ON USING MACHINE LEARNING TO IDENTIFY KNOWLEDGE IN API REFERENCE DOCUMENTATION

Davide Fucci
University of Hamburg
Hamburg
Germany
fucci@informatik.uni-hamburg.de

Alireza Mollaalizadehbahnemiri
University of Hamburg
Hamburg
Germany
alirezam.alizadeh@gmail.com

Walid Maalej
University of Hamburg
Hamburg
Germany
maalej@informatik.uni-hamburg.de

July 24, 2019

ABSTRACT

Using API reference documentation like JavaDoc is an integral part of software development. Previous research introduced a grounded taxonomy that organizes API documentation knowledge in 12 types, including knowledge about the *Functionality*, *Structure*, and *Quality* of an API. We study how well modern text classification approaches can automatically identify documentation containing specific knowledge types. We compared conventional machine learning (k-NN and SVM) and deep learning approaches trained on manually-annotated Java and .NET API documentation (n = 5,574). When classifying the knowledge types individually (i.e., multiple binary classifiers) the best AUPRC was up to 87%. The deep learning and SVM classifiers seem complementary. For four knowledge types (*Concept*, *Control*, *Pattern*, and *Non-Information*), SVM clearly outperforms deep learning which, on the other hand, is more accurate for identifying the remaining types. When considering multiple knowledge types at once (i.e., multi-label classification) deep learning outperforms naïve baselines and traditional machine learning achieving a MacroAUC up to 79%. We also compared classifiers using embeddings pre-trained on generic text corpora and StackOverflow but did not observe significant improvements. Finally, to assess the generalizability of the classifiers, we re-tested them on a different, unseen Python documentation dataset. Classifiers for *Functionality*, *Concept*, *Purpose*, *Pattern*, and *Directive* seem to generalize from Java and .NET to Python documentation. The accuracy related to the remaining types seems API-specific. We discuss our results and how they inform the development of tools for supporting developers sharing and accessing API knowledge. Published article: https://doi.org/10.1145/3338906.3338943

1 Introduction

Software developers reuse software libraries and frameworks through Application Programming Interfaces (APIs). They often rely on reference documentation to identify which API elements are relevant for the task at hand, how the API can be instantiated, configured, and combined [1]. Compared to other knowledge sources, such as tutorials and Q&A portals, reference documentation like JavaDoc and PyDoc are considered the official API technical documentation. They provide detailed and fundamental information about API elements, components, operations, and structures [2, 3].

Figure 1: A reference documentation page in the JDK API annotated with the knowledge types it contains.

As API documentation can be thousands of pages long [4, 5], accessing relevant knowledge can be tedious and time-consuming [1]. Moreover, the information necessary to accomplish a task can be scattered across the documentation pages of multiple elements, such as classes, methods, and properties. Thus, developers try to use other sources to fulfill their information needs. For example, although the Java Development Kit (JDK) API documentation contains more than 7,000 pages, as of early 2019, there are more than 3 million StackOverflow posts tagged as `java`.

Over the last decade, software engineering researchers studied what information developers need when consulting API documentation [3, 6, 7]. One line of research focuses on automatically matching information needs with the types of knowledge available in the documentation. Maalej and Robillard [3] took a first step in this direction by developing an empirically-validated taxonomy of 12 knowledge types found within API reference documentation. A single documentation page can include several knowledge types (Figure 1). *Functionality* and *Directive* are particular types of knowledge needed to accomplish a development task, whereas the *Non-information* type contains only uninformative boilerplate text [3]. Maalej and Robillard argue that such knowledge categorization allows for a) understanding and improving the documentation quality and b) satisfying developers' information needs. [3]

The research community has shown interest in studying specific knowledge types contained in API reference documentation. For example, Montperrus et al. [8] and Seid et al. [9] studied *Directive* to prevent the violation of API usage constraints. Robillard and Chhetri [5] filtered *Non-information* when recommending APIs to developers. However, these automated approaches are based either on linguistic features engineering [5] or on syntactic patterns [8].

This work investigates how well modern text classification approaches can automatically identify the knowledge types suggested by Maalej and Robillard in API documentation. Based on a dataset of 5,574 labelled Java and .NET documentation, we trained, tested, and compared conventional machine learning approaches—i.e., k-Nearest Neighbors (k-NN) and Support Vector Machines (SVM)—as well as deep learning approaches—i.e., recurrent neural network (RNN) with a Long Short-Term Memory (LSTM) layer. The RNN learns features from a semantic representation of general purpose text (i.e., embeddings). Hence, we studied how our results are impacted by training the network using software development-specific corpora from StackOverflow as opposed to a general purpose one. Finally, we studied the generalizability of the classifiers to an unseen dataset obtained from the Python standard library.

This paper makes three contributions. First, we present a detailed classification **benchmark** for API documentation. The settings include different machine learning approaches and configurations, different word embeddings for the RNN, different datasets for different APIs, as well as various evaluation metrics. Researchers and tool vendors can use the benchmark, for example, to select and optimize a specific classifier for a specific cofiguration of API and knowledge types. Second, as we share the code and data of this study,[1] several **top-performing classifiers** (e.g., AUPRC $\geq 80\%$) already have practical relevance. Third, our findings and discussion of related work provide insights to researchers, tool vendors, and practitioners on **how machine learning can help** better organize, access, and share knowledge about API.

The rest of the paper is organized as follows. Section 2 describes our research settings, Section 3 presents the configurations of the classifiers, and Section 4 reports their performance We discuss related work in Section 5 and the implication of our results in Section 6. Finally, Section 7 concludes the paper.

---

[1] https://zenodo.org/badge/latestdoi/194706952

2 Research Settings

This section introduces the research questions, method, and data.

2.1 Research Questions and Method

Maalej and Robillard [3] proposed an empirically-validated taxonomy of 12 knowledge types based on grounded theory and systematic content analysis (17 experienced coders, 279 person-hours effort). Table 1 reports the identified knowledge types which represent the basis for this work. Our primary goal is to study how well simple machine learning for text classification, without additional feature engineering or advanced natural language processing (NLP) techniques, can identify these knowledge types. That is, our classifiers label a document with one or more knowledge types.

Table 1: Twelve knowledge types included in reference documentation (adapted from Maalej and Robillard [3]).

Knowledge type	Brief description
Functionality	Describes the capabilities of the API, and what happens when it is used.
Concept	Explains terms used to describe the API behavior or the API implementation.
Directive	Describe what the user is allowed (not allowed) to do with the API.
Purpose	Explains the rationale for providing the API or for a design decision.
Quality	Describes non-functional attributes of the API, including its implementation.
Control	Describes how the API manages the control-flow and sequence of calls.
Structure	Describes the internal organization of API elements including their relationships.
Pattern	Explains how to get specific results using the API.
Example	Provides examples about the API usage.
Environment	Describes the API usage environment.
Reference	Pointers to external documents.
Non-information	Uninformative, boilerplate text.

There are two main text classification approaches which we study in this paper. Traditional approaches usually learn the classes from the occurrences of certain keywords or phrases in the training set. More computational intensive approaches, often referred to as deep learning, use the semantics of the keywords—i.e., the context of the keyword occurrences [10].

For traditional approaches, we study two algorithms frequently used for text classification, k-NN and SVM. For deep learning, we used RNN with an LSTM layer, which is particularly effective for text categorization problems [11]. This architecture is recommended over, for example, Convolutional Neural Network (CNN). While the latter is more suited for image recognition [12], RNN with LSTM handles more efficiently the dependencies between features [11]. We also compare these classifiers to naïve baselines.

The task tackled in this study is to assign knowledge types to an API document. As the document can contain more than one knowledge type, this task is modelled as a multiple binary classification problem consisting of independently train one binary classifier for each knowledge type. Another approach is to train a multi-label classifier—i.e., a classifier that outputs a set of knowledge types rather than a single one. We analyze and report the results for both approaches when answering the following research question.

RQ1. How well can text-based classifiers identify knowledge types in API reference documentation? In particular, can deep learning improve over traditional approaches?

For text classification tasks, the input layer of an RNN usually consists of embeddings trained on large unlabeled textual corpora necessary to capture rich semantic features [13]. Pre-trained embeddings are available and can be easily "plugged" in the network without further effort. However, while these embeddings save computational time and well represent common language tasks, they can miss software engineering or API-specific semantics. This motivates our second research question.

> **RQ2**. Do software development-specific text embeddings improve classification results compared to general purpose ones?

Finally, a common question for machine learning evaluation is whether a model trained on a certain dataset generalizes to other data. The original dataset includes documentation of the standard Java and .NET libraries [3]. Since we aim to assess the generalizability of our approach to API reference documentation written in a different style, we manually annotated a new datase sampled from the Python standard library documentation. We used this dataset as an additional test set to report our classifiers performance.

> **RQ3**. Can documentation classification based on knowledge types be generalized across API?

We assess models based on 10-fold cross-validation using 10% of the dataset as test set. When comparing individual knowledge types classifiers, we report Area Under Precision-Recall Curve (AUPRC). Precision-Recall curves are a common metric to evaluate binary classificaiton and are obtained by plotting precision and recall values at different probabilities thresholds [14]. In particular, they are used to evaluate machine learning model trained on imbalanced data sets [14]. Therefore, AUPRC is a summary measure of performance irrespective of a particular threshold.

When comparing classifiers for multiple knowledge types, we report performance according to two types of metrics, item-based and label-based. The item-based metrics are a) *Hamming Loss*, namely the ratio of wrongly classified labels to the total number of labels (its best value is zero) and b) *Subset Accuracy*, namely the percentage of exact matches between the predicted and the actual labelset.

The label-based metrics are precision, recall, F1-measure (Formula 1), and Area Under Receiving Operator Curve (AUC).

$$F1 = \frac{2 \times TruePositives}{2 \times TruePositives + FalsePositives + FalseNegatives} \tag{1}$$

The Receiving Operator Curve is created by plotting recall against false positive rate (FPR, Equation 2), at different probability thresholds. Accordingly, AUC does not depend on a particular threshold [15]. To calculate the value of True Positive, False Positives, and False Negatives we used 0.5 as probability threshold [10].

$$FPR = 1 - \frac{TrueNegatives}{TrueNegatives + FalsePositives} \tag{2}$$

The label-based metrics are macro-averaged. Macro-averaging applies the metric to the binary partition of each predicted label and then averages the results—i.e., labels have equal contribution in the final result. In contrast, micro-averaging first aggregates the individual metric components (i.e., true positives, false positives, true negatives, and false negatives) of each label and then averages them. Therefore, micro-averaging is biased toward the majority classes and should be avoided when evaluating unbalanced datasets [16].

We compare the results of the classifiers to naïve baselines, MF1, MF2, and RAND. The first two always assign the first (respectively one of the first two) most-frequent labels to each document, whereas the latter assigns a random label.

Table 2: Overview of the CADO dataset.

	#documents	Words max.	Words mean	Vocab. size
.NET	2,782	2,874	89	10,630
JDK	2,792	2,099	86	10,763
Total	5,574	2,874	87	17,758

2.2 Research Data

We use the **CADO dataset** created by Maalej and Robillard [3] as the result of their content analysis of the JDK 6 and .NET 4.0 API reference documentation. CADO contains 5,574 observations. The columns include the name of the API element (e.g., a class, a method, or a property), its documentation text, and 12 binary values indicating the presence (or absence) of the corresponding knowledge type. Table 2 summarizes the dataset textual properties.

The most frequent knowledge types are *Functionality* and *Non-information*, whereas *Quality* and *Environment* are the least frequent. We did not merge some of the knowledge types as Maalej and Robillard reported no significant evidence of their co-occurence [3].

The majority of the documents (90.5%) contains one to five of the 12 knowledge types. We use the SCUMBLE [17] score ($\in [0, 1]$) to report the level of unbalancedness. For a given label, a high SCUMBLE score represents a large difference between the frequencies of all the other co-occurring labels. In general, datasets with high scores are problematic for classification tasks [18]. However, for datasets characterized by low SCUMBLE score, resampling can reduce unbalancedness [18]. CADO mean SCUMBLE score is 0.11.

We applied random under- and over-sampling to 90% of the dataset (i.e., the training set). We did not resample the test set (10% of the dataset) to avoid sampling bias. For resampling, we removed 30% of the documents containing *Functionality* and *Non-Information* in their labelset and duplicated 50% of the documents containing *Environment* and *Quality*. The thresholds were obtained empirically based on the SCUMBLE score. After resampling, the training and test sets contain 3,876 and 430 observations respectively. Figure 2 presents the label frequencies in the dataset we used to train the models after re-sampling.

We prepared a new **PYTHON dataset** consisting of 100 API documentation pages (i.e., modules, types, attributes, and methods) from the Python 2.7 standard library.[2] We selected the Python standard library since its code is organized differently than Java or .NET as it makes extensive use of modules in which functions, classes, and variables are defined. The Python programming paradigm is more functional than Java and .NET which instead follow an object-oriented paradigm. Python is dynamically typed, and its reference documentation tends to focus on functions, whereas types documentation is embeddable in the source code (e.g., through Docstrings). Finally, its development and documentation are driven by an open source, non-profit community (the Python Software Foundation) whereas Java and .NET are owned by corporations.

We followed the sampling strategy suggested by Maalej and Robillard [3]—i.e., stratified random sampling. We first created strata for each of the base modules and then randomly sampled API documentation from each stratum proportionally to their frequencies.

---

[2]https://docs.python.org/2.7/library/

Figure 2: Knowledge types distribution in the CADO dataset after resampling.

Figure 3: Knowledge types distribution in the PYTHON dataset.

Two Ph.D. students in software engineering, accustomed to work with Python, manually labelled the knowledge types in each document. For this task, we provided them the same guidelines from Maalej and Robillard[3] with small adaptations, such as providing examples using the Python programming language. The agreement on the label set was 14%—i.e., 14 out of the 100 examples were labeled with the *exact same* set of knowledge types. The overall agreement was 75%—i.e., of the 1200 labels (100 examples × 12 labels), 300 were conflicting. Two of the authors addressed the conflicts and created the final dataset. Figure 3 shows the distribution of knowledge types in the PYTHON dataset. *Functionality* is the majority label. However, as PYTHON represents an additional test set (i.e., no examples from this dataset are used to train the classifiers), we did not resample it to avoid biased results.

Table 3: Summary of the corpora used to train the GloVe embedding.

ID	Name	Corpus description	#docs	#vocabulary
CC	Common Crawl	General purpose, high-quality text crawled from Internet pages	2.2M	∼220.000
CCOTF	Common Crawl on-the-fly	Common Crawl where missing words are learned from CADO	2.2M	∼220.000
SO	StackOverflow	StackOverflow questions and answers	20M	∼400.000
SOAPI	StackOverflow Java and .NET posts	StackOverflow questions and answers tagged as `java` or `.net`	4M	∼100.000

For both datasets, we performed several, simple operations to clean and prepare the textual data. We lower-cased, tokenized, and applied stop-words removal to the API documentation text. Then, we transformed terms in an order-preserving one-hot vectors.

For the deep learning classifiers in our benchmark, we train GloVe [19] embeddings based on four large corpora, summarized in Table 3. The Common Crawl (CC) is a pre-trained embedding downloaded in March 2018.[4] It includes 840B tokens and a vocabulary of 2.2M words. The corpus contains high-quality, general-purpose text crawled from the Internet. However, the CC corpus is missing domain-specific terms present in the CADO dataset. Accordingly, in the CCOTF embeddings, the missing words from the CC corpus are trained on-the-fly [20]. Finally, we obtained a completely domain-specific representation of the input by training embeddings on two additional corpora, StackOverflow (SO) and StackOverflow API (SOAPI). The former includes 20 million posts, while the latter includes 4 million posts tagged as `java` or `.net`.

3 Classifiers Configuration

This section reports information about the configuration of both machine learning and deep learning classifiers used in this study.

3.1 Traditional Machine Learning

The machine learning approaches we selected for our classification task are SVM [21] and *k*-NN [22] as well as their adaptations to multi-label problems, namely One-vs-Rest SVM (OvRSVM) and Multi Label *k*-NN (ML-*k*NN). We use unigrams and bigrams extracted from the CADO dataset as their input features as *n*-gram language models are easy to

---

[3] https://cado.informatik.uni-hamburg.de
[4] `https://commoncrawl.org`

compute and use. Moreover, they have been used in studies where machine learning and natural language processing are applied to software engineering contexts [23, 24].

SVM is one of the most investigated approaches for statistical document classification and it is considered state-of-the-art [25, 21]. Moreover, it showed good results in software engineering-specific text classification problems (e.g., [26, 27]). SVM finds the hyper-plane maximizing the margin between two classes in the feature space, and it can learn and generalize high-dimensional features typical for text classification tasks [21, 25]. When taking into account multiple knowledge types at once, we trained and reported the results of a SVM model adapted to such problem—i.e., OvRSVM using binary relevance [28]. OvRSVM considers additional parameters and constraints necessary to solve the optimization problem with several classes and to handle the separation of several hyper-planes [28]. For the SVM classifiers, we report the best model after hyper-parameters tuning using GridSearch [29].

k-NN is a widely-used approach in machine learning [30]. It determines the k nearest neighbors of a document using Euclidean distance. Then it assigns the document a label based on the document neighbors using Bayes decision rules [30]. For the multi-label classification, we use ML-kNN, which outperforms well-established multi-label classifiers [31].

3.2   RNN with LSTM layer

Deep learning has recently brought substantial improvements in the field of machine vision and natural language processing (NLP) [10]. The LSTM layer extends RNN capabilities by utilizing several gates and a memory cell in the recurrent module to alleviate the vanishing gradient problem and to handle more efficiently the long-term dependencies between features [11].

Figure 4: Architecture of the RNN used for classification of the knowledge types.

Figure 4 shows the architecture we used in this work. The network is composed of a single LSTM layer, two dense layers, and an output layer. The number of units in the LSTM layer is proportional to dimensions of the vectors used to represent each word. The dense layers contain 128 and 64 units respectively. The number of units in the output layer is the number of knowledge types (i.e., 12 units).

The core component of the LSTM layer is a memory cell which stores the information related to the previous analysis steps within the network. At each step of the training, the network predicts the output based on a) the new input, b) the previous state of the other hidden layers of the RNN, and c) and the current state of the memory cell. Accordingly, the role of the gates is to learn how to modify the memory cell to enhance prediction accuracy (see Figure 5).

The *forget gate* (f_t) processes the information from the previous hidden state layer (h_{t-1}) and the current input (X_t)—i.e., a representation of the API documentation text. It then decides what information should be discarded or kept from the previous state of the memory cell (C_{t-1}). The *input gate* (i_t) is responsible for selecting new information from the input (X_t) that should be stored in the cell state. The third gate is called *output gate* (o_t) and decides which part of the available information in the memory cell should be used to produce the final output (h_t).

The role of the forget cell in our network is to optimally discard information related to previous knowledge types when they change in the new input. For instance, when the knowledge type in the new input is *Directive*, the forget gate removes the pieces of information associated with other knowledge types. Consequently, the forget gate reduces the ambiguity of the memory cell when learning individual types. The features associated with the knowledge type in the current input document are moved into the memory cell. In the memory cell, the input gate decides what information should be stored. For example, when the current input contains *Directive*, its features will be extracted and stored in the memory cell using the input gate. Finally, the output gate selects the most significant features associated with the *Directive* type.

The **input layer** for the RNN consists of word embedding vectors trained using GloVe [19]. GloVe relies on the global occurrences of a word in a corpus by defining a word-to-word co-occurrence matrix. Each value of the matrix contains the probability P of word j appearing in the context of word i, as reported in Equation 3. In particular, X_{ij} denotes the number of times word j occurs in the context of word i and X_i denotes the number of times that any word

Figure 5: A single LSTM recurrent module containing input (i_t), output (o_t), and forgets gates (f_t).

k appears in the context of word i. Namely, GLoVe defines a learning function that estimates the probability ratio of the co-occurrence of two target words, i and j, given a context word k [19].

$$P_{ij} = P(j|i) = X_{ij}/X_i, \ X_i = \sum_k X_{ik}. \tag{3}$$

As input vectors with identical length speed up the process of building the embedding layer of the RNN [32], we considered 300 as the maximum vector length and padded shorter vectors with zeroes. Therefore, our input layer is a 2D matrix where each row is a 300-dimensional unit (artificial neuron). The number of units in the input layer depends on the vocabulary size of the corpus used to train the embeddings. When a document is fed to the RNN, the units associated with the document terms will be activated.

The first **hidden layer** is a LSTM unit which learns textual features of an input document. It is suited to learn long-term dependencies, such as in the case of the large API reference documentation text. To prevent over-fitting, we applied a dropout technique to the weights matrix and to the bias vector [33].

The output of the LSTM layer (i.e., a set of features that can be associated with a knowledge type) goes through two fully-connected dense layers. The dense layers provide deep representations of the features extracted by the LSTM layer and enable the network to learn their hierarchical and compositional characteristics. To alleviate feature loss due to the projection of the features from a high-dimensional space to the low-dimensional space of the output layer, the model smoothly reduces the number of units in the dense layers from 128 to 64 [10]. We use the ReLU activation function for the dense layers to prevent over-fitting [10].

The **output layer** provides the predicted knowledge types using a sigmoid activation function. Hence, the number of units in the output layer is the number of labels that the model learns. As the output of the sigmoid is a probability value between 0 and 1, each neuron in this layer learns to estimate the probability of observing one of the labels. To binarize the predicted probabilities we used different thresholds according to the different metrics.

We tuned the following network parameters.
Epoch is a complete pass (back and forward) of every sample through the neural network. As customary, we run 100 epochs [10].
Batch size is the number of samples passed through the network at once. As customary, we used a batch size of 32 [10].
Optimizer is the optimization method minimizing the prediction error. We use Adam, a state-of-the-art algorithm for training RNN [34].
Loss Function is the measure of the network prediction error. We use sigmoidal cross-entropy as it is efficient for text classification [35].
Learning Rate is the parameter controlling the adjustments to the weights with respect to the prediction error. We used a customary learning rate of .001 [10].

4   Results

In this section, we compare the performance of RNN and traditional machine learning approaches for the classification of API reference documentation. We contrast the performance of RNN classifiers trained using different embeddings. Moreover, we assess the classifiers generalizability to another test set.

Table 4: Comparison between Deep learning classifiers (trained with embeddings from general purpose and software development corpora), traditional machine learning, and naïve approaches for classifying *individual* knowledge types in the CADO dataset. Values report the Area Under Precision-Recall Curve (AUPRC).

Knowledge Type	Naïve baselines			Traditional approaches		Deep learning (General Purpose)		Deep learning (Software dev.)	
	MF1	MF2	RAND	k-NN	SVM	RNN_{CC}	$RNN_{CC_{OTF}}$	RNN_{SO}	$RNN_{SO_{API}}$
Functionality	0.69	0.73	0.72	0.76	0.39	0.86	0.84	**0.87**	**0.87**
Concept	0.11	0.14	0.12	0.13	**0.57**	0.25	0.28	0.28	0.28
Directive	0.26	0.16	0.17	0.22	0.04	0.40	0.41	0.41	**0.45**
Purpose	0.22	0.21	0.17	0.17	0.09	0.36	0.40	0.40	**0.41**
Quality	0.04	0.04	0.05	0.12	0.13	**0.78**	0.69	0.68	0.54
Control	0.08	0.12	0.09	0.08	**0.81**	0.28	0.32	0.30	0.30
Structure	0.37	0.37	0.35	0.38	0.42	0.61	0.56	**0.63**	0.60
Pattern	0.14	0.17	0.14	0.21	**0.59**	0.46	0.46	0.48	0.51
Example	0.24	0.23	0.20	0.25	0.60	**0.90**	0.85	**0.90**	**0.90**
Environment	0.04	0.03	0.06	0.16	0.43	0.68	**0.80**	0.66	0.51
Reference	0.11	0.14	0.16	0.13	0.15	0.35	0.35	**0.41**	0.30
Non-information	0.29	0.31	0.28	0.33	**0.71**	0.57	0.58	0.62	0.55

4.1 Knowledge Types Identification

Individual knowledge types. We trained two RNN-based classifiers using a general-purpose corpus to create the embeddings for the input layer—RNN_{CC} and $RNN_{CC_{OTF}}$. Table 4 reports the evaluation of our classifiers. RNN and traditional machine learning approaches improve over the naïve baselines for all the individual knowledge type classification by up to 74% (41% on average). SVM always performed better than k-NN.

Deep learning classify *Functionality*, *Example*, and *Environment* with high precision and high recall at different probability thresholds (AUPRC $\geq 80\%$) outperforming machine learning approaches. The RNNs yields subpar results for *Directive*, *Purpose*, *Reference*, *Concept*, and *Control* (AUPRC $< 50\%$). However, the best SVM outperforms the best RNN only for the latter two types.

The best classifiers for *Quality*, *Structure*, *Patterns*, and *Non-information* yield an AUPRC between 59% and 78%. Also in this case, the best machine learning approach (i.e., SVM) outperforms the best RNNs classifiers only for the latter two types. Compared to machine learning, RNNs classify better eight knowledge types.

Multiple knowledge types. The second step is to consider our task as a multi-label classification problem rather than building individual classifiers for each knowledge type. We compare the RNN classifiers to the multi-label adaptation of the same two machine learning models and two naïve baselines (see Table 5).

ML-kNN and OvRSVM perform worse than the baselines for the item-based metrics, whereas the RNNs shows the best performance. The RNNs outperform ML-kNN, OvRSVM, and the baselines for label-based metrics. There is an 11% improvement regarding the most strict metric (i.e., Subset Accuracy) between the best RNN and machine learning classifiers. Regarding MacroPrecision, MacroRecall, and MacroF1, there is an improvement between 25% and 28% for the RNN. MF1 performs better than traditional machine learning regarding MacroAUC, which RNNs improves by 17%.

> **Answer to RQ1.** One-third of the knowledge types can be automatically identified with good results (i.e., AUPRC $\geq 80\%$). RNN can more accurately ($¿ 10\%$) identify eight of the 12 knowledge type compared to traditional machine learning approaches. When considering multi-label classification, RNN outperforms traditional machine learning approaches for item- and label-based metrics.

4.2 Software Development-specific Corpus

Individual knowledge types. One RNN uses freely-available, pre-trained embeddings based on a general purpose textual corpus (i.e., RNN_{CC}), whereas $RNN_{CC_{OTF}}$ uses the same corpus but learns missing words on-the-fly from the CADO dataset. The assumption behind text statistical representations such as GloVe is that the meaning of a document is determined by the meaning of the words that appear in it [36]. Accordingly, RNN_{SO} and $RNN_{SO_{API}}$ use corpora in a domain closer to the one of API documentation.

As shown in Table 4, the best among these RNNs performs similarly to their general domain counterparts (ΔAUPRC = 4%). For *Functionality*, *Purpose*, *Control*, and *Structure* the differences are minimal (1-2%). However, for *Quality* and *Environment* there is a substantial decrease in performance when using software development-specific embeddings

Table 5: Comparison between Deep learning classifiers (trained with embeddings from general purpose and software development corpora), traditional machine learning, and naïve approaches for classifying *multiple* knowledge types in CADO.

Metric	Naïve baselines		Traditional approaches		Deep learning (General Purpose)		Deep learning (Software dev.)	
	MF1	MF2	ML-kNN	OvRSVM	RNN_{CC}	$RNN_{CC_{OTF}}$	RNN_{SO}	$RNN_{SO_{API}}$
Hamming Loss	0.17	0.20	0.18	0.30	0.16	**0.14**	**0.14**	**0.14**
Subset Accuracy	0.00	0.13	0.11	0.02	0.20	**0.22**	0.19	0.21
MacroPrecision	0.05	0.08	0.41	0.21	0.56	**0.66**	0.61	0.63
MacroRecall	0.16	0.16	0.24	0.27	**0.55**	0.39	0.30	0.33
MacroF1	0.10	0.10	0.27	0.24	**0.55**	0.44	0.40	0.43
MacroAUC	0.62	0.50	0.55	0.61	0.73	0.74	0.78	**0.79**

(10% and 14%, respectively). Overall, the improvement is rather limited given the overhead in obtaining the corpus and computing the embeddings.

Multiple knowledge types. Table 5 shows that RNN_{CC} and $RNN_{CC_{OTF}}$ outperform RNN_{SO} and $RNN_{SO_{API}}$ when considering label-based metrics (except for MacroAUC) and perform similarly when considering item-based metrics. The RNN trained using Java and .NET StackOverflow posts yields the best MacroAUC (79%).

> **Answer to RQ2.** RNN using software development-specific embeddings show slight to no improvement over RNN using general purpose embeddings for classification of individual knowledge types. When considering multi-label learning, except for MacroAUC, using general purpose embeddings yields better results across item- and label-based metrics.

4.3 Classifiers Generalizability

Individual knowledge types. Table 6 reports the performance of the individual RNN-based classifiers on the PYTHON test set. Also in this setting, no naïve baseline performs better than traditional or deep learning approaches. The RNNs are the best classifiers for seven knowledge types, whereas SVM shows the best results for the remaining five. Consistently with the CADO setting, SVM is the best classifier for *Concept*, *Pattern*, and *Non-information*. Classifiers for *Functionality*, *Concept*, and *Purpose* show some improved performance compared to the CADO settings (Δ AUPRC = 8.3%).

There is a large absolute difference (Δ AUPRC = 33%) between the two settings when considering *Directive*, *Quality*, *Control*, *Structure*, *Example*, and *Environment*, suggesting that these knowledge types are dependent on the settings. On average, the performance on the PYTHON dataset decrease by ~16% over the 12 knowledge types.

Multiple knowledge types. Table 7 present the results of the multi-label classification task. Regarding item-based metrics, our classifiers perform worse or on par with respect to the naïve baselines. The classifiers show low precision (40% for the best classifier, SVM) and recall (26% for the best classifiers, $RNN_{CC_{OTF}}$ and $RNN_{SO_{API}}$). SVM also achieves the best F1 (30%). $RNN_{SO_{API}}$ shows the best performance for MacroAUC (64%) .

> **Answer to RQ3.** Classifiers for *Functionality*, *Concept*, *Purpose*, *Pattern*, and *Directive* seem to generalize from Java and .NET to Python documentation. The generalization for multiple knowledge types classifiers is limited.

5   Related Work

To the best of our knowledge, this is the first study, addressing the automated identification of *several* knowledge types within API reference documentation. In this section, we report related work investigating some of the knowledge types individually. We present studies comparing traditional machine learning and deep learning approaches for text classification in software engineering.

5.1   Knowledge Types in API Documentation

Identifying a document based on the knowledge types it contains can support documentation quality assessment and improvement. For example, Ding et al. [37] systematic review of 60 primary studies investigates documentation quality attributes. The authors focus on knowledge-based approaches used to address quality issues of API documentation. Although retrievability is reported as an essential quality attribute, the authors show a lack of advanced ways to retrieve

Table 6: Comparison between Deep learning classifiers (trained with embeddings from general purpose and domain specific corpora), traditional machine learning, and naïve approaches for classifying API documents based on *individual* knowledge type in the PYTHON dataset. Values report the Area Under Precision-Recall Curve (AUPRC).

Knowledge Type	Naïve baselines			Traditional approaches		Deep learning (General Purpose)		Deep learning (Software dev.)	
	MF1	MF2	RAND	k-NN	SVM	RNN_{CC}	$RNN_{CC_{OTF}}$	RNN_{SO}	$RNN_{SO_{API}}$
Functionality	0.89	0.89	0.92	0.85	0.94	0.90	0.89	**0.95**	0.94
Concept	0.29	0.28	0.31	0.26	**0.64**	0.40	0.33	0.49	0.41
Directive	0.41	0.41	0.49	0.42	**0.71**	0.49	0.44	0.55	0.63
Purpose	0.28	0.28	0.25	0.30	0.13	0.46	0.40	**0.51**	0.39
Quality	0.17	0.17	0.19	0.17	0.27	0.20	0.17	0.20	**0.32**
Control	0.27	0.27	0.32	0.24	0.33	0.43	**0.46**	0.39	0.35
Structure	0.24	0.24	0.24	0.32	0.11	0.26	0.24	0.30	**0.32**
Pattern	0.22	0.22	0.24	0.29	**0.61**	0.50	0.30	0.41	0.43
Example	0.36	0.36	0.38	0.43	0.44	0.48	0.49	**0.51**	0.48
Environment	0.16	0.16	0.17	0.16	**0.37**	0.15	0.15	0.18	0.17
Reference	0.12	0.12	0.17	0.11	0.22	0.16	0.19	0.24	**0.25**
Non-information	0.23	0.23	0.24	0.27	**0.61**	0.30	0.39	0.30	0.28

Table 7: Comparison between Deep learning classifiers (trained with embeddings from general purpose and software development corpora), traditional machine learning, and naïve approaches for classifying *multiple* knowledge types in PYTHON.

Metric	Naïve		Traditional approaches		Deep learning (General Purpose)		Deep learning (Software dev.)	
	MF1	MF2	MLkNN	OvRSVM	RNN_{CC}	$RNN_{CC_{OTF}}$	RNN_{SO}	$RNN_{SO_{API}}$
Hamming Loss	0.23	**0.25**	0.28	0.35	0.27	0.30	0.26	0.27
Subset Accuracy	**0.05**	**0.05**	0.02	0.01	0.02	0.03	0.04	**0.05**
MacroPrecision	0.07	0.10	0.33	**0.40**	0.36	0.31	0.31	0.31
MacroRecall	0.08	0.16	0.24	0.24	0.24	**0.26**	0.21	**0.26**
MacroF1	0.07	0.13	0.28	**0.30**	0.29	0.28	0.25	0.28
MacroAUC	0.50	0.50	0.53	0.54	0.60	0.57	0.62	**0.64**

specific information from API documentation. On the one hand, our work represents a first step towards developing retrieval mechanisms for documents containing a set of knowledge types from the Java and .NET API reference documentation. On the other hand, the individual classifiers showing a performance (e.g., *Functionality*, *Control*, *Example*, and *Environment*) can be used to retrieve documents containing a specific knowledge type. Moreover, our classifiers can be used to retrieve documents containing *Functionality* from the Python standard library documentation.

Previous research tried to automatically retrieve particular knowledge from API documentation. Robillard and Chhetri [5] presented an approach to identify API-related information that developers should not ignore as well as non-critical information. Their approach—based on natural language analysis (i.e., part-of-speech tagging, word patterns)—shows 90% precision and 69% recall when applied to 1000 Java documentation units. However, the authors needed to manually assess, on top of the sensible knowledge items, also obvious, unsurprising, and predictable documentation—i.e., what we consider *Non-information* [5]. Our SVM classifier, trained using simple features, identifies *Non-information* with 71% accuracy.

Montperrus et al. [8] studied a particular knowledge type found in API reference documentation, *Directive*. They analyzed more than 4000 API documentation from open source libraries. To determine the documents containing *Directive*, they developed a set of syntactic patterns associated with concerns reported in the documentation. Finally, they manually created a taxonomy of 23 directives. Pandita et al. [38] proposed an NLP-based approach to verify the legal usage of API methods against its description extracted automatically from the documentation. Their approach uses features derived from part-of-speech tagging and chunking techniques to semantically analyze text. Moreover, using a domain dictionary, the authors extracted methods specifications as first-order logic expressions to verify their legal usage in client code.

Conversely, in this work, we attempted a simple approach based only on features which can be automatically extracted from the raw text. Our goal was to create a benchmark which can be improved by including, for example, natural language patterns specific for each knowledge types and domain-specific models. We show that some classifiers have already practical relevance.

5.2 Deep Learning in Software Engineering

Xu et al. [39] use CNN to semantically link together knowledge units from StackOverflow. Their approach focuses on predicting several classes of relatedness (e.g., duplicate, related information). The network input is the word2vec representation of 100,000 Java-related posts from StackOverflow, whereas the dataset includes 8,000 knowledge units balanced among relatedness types. The CNN outperformed machine learning baselines—i.e., SVM trained using tf-idf and word2vec. However, Fu and Menzies [27] replicated Xu et al. study comparing their results to the same SVM baselines optimized using hyper-parameter tuning. The authors showed improved results for the baselines which perform closely (if not better) to the CNN, although the latter required 84x more time to train. In this work, we also used a deep learning approach with a semantic representation of the input based on StackOverflow. We found that, for our task, there are only a few small improvements due to the software development-specific corpus, which may not be worth when considering the extra effort required to obtain and train the embeddings. We compared the deep learning approach to (among others) SVM models trained in line with the suggestions of Fu and Menzies [27]. We showed that the approaches are complementary as their performance depends on the specific knowledge types.

Fakhoury et al. [40] applied deep learning and traditional machine learning to the detection of language anti-patterns in software artifacts (e.g., poor naming conventions and documentation) using a dataset of 1,700 elements collected from 13 large Java system. The authors showed that using Bayesian optimization and model selection, traditional machine learning models can outperform deep learning not only in accuracy but also regarding the use of computational resources. They advise researchers and practitioners to explore traditional machine learning models with hyper-parameter tuning before turning to deep learning approaches. Our results for individual knowledge types partly support this conclusion. However, when tackling multi-label problems, our work shows that deep learning performs better than traditional machine learning for all the reported metrics.

6 Discussion

In this section, we discuss the implications for practitioners and researchers. Then, we present the limitations of this study.

6.1 Implications

Building automated knowledge extraction tools. Classifiers showing good performance (AUPRC $\geq$ 80%) can already be used in practice to tag documents containing crucial information for developers. Moreover, these classifiers are trained using either traditional machine learning algorithms with simple text features or using deep learning but with readily available embeddings.

A document containing *Functionality* can answer developers' information needs regarding what the API does, whereas *Control* and *Example* address how to accomplish a task using the API. The classifier for *Functionality* can be applied also to Python documentation. The *Environment* classifiers can be used to get information regarding an API usage context. Classifiers for *Quality* and *Non-information* showed encouraging results (AUPRC $\geq$ 70%). The former is relevant to understand API performance, whereas the latter is useful for suggesting information that a developer can ignore. Moreover, the *Non-information* classifier showed promising results generalizing to the Python documentation. Given its particular use case, we suggest research to focus on maximizing recall to ensure that *all* uninformative documents can be tagged appropriately. For the other knowledge types, we suggest maximizing precision to guarantee that fundamental information is correctly tagged.

The results for other knowledge types can be improved by adding NLP-based features. For example, *Structure* usually contains references to other API elements that can be identified using a specific named-entity tagger (e.g., [41]). *Concept* and *Pattern* are strongly characterized by explanations of specific terms and sequence of steps. These can be identified through specialized features based on linguistic inquiry [42], such as *drives* (e.g., "do this to achieve that") and *time orientation* (e.g., "do this, then do that"). As these classifiers showed similar results when applied to the Python documentation, their improvement can also increase their generalizability.

The classifiers showed the worst results (AUPRC < 50%) for *Directive*, *Purpose*, and *Reference* knowledge types. The first two can be the subject of further research. In particular, features for a *Directive* classifier can be extracted from Maalej and Robillard work [3] as well as from the specific taxonomy developed by Montperrus et al. [8]. Furthermore, previous work on rationale mining for other software engineering tasks (e.g., [43, 44]) can be adapted to improve the results for the *Purpose* knowledge type. The *Reference* classifier showed some of the weakest performance but it can be improved with simple syntactical features—e.g., the presence of links.

There is a variation in performance between the classifier configurations (e.g., traditional machine learning vs. deep learning) and between the individual knowledge types. We hypothesize that some knowledge types can be sensitive to specific keywords, such as "callback," "event," and "trigger" in the case of *Control*. On the other hand, knowledge types such as *Environment* and *Example* are characterized by a change in the language context. The former tends to interpolate text with numbers (as it includes information such as version and copyright year), while the latter contains sequences that do not occur in natural language (i.e., source code). We postulate that the RNN can capture this change of context. However, the explanations for some classifiers results are more subtle. For example, *Non-information* implies expressing in natural language information already provided by a method signature. This implies a mapping between source code tokens and natural language ones which need to be further investigated. Similarly, the *Purpose* knowledge type contains information—i.e., the answer to a "why" question—which can be difficult to identify, from a semantic perspective, using the simple configurations of our classifiers. Arguably, the intrinsic difficulty to identify a knowledge type, even for a human expert, can explain some of the poor results. For instance, Maalej and Robillard report low agreement for *Purpose*—a knowledge type showing subpar results (AUPRC = 41%).

Another explanation for the different performance between traditional machine learning and deep learning can lay in the parameters used to tune the latter. A suggested improvement is to create 12 binary RNNs (one for each knowledge type) and select different parameters for a) the activation function of the output layer (e.g., SoftMax [45]), b) the loss function (e.g., Categorical Cross-Entropy [46]), and c) the optimizer (e.g., RMSProp or ADA [47]).

Using knowledge types when developing software. One of the main applications for the classifiers presented in this study is documentation filtering. On top of the current option (e.g., by package or class), API websites can offer their users the possibility to search documentation based on specific knowledge types. For example, a developer fixing a specific performance bug (e.g., related to wireless connectivity) can search the network API documentation containing the *Quality* knowledge type. In this scenario, the classifier can be optimized for precision—i.e., the developer would consult a small number of documents which are likely to contain the information she needs. On the other hand, a developer exploring possible usage of a new set of APIs can filter them according to *Functionality* which describes the their capabilities. In this scenario, the classifier can be optimized for recall—i.e., the developer consults a substantial amount of documents which offer her a complete overview of the API functionalities, even if some may be irrelevant. Our benchmark is a starting point for selecting which classifier to optimize according to specific scenarios. Classifiers with AUPRC $\geq$ 80% can already be utilized the scenarios above. Classifiers with AUPRC $\geq$ 50% need further optimization.

The proposed benchmark is also a stepping stone to support software developers filtering API documentation based on *multiple* knowledge types of interest. Given the complexity of such a task, our best classifier ($RNN_{SO_{API}}$) showed good results (MacroAUC = 79%). However, when disregarding recall—based on the assumption that a developer will not read a large number of documents—the classifier with the highest precision (66%) is $RNN_{CC_{OTF}}$. Conversely, when a developer can tolerate noisy yet comprehensive results, we recommend using RNN_{CC}—i.e., the classifier with the best recall. In both cases, the classifiers rely on "affordable" embeddings.

Using knowledge types when authoring documentation. API documentation providers can leverage the results of this work to monitor their product. For example, they can use simple machine learning models (e.g., SVM) to find documents containing *Non-information* and remove irrelevant text that increase the developers' cognitive effort (e.g., the repetition of a method signature in textual form). Furthermore, they can monitor the presence of knowledge types containing crucial information for software developers, such as *Functionality*, *Control*, and *Example*. API documentation provider can also monitor the decrease of important knowledge types (e.g., *Functionality*) or increase of harmful ones (i.e., *Non-information*) before releasing new version of an API and its documentation. API defects can be diagnosed by identifying (and subsequently improving) documentation containing *Directive* and *Quality*.

Further research outlets. Researchers investigating documentation quality can benefit from the results of our work. For example, quality models can be devised based on the presence (or absence) of specific types. A first step is the identification of knowledge types in a set of documents. In our benchmark, the $RNN_{SO_{API}}$ model showed good results (MacroAUC = 0.79). The classifier correctly identifies documents containing a set of knowledge types with 60% false positives rate when maximizing recall.

Researchers should also consider the trade-off between using a pre-trained embedding while losing some performance (5%) in terms of MacroAUC. Given the results obtained on the Python standard library, we recommend researchers to be careful when applying our multi-label models to different API documentation. Researchers have shown interest in studying how the usage of particular elements in a framework is documented (e.g., [5, 9, 8]). This line of research can benefit from an approach to automatically retrieve API reference documentation containing the *Functionality* knowledge type using the RNN_{SO} model, as it showed good performance on different test sets.

6.2 Threats to Validity

The API reference documentation used to train our classifiers is based on two libraries, JDK and .NET. While the language paradigms are similar, their documentation styles are different [3]. Moreover, we directly addressed a threat to generalizability by investigating the less structured documentation of the Python programming language API [48]. We acknowledge that our results may not hold for API reference documentation in other domains (e.g., for a specific framework) or for a different programming paradigm (e.g., declarative programming). Although Maalej and Robillard taxonomy is general enough [3], other knowledge types may exist.

The labeling of our new test set can introduce a threat to internal validity. To mitigate such threat, two raters independently labeled the documents using validated guidelines [3]. We reconciled the disagreements (approximately 50% were clear mistakes) by discussing borderline cases and reaching consensus among the authors.

Our benchmark only includes two traditional machine learning algorithms, one specific deep learning architecture, and four representations for the RNN input layer. Nevertheless, there may be other algorithms, embeddings, and configurations worth of investigation.

The results can be biased due to the unbalancedness of the dataset. To reduce this threat, we applied common resampling techniques to the training set and reported the performance according to appropriate metrics. We did not observe a correlation between the classifiers performance and the distribution of the labels.

7 Conclusion and Future Work

In this paper, we built several classifiers, using machine learning and deep learning approaches, to automatically identify the 12 knowledge types proposed by Maalej and Robillard [3] in API documentation. We used Java and .NET manually-annotated API documentation (n = 5,574) as a dataset for training and testing the classifiers. We showed good results (i.e., AUPRC $\geq$ 80%) for one-third of the knowledge types. In particular, RNN identifies eight types more accurately than traditional machine learning. When considering multiple knowledge types at once (i.e., multi-label classification), RNN outperforms traditional machine learning approaches.

When word embeddings (i.e., the RNN input layer) are created from StackOverflow posts, rather than from general-purpose text, there is slight to no improvement in performance (ΔAUPRC = 4%). When considering multiple labels, software development-specific embeddings yield better results for MacroAUC (79% vs. 74%).

We applied the classifiers to a new test set (n = 100) obtained from the Python API documentation. Classifiers for *Functionality*, *Concept*, *Purpose*, *Pattern*, and *Directive* generalize to Python. However, the generalization of multi-label classifiers is limited.

Some of the classifiers presented in this work can be already used by practitioners (e.g., developers, API providers) in different application scenarios. Moreover, we propose possible improvements to the classifiers based on features specific for a knowledge type.

For further studies, we plan to evaluate other deep learning and hybrid approaches. Based on our benchmark, we plan to implement a tool (e.g., a browser plugin) to filter API documentation based on knowledge types and evaluate its usefulness with software developers. Finally, our long-term goal is to achieve knowledge identification in specific sentences or paragraphs of API documentation.

References

[1] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2010.

[2] U. Dekel and J. D. Herbsleb, "Improving api documentation usability with knowledge pushing," in *Proceedings of the 31st International Conference on Software Engineering*.   IEEE Computer Society, 2009, pp. 320–330.

[3] W. Maalej and M. P. Robillard, "Patterns of Knowledge in API Reference Documentation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1264–1282, 2013.

[4] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining api types using text classification," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*.   IEEE Press, 2015, pp. 869–879.

[5] M. P. Robillard and Y. B. Chhetri, "Recommending reference API documentation," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1558–1586, Jul. 2014.

[6] J. Stylos, B. Graf, D. K. Busse, C. Ziegler, R. Ehret, and J. Karstens, "A case study of api redesign for improved usability," in *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on.* IEEE, 2008, pp. 189–192.

[7] J. Stylos and B. A. Myers, "The implications of method placement on api learnability," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering.* ACM, 2008, pp. 105–112.

[8] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? An empirical study on the directives of API documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 2011.

[9] M. A. Saied, H. Sahraoui, and B. Dufour, "An observational study on api usage constraints and their documentation," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on.* IEEE, 2015, pp. 33–42.

[10] L. Deng and D. Yu, "Deep learning: methods and applications," *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, pp. 197–387, 2014.

[11] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[12] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[13] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[14] K. Boyd, K. H. Eng, and C. D. Page, "Area under the precision-recall curve: Point estimates and confidence intervals," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases.* Springer, 2013, pp. 451–466.

[15] J. Huang and C. X. Ling, "Using auc and accuracy in evaluating learning algorithms," *IEEE Transactions on knowledge and Data Engineering*, vol. 17, no. 3, pp. 299–310, 2005.

[16] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing & Management*, vol. 45, no. 4, pp. 427–437, 2009.

[17] F. Charte, A. Rivera, M. J. del Jesus, and F. Herrera, "Concurrence among imbalanced labels and its influence on multilabel resampling algorithms," in *International Conference on Hybrid Artificial Intelligence Systems.* Springer, 2014, pp. 110–121.

[18] F. Herrera, F. Charte, A. J. Rivera, and M. J. Del Jesus, "Multilabel classification," in *Multilabel Classification.* Springer, 2016, pp. 17–31.

[19] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation." in *EMNLP*, vol. 14, 2014, pp. 1532–1543.

[20] G. Halawi, G. Dror, E. Gabrilovich, and Y. Koren, "Large-scale learning of word relatedness with constraints," in *KDD.* New York, NY, USA: ACM, 2012, pp. 1406–1414. [Online]. Available: http://doi.acm.org/10.1145/2339530.2339751

[21] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," in *European conference on machine learning.* Springer, 1998, pp. 137–142.

[22] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is "nearest neighbor" meaningful?" in *International conference on database theory.* Springer, 1999, pp. 217–235.

[23] C. A. Cois and R. Kazman, "Natural language processing to quantify security effort in the software development lifecycle." in *SEKE*, 2015, pp. 716–721.

[24] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE).* IEEE, 2012, pp. 837–847.

[25] T. Joachims, "Training linear svms in linear time," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2006, pp. 217–226.

[26] F. Calefato, F. Lanubile, F. Maiorano, and N. Novielli, "Sentiment polarity detection for software development," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1352–1382, 2018.

[27] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* ACM, 2017, pp. 49–60.

[28] Y. Liu and Y. F. Zheng, "One-against-all multi-class svm classification using reliability measures," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 2. IEEE, 2005, pp. 849–854.

[29] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[30] T. M. Cover, P. E. Hart *et al.*, "Nearest neighbor pattern classification," *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.

[31] M.-L. Zhang and Z.-H. Zhou, "Ml-knn: A lazy learning approach to multi-label learning," *Pattern recognition*, vol. 40, no. 7, pp. 2038–2048, 2007.

[32] J. Patterson and A. Gibson, *Deep Learning: A Practitioner's Approach.*   O'Reilly Media, 2017.

[33] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014. [Online]. Available: http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

[34] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[35] J. Nam, J. Kim, E. L. Mencía, I. Gurevych, and J. Fürnkranz, "Large-scale multi-label text classification—revisiting neural networks," in *Joint european conference on machine learning and knowledge discovery in databases*.   Springer, 2014, pp. 437–452.

[36] G. A. Miller and W. G. Charles, "Contextual correlates of semantic similarity," *Language and cognitive processes*, vol. 6, no. 1, pp. 1–28, 1991.

[37] W. Ding, P. Liang, A. Tang, and H. Van Vliet, "Knowledge-based approaches in software documentation: A systematic literature review," *Information and Software Technology*, vol. 56, no. 6, pp. 545–567, 2014.

[38] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *Proceedings of the 34th International Conference on Software Engineering*.   IEEE Press, 2012, pp. 815–825.

[39] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*.   ACM, 2016, pp. 51–62.

[40] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*.   IEEE, 2018, pp. 602–611.

[41] M. Mäntylä, F. Calefato, and M. Claes, "Natural language or not (nlon)-a package for software engineering text analysis pipeline," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*.   IEEE, 2018, pp. 387–391.

[42] Y. R. Tausczik and J. W. Pennebaker, "The psychological meaning of words: Liwc and computerized text analysis methods," *Journal of language and social psychology*, vol. 29, no. 1, pp. 24–54, 2010.

[43] Z. Kurtanović and W. Maalej, "On user rationale in software engineering," *Requirements Engineering*, vol. 23, no. 3, pp. 357–379, 2018.

[44] B. Rogers, J. Gung, Y. Qiao, and J. E. Burge, "Exploring techniques for rationale extraction from existing documents," in *2012 34th international conference on software engineering (ICSE)*.   IEEE, 2012, pp. 1313–1316.

[45] G. E. Hinton and R. R. Salakhutdinov, "Replicated softmax: an undirected topic model," in *Advances in neural information processing systems*, 2009, pp. 1607–1614.

[46] Z. Zhang and M. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," in *Advances in Neural Information Processing Systems*, 2018, pp. 8792–8802.

[47] M. C. Mukkamala and M. Hein, "Variants of rmsprop and adagrad with logarithmic regret bounds," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*.   JMLR. org, 2017, pp. 2545–2553.

[48] B. Dagenais and M. P. Robillard, "Creating and evolving developer documentation: understanding the decisions of open source contributors," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*.   ACM, 2010, pp. 127–136.