Software Entity Recognition with Noise-Robust Learning

Tai Nguyen[‡*¶], Yifeng Di[†*], Joohan Lee[§¶], Muhao Chen[§], Tianyi Zhang[†]
[†]Purdue University, West Lafayette, USA
[‡]University of Pennsylvania, Philadelphia, USA
[§]University of Southern California, Los Angeles, USA
taing@seas.upenn.edu, di5@purdue.edu, joohanl@usc.edu, muhaoche@usc.edu, tianyi@purdue.edu

***Abstract*—Recognizing software entities such as library names from free-form text is essential to enable many software engineering (SE) technologies, such as traceability link recovery, automated documentation, and API recommendation. While many approaches have been proposed to address this problem, they suffer from small entity vocabularies or noisy training data, hindering their ability to recognize software entities mentioned in sophisticated narratives. To address this challenge, we leverage the Wikipedia taxonomy to develop a comprehensive entity lexicon with 79K unique software entities in 12 fine-grained types, as well as a large labeled dataset of over 1.7M sentences. Then, we propose *self-regularization*, a noise-robust learning approach, to the training of our software entity recognition (SER) model by accounting for many dropouts. Results show that models trained with self-regularization outperform both their vanilla counterparts and state-of-the-art approaches on our Wikipedia benchmark and two Stack Overflow benchmarks. We release our models[1], data, and code for future research.[2]**

***Index Terms*—Software Entity Recognition, Datasets, Noise-Robust Learning**

I. Introduction

Software entity recognition (SER) is an integral task for acquiring software-related knowledge. It serves as the backbone of many downstream software engineering applications, such as traceability link recovery [1]–[4], automated documentation [5]–[9], API recommendation [10]–[12], and bug fixing [13]–[16].

Early work in this research direction employs pattern-matching methods to identify software entities based on predefined linguistic patterns or predefined dictionaries [5], [17]–[19]. However, these methods lack the flexibility to handle the sophistication and ambiguity in free-form text [20]. Machine learning methods have been increasingly adopted to solve this task [21]–[26]. For example, S-NER [21] uses a feature-based Conditional Random Field (CRF) model to recognize software entities in five categories, including *Programming Language*, *Platform*, *API*, *Tool-library-framework* and *Software Standard*. However, S-NER is trained on a small dataset with 4,646 sentences and 2,404 named entities. It does not generalize well to commonly mentioned entities such as “AMD64” and “Memory Leak”. Furthermore, given the simplicity of its model design, it only achieves a 78% F1 score on a Stack Overflow dataset.

In general text domains, deep learning models, such as BiLSTM-CRF [27] and BERT-NER [28], have emerged as the current paradigms for Named Entity Recognition (NER). However, these models can only detect entities in general text domains, such as person names and locations. Due to the domain shift challenge, simply finetuning them to a highly specialized domain such as software engineering is not sufficient [23]. Recently, a new approach called SoftNER [23] has been proposed to detect fine-grained software entity types with BERTOverflow, a BERT model finetuned on Stack Overflow data. Their evaluation shows it has greatly outperformed BiLSTM-CRF and BERT-base models and can detect various types of software entities, such as operating systems and software libraries.

Despite the great stride, our assessment shows that existing models, including SoftNER, still fall short of addressing domain shift, limited vocabularies, and morphological names in software engineering. In particular, the training data of SoftNER is noisy, since it is constructed synthetically based on Stack Overflow (SO) tags, which can be created by any SO users and suffer from informal naming conventions and all sorts or randomness. Our manual analysis shows that the training data of SoftNER has a high labeling error rate of 17.79% (detailed in Section IV-E). We hypothesize that it may be due to the lack of widespread use of double annotation and metadata for automatic annotation. This motivates us to construct a new dataset with fewer labeling errors but more sentences and named entities.

To address this limitation, we develop an automated pipeline to develop a large, high-quality software entity dataset based on Wikipedia. We call this dataset WikiSER. Compared with Stack Overflow, Wikipedia strives to be a comprehensive online encyclopedia. It generally exhibits better structures, well-formedness, grammaticality, and semantic coherence in its natural language sentences [29]. Since Wikipedia contains articles in numerous domains, our approach first processes the Wikpedia taxonomy starting from the root category “Computing” and performs hierarchical pruning to only retain SE-related categories. Then, it extracts the titles of all articles belonging to these categories as well as their aliases curated

*Equal contribution.
¶Work done as remote research interns at Purdue University.
[1]https://huggingface.co/taidng/wikiser-bert-base; https://huggingface.co/taidng/wikiser-bert-large.
[2]https://github.com/taidnguyen/software_entity_recognition

by Wiki authors as the entity lexicon. In this way, we get overall 79K software entities in 12 fine-grained categories, e.g., algorithms, data structures, libraries, and OS.

Since Wikipedia articles often contain hyperlinks to other articles, each hyperlinked word or phrase can be treated as a mention of another entity, which can be leveraged to curate the text corpus with labeled entities. Based on our observation, Wiki authors typically only add hyperlinks to the first mention of an entity in a Wikipedia article. Thus, we further develop a matching method to automatically propagate entity types, so that we can obtain more sentences with labeled entities. In the end, we curate a large corpus with 1.7M sentences labeled with the 79K entities. Our manual validation demonstrates that the labeling error rate of our dataset is 9.17%, compared with an error rate of 17.79% in the SoftNER dataset (Section IV-E).

Furthermore, we propose a noise-robust learning framework called *self-regularization* that trains an SER model to be consistent with its predictions under a noisy setting. Specifically, to enhance the robustness of model training, our framework leverages the dropout mechanism to simulate the prediction inconsistency from multiple differently initialized models and incorporates an agreement loss as a regularization mechanism, encouraging prediction consistency in the presence of noisy labels. By relying solely on the training data, our framework offers several advantages over other noise-robust methods and can be easily adapted to any model initialization.

Our evaluation demonstrates that BERT models trained with our self-regularization framework outperform multiple baselines. Specifically, our self-regularized BERT$_{base}$ model outperforms a SOTA SER model called SoftNER [23] by 7.1% in terms of F1 score. Furthermore, self-regularization also outperforms co-regularization [30], a SOTA noise-robust learning method, by 2.9% in F1. This performance gain from self-regularization also generalizes to two existing SER datasets [23], [31] obtained from Stack Overflow. Finally, we observe that self-regularization is more effective for smaller models and in-domain training indeed plays a major role in boosting the performance gain of SER in different types of data, e.g., Wikipedia vs. Stack Overflow. Overall, these findings provide valuable insights into the strengths and limitations of our approach.

To sum up, we make the following contributions:

1) **Dataset.** We leverage Wikipedia to develop a comprehensive lexicon of 79K software entities in 12 fine-grained categories, as well as a large labeled dataset with 1.7M sentences and 3.4M entity labels. We make our dataset publicly available to cultivate future research.
2) **Model.** We propose a new noise-robust learning framework that regularizes the training of SER models via a dropout mechanism to account for labeling errors in SER datasets.
3) **Evaluation.** We conduct a comprehensive evaluation of the proposed approach against the state-of-the-art SER models on multiple datasets.

II. Related Works

A. Software Entity Recognition

Recognizing software entities from text documents has been a long-standing research problem in Software Engineering [1], [2]. Early approaches rely on keyword or rule-based pattern matching to identify software entities, and they mainly focus on identifying API names [1], [3]–[6], [17], [32]. For example, Bacchelli et al. design lightweight regular expressions based on common naming conventions to identify class and function names in email discussions [3]. Rigby et al. encode regular expressions into an island parser to recognize classes, methods, and fields mentioned in Stack Overflow posts [17].

More recently, machine learning, especially deep learning, has been increasingly adopted for software entity recognition [20], [21], [23], [24], [31], [33]. These approaches also recognize a richer set of software entities beyond API names. For example, Ye et al. proposed a Conditional Random Field (CRF) model to identify five types of software entities in Stack Overflow posts, including *programming languages*, *platforms*, *APIs*, *software libraries and frameworks*, *software standards* [21]. They further integrated word embeddings with the CRF model to address the challenges of polysemy and naming variations [31]. Zhou et al. proposed a similar word embedding-based CRF model but focused on identifying software entities in bug reports [33]. More recently, they proposed a BiLSTM-CRF model for software entity recognition [24]. Chen et al. proposed to identify morphological relations between software entities by analyzing and comparing the word embeddings learned from software-related documents and general text documents [34]. Tabassum et al. finetuned BERT with Stack Overflow posts and proposed a BERT-CRF model called SoftNER to detect software entities in Stack Overflow posts [23]. Huo et al. proposed to combine BiLSTM-CRF with a context-aware scoring mechanism to identify API mentions in free-form text [20].

Recently, Chew et al. [35] conducted a comparative evaluation on S-NER [21], Stanford-NER [36], BERT [28], and BERTOverflow [23]. They found that S-NER achieved the best performance while BERTOverflow achieved the worst performance. However, the best model only achieved 78.18% F1-score. Our work advances the state-of-the-art by addressing the data noise challenge in SER. To achieve this, we present a large NER dataset with fewer noisy labels and a noise-robust learning framework to account for data noises coming from annotation errors and ambiguous software entity names. Our evaluation shows that NER models trained with our new dataset and noise-robust learning framework achieved the best performance among six NER baselines.

B. Named Entity Recognition

Our work is also closely related to the general-domain Named Entity Recognition (NER) task in natural language processing (NLP) [37]. NER models aim to identify named entities in the general text domain, such as persons, organizations, and locations. Recently, deep learning models

have gained dominance in obtaining state-of-the-art results. These models capture complex interactions between words and their contexts by learning from large text corpora labeled with named entities. Huang et al. proposed a BiLSTM-CRF model to encode the contextual information in a sentence for NER [27]. Following this work, many BiLSTM-CRF-based models have been proposed [38]–[47].

Recently, Transformer-based language models [48] have become a new standard for developing NER models. Researchers have experimented with a range of pretrained language models for NER, such as BERT [28], RoBERTa [49], LUKE [50] and even autoregressive models such as GPT [51]. Typically, these language models are first pretrained on a large unlabeled text corpus through self-supervised learning and then finetuned for a specific task.

It remains challenging to reuse NER models trained on general text corpora to highly specialized domains such as biology and medicine, as shown by previous studies [52]–[55]. Several known challenges present, including domain-specific naming standards, common word polysemy [31], [56], and naming variations [35]. Thus, developers have spent great effort to develop domain-specific NER models, such as BioBERT [57] for biomedical literature, ClinicalBERT [58] for clinical documents, or SciBERT [59] for scientific literature. Our work focuses on doing NER for software documents.

C. Noise-robust Learning

Training data is often noisy and contains various types of errors, which can degrade the performance of ML models. Noise-robust learning has been widely studied in computer vision [60]–[67]. Recently, several approaches have investigated noise-robust learning in NLP tasks [30], [68]–[71]. Wang et al. [68] proposed CrossWeigh, a method that partitions the training data into several folds and trains independent NLP models to identify potential noisy labels. However, this approach requires training multiple models on different data folds and is thus computationally expensive and only supports fold-level noise estimation. Xiao et al. [69] proposed a Bayesian Neural Network (BNN) method that quantifies model and data uncertainties for NER and sentiment analysis tasks. Wang et al. [70] proposed NetAb, presupposing that noise can be simulated by flipping clean labels randomly. However, this presupposition is overturned by Cheng et al. [71], indicating that different datasets have different noise rates. Zhou and Chen [30] proposed a co-regularization framework that consists of two or more neural networks with the same structure but different initializations, which is particularly effective at reducing the impact of noise in the training data and improving the accuracy of the NER model. Inspired by this approach, we propose *Self-regularization*, a noisy-robust learning approach for NER in the SE domain. Self-regularization outperforms co-regularization in our evaluation while requiring training of a single model instead of simultaneously many models, making it more computationally efficient.

III. Problem Formulation

This section defines the research problem of recognizing software entities from text documents.

Definition 1. (Software Entity): Software entities are nouns and noun phrases that describe specific objects, concepts, and procedures related to software engineering, such as an algorithm name and a data structure name. To effectively perform software entity recognition, it is crucial to establish a well-defined and easily interpretable inventory of entity types. For software entities, our primary objective is to construct a domain-specific inventory of entity types that comprehensively cover various aspects of software engineering knowledge. Therefore, we design our inventory of entity types to cover software engineering concepts exclusively. In future work, one can extend our dataset with more software-related entities, such as software engineering conference names and computer scientist names, based on the downstream applications.

With this domain focus, three of the authors conduct an iterative process involving a focus group over three 2-hour sessions. Each author independently annotated 50 samples of software entities from our corpus. Then, they compared notes and reconciled differences through discussions. After three iterations of sampling, annotation, and consensus building, they ultimately reached an agreement to center our attention on the following 12 fine-grained software entity types that cover key software entities while balancing specificity versus coverage. These 12 types are *Algorithm*, *Application*, *Architecture*, *Data structure*, *Device*, *Error name*, *General concept*, *Language*, *Library*, *License*, *Operating system*, and *Protocol*. We provide a definition and examples for each software entity type below.

- **Algorithm.** This type includes computational procedures, algorithms, and paradigms that take inputs and perform defined operations to produce outputs, e.g., Bubble Sort, Auction Algorithm, and Collaborative Filtering.
- **Application.** This type includes computer software and programs designed to perform specific user-oriented tasks, e.g., Adobe Acrobat, Microsoft Excel, and Zotero.
- **Architecture.** This type includes computer architectures and other related computer system designs, e.g., IBM POWER architecture, Skylake (microarchitecture), and Front-side Bus.
- **Data structure.** This type includes standardized ways of organizing and accessing data in computer programs, e.g., Array, Hash table, and mXOR linked list.
- **Device.** This type includes physical computing components designed for specific functions, e.g., Samsung Gear S2, iPad, and Intel T5300.
- **Error name.** This type includes program errors, exceptions, and anomalous behaviors in computer software, e.g., Buffer Overflow, Memory Leak, and Year 2000 Problem.
- **General concept.** This type includes a broad range of programming strategies, paradigms, concepts, and design principles, e.g., Memory Management, Adversarial Machine Learning, and Virtualization.

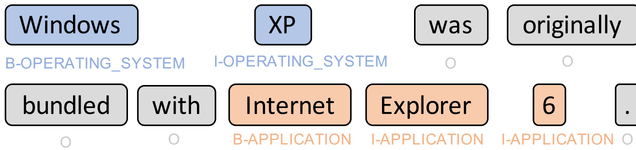

Fig. 1: Software entity tagging with the IOB scheme

- **Language.** This type includes programming languages and domain-specific languages designed to communicate instructions to computers, e.g., C++, Java, Python, and Rust.
- **Library.** This type includes software libraries, packages, frameworks, and other types of APIs, e.g., Beautiful Soup, FFmpeg, and FastAPI.
- **License.** This type includes legal terms governing the usage and distribution of software, e.g., Cryptix General License, GNU General Public License, and MIT License.
- **Operating system.** This type includes system software responsible for managing computer hardware and software resources and providing services for computer programs, e.g., Linux, Ubuntu, Red Hat OS, and MorphOS.
- **Protocol.** This type includes rules and standards that define communication between electronic devices, e.g., TLS, FTPS, and HTTP.

Definition 2. (Software Entity Recognition): We formulate the task of software entity recognition as a token-level classification problem. Given *T*, a free-form text in the context of software engineering, the software entity recognition task is to identify every span of words $s =< w_1 w_2 \cdots w_n >$ that refers to a software entity from *T* and classify each *s* into one of the 12 entity types we defined.

In our problem setting, we consider the IOB [72] scheme for entity labeling. IOB is a commonly used tagging format for annotating tokens in NER. It provides a simple way to identify entity boundaries. In the IOB scheme, each token in a sequence is labeled as either *B* (i.e., beginning of an entity), *I* (i.e., in the middle of an entity), or *O* (i.e., not an entity). Figure 1 illustrates the IOB labeling scheme with an example sentence from Wikipedia. In this sentence, “Windows XP” are labeled as *Operating System* and “Internet Explorer 6” are labeled as *Application*. Since both of them contain multiple words, the first words in them are labeled with *B-OPERATION_SYSTEM* and *B-APPLICATION* respectively, while the remaining words are labeled as *I-OPERATION_SYSTEM* and *I-APPLICATION*. In the software entity recognition task, an entity is considered correctly recognized *only if* the labels of all words in the entity span are correctly predicted.

IV. Dataset Construction

In this section, we outline the process of identifying and categorizing software entities in Wikipedia corpora. Figure 2 illustrates the data construction pipeline. The resulting dataset, which we refer to as WikiSER, comprises 1.7M sentences labeled with 79K unique software entities (3.4M labels in total).

A. Pruning Wikipedia Taxonomy

Since articles on Wikipedia cover a variety of domains, we need to first prune it to retain ones only related to software engineering. Wikipedia provides a hierarchical classification of its articles based on the domain and topic. In the hierarchy, more general categories appear closer to the root, while more specific categories appear at the bottom. To collect SE categories, we start from `Category:Computing`, which is the most general category related to SE. We use the MediaWikiAPI to recursively find all descending subcategories of `Category: Computing` in the taxonomy (a total of 2M categories).

Note that not all descendants of `Category: Computing` fall into the 12 SE types we focus on, since our category taxonomy focuses specifically on software entities rather than peripheral topics. For example, `Category: Computer specialists` is a descendant of `Category: Computing` but contains experts and researchers in computer science and adjacent fields, which are not our main focus. However, entities labeled as `Category: Computer specialists` during Wikipedia crawling could be retained in a separate type for future use. Our taxonomical design and ML architecture allow seamless integration of new entity categories if required. To prune the category taxonomy, two of the authors examine the sub-categories of `Category: Computing` in a top-down manner and create a list of 924 categories that are irrelevant to the 12 software entity types defined in Section III.We remove all categories that are descendants of categories in this list. Due to Wikipedia’s tree structure, removing a parent category translates to a removal of an entire branch. After pruning, we retain 8,469 categories.

We manually filter the remaining 8,469 categories to ensure their correctness. To facilitate a consensus on this manual filtering, the two authors had a 1-hour discussion regarding the filtering criteria. They also practiced on 50 categories together to reinforce their understanding of the selection criteria. The complete annotation process is divided into three steps. First, each author filters half of the 8,469 categories. They discuss with each other when they are uncertain about a category during the filtering process. Then, they cross-validate their filtering results and discuss the categories where they disagree. The agreement level between the two authors is 0.86 in terms of Cohen’s Kappa [73], indicating a substantial agreement level. Third, they work in pairs to resolve all categories that are flagged as questionable. This manual process took about 62 person-hours. A total of 945 categories are deemed unrelated to the 12 software entities by both authors and 7,524 categories remain after the manual filtering.

B. Collecting Software Entities

On Wikipedia, each article is labeled with one or more Wikipedia categories by default. We leverage this categorization to find Wikipedia articles that may describe software

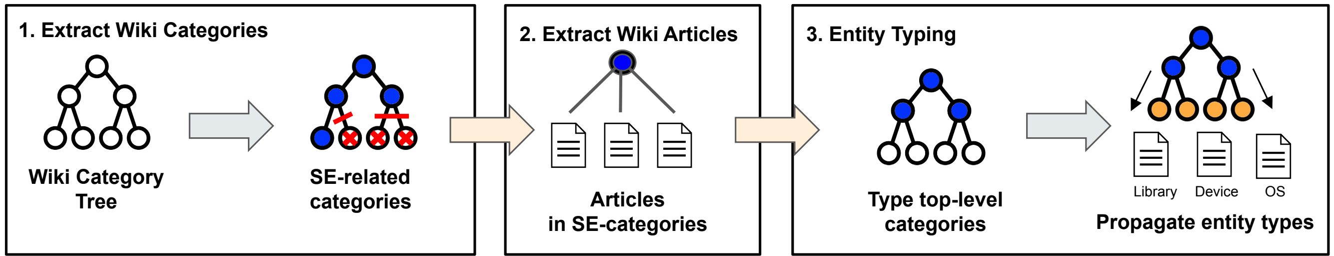

Fig. 2: WIKISER Construction Pipeline

TABLE I: Heuristics for Identifying Software Entities in Wikipedia taxonomy

Heuristics	Article #	Precision
Contains 1 SE category	139,752	79.3%
Contains 2 or more SE categories	79,899	**92.8%**
20% of labeled categories are SE	105,793	83.5%
50% of labeled categories are SE	38,043	88.2%
60% of labeled categories are SE	15,528	90.6%

entities. Specifically, we write a script to automatically extract Wiki articles labeled with at least one of the 7,524 SE categories identified from the previous step. 139,752 Wiki articles are extracted after this step.

However, we notice that this corpus is noisy. While many articles are labeled with a SE category, they do not actually describe a specific software entity. For example, "Software studies" is a Wiki article under the `Software` category, but it does not refer to a specific software entity. After manually examining 30 articles, we find that those articles are labeled with not only a SE category but also some categories not closely related to SE. For example, the "Software studies" article is labeled with `Computing culture`, `Cultural studies`, `Digital humanities`, in addition to `Software`.

Based on this insight, we experiment with several heuristics that filter Wikipedia articles by the number or percentage of SE categories among all the labeled categories. Table I describes each heuristic. We measure the precision of each heuristic. Here, precision refers to how many articles classified as SE-related are actually related to SE. The two authors sampled 385 articles from all the extracted Wiki articles. This sample size is considered statistically significant with a 95% confidence level and a margin of error of 5%. We then manually label whether the articles are SE-related and perform cross-validation. Given the results as the ground truth, we compare them with those obtained by filtering based on various heuristics and compute the precision. Consequently, we find that the heuristic of selecting articles labeled with two or more SE categories achieved the highest precision, 92.8%, among all heuristics. The filtered number of articles is 79,899, which has not decreased significantly compared to the number number of articles before filtering.

C. Labeling Software Entity Spans

As explained in Section III, we need to identify the span of each software entity mentioned in a Wiki article. We leverage the hyperlinks in a Wiki article, as well as keyword matching, to identify the mentions of software entities in a Wiki article. Specifically, we treat the title of each of the 79,899 articles found in the previous step as a software entity. If a word or a phrase in a sentence of a Wiki article is hyperlinked to a Wiki article in the 79,899 articles, we consider that it mentions a software entity.

We observe that not all mentions of a software entity are hyperlinked in a Wiki article. Specifically, many Wiki articles only hyperlink the first mention of an entity to its corresponding article. Based on this observation, we further develop a keyword-matching method to identify the mentions of software entities.

A major challenge in this step is that the same entity can be expressed in different forms. For example, "Long Short-Term Memory" is often written as "LSTM". To address this challenge, we leverage the page redirection mechanism in Wikipedia to recognize aliases, which is commonly adopted in prior work [74]–[77]. In Wikipedia, accessing an article can sometimes automatically send visitors to another article with the same concept but a different name, which is called a redirect. For example, "LSTM" is a redirect of "Long Short-Term Memory". Hence, when users want to visit the Wikipedia article of "LSTM", it will automatically jump to the article of "Long Short-Term Memory". Since they both point to the same article, we can safely assume that "LSTM" is an alias for "Long Short-Term Memory". Using this mechanism, we get aliases for all software entities in our dataset. Given a Wikipedia article, we first perform lemmatization to handle words in different forms and then identify the mentions of a software entity or its alias via exact keyword matching.

In total, we obtain 3.4M sentences from 79,899 articles. Many of these sentences do not mention any software name entities and provide little value for model training and evaluation. Thus, we remove over 1.7M of these instances along with duplicated sentences. Finally, the dataset results in 1,663,431 sentences that mention at least one software entity. Figure 3 shows the distribution of the number of name entities in the sentences in WIKISER.

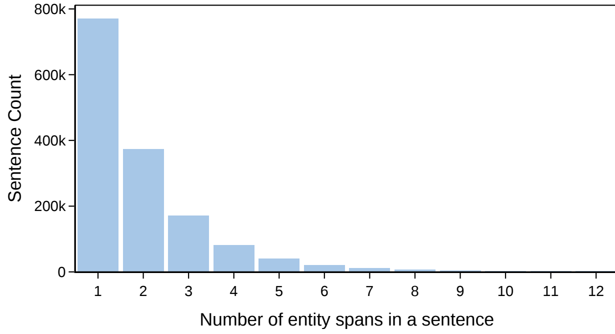

Fig. 3: Distribution of Entity Spans Per Sentence

D. Labeling Entity Types

As the final labeling step, we need to further assign each entity span to the corresponding entity type as defined in Section III. Note that under the Wikipedia taxonomy, an article belongs to one or more categories. Thus, an intuitive idea is to leverage the categories to infer the type of the entity an article refers to. Based on this idea, we first establish a mapping between the 7,524 categories from Section IV-A and the 12 software entity types defined in Section III. Then, we infer the entity types of an article based on the types of categories they belong to. We elaborate on these two steps below.

Map Wiki categories to entity types. To do this, the second and the third authors first collectively labeled the SE categories up to the 5th level in the Wikipedia taxonomy, resulting in a total of 1,160 categories. Similar to the manual labeling process in Section IV-A, they first discuss the categorization criteria for 1 hour and practice together on 50 categories to enhance consensus on categorization. Then, each of them is assigned half of the categories and is asked to report any disagreement they are uncertain about. The Cohen's Kappa [73] score is 0.82, indicating substantial agreement. They further discuss and resolve any disagreement. In this way, we manually establish a mapping between the 1,160 categories up to the 5th level to the 12 software entity categories. Then, we developed an automated script that performs a breadth-first traversal of the category hierarchy, starting from categories of level 6, and automatically assigns an entity type to a descending category based on the type of its parent category. Ultimately, we establish a mapping between all 7,524 categories to the 12 entity types.

Inferring Entity Types. Having obtained the inferred types for all SE categories, our goal is to classify the Wikipedia article of each entity based on its categories. However, as explained in Section IV-B, each article can have multiple categories, which may belong to different entity types. For example, consider the software entity "ChromeOS", which should be categorized as an *Operating System*. However, while it has categories that belong to the *Operating System* type, such as `Category: ARM operating systems` and `Category: Google operating systems`, it also has a category that belongs to the *Application* type (i.e., `Category: Google Chrome`).

```
{First sentence from the Wikipedia article of P}.
In Software Engineering, {P} belongs to the category of {TYPE}.
```

Fig. 4: The Prompt Input to Flan-T5 to Infer Entity Type (Content in curly brackets are placeholders)

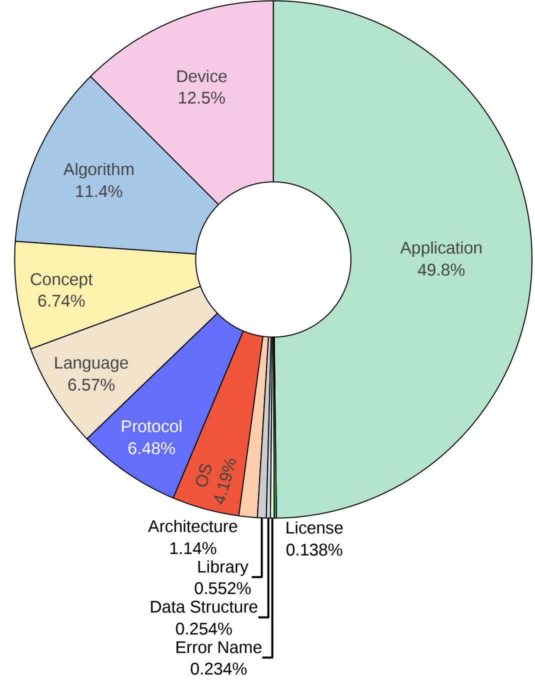

Fig. 5: Distribution of Software Entities by Type

To address this issue, we design three heuristics to decide the final entity type of an article with multiple categories. First, we simply assign the entity type of the most fine-grained category of the article, which is measured by the distance to the root `Category: Computing`. Second, if all categories of the article are at the same granularity in the hierarchy, we assign the entity type that the majority of categories belong to. Third, if there is still a tie, we infer the entity type of the article by prompting a large language model. Specifically, for a Wikipedia article P, we prompt Flan-T5 XL [78] with the format shown in Figure 4. Specifically, we prepend the prompt with the first sentence extracted from P. We substitute the second mask token with each candidate type and use Flan-T5 XL to calculate the perplexity of each completed prompt. Perplexity measures the degree of uncertainty of the language model when generating a new token. The candidate with the lowest perplexity is selected, indicating the highest confidence from the language model. These three heuristics apply to 39%, 56%, and 5% of the 79K software entities in WIKISER respectively.

Figure 5 shows the distribution of different types of software entities in our dataset. *Application* is the most frequently occurring entity type on Wikipedia, comprising 41% of all

software entities present in our dataset. The following are *Device*, *Algorithm*, *General Concept*, *Language* and *Protocol*, respectively accounting for 12.5%, 11.4%, 6.74%, 6.57%, and 6.48% of all entities.

E. Manual Validation

To evaluate the labeling accuracy of our method, we manually validated a random sample from the 1,663,431 sentences identified in Section IV-B. While selecting a sample that is too large is expensive and time-consuming, selecting a sample that is too small can lead to inaccurate conclusions. We use two sampling statistics—confidence level and margin of error—to decide on the proper sample size. We chose a 95% confidence level and a 5% margin of error, which are commonly used in empirical software engineering research [79]. As a result, we randomly sampled 387 sentences. From these sampled sentences, our method automatically generated 807 entity labels in the IOB format.

To validate the correctness of these labels, three of the authors held a 1-hour discussion to establish the span detection and entity categorization criteria and practiced on 20 sentences collaboratively to ensure their mutual agreement on the criteria. Then, each of them manually labels one-third of the 387 sentences. For the tokens that they are uncertain about, they mark them as "uncertain" and discuss them later. Then, through cross-validation and discussion, they eventually reach a consensus and complete the manual annotation of all sentences. In cases where sentences are marked as "uncertain", the three authors conduct a joint review and discussion to arrive at a final determination for their annotation.

By comparing the manual annotations and the auto-generated labels, we found that 74 auto-generated labels (9.17%) are incorrect. Though this error rate is higher than the 5.38% error rate of CoNLL [80], the most widely used NER in the general text domain, this is reasonable given WIKISER's fine-grained nature. CoNLL only contains 4 general entity types, while WIKISER includes 12 granular, domain-specific software entity types. The increased specificity makes entity disambiguation more challenging. Furthermore, software entities have high name overlap and aliasing, presenting additional difficulty. Considering WIKISER's more complex fine-grained distinctions, the labeling quality achieved is acceptable, especially given no existing fine-grained software NER datasets to compare against.

Likewise, we manually validate two state-of-the-art SER datasets, S-NER [21] and SoftNER [23], and compute their labeling error rates. For each dataset, we randomly sample 387 sentences, which ensures a 95% confidence level and a 5% margin of error, and follow the same procedure to manually label them. Our analysis reveals that the S-NER and SoftNER have 13.93% and 17.79% error rates, respectively, as shown in Table II. Thus, compared with S-NER and SoftNER, our new WIKISER dataset not only has the lowest error rate of 9.17% but also includes a comprehensive set of software entities and the most labeled sentences (1.7M), compared to 1,015 in S-NER and 7,438 in SoftNER. Though SoftNER has more entity types than WIKISER, 8 of the 20 types in SoftNER are code-related entity types, e.g., *class*, *variable*, *inline-code*, *function*, etc. Code-related entities are easier to detect compared with other types of software entities, such as library names and protocols, which have more aliases and naming ambiguity. There is also a large body of literature on recognizing code-related entities [1], [3]–[6], [17], [20], [32]. State-of-the-art techniques such as ARCLIN [20] have achieved high accuracy in detecting code-related entities. Thus, code-related entities are not of interest in this work.

TABLE II: Comparison between WIKISER and Two Existing SER Datasets from Stack Overflow

	Ye et al. [21]	Tabassum et al. [23]	WIKISER
Entity types	5	**20**	12
Sentences	4,646	6,510	**1.7M**
Unique entities	1,015	7,438	**79,899**
Labeling errors	13.93%	17.79%	**9.17%**

V. NOISE-ROBUST LEARNING

Although WIKISER has a lower labeling error rate compared with other benchmarks, it is not free of labeling errors. Thus, we propose a noise-robust learning framework to account for such labeling errors during model training. The key insight is that, compared to clean-label settings, noisy-label settings can benefit from a "delayed" learning curve [30]. This is due to the fact that neural models tend to learn quickly from clean instances that are more compatible with the task's inductive bias at the early stages of training. While doing so, they can become overly confident and less likely to learn from noisier instances that might diverge from the task's inductive bias in later epochs [81]. We can solve this problem by adding a loss term that discourages premature convergence and prevents the model from overfitting uncertain, noisy labels.

Based on this insight, we propose *self-regularization*, a noisy learning approach that leverages the dropout function to reduce overfitting. In deep learning, dropout [67] has long been used to improve generalization in neural networks by randomly dropping parts of a network layer. Different versions of the same model induced by the dropout might make different prediction distributions, especially in the presence of noisy instances. We can control this randomness by regularizing the model over its prediction divergence. In R-DROP, Liang et al. [82] was the first to use dropout as the main mechanism for regularization in noisy-label settings. Self-regularization differs from R-DROP in that it relaxes the number of forward passes through the model to be *multiple* instead of just two.

Recall that our approach adds a loss term of self-regularization to the model learning objective. This function accounts for the prediction inconsistency of the model's forward passes in training. We explain more details of our training framework below.

Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ be a dataset with pairs of an input sequence x_i and a label sequence y_i. In the NER setting, an instance in (x_i, y_i) could be considered mislabeled if a token

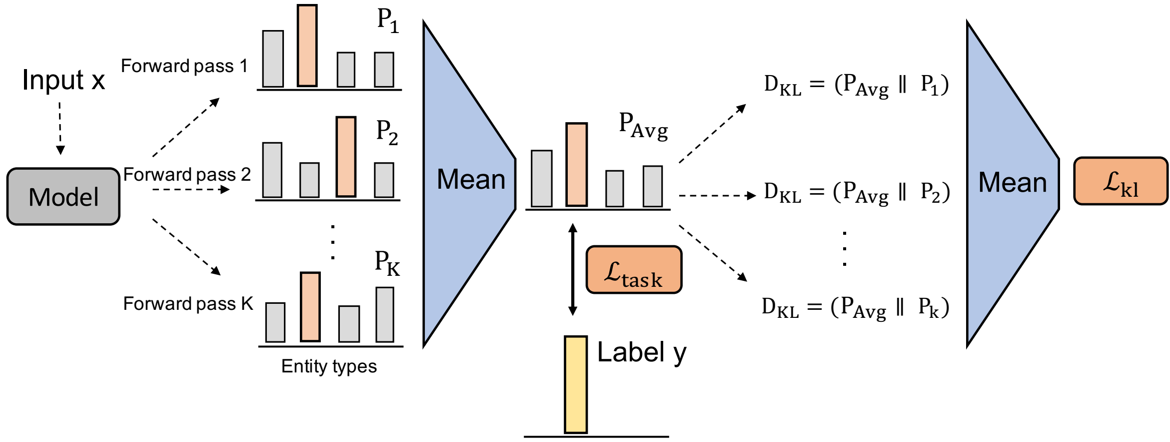

Fig. 6: Overview of Self-regularization Framework

Algorithm 1 Training a SER model with self-regularization

Input: Train set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, agreement loss weight α (hyperparameter), number of forward passes K (hyperparameter).
Output: Trained model M.

1: Initialize model M from a pretrained language model
2: Warm-up M for 10% of training steps on $\mathcal{L}_{task}$
3: **while** M not converged **do**
4: Randomly sample a batch $\mathcal{B} \in \mathcal{D}$
5: Forward pass K times through M to obtain probability distributions over label space $\mathcal{P} = \{P_j\}_{j=1}^{K}$
6: Compute task loss $\mathcal{L}_{task}$ by Equation 2
7: Compute KL-divergence loss $\mathcal{L}_{kl}$ by Equation 1
8: Compute $\mathcal{L}_{agree}$ by Equation 3
9: Update parameters in M by minimizing $\mathcal{L}_{agree}$
10: **end while**

in x_i is wrongly typed (ie. iPhone as *Algorithm* instead of *Device*), or incorrectly assigned a non-named entity ("O"). The goal is to learn a noise-robust model M that tolerates the inevitable training noise.

We initialize M from a pretrained language model such as BERT$_{base}$, where dropout is by default incorporated. At each training step, we sample a batch $\mathcal{B} = \{(x_i, y_i)\}_{i=1}^{|\mathcal{B}|}$ in $\mathcal{D}$ for inference on M. After each random dropout, M becomes a new submodel with a fraction of the original units in the network. The same instance input x_i can outputs different results when passing through M. The left-most block in Figure 6 illustrates this process.

Training of the NER model with self-regularization optimizes two objectives. The first objective is the divergence loss $\mathcal{L}_{kl}$. Given the dropout randomness, we obtain a set of $\mathcal{P} = \{P_j\}_{j=1}^{K}$ distributions over the label space when inputting an instance to M over K forward passes. In noisy settings, $\mathcal{P}$ is likely to have high variance. We can control for such variance by taking the bidirectional Kullback-Leibler (KL) divergence between the average target probability distribution of $\mathcal{P}$ and each P_j:

$$\mathcal{L}_{kl} = \frac{1}{K}\sum_{j=1}^{K} D_{KL}(P_j || \frac{1}{K}\sum_{j=1}^{K} P_j) \tag{1}$$

where $\frac{1}{K}\sum_{j=1}^{K} P_j$ is the average of K probability distributions obtained from the Softmax, denoted as P_{Avg} in Figure 6.

The second objective optimizes the cross-entropy task loss $\mathcal{L}_T$ for x_i for doing NER label classification:

$$\mathcal{L}_{task} = -\frac{1}{K}\sum_{j=1}^{K}\sum_{l=1}^{|x_i|} y_{i,l} \log P_{j,l} \tag{2}$$

where $y_{i,l}$ is the true label of the l-th token in input x_i and $P_{j,l}$ is the probability distribution over the label space for the l-th token obtained from the j-th forward pass. We achieve the standard task loss by averaging the cross-entropy loss over all forward passes.

Finally, the combined agreement loss accounts for both Equation 2 and Equation 1 as the single learning objective to optimize M:

$$\mathcal{L}_{agree} = \mathcal{L}_{task} + \alpha \times \mathcal{L}_{kl} \tag{3}$$

where α is a positive multiplier used to weight the agreement loss. $\mathcal{L}_{agree}$ is high when the feels uncertain about its prediction and gets smaller when the output probability distribution is more consistent. Algorithm 1 describes the full pipeline of self-regularization.

Self-regularization has a few benefits over previous methods. First, compared to existing noisy-label learning methods such as co-regularization and CrossWeigh, [30], [68], self-regularization only requires training a single model instead of multiple. This translates to less training time and memory overhead. Second, compared to previous NER models in the software domain, our approach demands neither the use of an external gazette [21], nor auxiliary models [23].Last, the method is versatile in that it can synergize well with any pretrained or randomly initialized models.

VI. EXPERIMENTS

We design multiple experiments to evaluate our dataset and noisy label learning method. We aim to answer the following research questions:

- RQ1: How well does self-regularization work as a denoising measure on WIKISER?
- RQ2: How does self-regularization generalize to Stack Overflow benchmarks?
- RQ3: What entity types are most difficult to learn?
- RQ4: How does the number of forward passes impact self-regularization?
- RQ5: How efficient is self-regularization compared to co-regularization?

A. Experimental Setup

Dataset. Given the massive size of the WIKISER dataset, we create a subset of it - WIKISER$_{small}$ - to train and test SER models. This set strives for a uniform distribution of each entity type. The resulting WIKISER$_{small}$ consists of 50K sentences for *training*, 8K for *validation*, and 8k for *test*.

Baselines. We describe the following baselines to compare with our noisy label learning method.

1) **SoftNER.** SoftNER is the state-of-the-art SER model proposed in [23]. It uses an architecture that combines three embedding attention layers: BERTOverflow, an auxiliary code classifier, and an auxiliary segmentation model to identify name spans. The segmentation model uses extra information such as HTML tags in a SO post, which does not apply to our Wikipedia data. We maintain the code classifier and segmentation model as given, but instead finetuning the entire model on WIKISER$_{small}$.
2) **Co-regularization.** Zhou and Chen [30] propose a co-regularization framework that reduces overfitting by regularizing the output divergence of many models, outperforming many methods in information extraction for the general domain. We finetune BERT$_{base}$ with their co-regularization denoising objective as a comparison baseline for self-regularization.
3) **BERT$_{base}$.** We finetune pretrained BERT$_{base}$ cased version on our WIKISER$_{small}$. BERT$_{base}$ commonly serves as the standard baseline for many downstream language tasks in the general domain [30], [83].
4) **RoBERTa$_{base}$.** RoBERTa [49] is another Transformer-based language model that is pretrained from large-scale text corpora. It improves over BERT on many benchmarks by having more diverse training data and modified architecture. We finetune pretrained RoBERTa on WIKISER$_{small}$.
5) **BERTOverflow.** Initialized from BERT$_{base}$, BERTOverflow is trained on an additional 152M sentences from Stack Overflow [23]. In contrast to general-purpose pretrained models, BERTOverflow serves is in-domain for software engineering. We finetune its checkpoint from [23] on WIKISER$_{small}$.
6) **Larger model.** BERT$_{large}$ [28] is a bigger variant of BERT$_{base}$ with 340M parameters, whereas the base model has 110M parameters. As the models discussed thus far share the same size, we include a finetuned BERT$_{large}$ baseline to demonstrate the performance of a bigger model on WIKISER and the gains from self-regularization.

TABLE III: Evaluation Results on WIKISER

	P	R	F1
SoftNER [23]	64.4	69.1	66.6
BERTOverflow [23]	66.4	68.5	67.4
RoBERTa$_{base}$	68.2	71.0	69.6
BERT$_{base}$	68.1	**73.1**	70.5
BERT$_{base}$ + Co-reg. [30]	72.7	69.1	70.8
BERT$_{base}$ + Self-reg.	**74.9**	72.0	**73.7**
BERT$_{large}$	69.8	**74.5**	72.1
BERT$_{large}$ + Self-reg.	**73.3**	**74.5**	**73.9**

Training details. All models use Adam optimizer, learning rate $1e-5$, batch size 16, dropout rate of 10%, and are trained on the NVIDIA RTX A6000 for 30 epochs. For self-regularization, we choose a warm-up rate of 10% and $\alpha = 10$.[3]

B. RQ1: SER Accuracy on WIKISER

Table III shows the results of models trained by our self-regularization framework in comparison to baseline models. Overall, models trained with self-regularization outperform all baselines. This includes BERT$_{base}$ trained with co-regularization by 2.9%. The efficacy of both denoising method suggests that we are able to reduce overfitting while learning on WIKISER$_{small}$, which comes with a certain level of noise. We highlight that BERT$_{base}$ model trained with our self-regularization framework outperforms SoftNER, the SOTA SER model [23] by 7.1% in F1 score. SoftNER also performs worst than its pretrained model in BERTOverflow. A plausible reason is that, despite an adaptation to WIKISER$_{small}$ via finetuning, SoftNER's auxiliary models might have provided irrelevant signals on clean texts from Wikipedia.

We provide more in-depth analyses of the results.

Impact of pretrained models. Many NLP studies point to the importance of having pretraining data more closely aligned to the distribution of the downstream task, allowing the model to adapt more easily to the target domain. For SE, BERTOverflow is a pretrained language model finetuned from BERT$_{base}$ on 152M Stack Overflow posts. Surprisingly, Table III shows that BERTOverflow performs the worst, while BERT$_{base}$ performs the best. Notably, BERT$_{base}$ is trained on many general-domain corpora [28] that also includes Wikipedia, the same source in which we construct WIKISER. We suspect that BERT$_{based}$'s good performance can be attributed to the fact that its underlying distribution is closer to WIKISER than that of BERTOverflow. However, BERTOverflow is still effective

[3]We tune α over $\{10, 30, 50\}$ on WIKISER$_{small}$ and find $\alpha = 10$ to work best. Thus, we use $\alpha = 10$ for all models where self-regularization and co-regularization apply.

TABLE IV: Evaluation Results on two Stack Overflow datasets

	SoftNER-9 [23]			S-NER [21]		
	P	R	F1	P	R	F1
BERT$_{base}$	64.7	64.2	64.4	77.0	80.9	78.9
BERT$_{base}$+Self-reg.	65.2	62.4	64.8	81.8	81.1	81.4
SoftNER	74.6	72.9	73.7	81.3	**84.6**	82.9
BERTOverflow	65.2	73.1	74.0	84.3	83.9	84.1
BERTOverflow+Self-reg.	**75.8**	**76.5**	**76.1**	**86.0**	84.3	**85.2**

as a base model for Stack Overflow data, as our experiments demonstrate in Section VI-C.

Impact of model size. Compared to BERT$_{base}$ (110M parameters) and RoBERTa$_{base}$ (125M), BERT$_{large}$ (340M) trained with self-regularization sees an improvement over its vanilla counterpart. This suggests that gains are possible when the model size increases, though self-regularization shows more effectiveness for smaller models. Specifically, in comparison with gains from BERT$_{large}$, BERT$_{base}$ with self-regularization sees an improvement of 3.2% (instead of 1.8%) in F1.

C. RQ2: SER Accuracy on Stack Overflow

Since Wikipedia imposes strict guidelines and standards for editing and verifying written content, WIKISER tends to not suffer from data noise in the form of spelling mistakes, naming conventions, and others [21]. In this section, we investigate how our approach generalizes to more noisy benchmarks such as SoftNER [23] and S-NER [21].

Datasets. We use two different Stack Overflow corpora annotated by SoftNER and S-NER. As explained at the end of Section IV, SoftNER has 8 code-related entity types, which is not of interest in this work. Thus, we do not consider them in this experiment. For the remaining 12 entity types, we map them to our entity types for consistency. Please see Supplementary Materials for the full mapping. This results in 9 final entity types, since 3 types are very fine-grained and are merged with other types, e.g., *website* merging into *application*. For S-NER, which has just 5 entity types (*API*, *Language*, *Platform*, *Framework*, and *Software Standard*), we keep the dataset as it comes, and randomly sample 15% of its data as the test set. We finetune all baseline models separately on these two datasets.

Results. Table IV shows that the positive gains from self-regularization also apply to SoftNER and S-NER. BERTOverflow trained with self-regularization performs the best on both datasets. Specifically, it outperforms the self-regularized BERT$_{base}$ model by 11.3% on SoftNER-9 and 3.8% on S-NER in F1. This result makes sense since BERTOverflow is fine-tuned on 152M SO posts and its distribution is more aligned with SoftNER and S-NER than BERT$_{base}$. This implies that in-domain pretraining is still helpful. Additionally, Table IV also suggests that SoftNER performs worse than vanilla BERTOverflow. Since SoftNER adopts auxiliary models to recognize code entities, this implies that these auxiliary models are not as helpful in the absence of code-related entities.

TABLE V: Results by entity type. Second column shows the number of entity label occurrences in the test set of WIKISER$_{small}$

	# Spans	P	R	F1
General Concept	2,456	67.0	62.2	64.5
Algorithm	2,018	67.7	66.2	66.9
Application	6,861	67.9	69.7	68.7
Device	3,299	73.6	69.3	71.4
Language	2,525	73.2	74.4	73.8
Protocol	2,629	74.5	73.5	74.0
Architecture	1,538	78.0	73.8	75.8
Operating System	2,765	80.1	78.6	79.3
Library	991	81.2	84.8	82.9
Data Structure	1,051	83.1	87.4	85.2
Error Name	1,088	86.0	90.8	88.3
License	1,140	86.6	90.9	88.7
Micro Avg.	-	73.8	73.5	73.7
Macro Avg.	-	76.6	76.8	76.6

TABLE VI: Impact of the number of forward passes

# Forward Passes (K)	2	3	4
BERT$_{base}$ + Self reg.	70.7	73.7	73.8
BERT$_{large}$ + Self reg.	73.9	74.0	73.8

D. RQ3: SER Accuracy by Entity Type

Table V shows the results of BERT$_{base}$ trained with self-regularization across all entity types. Overall, the model performs fairly well on *License*, *Error Name*, *Data Structure*, *Library*, and *Operating Systems*. Other entity types, such as *Application*, *Algorithm*, and *General Concept*, appear to be more challenging than others. One possible reason is that entities in those types share more common words with non-SE words, making them more ambiguous to recognize. In contrast, *License* and *Error Name* tend to be more unique and standardized, making them easier to detect.

E. RQ4: Choice of K for Self-regularization

How does increasing the number of forward passes in self-regularization affect model performance? Results from Table VI show a major improvement for BERT$_{base}$ when the model regularizes over $K = 3$ outputs instead of $K = 2$. At $K = 4$, BERT$_{base}$ sees a marginal improvement of 0.1%. For BERT$_{large}$, model performance does not vary greatly with different values of K (within 0.2%). Here, we note a potential tradeoff between performance and computational resources. The choice of forward passes can vary by the task and model architecture, and a high K does not necessarily lead to better results compared to smaller values of K.

F. RQ5: Training Time and GPU Memory Usage

Table VII shows the training time and GPU memory usage of models trained with self-regularization compared to co-regularization [30] and various model sizes averaged over all epochs. Results show that self-regularization incurs minimal memory overhead, whereas co-regularization requires 2X the GPU memory when trained with two models for regularization

TABLE VII: Training Time and GPU Memory Usage

	Wall-clock time (s)	GPU (MB)
BERT$_{base}$	287	1697
BERT$_{base}$ + Self-reg. (K=2)	433	1712
BERT$_{base}$ + Self-reg. (K=3)	578	1718
BERT$_{base}$ + Co-reg. (N=2)	512	3335
BERT$_{base}$ + Co-reg. (N=3)	732	5029
BERT$_{large}$	756	5079
BERT$_{large}$ + Self-reg (K=2)	1155	5079

and 3X when trained with three models. Furthermore, compared with co-regularization, self-regularization also translates to faster training time.

VII. Discussion

A. Threats to Validity

We discuss the validity of our approach in both data construction and denoising method. First, labeling software entities in sentences could be subjective. Despite our efforts to narrow down a Wikipedia tree for the software engineering domain, we recognize that our annotation still cannot guarantee perfect precision or recall of all relevant named entities from Wikipedia. We estimate the precision of our WikiSER in Section IV and compare our dataset against previous work, which shows that WikiSER's size and comparatively low label error rate position it as a beneficial contribution for both the software engineering and NLP community. In addition, we understand the limitations of the Wikipedia taxonomy as a rich but not exhaustive source of relevant software entities. Besides Wikipedia, data sources such as GitHub and source code documentation could provide fruitful information.

In model training, we use mostly the same hyperparameters for baseline methods without individual tuning. A more thorough grid search could improve the results for some models in our evaluation. However, we note that while our main method experiments with different α, an important hyperparameter for self-regularization, we minimally tune other hyperparameters.

The task of software entity recognition poses many challenges in entity confusion and ambiguity, noisy user-input texts, and constant distribution shifts [21]. While we demonstrate the efficacy of the self-regularization framework in recognizing entities for clean (Wikipedia) and noisier user-input texts (Stack Overflow), it is difficult to guarantee that our approach would generalize well to *any* domain. However, we highlight that our framework can be easily adapted to new domains and any pretrained language models without requiring heavy supervision from auxiliary resources.

B. Limitations & Future Work

We evaluate our proposed method on WikiSER$_{small}$ rather than the entire WikiSER dataset, which is too large to train and evaluate in our GPU server. Future work could look into training and evaluating SER models with a larger sample from WikiSER or even the entire dataset on more GPUs.

Future work can also look into ways to improve upon more domain-specific methods for noisy label learning and further leverage the massive source of labeled data from WikiSER. The fact that our corpus contains 1.7M sentences makes it an attractive resource for exploring language model pretraining and multi-task learning [84]. Furthermore, there are many downstream software engineering tasks that could benefit from our noise-robust learning methods for SER, such as traceability link recovery [1]–[4], automated documentation [5]–[9], API recommendation [10]–[12], and bug fixing [13]–[16]. It is worthwhile to augment existing solutions in these downstream tasks with our SER model. Finally, given the recent advancement in Large Language Models (LLMs) such as ChatGPT, it is interesting to investigate how well LLMs perform in software entity recognition tasks.

VIII. Conclusion

In this work, we construct WikiSER, a large and high-quality software entity recognition dataset by leveraging the Wikipedia corpus. To account for labeling errors in SER datasets, we propose a new noise-robust learning method called self-regularization. Compared with multiple baseline models, including a SOTA SER model [23] and a SOTA noise-robust learning method [30], models trained with self-regularization perform the best while being more computationally efficient. Furthermore, self-regularization also generalizes well to two existing SER datasets from Stack Overflow. Finally, we highlight several improvement opportunities and outline future work.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. This research was in part supported by an Amazon Research Award and a Cisco Research Award.

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *TSE*, vol. 28, no. 10, pp. 970–983, 2002.

[2] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *ICSE*. IEEE, 2003, pp. 125–135.

[3] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *ICSE*, 2010, pp. 375–384.

[4] B. Dagenais and M. P. Robillard, "Recovering traceability links between an api and its learning resources," in *ICSE*. IEEE, 2012, pp. 47–57.

[5] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *ICSE*, 2014, pp. 643–652.

[6] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *ICSE*, 2016, pp. 392–403.

[7] C. Chen and Z. Xing, "Mining technology landscape from stack overflow," in *ESEM*, 2016, pp. 1–10.

[8] C. Chen, Z. Xing, and L. Han, "Techland: Assisting technology landscape inquiries with insights from stack overflow," in *ICSME*. IEEE, 2016, pp. 356–366.

[9] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving api caveats accessibility by mining api caveats knowledge graph," in *ICSEM*. IEEE, 2018, pp. 183–193.

[10] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," in *ASE*, 2018, pp. 293–304.

[11] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *SANER*, vol. 1. IEEE, 2016, pp. 349–359.

[12] W. Xie, X. Peng, M. Liu, C. Treude, Z. Xing, X. Zhang, and W. Zhao, "Api method recommendation via explicit matching of functionality verb phrases," in *ESEC/FSE*, 2020, pp. 1015–1026.

[13] F. Chen and S. Kim, "Crowd debugging," in *ESEC/FSE*, 2015, pp. 320–332.

[14] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing q&a sites (t)," in *ASE*. IEEE, 2015, pp. 307–318.

[15] S. Mahajan, N. Abolhassani, and M. R. Prasad, "Recommending stack overflow posts for fixing runtime exceptions using failure scenario matching," in *ESEC/FSE*, 2020, pp. 1052–1064.

[16] S. Mahajan and M. R. Prasad, "Providing real-time assistance for repairing runtime exceptions using stack overflow posts," in *ICST*. IEEE, 2022, pp. 196–207.

[17] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *ICSE*. IEEE, 2013, pp. 832–841.

[18] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker, "Part-of-speech tagging of program identifiers for improved text-based software engineering tools," in *ICPC*. IEEE, 2013, pp. 3–12.

[19] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Improving ir-based traceability recovery via noun-based indexing of software artifacts," *Journal of Software: Evolution and Process*, vol. 25, no. 7, pp. 743–762, 2013.

[20] Y. Huo, Y. Su, H. Zhang, and M. R. Lyu, "Arclin: automated api mention resolution for unformatted texts," in *ICSE*, 2022, pp. 138–149.

[21] D. Ye, Z. Xing, C. Y. Foo, Z. Q. Ang, J. Li, and N. Kapre, "Software-specific named entity recognition in software engineering social content," in *SANER*. IEEE, 2016, pp. 90–101.

[22] X. Chen, C. Chen, D. Zhang, and Z. Xing, "Sethesaurus: Wordnet in software engineering," *TSE*, vol. 47, no. 9, pp. 1960–1979, 2019.

[23] J. Tabassum, M. Maddela, W. Xu, and A. Ritter, "Code and named entity recognition in stackoverflow," in *ACL*, 2020, pp. 4913–4926.

[24] C. Zhou, B. Li, and X. Sun, "Improving software bug-specific named entity recognition with deep neural network," *Journal of Systems and Software*, vol. 165, p. 110572, 2020.

[25] G. Malik, M. Cevik, S. Bera, S. Yildirim, D. Parikh, and A. Basar, "Software requirement-specific entity extraction using transformer models," in *CAI*, 2022.

[26] A. Gorbatovski and S. Kovalchuk, "Bayesian networks for named entity prediction in programming community question answering," *arXiv:2302.13253*, 2023.

[27] Z. Huang, W. Xu, and K. Yu, "Bidirectional lstm-crf models for sequence tagging," *arXiv:1508.01991*, 2015.

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL*, 2019, pp. 4171–4186.

[29] T. Yano and M. Kang, "Taking advantage of wikipedia in natural language processing," 2016.

[30] W. Zhou and M. Chen, "Learning from noisy labels for entity-centric information extraction," in *EMNLP*, 2021.

[31] D. Ye, Z. Xing, C. Y. Foo, J. Li, and N. Kapre, "Learning to extract api mentions from informal natural language discussions," in *ICSME*. IEEE, 2016, pp. 389–399.

[32] S. Ma, Z. Xing, C. Chen, C. Chen, L. Qu, and G. Li, "Easy-to-deploy api extraction by multi-level feature embedding and transfer learning," *TSE*, vol. 47, no. 10, pp. 2296–2311, 2019.

[33] C. Zhou, B. Li, X. Sun, and H. Guo, "Recognizing software bug-specific named entity in software bug repository," in *ICPC*, 2018, pp. 108–119.

[34] C. Chen, Z. Xing, and X. Wang, "Unsupervised software-specific morphological forms inference from informal discussions," in *ICSE*. IEEE, 2017, pp. 450–461.

[35] M. Y. Chew, Y. J. Cheng, O. Mahan, and M. R. Islam, "A comparative study of name entity recognition techniques in software engineering texts," in *SAC*, 2022, pp. 1611–1614.

[36] J. R. Finkel, T. Grenager, and C. D. Manning, "Incorporating non-local information into information extraction systems by gibbs sampling," in *ACL*, 2005, pp. 363–370.

[37] D. Nadeau and S. Sekine, "A survey of named entity recognition and classification," *Lingvisticae Investigationes*, vol. 30, no. 1, pp. 3–26, 2007.

[38] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, "Neural architectures for named entity recognition," pp. 260–270, 2016.

[39] J. P. Chiu and E. Nichols, "Named entity recognition with bidirectional lstm-cnns," *TACL*, vol. 4, pp. 357–370, 2016.

[40] T. H. Nguyen, A. Sil, G. Dinu, and R. Florian, "Toward mention detection robustness with recurrent neural networks," *arXiv:1602.07749*, 2016.

[41] S. Zheng, F. Wang, H. Bao, Y. Hao, P. Zhou, and B. Xu, "Joint extraction of entities and relations based on a novel tagging scheme," in *ACL*, 2017, pp. 1227–1236.

[42] P. Zhou, S. Zheng, J. Xu, Z. Qi, H. Bao, and B. Xu, "Joint extraction of multiple relations and entities by using a hybrid neural network," in *CCL*. Springer, 2017, pp. 135–146.

[43] X. Ma and E. Hovy, "End-to-end sequence labeling via bi-directional lstm-cnns-crf," in *ACL*, 2016, pp. 1064–1074.

[44] Q. H. Tran, A. MacKinlay, and A. J. Yepes, "Named entity recognition with stack residual lstm and trainable bias decoding," in *IJCNLP*, 2017, pp. 566–575.

[45] M. Rei, G. Crichton, and S. Pyysalo, "Attending to characters in neural sequence labeling models," in *COLING*, 2016, pp. 309–318.

[46] Q. Wei, T. Chen, R. Xu, Y. He, and L. Gui, "Disease named entity recognition by combining conditional random fields and bidirectional recurrent neural networks," *Database*, vol. 2016, 2016.

[47] B. Y. Lin, F. F. Xu, Z. Luo, and K. Zhu, "Multi-channel bilstm-crf model for emerging named entity recognition in social media," in *Proceedings of the 3rd Workshop on Noisy User-generated Text*, 2017, pp. 160–165.

[48] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *NeurIPS*, vol. 30, 2017.

[49] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv:1907.11692*, 2019.

[50] I. Yamada, A. Asai, H. Shindo, H. Takeda, and Y. Matsumoto, "Luke: Deep contextualized entity representations with entity-aware self-attention," in *EMNLP*, 2020, pp. 6442–6454.

[51] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[52] Z. Yang, R. Salakhutdinov, and W. W. Cohen, "Transfer learning for sequence tagging with hierarchical recurrent networks," in *ICLR*, 2017.

[53] C. Jia, X. Liang, and Y. Zhang, "Cross-domain ner using cross-domain language modeling," in *ACL*, 2019, pp. 2464–2474.

[54] C. Jia and Y. Zhang, "Multi-cell compositional lstm for ner domain adaptation," in *ACL*, 2020, pp. 5906–5917.

[55] Z. Liu, Y. Xu, T. Yu, W. Dai, Z. Ji, S. Cahyawijaya, A. Madotto, and P. Fung, "Crossner: Evaluating cross-domain named entity recognition," in *AAAI*, vol. 35, no. 15, 2021, pp. 13 452–13 460.

[56] M. R. Islam and M. F. Zibran, "A comparison of software engineering domain specific sentiment analysis tools," in *SANER*. IEEE, 2018, pp. 487–491.

[57] J. Lee, W. Yoon, S. Kim, D. Kim, S. Kim, C. H. So, and J. Kang, "Biobert: a pre-trained biomedical language representation model for biomedical text mining," *Bioinformatics*, vol. 36, no. 4, pp. 1234–1240, 2020.

[58] E. Alsentzer, J. Murphy, W. Boag, W.-H. Weng, D. Jindi, T. Naumann, and M. McDermott, "Publicly available clinical bert embeddings," in *ClinicalNLP*, 2019, pp. 72–78.

[59] I. Beltagy, K. Lo, and A. Cohan, "Scibert: A pretrained language model for scientific text," pp. 3615–3620, 2019.

[60] Z. Zhang and M. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," *NeurIPS*, vol. 31, 2018.

[61] Y. Wang, X. Ma, Z. Chen, Y. Luo, J. Yi, and J. Bailey, "Symmetric cross entropy for robust learning with noisy labels," in *ICCV*, 2019, pp. 322–330.

[62] S. Sukhbaatar, J. Bruna, M. Paluri, L. Bourdev, and R. Fergus, "Training convolutional networks with noisy labels," *arXiv:1406.2080*, 2014.

[63] J. Goldberger and E. Ben-Reuven, "Training deep neural-networks using a noise adaptation layer," in *ICLR*, 2017.

[64] R. Wang, T. Liu, and D. Tao, "Multiclass learning with partially corrupted labels," *TNNLS*, vol. 29, no. 6, pp. 2568–2580, 2017.

[65] H.-S. Chang, E. Learned-Miller, and A. McCallum, "Active bias: Training more accurate neural networks by emphasizing high variance samples," *NeurIPS*, vol. 30, 2017.

[66] R. Müller, S. Kornblith, and G. E. Hinton, "When does label smoothing help?" *NeurIPS*, vol. 32, 2019.

[67] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *JMLR*, vol. 15, no. 1, pp. 1929–1958, 2014.
[68] Z. Wang, J. Shang, L. Liu, L. Lu, J. Liu, and J. Han, "Crossweigh: Training named entity tagger from imperfect annotations," in *EMNLP-IJCNLP*, 2019, pp. 5154–5163.
[69] Y. Xiao and W. Y. Wang, "Quantifying uncertainties in natural language processing tasks," in *AAAI*, vol. 33, no. 01, 2019, pp. 7322–7329.
[70] C. Wang, X. Peng, M. Liu, Z. Xing, X. Bai, B. Xie, and T. Wang, "A learning-based approach for automatic construction of domain glossary from source code and documentation," in *ESEC/FSE*, 2019, pp. 97–108.
[71] J. Cheng, T. Liu, K. Ramamohanarao, and D. Tao, "Learning with bounded instance and label-dependent label noise," in *ICML*. PMLR, 2020, pp. 1789–1799.
[72] L. A. Ramshaw and M. P. Marcus, "Text chunking using transformation-based learning," *Natural language processing using very large corpora*, pp. 157–176, 1999.
[73] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.
[74] K. Torisawa *et al.*, "Exploiting wikipedia as external knowledge for named entity recognition," in *EMNLP-CoNLL*, 2007, pp. 698–707.
[75] S. Cucerzan, "Large-scale named entity disambiguation based on wikipedia data," in *EMNLP-CoNLL*, 2007, pp. 708–716.
[76] A. E. Richman and P. Schone, "Mining wiki resources for multilingual named entity recognition," in *ACL-HLT*, 2008, pp. 1–9.
[77] J. Nothman, N. Ringland, W. Radford, T. Murphy, and J. R. Curran, "Learning multilingual named entity recognition from wikipedia," *Artificial Intelligence*, vol. 194, pp. 151–175, 2013.
[78] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, E. Li, X. Wang, M. Dehghani, S. Brahma *et al.*, "Scaling instruction-finetuned language models," *arXiv:2210.11416*, 2022.
[79] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, and A. Pohthong, "Robust statistical methods for empirical software engineering," *Empirical Software Engineering*, vol. 22, pp. 579–630, 2017.
[80] E. F. Sang and F. De Meulder, "Introduction to the conll-2003 shared task: Language-independent named entity recognition," 2003.
[81] D. Arpit, S. Jastrzębski, N. Ballas, D. Krueger, E. Bengio, M. S. Kanwal, T. Maharaj, A. Fischer, A. Courville, Y. Bengio *et al.*, "A closer look at memorization in deep networks," in *ICML*. PMLR, 2017, pp. 233–242.
[82] X. Liang, L. Wu, J. Li, Y. Wang, Q. Meng, T. Qin, W. Chen, M. Zhang, and T.-Y. Liu, "R-drop: Regularized dropout for neural networks," in *NeurIPS*, 2021.
[83] T. Zhang, D. P. Chandrasekaran, F. Thung, and D. Lo, "Benchmarking library recognition in tweets," in *ICPC*, 2022, pp. 343–353.
[84] A. Aghajanyan, A. Gupta, A. Shrivastava, X. Chen, L. Zettlemoyer, and S. Gupta, "Muppet: Massive multi-task representations with pre-finetuning," in *EMNLP*, 2021, pp. 5799–5811.