POSIT: Simultaneously Tagging Natural and Programming Languages

Profir-Petru Pârţachi
profir-petru.partachi.16@ucl.ac.uk
University College London
London, United Kingdom

Santanu Kumar Dash
s.dash@surrey.ac.uk
University of Surrey
Guildford, Surrey, United Kingdom

Christoph Treude
christoph.treude@adelaide.edu.au
University of Adelaide
Adelaide, South Australia, Australia

Earl T. Barr
e.barr@ucl.ac.uk
University College London
London, United Kingdom

ABSTRACT

Software developers use a mix of source code and natural language text to communicate with each other: Stack Overflow and Developer mailing lists abound with this mixed text. Tagging this mixed text is essential for making progress on two seminal software engineering problems — traceability, and reuse via precise extraction of code snippets from mixed text. In this paper, we borrow code-switching techniques from Natural Language Processing and adapt them to apply to mixed text to solve two problems: language identification and token tagging. Our technique, POSIT, simultaneously provides abstract syntax tree tags for source code tokens, part-of-speech tags for natural language words, and predicts the source language of a token in mixed text. To realize POSIT, we trained a biLSTM network with a Conditional Random Field output layer using abstract syntax tree tags from the CLANG compiler and part-of-speech tags from the Standard Stanford part-of-speech tagger. POSIT improves the state-of-the-art on language identification by 10.6% and PoS/AST tagging by 23.7% in accuracy.

KEYWORDS

part-of-speech Tagging, Mixed-Code, Code-Switching, Language Identification

ACM Reference Format:
Profir-Petru Pârţachi, Santanu Kumar Dash, Christoph Treude, and Earl T. Barr. 2020. POSIT: Simultaneously Tagging Natural and Programming Languages . In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Programmers often mix natural language and code when talking about the source code. Such mixed text is commonly found in mailing lists, documentation, bug discussions, and online fora such as Stack Overflow. Searching and mining mixed text is inevitable when tackling seminal software engineering problems, like traceability and code reuse. Most development tools are monolingual or work at a level of abstraction that does not exploit language-specific information. Few tools directly handle mixed text because the differences between natural languages and formal languages call for different techniques and tools. Disentangling the languages in mixed text, while simultaneously accounting for cross language interactions, is key to exploiting mixed text: it will lay the foundation for new tools that directly handle mixed text and enable the use of existing monolingual tools on pure text snippets extracted from mixed text. Mixed-text-aware tooling will help bind specifications to their implementation or help link bug reports to code.

The *mixed text tagging* problem is the task of tagging each token in a text that mixes at least one natural language with several formal languages. It has two subproblems: identifying a token's origin language (language tagging) and identifying the token's part of speech (PoS) or its abstract syntax tree (AST) tag (PoS/AST tagging). A token may have multiple PoS/AST tags. In the sentence "I foo-ed the String 'Bar'.", 'foo' is a verb in English and a method name (of an object of type String). Therefore, PoS/AST tagging involves building a map that pairs a language to the token's PoS/AST node in that language, for each language operative over that token.

We present POSIT to solve the 1+1 mixed text tagging problem: POSIT distinguishes a Natural language (English) from programming language snippets and tags each text or code snippet under its language's grammar. To this end, POSIT jointly solves both the language segmentation and tagging subproblems. POSIT employs techniques from Natural Language Processing (NLP) for code-switched[1] text. Code-switching occurs when multilingual individuals simultaneously use two (or more) languages. This happens when they want to use the semantics of the embedded language in the host language. Within the NLP space, such mixed text data tends to be bi- and rarely tri-lingual. Unique to our setting is, as our data taught us, the mixing of more than three languages, one natural and often many formal ones — in our corpus, many posts combine a programming language, file paths, diffs, JSON, and URLs.

To validate POSIT, we compare it to Ponzanelli *et al.*'s pioneering work StORMeD [22], the first context-free approach for mixed

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] The fact that the NLP literature uses the word "code" in their name for the problem of handling text that mixes multiple natural languages is unfortunate in our context. They mean code in the sense of coding theory.

text. They use an island grammar to parse Java, JSON and XML snippets embedded within English. As English is relegated to water, StORMeD neglects natural language, builds ASTs for its islands, then augments them with natural language snippets to create heterogenous ASTs. POSIT tags both natural languages and formal languages, but does not build trees. Both techniques identify language shifts and both tools label code snippets with their AST labels. POSIT is designed from the ground up to handle *untagged* mixed text after training. StORMeD looks for tags and resorts to heuristics in their absence. On the language identification task, StORMeD achieves an accuracy of 71%; on the same dataset, POSIT achieves 81.6%. To compare StORMeD and POSIT on the PoS/AST tagging task, we extracted AST tags from the StORMeD output. Despite not being designed for this task, StORMeD establishes the existing state of the art and achieves 61.9% against POSIT's 85.6%. POSIT outperforms StORMeD here, in part, because it finds many more small code snippets in mixed text. In short, POSIT advances the state-of-the-art on mixed text tagging.

POSIT is not restricted to Java. On the entire Stack Overflow corpus (Java and non-Java posts), POSIT achieves an accuracy of 98.7% for language identification and 96.4% for PoS or AST tagging. A manual examination of POSIT's output on Stack Overflow posts containing 3,233 tokens showed performance consistent with POSIT's results on the evaluation set: 95.1% accuracy on language tagging and 93.7% on PoS/AST tagging. To assess whether POSIT generalises beyond its two training corpora, we manually validated it on e-mails from the Linux Kernel mailing list. Here, POSIT achieved 76.6% accuracy on language tagging and 76.5% on PoS/AST tagging.

POSIT is directly applicable to downstream applications. First, its language identification achieves 95% balanced accuracy when predicting missed code labels and could be the basis of a tool that automatically validates posts before submission. Second, TaskNav [29] is a tool that extracts mixed text for development tasks. POSIT's language identification and PoS/AST tagging enables TaskNav to extract more than two new, reasonable tasks per document: on a corpus of 30 LKML e-mails, it extracts 97 new tasks, 65 of which are reasonable.

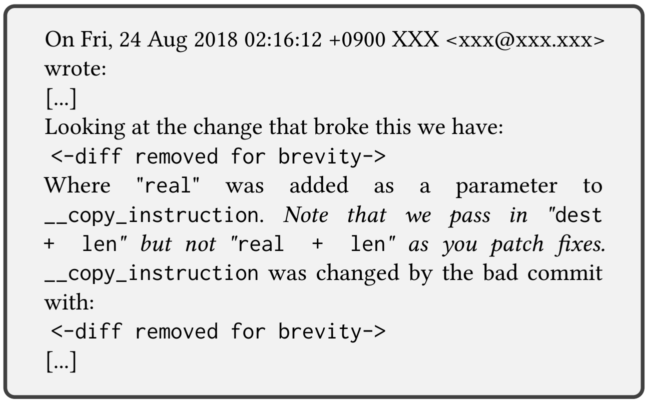
Our main contributions follow:

- We have built the first corpus for mixed text that is tagged at token granularity for English and C/C++.
- We present POSIT, an NLP-based code-switching approach for the mixed text tagging problem;
- POSIT can directly improve downstream applications: it can improve the code tagging of Stack Overflow posts and it improves TaskNav, a task extractor.

We make our implementation and the code-comment corpus used for evaluation available at https://github.com/PPPI/POSIT.

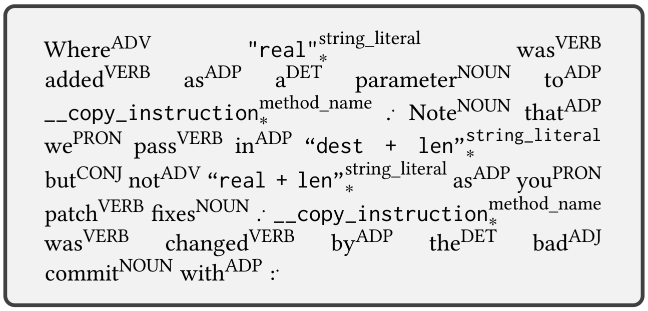
2 MOTIVATING EXAMPLE

The mix of source code and natural language in the various documents produced and consumed by software developers presents many challenges to tools that aim to help developers make sense of these documents automatically. An example is TaskNav [29], a tool that supports task-based navigation of software documentation by automatically extracting task phrases from a documentation corpus



On Fri, 24 Aug 2018 02:16:12 +0900 XXX <xxx@xxx.xxx> wrote:
[...]
Looking at the change that broke this we have:
`<-diff removed for brevity->`
Where `"real"` was added as a parameter to `__copy_instruction`. *Note that we pass in* `"dest + len"` *but not* `"real + len"` *as you patch fixes.* `__copy_instruction` was changed by the bad commit with:
`<-diff removed for brevity->`
[...]

Figure 1: Example e-mail snippet from the Linux Kernel mailing list. It discusses a patch that fixes a kernel freeze. Here the fix is performed by updating the RIP address by adding `len` to the `real` value during the copying loop. Code tokens are labelled by the authors using the patches as context and rendered using `monospace`.



Where$^{\mathrm{ADV}}$ `"real"`$^{\mathrm{string\_literal}}_{*}$ was$^{\mathrm{VERB}}$ added$^{\mathrm{VERB}}$ as$^{\mathrm{ADP}}$ a$^{\mathrm{DET}}$ parameter$^{\mathrm{NOUN}}$ to$^{\mathrm{ADP}}$ `__copy_instruction`$^{\mathrm{method\_name}}_{*}$.$^{\cdot}$ Note$^{\mathrm{NOUN}}$ that$^{\mathrm{ADP}}$ we$^{\mathrm{PRON}}$ pass$^{\mathrm{VERB}}$ in$^{\mathrm{ADP}}$ `“dest + len”`$^{\mathrm{string\_literal}}_{*}$ but$^{\mathrm{CONJ}}$ not$^{\mathrm{ADV}}$ `“real + len”`$^{\mathrm{string\_literal}}_{*}$ as$^{\mathrm{ADP}}$ you$^{\mathrm{PRON}}$ patch$^{\mathrm{VERB}}$ fixes$^{\mathrm{NOUN}}$.$^{\cdot}$ `__copy_instruction`$^{\mathrm{method\_name}}_{*}$ was$^{\mathrm{VERB}}$ changed$^{\mathrm{VERB}}$ by$^{\mathrm{ADP}}$ the$^{\mathrm{DET}}$ bad$^{\mathrm{ADJ}}$ commit$^{\mathrm{NOUN}}$ with$^{\mathrm{ADP}}$:$^{\cdot}$

Figure 2: POSIT's output from which TaskNav++ extracts the tasks (pass in “dest + len”) and (pass in “real + len”). We show the PoS/AST tags as superscript and mark tokens with $_*$ if they are identified as code. POSIT spots the two mention-roles of code tokens as ‘string_literal’s.

and by surfacing these task phrases in an interactive auto-complete interface. For the extraction of task phrases, TaskNav relies on grammatical dependencies between tokens in software documentation that, in turn, relies on correct parsing of the underlying text. To handle the unique characteristics of software documentation caused by the mix of code and natural language, the TaskNav developers hand-crafted a number of regular expressions to detect code tokens as well as a number of heuristics for sentence completion, such as adding “This method” at the beginning of sentences with missing subject. These heuristics are specific to a programming language (Python in TaskNav's case) and a particular kind of document, such as API documentation dominated by method descriptions.

POSIT has the potential to augment tools such as TaskNav to reliably extract task phrases from any document that mixes code and natural language. As an example, in Figure 1, we can see an e-mail excerpt from the LKML[2]. TaskNav only manages to extract trivial task phrases from this excerpt (e.g., "patch fixes") and misses task phrases related to the code tokens of `dest`, `real`, and `len` due to incorrect parsing of the sentence beginning with "Note that ...". After augmenting TaskNav with POSIT, the new version, which we call TaskNav++, manages to extract two additional task phrases: (pass in "dest + len") and (pass in "real + len"); we present POSIT's output on this sentence in Figure 2. These additional task phrases extracted with the help of POSIT will help developers find resources relevant to the tasks they are working on, e.g., when they are searching for resources explaining which parameters to use in which scenario. We discuss the performance of TaskNav++ in more detail in Section 6.2.

3 MIXED TEXT TAGGING

Tags are the non-terminals that produce terminals in a language's grammar. Given mixed text with k natural languages and l formal languages, let a token's tag map bind the token to a tag for each of the $k + l$ languages. We consider a formal language to be one which, to a first approximation, has a context-free grammar. The *mixed text tagging problem* is then the problem of building a token's tag map. For example, in the sentence, "'lieben' means love in German", 'lieben' is a subject in the frame language English and a verb in German. Moving to a coding example, in a sentence such as "I foo-ed the String 'Bar'.", we observe 'foo' to be a verb in English and a method name (of an object of type String).

A general solution produces a list of pairs: part-of-speech tags for each of the k natural languages together with the natural language for which we have the tag, and AST tags for each of the l formal languages together with the language within which we have the AST tag. We also consider two special tags Ω and ϵ that are fresh relative to the set of all tags within all natural and formal languages. We use ϵ to indicate that a particular language has no candidate tag, while Ω is paired with the origin language, answering the first task of our problem. In the first example above, 'lieben''s tag map is [(Ω, De), (Verb, De), (Noun, En), (ϵ, C)], if we consider English, German and C. In the code example, 'foo's' tag map is [(Ω, C), (ϵ, De), (Verb, En), (method_name, C)]. In multilingual scenarios, a token might have a tag candidate for every language.

The mixed text tagging problem is context-sensitive. We argue below that determining the token's origin language is context-sensitive for a single token code-switch. The proof rests on a series of definitions from linguistics which we state next. To bootstrap, a *morpheme* is an atomic unit of meaning in a language. Morphemes differ from words in that they may or may not be free, or stand alone. We source these definitions from Poplack [24].

"*Code-switching* is the alternation of two languages in a single discourse, sentence or constituent. . . . [deletia] . . . [It] was characterised according the degree of integration of items from one language (L_1) to the phonological, morphological, and syntactic patterns of the other (L_2)" [24, §2 ¶2]. We use L_1 to refer to the frame language and L_2 to the embedded one. Further, context-switching has two restrictions on when it may occur. It can only occur after free morphemes. The second restriction is that code-switching occurs at points where juxtapositions between L_1 and L_2 do not violate the syntactic rules of either language. Code-switching allows integrating items from L_2 into L_1 along any one of phonological, morphological, or syntactic axis, but not all three simultaneously. This last case is considered to be mono-lingual L_1.

Adaptation occurs when an item from L_2 changes when used in L_1 to obey L_1's rules. Adaptation has three forms: morphological, phonological, and syntactical. *Morphological adaptation* represents modifying the spelling of L_2 items to fit L_1 patterns. *Phonological adaptation* represents changing the pronunciation of an L_2 item in an L_1 context. *Syntactic adaptation* represents modifying L_2 items embedded in a discourse, sentence, or constituent in L_1 to obey L_1's syntax. Finally, L_2 items can be used in L_1 *without adaptation*. In this case, these items often reference the code-entity by name and are used as a 'noun' in L_1.

We now consider three cases: (I) L_2 items are morphologically adapted to L_1, (II) L_2 items are syntactically adapted to L_1, and (III) no adaptation of L_2 items occurs before their use in L_1. We do not consider phonological adaptation of L_2 items into L_1 as that is not observable in text.

Case I: Morphological Adaptation. Consider using affixation to convert foo/ class to foo−ify/verb to denote the action of converting to the class foo. In this case, foo−ify behaves as a bona fide word in L_1. Such examples obey the free-morpheme restriction mentioned above. This enables it to be a separate, stand-alone morpheme/item within L_1. The juxtaposition restriction, further ensures that this parses within L_1. Lacking a context to indicate foo's origin, a parsers would need to assume that it is from L_1.

Case II: Syntactic Adaptation. This case manifests similarly to morphological adaptation, such as tense agreement, or, potentially, as word order restrictions. If spelling changes do occur, this case reprises the morphological adaptation case. If the only adaptation is word order, then the task becomes spotting a L_2 token that has stayed unchanged in a L_1 sentence or constituent. This is impossible in general if the two language's vocabularies overlap.

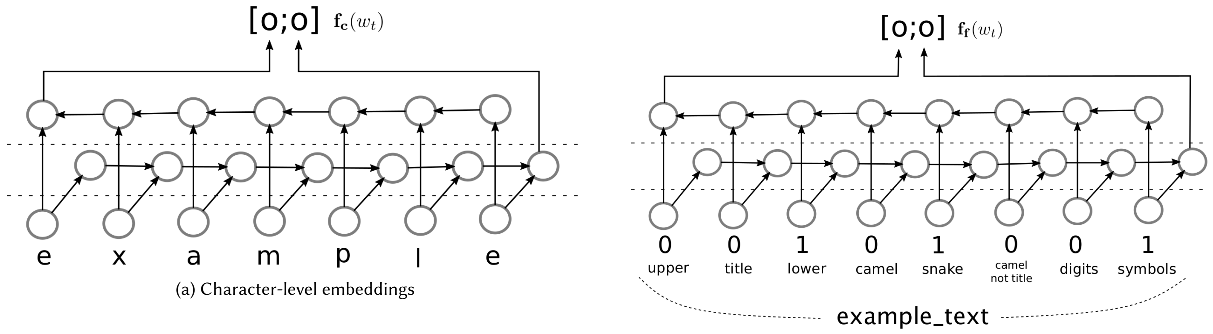
Case III: No Adaptation. If no adaptation occurs, then the formal token occurs in L_1. This reduces to the second subcase of the syntactic adaptation case.

4 POSIT

POSIT starts from the biLSTM-CRF model presented in Huang *et al.* [14], augments it to have a character-level encoding as seen in Winata *et al.* [32] and adds two learning targets as in Soto and Hirschberg [26]. Figure 4 presents the resulting network. The network architecture employed by POSIT is capable of learning to provide a language tag for any $k + l$ languages considered. This model is capable of considering the context in the input using the LSTMs, it can bias its subsequence choices as it predicts tags based on the predictions made thus far, and the character-level encoding allows it to learn token morphology features beyond those that we may expose to it directly as a feature vector.

Feature Space. We rely on source code attributes to separate code from natural language while tagging both simultaneously. We derive vector embeddings for individual characters to model subtle variations in how natural language is used within source code

[2] https://lkml.org/lkml/2018/8/24/19



(a) Character-level embeddings

(b) Feature vector embeddings

Figure 3: Computation of embeddings at the character level and from coding naming and spelling convention features. In the bottom most layer, the circles represent an embedding operation on characters or features to a high-dimensional space. The middle layer represents the forward LSTM and the top most layer — the backward LSTM. At the word level, character and feature vector embeddings are represented by the concatenation of the final states of the forward and backward LSTMs represented by $[\cdot;\cdot]$ in the diagrams above.

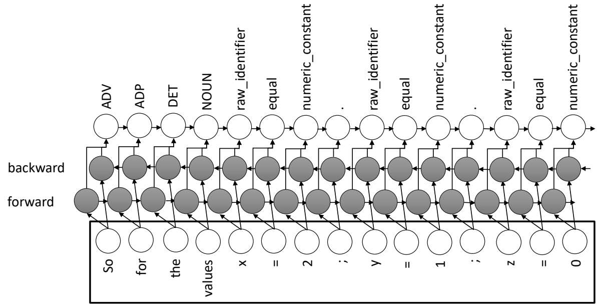


Figure 4: A representation of the neural network used for predicting English PoS tags together with compiler derived AST tags. The shaded cells represent LSTM cells, arrows represent the flow of information in the network. The top layer represents a linear Conditional Random Field (CRF) and the transition probabilities are used together with a Viterbi decode to obtain the final output. The first layer is represented by Equation (1) and converts the tokenised sentences into vector representations.

snippets. Examples of such variations are numbered variables such as i1 or i2 that often index axes during multi-dimensional array operations. Another such variation arises in the naming of loop control variables where the iterator could be referred to in diverse, but related ways, as i, it or iter. These variations create out-of-vocabulary (OOV) words which inhibit modelling of the mixed text. The confounding effects of spelling mistakes and inconsistencies in the NLP literature have been independently observed by Winata *et al.* [32]. They proposed a bilingual character bidirectional RNN to model OOV words. POSIT uses this approach to capture character level information and address diversity in identifier names.

Additionally, we consider the structural morphology of the tokens. Code tokens are represented differently to natural language tokens. This is due to coding conventions in naming variables. We utilise these norms in developing a representation for the token. Specifically, we encode common conventions and spelling features into a feature vector. We record if the token is: (1) UPPER CASE, (2) Title Case, (3) lower case, (4) CamelCase, (5) snake_case; or if any character: (6) other than the first one is upper case, (7) is a digit, or (8) is a symbol. It may surprise you that font, while often used by humans to segment mixed text, is not in our token morphology feature vector. We did not use it as it is not available in our datasets. For the purposes of code reuse, we use a sequential model over this vector as well, similar to the character level vector, although there is no inherent sequentiality to this data. By ablating the high-level model features, we found that this token morphology feature vector did not significantly improve model performance (Section 5.3).

Encoding and Architecture. At a glance, our network, which we present diagrammatically in Figure 4, works as follows:

$$\mathbf{x}(t) = [\mathbf{f_w}(w_t); \mathbf{f_c}(w_t); \mathbf{f_f}(w_t)], \tag{1}$$

$$\mathbf{h}(t) = f(\mathbf{W}\mathbf{x}(t) + \mathbf{U}\mathbf{h}(t-1)), \tag{2}$$

$$\mathbf{y}(t) = g(\mathbf{V}\mathbf{h}(t)). \tag{3}$$

In Equation (1), we have three sources of information: character-level encodings ($\mathbf{f_c}(w_t)$), token-level encodings ($\mathbf{f_w}(w_t)$) and a feature vector over token morphology ($\mathbf{f_f}(w_t)$). Each captures properties at a different level of granularity. To preserve information, we embed each source independently into a vector space, represented by the three f functions. For both the feature vector and the characters within a word, we compute a representation by passing them as sequences through the biLSTM network in Figure 3. This figure represents the internals of $\mathbf{f_c}(w_t)$ and $\mathbf{f_f}(w_t)$ from Equation (1) and allows the model to learn patterns within sequences of characters as well as coding naming or spelling conventions cooccurrence patterns. The results of these two biLSTMs together with a word embedding function $\mathbf{f_w}$ are concatenated to become the input to the main biLSTM, $\mathbf{x}(t)$ in Equation (1). This enables the network to learn, based on a corpus, semantics for each token. This vector represents the input cells in our full network overview in Figure 4, which is enclosed in the box.

We pass the input vector $\mathbf{x}(t)$ through a biLSTM. The biLSTM considers both left and right tokens when predicting tags. Each cell performs the actions of Equation (2) and Equation (3), with the remark that the backwards-LSTM has the index reversed. This allows the network to consider context up to sentence boundaries. We then make use of the standard softmax function:

$$\mathrm{softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for } j = 1, ..., K; \tag{4}$$

which allows us to generate output probabilities over our learning targets as such:

$$\mathbf{p}(\mathrm{tag}_t \mid \mathrm{tag}_{t-1}) = \mathrm{softmax}(\mathbf{y}(t)), \tag{5}$$

$$\mathbf{p}(\mathrm{l}_{\mathrm{id}_t} \mid \mathrm{l}_{\mathrm{id}_{t-1}}) = \mathrm{softmax}(2\mathrm{LP}(\mathbf{h}(t))), \tag{6}$$

Equation (6) represents language ID transition probabilities, and Equation (5) — tag transition probabilities. In Equation (6), 2LP represents a 2-layer Multi Layer Perceptron. We make use of these transition probabilities in the CRF layer to output Language IDs and tags for each token while considering predictions made thus far. The trained eye may recognise in Equation (5) and Equation (6) the transition probabilities of two Markov chains. Indeed, we obtain the optimal output sequence by Viterbi-decoding [30]. While Equation (5) may seem to indicate that only single tags can be output by this architecture, this is not true. Given enough data, we can map tuples of tags to new fresh tags and decode at output time. This may not be as efficient as performing multi-tag output directly.

To train the network, we use the negative log-likelihood of the actual sequence being decoded from the CRF network and we backpropagate this through the network. Since we have two training goals, we combine them in the loss function by performing a weighted sum of the negative log-likelihood losses for each individual task, then train the network to perform both tasks jointly. When deployed, POSIT makes use of the CLANG lexer python port to generate the token input required by f_w, f_c, and f_f.

5 EVALUATION

For each token, POSIT makes two predictions: language IDs and PoS/AST tags. The former task represents correctly identifying where to add </code>-tags. This measures how well POSIT segments English and code. Section 5.2 reports POSIT's performance on this task on the evaluation set. For PoS/AST tag prediction, we focus on POSIT's ability to provide tags describing the function of tokens for both modalities reliably. To measure POSIT's performance here, we consider how well the model predicts the tags for a withheld evaluation dataset, which Section 5.2. presents along with the English-code segmentation result.

POSIT implements the network discussed in Section 4 in TensorFlow [4]. It uses the Adaptive Moment Estimation (Adam) [16] optimiser to assign the weights in the network. We trained it up to 30 epochs or until we did not observe improvement in three consecutive epochs. We used micro-batches of 64, a learning rate of 10^{-2}, and learning decay rate of 0.95. We use a 100 dimensional word embedding space and a 50 dimensional embedding space for characters. The LSTM hidden state is 96 dimensional for the word representation, 48 dimensional for characters and 4 for the token morphology feature vector. The output of the tag CRF is the concatenation of all final biLSTM states. We use a 2 layer perceptron with 64 and 8 dimensional hidden layers for language ID prediction. We apply a dropout of 0.5. Section 5.2 uses this implementation for validation and Section 5.3 uses it for ablation. The model's source code is available at https://github.com/PPPI/POSIT.

All POSIT runs, training and evaluation, were performed on a high-end laptop using an Intel i7-8750H CPU clocked at 3.9GHz, 24.0 GB of RAM and a Nvidia 1070 GPU with 8 GB of VRAM.

The state-of-the-art tool StORMeD, which we use for comparison, is available as a webservice, which we use by augmenting the demo files made available at https://stormed.inf.usi.ch/#service.

5.1 Corpus Construction

For our evaluation, we make use of two corpora. We use both to train POSIT, and we evaluate on each to see the performance in two important use-cases, a natural language frame language with embedded code and the reverse. Table 1 presents their statistics.

The first corpus is the Stack Overflow Data-dump [2] that Stack Overflow makes available online as an XML-file. It contains the HTML of Stack Overflow posts with code tokens marked using `</code>`- as well as `</pre class="code">`-tags. These tags enable us to construct a ground-truth for the English-code segmentation task. To obtain the PoS tags for English tokens, we use the tokeniser and Standard Stanford part-of-speech tagger present in NLTK [8]. For AST tags, we use a python port of the CLANG lexer and label tokens using a frequency table built from the second, CodeComment corpus. This additionally ensures that both corpora have the same set of AST tags. We allow matches up to a Levenstein distance of three for them; we choose three from spot-checking the results of various distances: after three, the lists were long and noisy. We address the internal threat introduced by our corpus labelling in Section 7.2.

We built the second, CodeComment corpus [3], from the CLANG compilation of 11 native libraries from the Android Open Source Project (AOSP): boringssl, libcxx, libjpeg-turbo, libmpeg2, libpcap, libpng, netcat, netperf, opencv, tcpdump and zlib. We chose these libraries in a manner that diversifies across application areas, such as codecs, network utilities, productivity, and graphics. We wrote a CLANG compiler plugin to harvest all comments and the snippets in the source code around those comments. Our compiler pass further harvests token AST tags for individual tokens in the source code snippets. In-line comments are often pure English; however, documentation strings, before the snippets with which they are associated, contain references to code tokens in the snippet. We further process the output of the plugin offline where we parse doc-strings to decompose intra-sentential mixed text and add part-of-speech tags to the pure English text. Thus, by construction, we have both tag and language ID ground-truth data. We allow matches up to 3 edits away to account for misspellings that may exist in doc-strings. The former ground-truth is obtained from CLANG during the compilation of the projects, while English comments are tokenised and labelled using NLTK as above. For code tokens in comments, we override their language ID and tag using information about them from the snippet associated with the comment.

Table 1: Corpus statistics for the two corpora considered together with the Training and Development and Evaluation splits. We performed majority class (English) undersampling only for the Stack Overflow training corpus.

Corpus Name	Tokens		Sentences		English Only Sentences		Code Only Sentences		Mixed Sentences	
	Train&Dev	Eval	Train&Dev	Eval	Train&Dev	Eval	Train&Dev	Eval	Train&Dev	Eval
Stack Overflow	7645103	2612261	214945	195021	55.8%	57.0%	32.6%	38.0%	11.6%	4.9%
CodeComments	132189	176418	21681	8677	11.3%	11.0%	79.4%	79.6%	9.4%	9.3%
Total	7777292	2788679	236626	203698	51.7%	55.1%	36.9%	39.7%	11.4%	5.1%

A consequence of using CLANG to source our AST tags is that we are limited to the mixed text tagging problem with a single natural language ($k = 1$) and a single formal language ($l = 1$). This limitation is a property of the data and not the model.

5.2 Predicting Tags

Here, we explore POSIT's accuracy on the language identification and PoS/AST tagging subtasks of the mixed text tagging problem. We adapt StORMeD for use as our baseline and compare its performance against that of POSIT. We note, even after adaption for the AST tagging task for which it was not designed, StORMeD is the existing state-of-the-art. We close by reporting POSIT's performance on non-Java posts.

To compare with the existing tool StORMeD, we restrict our Stack Overflow corpus to Java posts, because Ponzanelli *et al.* designed StORMeD to handle Java, JSON, and XML. Further, StORMeD and POSIT do not solve the same problems. StORMeD parses mixed posts into HAST trees; POSIT tags sequences. Thus, we flatten StORMeD's HASTs and use the AST label of the parent of terminal nodes as the tag. Because StORMeD builds HASTs for Java, JSON or XML while POSIT uses CLANG to tag code tokens, we built a map from StORMeD's AST tag set to ours[3]. As this mapping may be imperfect, StORMeD's observed performance on the PoS/AST tagging task is a lowerbound (Section 7.2).

For the language identification task, StORMeD exposes a 'tagger' webservice. Given mixed text in HTML, it replies with a string that has `</code>` HTML-tags added. We parse this reply to obtain our token-level language tags as in Section 5.1. For PoS/AST tagging, StORMeD exposes a 'parse' webservice. Given a Stack Overflow post with with correctly labelled code in HTML (Section 5.1), this service generates HASTs. We flatten and translate these HASTs as described above. To use these services, we break our evaluation corpus up into 2000 calls to StORMeD's webservices, 1000 for the language identification task, and the other 1000 for HAST generation. This allows us to comply with its terms of service.

Language Tagging. Here, we compare how well StORMeD and POSIT segment English and code in the Java Stack Overflow corpus. Unlike StORMeD's original setting, we elide user-provided code token labels, both from StORMeD and POSIT to avoid data leakage. Predicting them is the very task we are measuring. The authors of StORMeD account for this scenario [22, §II.A]. Although StORMeD must initially treat the input as a text fragment node, StORMeD still runs an island grammar parser to find code snippets embedded within it. Despite being asked to perform on a task for which it was not designed, due to the elision of user-provided code labels, StORMeD performs very well on our evaluation set and, indeed, as pioneering, post-regex work, defined the previous state of the art on this task. In this setting, StORMeD obtains 71% accuracy, POSIT achieves 81.6%.

PoS/AST Tagging. Here, we use StORMeD as a baseline to benchmark POSIT's performance on predicting PoS/AST tags for each token. Granted, on the text fragment nodes, we are actually measuring the performance of the NLTK PoS tagger. Unlike the first task, we allow StORMeD to use user-provided code-labels for this subtask. POSIT, however, solves the two subtasks jointly, so giving it these labels as input remains a data leak. Therefore, we do not provide them to POSIT. After flattening and mapping HAST labels to our label universe, as described above, StORMeD achieves a more than respectable accuracy of 61.9%, while POSIT achieves 85.6%.

On a uniform sample set of 30 posts from queries to StORMeD, we observed StORMeD to struggle with single word tokens or other short code snippets embedded within a sentence, especially when these, like foo, bar, do not match peculiar-to-code naming conventions. While this is also a more difficult task for POSIT as well, it fares better. Consider the sentence 'Class A has a one-to-many relationship to B. Hence, A has an attribute collectionOfB .'. Here, StORMeD spots Class A and collectionOfB , the uses of A and B as stand-alone tokens slips passed the heuristics. POSIT manages to spot all four code tokens. POSIT's use of word embeddings, allows it to learn typical one word variable names and find unmarked code tokens that escape StORMeD's heuristics, such as all lowercase function names that are defined in a larger snippet within the post. For its part, StORMeD handled documentation strings well, identifying when code tokens are referenced within them. POSIT preferred to treat the doc-string as being fully in a natural language, missing code references that existed within them even when they contained special mark-up, such as @.

Beyond Java, JSON, and XML. POSIT is not restricted to Java, so we report its performance on the entire Stack Overflow corpus and on the CodeComment corpus. The former measures the performance on mixed text which has English as a frame language; the latter measures the performance on mixed text with source code as the frame language. On the complete Stack Overflow corpus, POSIT achieves an accuracy of 97.7% when asked to identify the language of the token and an accuracy of 93.8% when predicting PoS/AST tags. We calculated the first accuracy against user-provided code tags and the second against our constructed tags (Section 5.1). On the CodeComment corpus, we tweak POSIT's training. As examples within this corpus tend to be longer, we reduce the number

[3] The mapping can be found online at https://github.com/PPPI/POSIT/blob/92ef801e5183e3f304da423ad50f58fdd7369090/src/baseline/StORMeD/stormed_evaluate.py#L33.

of micro-batches to 16. After training on CodeComment, POSIT achieves an accuracy of 99.7% for language identification and an accuracy of 98.9% for PoS/AST tag predictions.

5.3 Model Ablation

POSIT depends on three kinds of embeddings — character, token, and token morphology — and CRF layer prior to decoding (Equation (1)). We can ablate all except the token embeddings, our bedrock embedding. We used the same experimental set-up described at the beginning of this section, with one exception: When ablating the CRF layer, we replaced it with a 2-Layer Perceptron whose output we then apply softmax to.

Table 2 shows the results. Keeping only the CRF-layer reduces the time per epoch from 3:30 hours to 1:03 hours (the first bolded row). On average, POSIT's model requires 6 to 7 epochs until it stops improving on the development set, so we stop. This configuration reduces training time by ∼14 hours. Further, it slightly increases performance. Only using the CRF, however, manual spot-checking reveals that POSIT incorrectly assigns token that obey common coding conventions and method call tokens as English. This is due to English to code class imbalance, and inspecting Table 1 makes this clear. The best performing model under human assessment of uniformly sampled token (the second bolded row) removes only the token morphology feature vector. Essentially, this model drops precisely those heuristics that we anecdotally know humans use when performing this task. Since dropping either the character or the token morphology embeddings yields almost identical performance, we hypothesise that POSIT learns these human heuristics, and perhaps others, in the character embeddings. We choose to keep character embeddings, despite training cost, for this reason.

6 POSIT APPLIED

POSIT can improve downstream tasks. First, we show how POSIT accurately suggests code tags to separate code from natural language, such as Stack Overflow's backticks. POSIT achieves 95% balanced accuracy on this task. Developers could use these accurate suggestions to improve their posts before submitting them; researchers could use them to preprocess and clean data before using it in downstream applications. For instance, Yin *et al.* [33] start from a parallel corpus of Natural Language and Snippet pairs and seek to align it. POSIT could help them extend their initial corpus beyond StackOverflow by segmenting mixed text into pairs. Second, we show how POSIT's language identification and PoS tagging predictions enable TaskNav — a tool that supports task-based navigation of software documentation by automatically extracting task phrases from a documentation corpus and by surfacing these task phrases in an interactive auto-complete interface — to extract new and more detailed tasks. We conduct these demonstrations using POSIT's best performing configuration, which ignores token morphology (Section 5.3).

6.1 Predicting Code Tags

Modern developer fora, notably Stack Overflow, provide tags for separating code and NL text. These tags are an unusual form of punctuation, so it is, perhaps, not surprising that developers often neglect to add them. Whatever the reason, these tags are often missing [21]. POSIT can help improve post quality by serving as the basis of a post linter that suggests code tags. A developer could use such a linter before submitting their post or the server could use this linter to reject posts.

Our Stack Overflow corpus contains posts that have been edited solely to add missing code tags. To show POSIT accuracy at suggesting missing code tags, we extracted these posts using the SOTorrent dataset [6]. First, we selected all posts that contain a revision with the message "code formatting". We uniformly, and without replacement, sampled this set for 30 candidates. We kept only those posts that made whitespace edits and introduced single or triple backticks. By construction, this corpus has a user-defined ground truth for code tags. We use the post before the revision as input and compare against the post after the revision to validate. POSIT manages to achieve a balanced accuracy of 95% on the code label prediction task on this corpus.

6.2 TaskNav++

To demonstrate the usefulness of POSIT's code-aware part-of-speech tagging, we augment Treude *et al.*'s TaskNav [29] to use POSIT's language identification and its PoS/AST tags.

To construct TaskNav++, we replaced TaskNav's Stanford NLP PoS tagger with POSIT. Like TaskNav, TaskNav++ maps AST tags to "NN". TaskNav uses the Penn Treebank [17] tag set; POSIT uses training data labelled with Universal tag set tags [19]. These tags sets differ; notably, the Penn Treebank tags are more granular. To expose POSIT's tags to TaskNav's rules to use those rules in TaskNav++, we converted our tags to the Penn Treebank tag set. This conversion harms TaskNav++'s performance, because it uses the Java Stanford Standard NLP library which expects more granular tags, although it can handle the coarser tags POSIT gives it.

To compare TaskNav and TaskNav++, we asked both systems to extract tasks from the same Linux Kernel Mailing List corpus that we manually analysed (Section 7.1). TaskNav++ finds 97 new tasks in the 30 threads or 3.2 new tasks per thread. Of these, 65 (67.0%) are reasonable tasks for the e-mail they were extracted from. Two of the authors performed the labelling of these tasks, we achieved a Cohen Kappa of 0.21, indicating fair agreement. Treude *et al.* also report low agreement regarding what is a relevant task [29]. To resolve disagreements, we consider a task reasonable if either author labelled it as such. The ratio of reasonable tasks is in the same range as that reported in the TaskNav work, *viz.* 71% of the tasks TaskNav extracted from two documentation corpora were considered meaningful by at least one of two developers of the respective systems. TaskNav prioritises recall over precision to enable developers to use the extracted tasks as navigation cues. POSIT's ability to identify more than two additional reasonable tasks per email thread contributes towards this goal.

Inspecting the tasks extracted, we find that some tasks benefit from POSIT's tokenisation. For example in 'remove excessive untagging in gup' vs 'remove excessive untagging in gup.c' the standard tokeniser assumed that the use of '.' in 'gup.c' indicates the end of a sentence. Our tokenisation also helps correctly preserve mention uses of code tokens: 'pass in "real + len"' and 'pass in "dest + len"', and even English-only mention uses: 'call writeback bits "tags"', 'split trampoline buffer into "temporary" destination buffer'. In all

Table 2: The result on the evaluation set for the different ablation configurations. All configurations use the token embedding as it is our core embedding. Observe that using only the CRF layer performs best on the language identification and the POSIT's tagging tasks.

High-level features	Language ID Accuracy	Tagging Accuracy	Mean Accuracy	Time per Epoch (hh:mm)
Only token embeddings	0.209	0.923	0.568	00:37
Only CRF	**0.970**	**0.926**	**0.948**	**01:03**
Only feature vector	0.450	0.928	0.689	00:45
Only character embeddings	0.312	0.923	0.617	02:32
No CRF	0.409	0.913	0.661	02:59
No feature vector	**0.966**	**0.924**	**0.945**	**03:19**
No character embeddings	0.966	0.919	0.943	01:39
All features	0.970	0.917	0.944	03:30

these cases, either TaskNav finds an incorrect version of the task ('add len to real rip') or simply loses the double-quotes indicating a mention use (for the English-only mention cases).

POSIT's restriction to a single formal language proved to be a double-edged sword. It helped separate patches that are in-lined with e-mails in our manual analysis of POSIT on the LKML (Section 7.1), while here we can see that it is problematic. By training only on a single programming language, POSIT misidentifies change-log and file-path lines as code. This propagates to TaskNav++, which in turn incorrectly adds these as tasks since POSIT stashes the path or change-log into a single code element. At times, this behaviour was also beneficial, such as annotating the code in the task: 'read <tt>extent [i]</tt>', this comes at the cost of generating incorrect tasks such as: 'change android <tt>/ ion / ion.c | 60 +++ +++ +++ +++ +++ +++ +</tt>'. We hypothesise that a solution to the general mixed text tagging problem would avoid this problem by explicitly training to identify file paths.

7 DISCUSSION

In this section, we first perform a deep dive into POSIT's output and performance. Then we address threats to POSIT's model, its training, and methodology.

7.1 POSIT Deep Dive

POSIT is unlikely to be the last tool to tackle the mixed text tagging problem. To better understand what POSIT does well and where it can be improved, we manually assessed its output on two corpora: a random uniform sample of 10 Stack Overflow posts from our evaluation set and a random uniform sample of 10 e-mails from the Linux Kernel Mailing List sent during August 2018. The Stack Overflow sample contains 3,233 tokens while the LKML — 17,451. We finish by showing POSIT's output on a small Stack Overflow post. Broadly, POSIT's failures are largely due to tokenisation problems, class imbalance, and lack of labels. Concerning the label problem, our data actually consists of a single natural language and several formal languages, one of which is a programming language, the others include diffs, URLs, mail headers, and file paths. This negatively impacted TaskNav++ by exposing diff headers and file paths as code elements, inducing incorrect tasks to be extracted. Our deep dive also revealed that POSIT accurately PoS-tags English, accurately AST-tags lone code tokens, and learned to identify diffs as formal, despite lack of labels.

To pre-process training data, POSIT uses two tokenisers: the standard NLTK tokeniser and a Python port of the CLANG lexer. POSIT uses labels (Stack Overflow's code tags) in the training to switch between them. In the data, we observed that POSIT had tagged some double-quotation marks as Nouns. Since the user-provided code labels are noisy [21], we hypothesise the application of the code tokeniser to English caused this misprediction. Designed to dispense with code-labels, POSIT exclusively relies on the CLANG lexer port during evaluation. Unsurprisingly, then, we observed POSIT incorrectly tagging punctuation as code-unknown as multiple punctuation tokens are grouped into single tokens that do not normally exist in English. We suspect this to due to applying English tokenisation to code snippets. Clearly, POSIT would benefit from tokenisation tailored for mixed text.

Within code segments, we also observed that POSIT had a proclivity to tag tokens as 'raw_identifier'. This indicates that context did not always propagate the 'method_name', or 'variable' tags across sentence boundaries. As the 'raw_identifier' tag was the go-to AST label for code, it suggests a class imbalance in our training data with regards to this label. Indeed, we observed POSIT to only tag a token as 'method_name' if it was followed by tokens that signify calling syntax — argument lists, including the empty argument list `()`.

This deep dive revealed a double-edged sword. Our sample contained snippets that represent diffs, URLs or file paths. POSIT's training data does not labels these formal languages nor did tokenisation always preserve URLs and file paths. Nonetheless, POSIT managed to correctly segment diffs by marking them as code, performing this task exceptionally well on the LKML sample. URLs and file paths were seen as English unless the resource names matched a naming convention for code. For URLs, POSIT tagged key-argument pairs (post_id=42) as ('variable', 'operation', 'raw_identifier'). Later in Section 6.2, POSIT's tendency to segment diffs as code was detrimental, since it stashed diff headers into a single code token, causing TaskNav++ to produce incorrect tasks.

An additional observation during our manual investigation is the incorrect type of tag relative to the language of the token. Consider the following:

	English	Code
PoS output	33.4%	1.5%
AST output	5.6%	59.5%

There are two ways of fixing the problem. The first is to use a comma to sequence statements within the macro without robbing it of its ability to act like an expression.

```
#define BAR(X) f(X), g(X)
```

The above version of bar `BAR` expands the above code into what follows, which is syntactically correct.

```
if (corge)
    f(corge), g(corge);
else
    gralt();
```

This does not work if instead of `f(X)` you have a more complicated body of code that needs to go in its own block, say for example to declare local variables. In the most general case the solution is to use something like `do ... while` to cause the macro to be a single statement that takes a semicolon without confusion.

Figure 5: Example sentence taken from Stack Overflow which freely mixes English and very short code snippets, here rendered using monospaced font. We can see both inter-sentential code-switching, such as the macro definition and the short example if statement snippet, as well as intra-sentential code-switching, the mention of the code token `f(X)` and the code construct '`do ... while`'.

We obtain these numbers by considering the agreement between the language identification task and the type of tag output for the Stack Overflow posts and LKML mails used in this deep dive. We can see that for 7.1% of the tokens (908) in our manual investigation POSIT outputs the wrong type of label given the language prediction. This was also observed by the authors for cases where one of the predictions was wrong while the other was correct, such as tagging a Noun as such while marking it as code. This is because we separated the two tasks after the biLSTM and trained them independently. We hypothesise that adding an additional loss term that penalises desynchronising these tasks would solve this problem. Alternatively, one could consider a more hierarchical approach, for example first predicting the language id, then predicating the tag output conditioned on this language id prediction.

For monolingual sentences, either English or code, POSIT correctly PoS- or AST-tagged the sequences. Spare the occasional hiccup at switching from English to code a single token too late, POSIT correctly detected the larger contiguous snippets. As code snippets ended, POSIT was almost always immediate to switch back to identifying tokens as English. For smaller embedded code snippets, POSIT correctly identified almost all method calls that were followed by argument lists, including '()'. POSIT almost always correctly identified operators and keywords even when used on their own in a mention role in the host language. Further, single token mentions of typical example function names, like foo or bar, code elements that followed naming conventions, or code tokens that were used in larger snippets within the same post were correctly identified as code.

In Figure 5, we observe 91% tag accuracy for English and 66.7% tag accuracy for code. The language segmentation is 76.7% accurate. POSIT correctly identifies the two larger code snippets as code except for the first token in each: '#define' and 'if'. It fails to spot do ... while as code, perhaps due to do and while being used within English sufficiently often to obscure the mention-role of the construct. On the other hand, it correctly spots f(X) as code since f and X are rarely used on their own in English.

7.2 Threats to Validity

The external threats to POSIT's validity relate mainly to the corpora, including the noisy nature of StackOverflow data [20], and the potential of the model to overfit. POSIT generalises to the extent to which its training data is representative. To avoid overfitting, we use a development set and an early stopping criterion (three epochs without improvement), as is conventional.

In Section 6.1, we show that despite the noisy training labels, POSIT is capable of predicating code-tags/spans that users originally forgot to provide. We also explore POSIT's performance on a corpus that is likely to differ from both training corpora, the Linux Kernel Mailing List (LKML) which was used during the deep dive (Section 7.1). This validation was performed manually due to lack of a ground truth; automatically generating a ground truth for this data would not escape the internal threats presented below. On this corpus, POSIT achieves a language identification accuracy of 76.6% and a PoS/AST tagging accuracy of 76.5%. Two of the authors have performed the labelling of this task and the Cohen kappa agreement [10] for the manual classification is 0.783, which indicates substantial agreement. We resolved disagreements by considering an output correct if both authors labelled it as such.

Neural networks are a form of supervised learning and require labels. We labelled our training in two ways, using one procedure for language labels and another for PoS/AST tags. Both procedures are subject to a threat to their construct validity. The language labels are user-provided and thus subject to noise, PoS tags are derived from an imperfect PoS tagger, and AST tags are added heuristically. For language labels, we both manually labelled data and exploited a human oracle. We manually labelled a uniformly sampled subset of 10 posts with 3,233 tokens from our Stack Overflow evaluation data, then manually assessed POSIT's performance on this subset. Two authors performed the manual labelling and achieved a Cohen kappa of 0.711 indicating substantial agreement. A similar procedure was applied to the LKML labelling task. On this validation, POSIT achieved 93.8 accuracy. Our Stack Overflow corpus contains revision histories. We searched this history for versions whose edit comment is "code formatting". We then manually filtered the resulting versions to those that only add code token labels (defined in Section 6.1). POSIT achieved 95% balanced accuracy on this validation. For the PoS/AST tagging task, we manually added the PoS/AST tags on the same 10 Stack Overflow posts, we used above. Here, POSIT achieved 93.7%.

In Section 5.2, we used StORMeD as a baseline for POSIT. As previously discussed, StORMeD was not designed for our task and handles Java, JSON, and XML. Adapting to our setting introduces an internal threat. To address this threat, we evaluated both StORMeD and POSIT only on those Stack Overflow posts tagged as Java. These tags are noisy [11, 20]. When evaluating StORMeD, its authors, Ponzanelli *et al.* used the same filter. We also map the StORMeD AST tag set to ours[4]. If the true mapping is a relation, not a function, then this would understate StORMeD's performance. This is unlikely because Java ASTs and CLANG ASTs are not that dissimilar. Further, POSIT must also contend with this noise. When building TaskNav++ (Section 6.2), we use a more coarse grained PoS tag set than the original TaskNav potentially reducing its performance.

8 RELATED WORK

In software engineering research, part-of-speech tagging has been directly applied for identifier naming [7], code summarisation [12, 13], concept localisation [5], traceability-link recovery [9], and bug fixing [27]. We first review natural language processing (NLP) research on code-switching, the natural language analogue of the mixed text problem. This is work on which we based POSIT. Then we discuss initial efforts to establish analogues for parts of speech categories for code and use them to tag code tokens. We close with the pioneering work on StORMeD, the first context-free work to automatically tackle the mixed text tagging problem.

NLP researchers are growing more interested in code-switching text and speech[5]. The main roadblock had been the lack of high-quality labelled corpora. Previously, such data was scarce because code-switching was stigmatised [24]. The advent of social media, has reduced the stigma and provided code-switching data, especially text that mixes English with another language [31]. High quality datasets of code-switched utterances are now under production [1]. For the task of part-of-speech (PoS) tagging code-switching text, Solorio and Liu [25] presented the first statistical approach to the task of part-of-speech (PoS) tagging code-switching text. On a Spanglish corpus, they heuristically combine PoS taggers trained on larger monolingual corpora and obtain 85% accuracy. Jamatia *et al.* [15], working on an English-Hindi corpus gathered from Facebook and Twitter, recreated Solorio's and Liu's tagger and they proposed a tagger using Conditional Random Fields. The former performed better at 72% vs 71.6%. In 2018, Soto and Hirschberg [26] proposed a neural network approach, opting to solve two related problems simultaneously: part-of-speech tagging and Language ID tagging. They combined a biLSTM with a CRF network at both outputs and fused the two learning targets by simply summing the respective losses. This network achieves a test accuracy of 90.25% on the inter-sentential code-switched dataset from Miami Bangor [1]. POSIT builds upon their model extended with Winata *et al.*'s [32] handling of OOV tokens, as discussed in Section 4.

Operating directly on source code (not mixed text), Newman *et al.* [18] sought to discover categories for source code identifiers analogous to PoS tags. Specifically, they looked for source code equivalents to Proper Nouns, Nouns, Pronouns, Adjectives, and Verbs. They derive their categories from 1) Abstract syntax trees, 2) how the tokens impact memory, 3) where they are declared, and 4) what type they have. They report the prevalence of these categories in source-code. Their goal was to map these code categories to PoS tags, thereby building a bridge for applying NLP techniques to code for tasks such as program comprehension. Treude *et al.* [28] described the challenges of analysing software documentation written in Portuguese which commonly mixes two natural languages (Portuguese and English) as well as code. They suggested the introduction of a new part-of-speech tag called Lexical Item to capture cases where the "correct" tag cannot be determined easily due to language switching.

Ponzanelli *et al.* are the first to go beyond using regular expressions to parse mixed text. When customising LexRank [23], a summarisation tool for mixed text, they employed an island grammar that parses Java and stack-trace islands embedded in natural language, which is relegated to water. They followed up LexRank with StORMeD, a tool that uses an island grammar to parse Java, JSON, and XML islands in mixed text Stack Overflow posts, again relegating natural language to water [22]. StORMeD produces heterogeneous abstract syntax trees (AST), which are ASTs decorated with natural language snippets.

StORMeD relies on Stack Overflow's code tags; when these tags are present, island grammars are a natural choice for parsing mixed text. Mixed text is noisy and Stack Overflow posts are no exception [21]. To handle this noise, StORMeD resorts to heuristics (anti-patterns in the nomenclature of island grammars), which they build into their island grammar's recognition of islands. For instance, if whitespace separates a method identifier from its '(', they toss that method identifier into water. To identify class names that appear in isolation, they use three heuristics: the name is a class if it is a fully qualified name with no internal spaces, contains two instances of CamelCase, or syntactically matches a Java generic type annotation over builtins. They use similar rules to handle Java annotation because Stack Overflow also uses '@' to mention users in posts. Heuristics, by definition, do solve a problem in general. For example, the generic method names often used in examples — foo, bar, or buzz — slip past their heuristics when appearing alone in the host language. This is true even when the post defines the method. Indeed, we show that no island grammar, which, by definition, extend a context-free grammar, can solve this Sisyphean task for mixed text, because we show this task to be context-sensitive in Section 3. Island grammar's anti-patterns do not make island grammars context-sensitive.

StORMeD and POSIT solve related but different mixed text problems. StORMeD recovers natural language, unprocessed, from the water, builds ASTs for its islands, then decorates those ASTs with natural language snippets to build its HAST. In contrast, POSIT tags both natural languages and formal languages, but does not build trees. StORMeD and POSIT do overlap on two subtasks of mixed text tagging: language identification and AST-tagging code. To compare them on these tasks, we had to adapt StORMeD. Essentially, we traverse the HASTs and consider the first parent of a terminal node to be the AST tag. We map these from the StORMeD tag set to ours (Section 5.2). POSIT advances the state of the art on these two tasks (Section 5.2).

As a notable service to the community, Ponzanelli *et al.* both provided a corpus of HASTs and StORMeD as an excellent, well-maintained web service. Their HAST corpus is a structured dataset that allows researchers a quick start on mining mixed text: it spares them from tedious pre-processing and permits the quick extraction and processing of code-snippets, for tasks like summarisation. We

[4] The mapping can be found online at https://github.com/PPPI/POSIT/blob/92ef801e5183e3f304da423ad50f58fdd7369090/src/baseline/StORMeD/stormed_evaluate.py#L33.

[5] The NLP term for text and speech that mixed multiple natural languages.

have published our corpus at https://github.com/PPPI/POSIT to complement theirs. Our project, which culminated in POSIT, would not have been possible without these contributions.

9 CONCLUSION

We have defined the problem of tagging mixed text. We present POSIT, implemented using a biLSTM-CRF Neural Network and compared it to Ponzanelli *et al.*'s pioneering work, StORMeD [22] on Java posts in Stack Overflow. We show that POSIT accurately identifies English and code tokens (81.6%), then accurately tags those tokens with their part-of-speech tag for English or their AST tag for code (85.6%). We show that POSIT can help developers by improving two downstream tasks: suggesting missing code labels in mixed text (with 95% accuracy) and extracting tasks from mixed text through TaskNav++, which exploits POSIT's output to find more than two new reasonable tasks per document.

POSIT and our CodeComment corpus are available at https://github.com/PPPI/POSIT.

10 ACKNOWLEDGEMENTS

We thank Ponzanelli *et al.* for developing and maintaining StORMeD, a powerful and easy-to-use tool, and for their prompt technical assistance with the StORMeD webservice. This research is supported by the EPSRC Ref. EP/J017515/1.

REFERENCES

[1] 2011. Bangor Talk Miami Corpus. http://www.bangortalk.org.uk/speakers.php?c=miami.

[2] 2018. Stack Exchange Data Dump. https://archive.org/details/stackexchange. [Online; accessed 05-Sep-2018].

[3] 2019. Code Comment Corpus. https://github.com/PPPI/POSIT/blob/master/data/corpora/lucid.zip. [Online; accessed 24-Jan-2020].

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[5] Surafel Lemma Abebe and Paolo Tonella. 2010. Natural language parsing of program element names for concept extraction. In *IEEE 18th Int. Conf. on Prog. Comp. (ICPC) 2010*. IEEE, 156–159.

[6] Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. 2018. Sotorrent: Reconstructing and analyzing the evolution of Stack Overflow posts. In *Proc. 15th Int. Conf. Min. Soft. Rep.* ACM, 319–330.

[7] Dave Binkley, Matthew Hearn, and Dawn Lawrie. 2011. Improving identifier informativeness using part of speech information. In *Proceeding 8th Work. Conf. Min. Softw. Repos. - MSR '11*. ACM Press, New York, New York, USA, 203. https://doi.org/10.1145/1985441.1985471

[8] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python.* O'Reilly Media.

[9] Giovanni Capobianco, Andrea De Lucia, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. 2013. Improving IR-based traceability recovery via noun-based indexing of software artifacts. *Journal of Software: Evolution and Process* 25, 7 (2013), 743–762.

[10] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. https://doi.org/10.1177/001316446002000104 arXiv:https://doi.org/10.1177/001316446002000104

[11] Jens Dietrich, Markus Luczak-Roesch, and Elroy Dalefield. 2019. Man vs machine: a study into language identification of stack overflow code snippets. In *Proc. 16th Int. Conf. Min. Soft. Repo.* IEEE Press, 205–209.

[12] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting Program Comprehension with Source Code Summarization. In *Proc. 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2* (Cape Town, South Africa) *(ICSE '10)*. ACM, New York, NY, USA, 223–226. https://doi.org/10.1145/1810295.1810335

[13] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *2010 17th Work. Conf. on Rev. Eng.* 35–44. https://doi.org/10.1109/WCRE.2010.13

[14] Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF Models for Sequence Tagging. (2015). https://doi.org/10.1061/(ASCE)CO.1943-7862.0000274. arXiv:1508.01991

[15] Anupam Jamatia, Björn Gambäck, and Amitava Das. 2015. part-of-speech tagging for code-mixed english-hindi twitter and facebook chat messages. In *Proc. Int. Conf. Rec. Adv. in Nat. Lang. Proc.* 239–248.

[16] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014). http://dblp.uni-trier.de/db/journals/corr/corr1412.html#KingmaB14

[17] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. (1993).

[18] Christian D Newman, Reem S Alsuhaibani, Michael L Collard, and Jonathan I Maletic. 2017. Lexical Categories for Source Code Identifiers. *SANER'17* (2017).

[19] Slav Petrov, Dipanjan Das, and Ryan McDonald. 2011. A universal part-of-speech tagset. *arXiv preprint arXiv:1104.2086* (2011).

[20] Luca Ponzanelli. 2014. Holistic recommender systems for software engineering. In *Companion Proc. 36th Int. Conf. Soft. Eng.* 686–689.

[21] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, Michele Lanza, and David Fullerton. 2014. Improving low quality stack overflow post detection. In *2014 IEEE Int. Conf. Soft. Maint. Evol.* IEEE, 541–544.

[22] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. 2015. StORMeD: Stack overflow ready made data. *IEEE Int. Work. Conf. Min. Softw. Repos.* 2015-Augus (2015), 474–477. https://doi.org/10.1109/MSR.2015.67

[23] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. 2015. Summarizing complex development artifacts by mining heterogeneous data. *IEEE Int. Work. Conf. Min. Softw. Repos.* 2015-Augus (2015), 401–405. https://doi.org/10.1109/MSR.2015.49

[24] Shana Poplack. 1980. Sometimes I'll start a sentence in Spanish Y TERMINO EN ESPAÑOL: toward a typology of code-switching 1. *Linguistics* 18 (01 1980), 581–618. https://doi.org/10.1515/ling.1980.18.7-8.581

[25] Thamar Solorio and Yang Liu. 2008. part-of-speech tagging for English-Spanish code-switched text. In *Proc. Conf. on Emp. Meth. in Nat. Lang. Proc.* Association for Computational Linguistics, 1051–1060.

[26] Victor Soto and Julia Hirschberg. 2018. Joint part-of-speech and Language ID Tagging for Code-Switched Data. (2018), 1–10.

[27] Yuan Tian and David Lo. 2015. A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports. In *SANER'15*. IEEE, 570–574.

[28] Christoph Treude, Carlos A Prolo, and Fernando Figueira Filho. 2015. Challenges in analyzing software documentation in Portuguese. In *Proc. 29th Bra. Sym. Soft. Eng.* IEEE, 179–184.

[29] Christoph Treude, Mathieu Sicard, Marc Klocke, and Martin Robillard. 2015. TaskNav: Task-based Navigation of Software Documentation. In *Proc. 37th Int. Conf. Soft. Eng. - Volume 2* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 649–652. http://dl.acm.org/citation.cfm?id=2819009.2819128

[30] A. Viterbi. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory* 13, 2 (April 1967), 260–269. https://doi.org/10.1109/TIT.1967.1054010

[31] Yogarshi Vyas, Spandana Gella, Jatin Sharma, Kalika Bali, and Monojit Choudhury. 2014. Pos tagging of english-hindi code-mixed social media content. In *Proc. 2014 Conf. on Emp. Meth. in Nat. Lang. Proc. (EMNLP)*. 974–979.

[32] Genta Indra Winata, Chien-Sheng Wu, Andrea Madotto, and Pascale Fung. 2018. Bilingual Character Representation for Efficiently Addressing Out-of-Vocabulary Words in Code-Switching Named Entity Recognition. In *Proc. Third Workshop Comp. Appr. Ling. Code-Switching*. Association for Computational Linguistics, Melbourne, Australia, 110–114. https://doi.org/10.18653/v1/W18-3214

[33] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. *Proc. - Int. Conf. Softw. Eng.* (2018), 476–486. https://doi.org/10.1145/3196398.3196408 arXiv:1805.08949