as first-order Markov model because of a memory of size one, i.e., the current state depends only on a history of previous states of length one. The order of a Markov model is the length of the history or context upon which the probabilities of the possible values of the next state depend, making high order HMMs strictly related to n-gram models. We believe that such a capability may be useful to more precisely model language syntax and capture, for example, specific programming styles, and the “naturalness” of software that is likely to be repetitive and predictable [23].

References


