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

Luigi Cerulo*‡, Michele Ceccarelli*‡, Massimiliano Di Penta†, Gerardo Canfora†
*Dep. of Science and Technology, University of Sannio, Benevento (Italy)
†Dep. of Engineering, University of Sannio, Benevento (Italy)
‡BioGeM s.c.a r.l., Institute of Genetic Research "Gaetano Salvatore", Ariano Irpino, AV (Italy)

***Abstract*—Emails and issue reports capture useful knowledge about development practices, bug fixing, and change activities. Extracting such a content is challenging, due to the mix-up of source code and natural language, unstructured text.**

**In this paper we introduce an approach, based on Hidden Markov Models (HMMs), to extract coded information islands, such as source code, stack traces, and patches, from free text at a token level of granularity. We train a HMM for each category of information contained in the text, and adopt the Viterbi algorithm to recognize whether the sequence of tokens—e.g., words, language keywords, numbers, parentheses, punctuation marks, etc.—observed in a text switches among those HMMs. Although our implementation focuses on extracting source code from emails, the approach could be easily extended to include in principle any text-interleaved language.**

**We evaluated our approach with respect to the state of art on a set of development emails and bug reports drawn from the software repositories of well known open source systems. Results indicate an accuracy between 82% and 99%, which is in line with existing approaches which, differently from ours, require the manual definition of regular expressions or parsers.**

**Keywords:** *HMM; Natural Language Parsing; Mailing list mining.*

## I. Introduction

Data available in software repositories is useful for software analysis and program comprehension activities. In particular, mailing lists and issue tracking systems are widely adopted in open source projects to exchange information about implementation details, high-level design, bug reports, code practices, patch proposals, and malfunctioning details, such as stack traces. A common practice in open source projects is to adopt mailing lists or bug tracking systems as the sole repository of software technical documentation available.

Extracting useful, relevant, and unbiased information from such free text archives is not straightforward. In Information Retrieval (IR) free text data is usually treated as vector of words counts. This is a representation usually adopted in the treatment of free text in many software engineering approaches [1]. Although this simplification works well for pure natural language texts, in software engineering it may fail as free text, generated by developers, is often not well-formed and interleaved with different information contents written in different coding syntaxes.

In this paper we introduce an approach, based on Hidden Markov Models (HMM), to extract coded information islands, e.g. source code and natural language, from free text at a token level of granularity. We consider the sequence of tokens—e.g., English words, programming language keywords, digits, parentheses, punctuation symbols, etc.—of a development email as the emissions generated by the *hidden states* of a HMM. We use the hidden states to model a specific coded information content, e.g., source code and natural language text. The Viterbi algorithm allows us to search for the path that maximizes the probability of switching between text and source code hidden states given an emission sequence [2]. Such a path allows us to classify each observed token in the corresponding coded information category. The approach could be easily extended to include also other text interleaved languages, such as stack trace and patches. The primary novelty and point of strength of the proposed approach is that it *does not require the manual definition (case by case) of regular expressions or parsers*, generally required for alternative approaches [3], [4], [5].

The automatic detection of information contained in the free text of a development email is useful for various tasks. For example, recovering the traceability between source code and software documentation stored in email has been tackled with methods based on vector of terms [6], [7]. Such methods are not able to recognize whether a term refers to natural language or source code contexts. Instead, knowing which is the context provenance of an index term could be beneficial to improve the precision of the traceability relations by weighting more meaningful terms [3]. In particular, summarization techniques are usually designed for specific types of artifacts and cannot be applied to emails where a mixture of languages may co-exist. Thus a method able to distinguish terms coming from different email contexts is crucial.

Generally speaking, removing irrelevant information from data may improve the quality of data extraction in approaches based on information retrieval techniques where a term's indexing procedure is adopted to build a language model [8]. The detection of different coding information in textual contents allows for excluding noisy data from the indexing process. Traceability recovery [6], impact analysis [9], bug report assignment [10], [11], code/text summarization [12], [13] are approaches that could benefit from methods that are able to discriminate noisy data and relevant information.

To evaluate our approach, we adopted a set of freely available HTML books, fully annotated with source code fragments (e.g., "Thinking in Java" by Bruce Eckel), random generated text files, and sets of emails and bug reports from two well-known open source software systems (Linux Kernel and Apache httpd). Results indicate an accuracy ranging between 82% and 99%, generally in line with alternative approaches (e.g., the one by Bacchelli *et al.* [3]) that require a manual customization and/or the definition of regular expressions or parsers.

The paper is organized as follows: Section II describes related work. Section III describes the proposed approach. Section IV provides details about the empirical evaluation procedure. Section V reports and discusses the obtained results. Section VI discusses threats to the validity of the evaluation, while Section VII concludes the paper and outlines directions for future work.

## II. Related Work

The problem of extracting useful models from textual software artifacts has been approached mainly by combining three different techniques: regular expressions, island parsers, and machine learning. A first approach has been proposed by Murphy and Notkin [5]. They outlined a lightweight lexical approach based on regular expressions that a practitioner should follow to extract patterns of interests (e.g., source code, function calls or definitions). Bettenburg *et al.* developed *infoZilla*, a tool that allows to detect and extract patches, stack traces, source code, and enumerations from bug reports and their discussions [4]. 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. Tang *et al.* proposed an approach to clean e-mail data for subsequent text mining [14]. 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.* [15] 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 technique, obtaining a total accuracy ranging between 89% and 94%.

Although such approaches are lightweight and exhibit promising level 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, such approaches work well on the corpus adopted for the construction of the parser, however are not generalizable. Our method learns directly from data 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 require 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 in extracting structured information from unstructured free text has been proposed by Skounakis *et al.* [16]. 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.

## III. Methods

In the following, we first report background notions about HMM. Then, we describe our approach for identifying source code fragments in natural language text by describing a basic HMM, an improved model able to detect islands of other languages than source code (e.g., logs, XML), and by providing details about model calibration.

### A. Background notions on Hidden Markov Models

A Markov model is a stochastic process where the state of a system is modeled as a random variable that changes through time, and the distribution for 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 only partially or none observable [17]. In other words, the state is not directly visible, while outputs, dependent on the state, are 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.

Hidden Markov models are especially known for their application in temporal pattern recognition such as speech, handwriting, and gesture recognition [18], [19], part-of-speech tagging [20], musical score following [21], and bioinformatics analyses, such as CpG island detection and splice site recognition [22].

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

- $\Sigma$ is an alphabet of output symbols;

- $Q$ is a finite set of states capable of emitting output symbols from alphabet $\Sigma$;
- $T$ 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 $b \in \Sigma$, such that for each $k \in Q$, $\sum_{i \in \Sigma} e_{ki} = 1$.

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, the probability that the sequence $X$ was generated by the HMM, given the path $\Pi$, is determined by:

$$P(X|\Pi) = t_{\pi_0 \pi_1} \prod_{i=1}^{L} e_{\pi_i x_i} t_{\pi_i \pi_{i+1}}$$

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:

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

The Viterbi algorithm is able to find such a path in $O(L \cdot |Q|^2)$ time [2]. As a clarification example, let us consider the classical unfair-casino problem [22]. To improve the chance of winning, a dishonest casino uses loaded dice occasionally, while most of the time using 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 equal to $1/6$. Suppose the loaded dice has the following probability distribution: $P(1) = P(2) = P(3) = P(4) = P(5) = 1/10$ and $P(6) = 1/2$. We are interested to unmask the casino. Thus, given a sequence of throw data, we would like to known when the casino uses a fair dice and when a loaded dice.

The HMM representing the rolling dice game is shown in Fig. 1, where the alphabet is $\Sigma = \{1, 2, 3, 4, 5, 6\}$, and the state space is $Q = \{FairDice, LoadedDice\}$. Suppose the dishonest casino switches between the two hidden states, fair dice and loaded dice, with the transition probabilities shown in the figure. When the casino uses the fair dice the emission probabilities are those of the fair dice, while when it uses the loaded dice the emission probabilities are those of the loaded dice. Let us assume that we observe the following sequence of rollings: 1, 2, 6, 4, 3, 6, 5, 2, 6, 6, 4, 1, 3, 6, 6, 6, 6, 6, 6, 6, 6, 5, 4, 6, 1, 6. We cannot tell which state each rolling is in. For example, the subsequence 6, 6, 6, 6, 6, 6 may happen using the loaded dice or it can happen using the fair dice even though the later case has less probability. The state is hidden from the sequence, e.g., we cannot determine the sequence of states from the given sequence. The Viterbi algorithm is able to determine (decode) the most probable sequence of states emitting the observed sequence of symbols [2].

TABLE I
THE TOKEN ALPHABET $\Sigma$.

| Symbol | Token | Regexp |
|---|---|---|
| WORD | any alphanumeric character | `[a-zA-Z0-9]+` |
| KEY | a WORD token that is also a language keyword (eg. C/C++, Java, Perl, SQL, . . . ) | |
| UNDSC | the underscore character | `\_+` |
| NEWLN | the newline character | `[\n\r]` |
| NUM | a WORD token that is a pure sequence of digits | `\d+` |
| the char itself | any other character not matching the previous patterns | `[^\s\w]` |

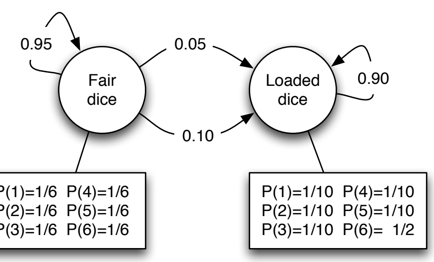


Fig. 1. The unfair-casino HMM.

### *B. A basic Hidden Markov Model*

To model our problem we adopt an HMM defined as follows. We model a text as a sequence of tokens, *i.e.* sequences of characters separated by spaces (`\s`). Each token belongs to a symbol class that constitutes the alphabet of our HMM. Table I shows the alphabet and the regular expression adopted to detect those symbols in text.

Table II shows some examples of texts and their representation as a sequence of tokens. The goal is to detect whether a symbol encountered in a text sequence comes from the natural language text or it is part of a source code fragment. For example, encountering a source code KEY symbol does not guarantees that the portion of the analyzed text is a source code fragment as many keywords could be also part of the English natural language (e.g., `while`, `for`, `if`, `function`, `select`, . . . ). We model this behavior assuming that each symbol could be emitted by two states, one modeling natural text language, and one modeling source code text. The HMM is in fact composed by 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. Clearly, the transition

TABLE II
SOME TOKEN SEQUENCE EXAMPLES.

| Text | Token sequence |
|---|---|
| "My dear Frankenstein," exclaimed he, "how glad I am to see you!" | " WORD WORD WORD , " NEWLN WORD WORD , " WORD WORD WORD WORD NEWLN WORD WORD WORD ! " |
| for(int i=0;i<10;i++) s+=1; | KEY ( WORD WORD = NUM ; WORD < NUM ; WORD + + ) WORD + = NUM ; |

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 the source code text it is more usual to find a punctuation mark symbol or special character, such as opening parenthesis. Formally, the HMM state space is defined as:

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

where $\Sigma_{TXT} = \{\text{WORD}_{TXT}, \text{KEY}_{TXT}, \dots\}$, and $\Sigma_{SRC} = \{\text{WORD}_{SRC}, \text{KEY}_{SRC}, \dots\}$. Each state emits the corresponding alphabet symbol without subscript label $_{TXT}$ or $_{SRC}$. For example, the KEY symbol can be emitted by $\text{KEY}_{TXT}$ or $\text{KEY}_{SRC}$ with a probability equal to 1. If the probability for staying in a natural language text is $p$ and the probability of staying in source code text is $q$, then the transition from a state in $\Sigma_{TXT}$ to a state in $\Sigma_{SRC}$ is $1-p$, instead the inverse transition is $1-q$. The above defined HMM emits the sequence of symbols observed in a text by evolving through a sequence of states $\{\pi_1, \pi_2, \dots, \pi_i, \pi_{i+1}, \dots\}$ with the transition probabilities $t_{kl}$ defined as:

$$t_{kl} = P(\pi_i = l|\pi_{i-1} = k) \cdot p, \text{ if } k, l \in \Sigma_{TXT}$$
$$t_{kl} = P(\pi_i = l|\pi_{i-1} = k) \cdot q, \text{ if } k, l \in \Sigma_{SRC}$$
$$t_{kl} = \frac{1-p}{|\Sigma|}, \text{ if } k \in \Sigma_{TXT}, l \in \Sigma_{SRC}$$
$$t_{kl} = \frac{1-q}{|\Sigma|}, \text{ if } k \in \Sigma_{SRC}, l \in \Sigma_{TXT}$$

and the emission probabilities defined as:

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

Fig. 2 shows the global HMM composed by two sub-HMM, one modeling natural language text and another modeling source code. Fig. 3 and Fig. 4 show their transition probabilities, estimated on the Frankenstein novel and PostgreSQL source code respectively. We detail in Section III-D how these probabilities could be estimated.

It is interesting to observe how typical token sequences are modeled by each HMM. For example, in the source code HMM a number (NUM) is typically preceded by open braces or brackets modeling arguments in a function call or array indexing, the underscore character follows and is followed by a WORD modeling typical variable naming convention. Instead, in the natural language HMM, numbers (NUM) are preceded just by the dollar symbol ($) indicating currency, and likely followed by a dot, indicating text item enumerations. Instead, in the source code HMM it is noticeable that numbers are part of an arithmetic/logic expressions, array indexing, or function argument enumeration.

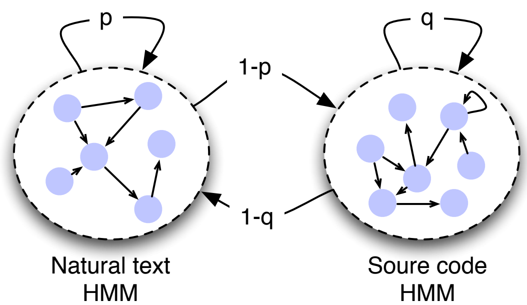


Fig. 2. The source code – natural text island HMM.

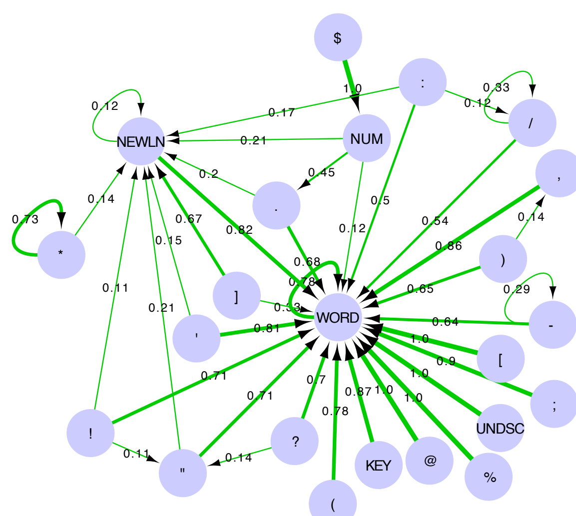


Fig. 3. A natural text HMM trained on the Frankenstein novel (transition probabilities less than 0.1 are not shown).

### C. An extension of the basic model

The basic HMM can be extended to include other language syntaxes usually adopted in development emails, such as patches, log messages, configuration parameters, failure

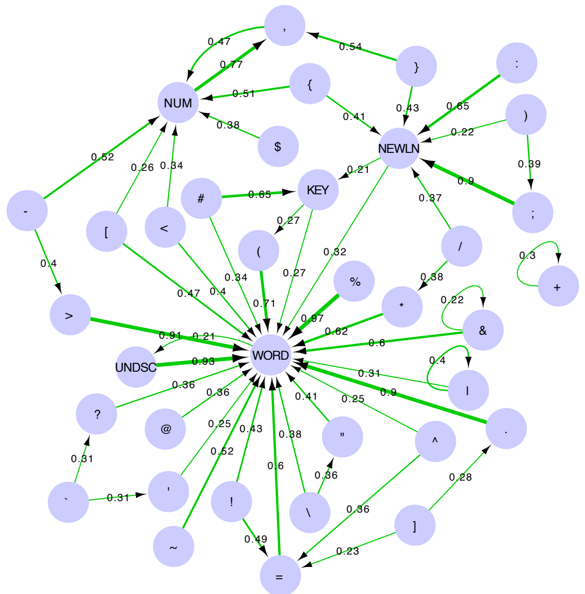


Fig. 4. A source code HMM trained on PosgreSQL source code (transition probabilities less than 0.2 are not shown).

reproducing steps, XML, and so on. In general to include $n$ language syntaxes we introduce the language transition probability matrix $W = w_{ij}$, for $i, j \in \{1, 2, \ldots, n\}$ that 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 to stay into a language syntax $i$ then the probability to switch 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, such as natural text and source code, the language transition probability matrix becomes that of the above described basic HMM (where $w_{11} = p$ and $w_{22} = q$):

$$W = \begin{vmatrix} w_{11} & (1 - w_{11}) \\ 1 - w_{22} & w_{22} \end{vmatrix}$$

### D. Model parameter estimation

The model parameters that need to be estimated are the transition probability matrix $T = \{t_{kl}\}$ and the language transition probability matrix $W = \{w_{ij}\}$. In a specific language context $t_{kl} = P(\pi_i = l | \pi_{i-1} = k)$ can be easily estimated by computing the number of transitions between two subsequent token symbols on training sample texts. For example, Fig. 3 shows the transition probabilities estimated on the Frankenstein novel, downloaded from the "Free ebooks – Project Gutenberg" (http://www.gutenberg.org). Fig. 4 shows the transition probabilities estimated inside a collection of C source code files extracted from the PosgreSQL source code repository (http://www.postgresql.org).

Language transition probabilities strongly depend on the nature of the text and the writing style adopted. This information may not be known in advance. The aim is to estimate how many transitions could happen between two language syntaxes. For example, in development mailing list messages usually there is no more than one source code fragment in a text message. If this is the case, we can assume, *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. It could happens that the transition between two language syntaxes could never occur. This may be the case of stack traces and patches. By observing developers' emails, it is usual to notice that, after a stack trace is provided, the resolution patch is introduced with natural language phrases, such as *"Here is the change that we made to fix the problem"*. This leads to a null transition probability from stack trace to patch code. Other heuristics could be adopted to refine the estimation of transition probabilities that reflects specific properties or styles adopted [23].

## IV. Empirical evaluation procedure

To evaluate the effectiveness of our approach we adopt the Precision and Recall metrics, known respectively also as Positive Predictive Value (PPV) and Sensitivity [24]. In particular, for each language syntax, $i$, we compute:

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

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 $i$. The total classification effectiveness is computed in term of classification accuracy, as:

$$ACC = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} (TP_i + FP_i)}$$

where $n$ is the number of considered language syntaxes.

We set up three experiments: i) in the first (E1) we adopt public available html/latex textbooks with fully annotated source code fragments; ii) in the second (E2) we build a corpus of random text files pasting together natural language text, code fragments, and patches; and iii) in the third (E3) we evaluate the method in a real mailing list context.

### A. **E1:** *Annotated textbooks*

In this experiment we use three textbooks related to computer programming topics containing Natural Language Text (NLT) and Source Code fragments (SRC). We exploit the fact that source code fragments are surrounded with html/latex tags, as shown in Table III, making the evaluation of classification accuracy feasible. The first is a programming book on Java with a lot of source code fragment examples; the second is a tutorial C programming with many coding style examples; the last is the documentation of an R package, Biostrings, containing usage examples on DNA string manipulation.

For each textbook, 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 a software system that are representative of

TABLE III
ANNOTATED TEXTBOOKS ADOPTED IN EXPERIMENT **E1**.

| | Text Book | Source code tagged with |
|---|---|---|
| 1. | Thinking in Java, 3rd Edition (Bruce Eckel) | `<p class=code>...</p> <p class=codeInline>...</p>` |
| 2. | Programming in C: A Tutorial (Brian W. Kernighan) | `<pre>...</pre>` |
| 3. | R Biostrings package manuals | `\example{...} \code{...}` |

TABLE IV
CROSS MAILING LIST VALIDATION - TRAINING CONDITIONS.

| | PostgreSQL | Linux Kernel | Frankenstein novel | Patchwork comments | Patchwork Patches |
|---|---|---|---|---|---|
| E2.1 | × | | | × | × |
| E2.2 | × | | × | | × |
| E2.3 | | × | × | | × |
| E2.4 | | × | | × | × |

the programming language encompassed in the textbook. In particular, we use JEdit (*version 3*) Java source code files for "Thinking in Java"; the Linux kernel (*version 3.9-rc6*) C source code files for "Programming in C tutorial"; and the *kernlab* R package source code files for "Biostrings package manuals".

## B. **E2:** *Random generated text*

In this experiment 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). A text file is generated by pasting together randomly selected pieces of information coming from the following repositories: i) source code C files from Linux kernel version, *3.9-rc6*, available at www.kernel.org; ii) patch proposals and natural language text from four Linux patchwork repositories available at https://patchwork.kernel.org. Natural language text is extracted from the patch textual comments after a manual purification of the selected sample that consists of eliminating automatic mailman directives (e.g., *from*, *reply*, *suggested by*) and inside code and stack trace fragments, if present. An example of random text adopted in this experiment is shown in Fig. 5. 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 patch messages from each of the following Linux patchwork projects:

- *linux-pci*, Linux PCI development list;
- *linux-pm*, Linux power management;
- *linux-nfs*, Linux NFS mailing list;
- *LKML*, Linux Kernel Mailing List.

We perform a cross-mailing list validation by leaving out 3 of the four considered mailing lists with the remaining one adopted 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 because of different programming styles no matter whether the source code language is the same. The goal of this experiment is to evaluate to what extent different training condition affects the classification performance. In particular, we consider the following four training conditions, namely E2.1, E2.2, E2.3, and E2.4.

- *E2.1*: source code transition probabilities estimated on a collection of PostgreSQL (*version 9.2*) C source code files; natural language text and patches transition probabilities estimated on the leaved out 3 of the four considered mailing lists.
- *E2.2*: source code transition probabilities estimated on a collection of PostgreSQL (*version 9.2*) C source code files; natural Language transition probabilities estimated on the Frankenstein novel; and patches transition probabilities estimated on the leaved out 3 of the four considered mailing lists.
- *E2.3*: source code transition probabilities estimated on a collection of Linux kernel (*version 3.9-rc6*) C source code files not adopted for random text generation; natural Language transition probabilities estimated on the Frankenstein novel; and patches transition probabilities estimated on the leaved out 3 of the four considered

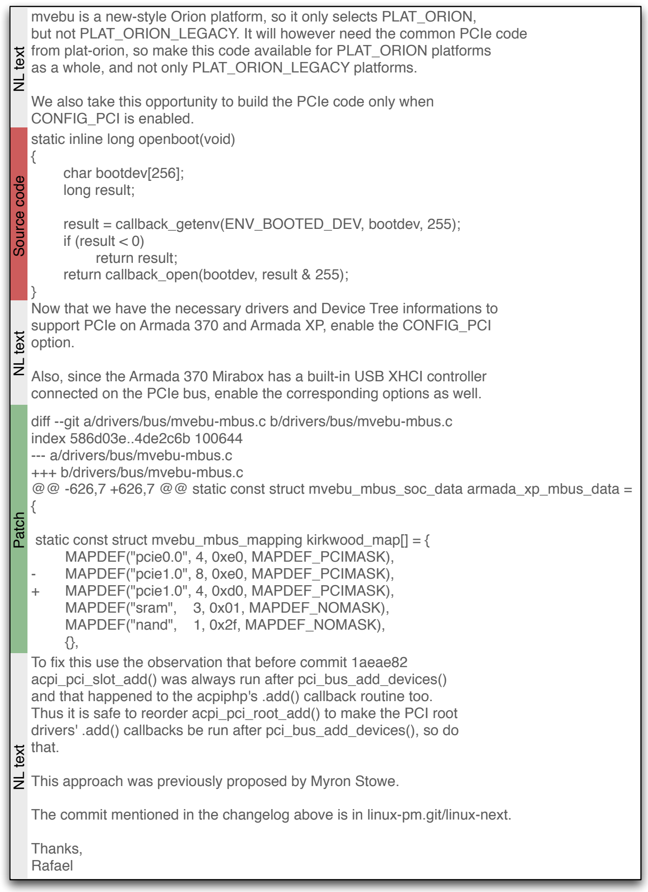


Fig. 5. An example of random generated text file.

files; natural language text and patches transition probabilities estimated on the leaved out 3 of the four considered mailing lists.

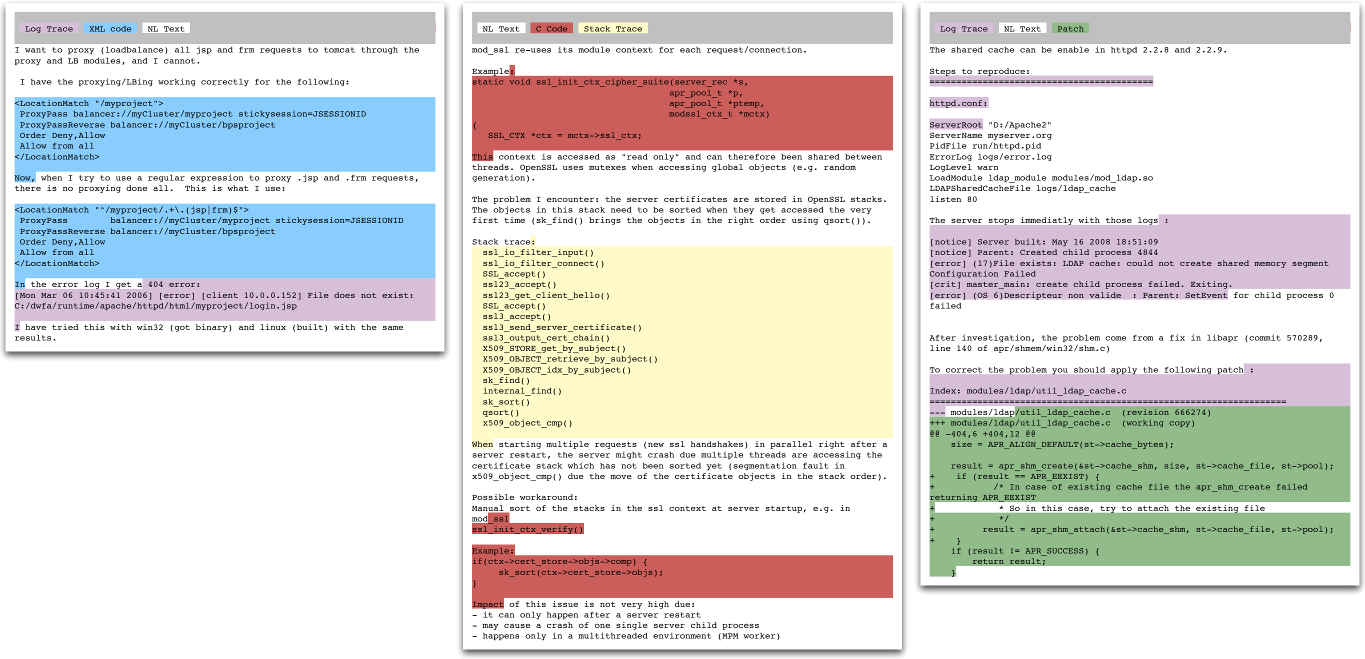


Fig. 6. Annotated message examples adopted for manual inspection.

mailing lists.

- *E2.4*: source code transition probabilities estimated on a collection of Linux kernel (*version 3.9-rc6*) C source code files not adopted for random text generation; natural language text and patches transition probabilities estimated on the leaved out 3 of the four considered mailing lists.

The training condition *E2.4* 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 adopted for testing. Instead, the training conditions *E2.1*, *E2.2*, and *E2.3* adopts source code and natural language text taken from a completely or partially different domain. Table IV summarizes how training sample are combined for each training condition.

### *C.* **E3:** *Mailing list and bug report classification*

In this experiment we evaluate the approach on messages drawn from real mailing lists and bug tracking systems. For this purpose we use the Linux-CEPH filesystem mailing list and the Apache httpd bug tracking system. We consider a number of language syntaxes that is representative for a typical message exchanged in those systems. For Linux-CEPH filesystem mailing lists we consider three language syntax categories: Natural Language text (NLT), Source Code (SRC), and Patches (PCH). Apache httpd bug reports have a more complex structure that ensemble almost six language syntax categories: Natural Language text (NLT), Source Code (SRC), Patches (PCH), Stack Traces (STR), XML code (XML), and log messages (LOG). The source code category comprises: C language, shell scripts, and Javascript. Stack traces are C function call stacks with memory references and memory dumps. XML code usually refers to Apache httpd configuration files issues, and httpd server error response. Finally, LOG traces are typical error/log messages printed out by the Apache httpd server.

We try to reproduce a typical real usage condition by adopting the following evaluation protocol: i) we extract the last 200 messages from a mailing list/bug tracking; ii) one of the authors trains the model by using a half set of the extracted messages (i.e., 100 random messages leaved out from the corpus); iii) another author annotates the remaining messages with the trained model and counts the number of false positives, i.e., the sum of all wrong classified tokens, by manual inspection. This allows us to compute the total accuracy of classification. To this aim, we developed a tool that shows in a browser the annotated messages by using different background colors. Fig. 6 shows some annotated message examples adopted in the manual inspection.

## V. Results

Table V reports results about experiment E1. The overall accuracy ranges from 0.86 to 0.97, attesting a promising performance on textbook encompassing different programming languages. The number of tokens in favor to natural language is higher in the first two textbooks and lower in the last one, making the dataset unbalanced in all cases. Precision and recall measures, and their harmonic mean (F-measure) are more appropriate for unbalanced datasets. The prediction accuracy of natural language outperforms source code in the first two textbook and is almost similar to source code for the last one. This leads us to conclude that the natural language model trained on the Frankenstein novel is almost accurate for modeling the natural language content of such textbooks. For source code, we can observe that the best performance

TABLE V
RESULTS ON ANNOTATED TEXTBOOKS (EXPERIMENT E1).

| | Text Book | No. of tokens NLT | No. of tokens SRC | Correctly classified NLT | Correctly classified SRC | Accuracy | NLT Pr | NLT Rc | NLT Fm | SRC Pr | SRC Rc | SRC Fm |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | Thinking in Java, 3rd Edition (Bruce Eckel) | 65055 | 43044 | 62244 | 31440 | 0.867 | 0.843 | 0.957 | 0.896 | 0.918 | 0.730 | 0.813 |
| 2. | Programming in C: A Tutorial (Brian W. Kernighan) | 9754 | 3429 | 9046 | 2667 | 0.888 | 0.922 | 0.927 | 0.924 | 0.790 | 0.778 | 0.784 |
| 3. | R Biostrings package manuals | 2195 | 11577 | 1932 | 11439 | 0.971 | 0.933 | 0.880 | 0.905 | 0.978 | 0.988 | 0.982 |

has been obtained for the R language (Precision 0.978 and Recall 0.988), while the worst for the C language (Precision 0.790 and Recall 0.778). An average performance has been obtained for the Java language (Precision 0.918 and Recall 0.730). We believe that this is mainly due to the programming style adopted, as we show with the next experiment (E2). The source code HMM was trained with real software systems (PosgreSQL for the C language, Jedit for the Java language, and Biostrings for the R language). It is likely that the coding style adopted, usually in textbooks, to teach basic programming techniques differs from the coding practices adopted by senior developers.

Tables VI, VII, VIII, and IX show results of cross mailing list validation (E2) obtained with the training conditions E2.1, E2.2, E2.3, and E2.4 respectively. The first training condition (E2.1) uses, for source code, training samples coming from a different test set domain. The second training condition (E2.2) uses, for source code, training samples coming from a different test set domain (PostgreSQL development community). The third training condition (E2.3) uses, for natural language, training samples coming from a different domain (Frankenstein novel). The fourth training condition (E2.4) 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 is obtained under the training condition E2.4 (F-measure ranging between 0.87 and 0.99 in all mailing lists and for each language syntax), 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 HMM is trained on examples of a different domain (E2.2 and E2.3) the corresponding NLT prediction performance decreases (F–measure less than 0.8 in almost all cases). Instead, the NLT prediction performance persists on almost the same level in E2.1 and E2.4 (F-measure ranging from 0.86 to 0.96). A different behavior can be observed for source code HMM. When source code HMM is trained with examples of a different domain (E2.1 and E2.2) the decrement in prediction performance affects both source code and patches (F-measure drops to around 0.7 for SRC and 0.8 for PCH in almost all cases). This is because the source code snippets written by developers in the mailing list messages are usually confused with patches. Biasing in fact the detection of source code snippets by the source code HMM trained on PostgreSQL source code samples, as shows by the confusion matrices reported in Tables VII and VIII.

Table X reports results of experiment E3. The overall performance is terms of classification accuracy is 0.824 for

TABLE VI
CROSS MAILING LIST VALIDATION - TRAINING CONDITION E2.1.

| | Classified as NLT | SRC | PCH | Precision | Recall | F-measure |
|---|---|---|---|---|---|---|
| | *linux-LKML* (Accuracy: 0.854) | | | | | |
| NLT | 3356 | 117 | 162 | 0.953 | 0.923 | 0.938 |
| SRC | 34 | 2066 | 1251 | 0.883 | 0.617 | 0.726 |
| PCH | 132 | 156 | 5439 | 0.794 | 0.950 | 0.865 |
| | *linux-nfs* (Accuracy: 0.859) | | | | | |
| NLT | 2210 | 11 | 23 | 0.940 | 0.985 | 0.962 |
| SRC | 27 | 2453 | 1254 | 0.982 | 0.657 | 0.787 |
| PCH | 113 | 33 | 4259 | 0.769 | 0.967 | 0.857 |
| | linux-pci (Accuracy: 0.841) | | | | | |
| NLT | 4516 | 114 | 128 | 0.962 | 0.949 | 0.955 |
| SRC | 74 | 2649 | 1716 | 0.955 | 0.597 | 0.735 |
| PCH | 106 | 10 | 4206 | 0.695 | 0.973 | 0.811 |
| | linux-pm (Accuracy: 0.756) | | | | | |
| NLT | 1823 | 17 | 26 | 0.778 | 0.977 | 0.866 |
| SRC | 44 | 1353 | 1518 | 0.975 | 0.464 | 0.629 |
| PCH | 477 | 18 | 3338 | 0.684 | 0.871 | 0.766 |

TABLE VII
CROSS MAILING LIST VALIDATION - TRAINING CONDITION E2.2.

| | Classified as NLT | SRC | PCH | Precision | Recall | F-measure |
|---|---|---|---|---|---|---|
| | *linux-LKML* (Accuracy: 0.789) | | | | | |
| NLT | 2499 | 286 | 850 | 0.940 | 0.687 | 0.794 |
| SRC | 36 | 2092 | 1223 | 0.825 | 0.624 | 0.711 |
| PCH | 123 | 159 | 5445 | 0.724 | 0.951 | 0.822 |
| | *linux-nfs* (Accuracy: 0.825) | | | | | |
| NLT | 1924 | 35 | 285 | 0.964 | 0.857 | 0.907 |
| SRC | 21 | 2322 | 1391 | 0.971 | 0.622 | 0.758 |
| PCH | 50 | 34 | 4321 | 0.721 | 0.981 | 0.831 |
| | linux-pci (Accuracy: 0.750) | | | | | |
| NLT | 3224 | 327 | 1207 | 0.976 | 0.678 | 0.800 |
| SRC | 46 | 2636 | 1757 | 0.887 | 0.594 | 0.712 |
| PCH | 33 | 10 | 4279 | 0.591 | 0.990 | 0.740 |
| | linux-pm (Accuracy: 0.746) | | | | | |
| NLT | 1369 | 75 | 422 | 0.820 | 0.734 | 0.775 |
| SRC | 22 | 1499 | 1394 | 0.952 | 0.514 | 0.668 |
| PCH | 278 | 0 | 3555 | 0.662 | 0.927 | 0.772 |

TABLE VIII
CROSS MAILING LIST VALIDATION - TRAINING CONDITION E2.3.

| | Classified as NLT | SRC | PCH | Precision | Recall | F-measure |
|---|---|---|---|---|---|---|
| | *linux-LKML* (Accuracy: 0.892) | | | | | |
| NLT | 2493 | 339 | 803 | 0.946 | 0.686 | 0.795 |
| SRC | 19 | 3332 | 0 | 0.887 | 0.994 | 0.937 |
| PCH | 123 | 85 | 5519 | 0.873 | 0.964 | 0.916 |
| | *linux-nfs* (Accuracy: 0.966) | | | | | |
| NLT | 1946 | 53 | 245 | 0.975 | 0.867 | 0.918 |
| SRC | 7 | 3727 | 0 | 0.986 | 0.998 | 0.992 |
| PCH | 43 | 0 | 4362 | 0.947 | 0.990 | 0.968 |
| | linux-pci (Accuracy: 0.887) | | | | | |
| NLT | 3284 | 234 | 1240 | 0.986 | 0.690 | 0.812 |
| SRC | 13 | 4426 | 0 | 0.948 | 0.997 | 0.972 |
| PCH | 33 | 10 | 4279 | 0.775 | 0.990 | 0.869 |
| | linux-pm (Accuracy: 0.908) | | | | | |
| NLT | 1374 | 82 | 410 | 0.822 | 0.736 | 0.777 |
| SRC | 19 | 2896 | 0 | 0.972 | 0.993 | 0.982 |
| PCH | 278 | 0 | 3555 | 0.897 | 0.927 | 0.912 |

TABLE IX
CROSS MAILING LIST VALIDATION - TRAINING CONDITION E2.4.

| | Classified as NLT | SRC | PCH | Precision | Recall | F-measure |
|---|---|---|---|---|---|---|
| | *linux-LKML* (Accuracy: 0.950) | | | | | |
| NLT | 3280 | 183 | 172 | 0.955 | 0.902 | 0.928 |
| SRC | 21 | 3330 | 0 | 0.916 | 0.994 | 0.953 |
| PCH | 135 | 124 | 5468 | 0.970 | 0.955 | 0.962 |
| | *linux-nfs* (Accuracy: 0.987) | | | | | |
| NLT | 2229 | 7 | 8 | 0.947 | 0.993 | 0.969 |
| SRC | 9 | 3725 | 0 | 0.998 | 0.998 | 0.998 |
| PCH | 116 | 0 | 4289 | 0.998 | 0.974 | 0.986 |
| | linux-pci (Accuracy: 0.972) | | | | | |
| NLT | 4534 | 119 | 105 | 0.975 | 0.953 | 0.964 |
| SRC | 6 | 4433 | 0 | 0.964 | 0.999 | 0.981 |
| PCH | 108 | 47 | 4167 | 0.975 | 0.964 | 0.969 |
| | linux-pm (Accuracy: 0.935) | | | | | |
| NLT | 1831 | 10 | 25 | 0.785 | 0.981 | 0.872 |
| SRC | 19 | 2896 | 0 | 0.988 | 0.993 | 0.990 |
| PCH | 482 | 25 | 3326 | 0.993 | 0.868 | 0.926 |

Apache httpd bug reports and 0.974 for linux–CEPH mailing list. In Apache httpd bug reports, we detected up to six language syntaxes, while in the linux-CEPH mailing list we detected three. The complexity of bug reports makes the detection of languages more problematic. The manual inspection revealed that sometimes log messages are confounded with stack traces, while source code is almost detected correctly and also patch proposals. XML tags are correctly detected, however sometimes the approach fails because the free text contained between XML tags is recognized as natural language text.

## VI. THREATS TO VALIDITY

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

### A. Construct Validity

Threats to *construct* validity may be related to imprecisions in our measurements. In particular, metrics adopted to evaluate

TABLE X
RESULTS OF MANUAL VALIDATION (EXPERIMENT E3).

| Mailing list | No. of tokens | False positives | Accuracy |
|---|---|---|---|
| Linux–CEPH filesystem discussion | 66298 | 1687 | 0.974 |
| Apache httpd bug reports | 32991 | 5809 | 0.824 |

the approach performance may not measure the concepts we want to assess. The third experiment (E3) is particularly affected by this threat, as the difference in opinions on what qualifies as information encoded in the documents may affect the estimation of classification accuracy. For example, source code comments may or may not be considered as natural language text, or the presence of references to method calls in a sentence may or may not qualify as a code snippet. In computing the overall accuracy of experiment E3, we considered multi-line source code comments as natural language text, and we did not consider function/method calls mentioned in natural language sentences as real source code fragments.

### B. 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. In experiment E3, over 100,000 tokens were evaluated manually, while in experiment E2 we classified over 400 mailing list messages, and in experiment E1 we classified over 130,000 tokens from three textbooks.

### C. Reliability validity

Threats to reliability validity concern the capability of replicating this study and obtaining the same results. Scripts and datasets adopted to run the experiments are available on http://www.rcost.unisannio.it/cerulo/dataset-scam2013.tgz.

### D. 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. In our experiments. we chose contexts with unrelated characteristics, software systems developed by separate communities, and in E3 we considered up to six language syntaxes. Moreover, in experiment E2 we performed a cross mailing list validation which is more powerful than a regular internal k-fold cross-validation. We are aware that a further empirical validation on a larger set of free text repositories would be beneficial to better support our findings.

## VII. CONCLUSIONS AND FUTURE WORK

We introduced a method based on Hidden Markov Models to identify code information typically included in development mailing list and bug report free text. We performed a cross mailing list evaluation on text assembled with random pieces of natural language text, source code, and patches obtaining promising results when the training samples are from the same mailing list information domain (accuracy ranging from 0.87 to 0.99). Furthermore, we performed a manual evaluation on a sample of real mailing list messages and bug reports obtaining, also in such a case, an overall classification accuracy ranging from 0.82 to 0.97, which is similar to the performance obtained by Bacchelli *et al.* in a comparable context [3].

The approach has, in fact, a performance level that in similar conditions is nearly equivalent to the most of literature methods generally based of regular expression and/or island parser. In addition, the method does not need the expertise required for the construction of the parser/regular expressions, as it learns directly from data. This last property allows the approach to be robust, especially with noisy data where regular expressions may fail, because of unexpected pattern variations from the original scheme on which the regular expression was designed.

The method opens for new opportunities in the context of software engineering free text mining. To this aim, we plan to investigate towards the following research directions:

- *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 1, i.e., the current state may depends only on a history of previous states of length 1. 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 model more precisely language syntaxes and capture, for example, specific programming styles, and the "naturalness" of software which is likely to be repetitive and predictable [23].
- *Evaluation dataset*. Last, but not least, we plan to evaluate the proposed approach on further datasets.

## References


[1] D. Binkley and D. Lawrie, "Development: Information retrieval applications," in *Encyclopedia of Software Engineering*, 2010, pp. 231–242.

[2] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *Information Theory, IEEE Transactions on*, vol. 13, no. 2, pp. 260–269, 1967.

[3] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 375–384.

[4] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of the 2008 international working conference on Mining software repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 27–30.

[5] G. C. Murphy and D. Notkin, "Lightweight lexical source model extraction," *ACM Transactions on Software Engineering and Methodology*, vol. 5, pp. 262–292, 1996.

[6] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, Oct. 2002.

[7] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*. IEEE Computer Society, 2003, pp. 125–137.

[8] B. Ribeiro-neto and Baeza-yates, *Modern Information Retrieval*. Addison Wesley, 1999.

[9] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *Proceedings of the 11th IEEE International Software Metrics Symposium*, ser. METRICS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 29–.

[10] ——, "Supporting change request assignment in open source development," in *Proceedings of the 2006 ACM symposium on Applied computing*, ser. SAC '06. New York, NY, USA: ACM, 2006, pp. 1767–1772.

[11] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370.

[12] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 223–226.

[13] K. Spärck Jones, "Automatic summarising: The state of the art," *Inf. Process. Manage.*, vol. 43, no. 6, pp. 1449–1481, Nov. 2007.

[14] J. Tang, H. Li, Y. Cao, and Z. Tang, "Email data cleaning," in *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. New York, NY, USA: ACM, 2005, pp. 489–498.

[15] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza, "Content classification of development emails," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 375–385.

[16] M. Skounakis, M. Craven, and S. Ray, "Hierarchical hidden markov models for information extraction," in *In Proceedings of the 18th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 2003, pp. 427–433.

[17] L. E. Baum, T. Petrie, G. Soules, and N. Weiss, "A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains," *The Annals of Mathematical Statistics*, vol. 41, no. 1, pp. 164–171, 1970.

[18] X. Huang, Y. Ariki, and M. Jack, *Hidden Markov Models for Speech Recognition*. New York, NY, USA: Columbia University Press, 1990.

[19] T. Starner and A. Pentl, "Visual recognition of american sign language using hidden markov models," in *In International Workshop on Automatic Face and Gesture Recognition*, 1995, pp. 189–194.

[20] S. M. Thede and M. P. Harper, "A second-order hidden markov model for part-of-speech tagging," in *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, ser. ACL '99. Stroudsburg, PA, USA: Association for Computational Linguistics, 1999, pp. 175–182.

[21] B. Pardo and W. Birmingham, "Modeling form for on-line following of musical performances," in *Proceedings of the 20th national conference on Artificial intelligence - Volume 2*, ser. AAAI'05. AAAI Press, 2005, pp. 1018–1023.

[22] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Jul. 1998.

[23] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847.

[24] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.