A Text Classification Approach to API Type Resolution for Incomplete Code Snippets

Camilo Velázquez-Rodríguez[a], Dario Di Nucci[b], Coen De Roover[a]

[a] *Vrije Universiteit Brussel, Brussels, Belgium*
[b] *University of Salerno, Fisciano (SA), Italy*

Abstract

The Stack Overflow Q&A platform boasts an active community of users who often include code snippets in their questions and answers. Several development tools rely on these code snippets as a source of information. Although code snippets are intended as examples for humans, they may not form compilation units. For instance, snippets illustrating how to use an API might lack the import statements for the corresponding API types. Thus, it becomes essential to determine the fully-qualified name of API types in incomplete snippets.

We present RESICO, a machine learning-based text classification approach to resolving the simple name of API types to their fully-qualified names. RESICO is trained on a corpus of Java programs for which a compiler can determine the fully-qualified names. For four machine learning classifiers, we evaluate the type resolution accuracy of the resulting models on the original and an extended version of datasets of snippets previously used to evaluate the current state-of-the-art approach based on information retrieval. Results show that our approach outperforms the state-of-the-art one, although the training phase is slightly slower. We observe that most of the incorrect type resolutions are not due to ambiguities among the simple names for API types but due to similarities among the contexts in which these types are used, representing a future research challenge.

Keywords: Fully Qualified Name Resolution, Machine Learning, Text Classification, Stack Overflow

1. Introduction

Developers may consult several sources of information online. Stack Overflow (SO) is a platform where users can post questions answered by others with expertise in the domain. SO posts often contain code snippets, e.g., to illustrate

. *Email addresses:* camilo.ernesto.velazquez.rodriguez@vub.be (Camilo Velázquez-Rodríguez), ddinucci@unisa.it (Dario Di Nucci), coen.de.roover@vub.be (Coen De Roover)

how to use the APIs of a library. However, such code snippets may miss the class declarations and package import statements that constitute a compilation unit. They may contain API usages without any syntactic reference to the library that provides the APIs. Even if the library name is mentioned in the text surrounding the code snippet (e.g., Guava), the fully-qualified name (FQN) of the types to import (e.g., `com.google.common.*`) can be challenging to resolve.

Determining which library types to import for a given code snippet is a problem shared by several tools that rely on SO. For instance, SISE [1] and POME [2] overcome the problem through a series of regular expressions. Solutions dedicated solely to this problem have also been proposed:

- Baker [3] takes a *deductive reasoning* approach for which it first populates a database from the JAR files of candidate libraries with semantic information about the provided API elements. For a given ambiguous API name, Baker can return a list of candidate FQNs that satisfy the semantic constraints imposed by later expressions in the snippet. However, the approach often cannot return such a list or provides several candidate names for small or incomplete snippets. StatType [4] and COSTER [5], in contrast, prefer the crowd's wisdom over deductive reasoning.

- StatType [4] treats type resolution as an instance of *statistical machine translation* from snippets with partially-qualified names to snippets with FQNs. FQNs are learned from co-occurrences in a corpus of projects that compile and use the APIs of the same libraries and are refined based on the local context surrounding the name to be resolved. StatType is entirely data-driven yet manages to outperform Baker in the evaluation by Phan et al. Nevertheless, training the underlying models may be computationally expensive.

- COSTER [5] takes an *information retrieval* approach to the problem, for which it first populates a database with the co-occurrence likelihood of tokens and FQNs. Name and context similarities are leveraged to refine the co-occurrence likelihood of candidates returned from the database. This refinement is context-sensitive as, for each token, both the local context of surrounding tokens and a global context of semantically-related usages are considered. In a detailed evaluation [5], COSTER outperforms both StatType and Baker; therefore, we consider it the state of the art.

This paper presents RESICO (RESolution in Incomplete COde), a new learning-based *text classification* approach to the problem of resolving API types in incomplete code snippets. The approach embraces the hypothesis that the API elements used within a snippet and the context in which this usage occurs often suffice to resolve the simple name of a type within the snippet to its fully qualified one. Word2Vec plays a fundamental role in RESICO, where it is used to learn vector representations for the API elements and their usage contexts within a dataset, so relying on a suboptimal predetermined one can be avoided. Once vector representations or word embeddings have been

learnt, RESICO vectorises the dataset and trains a multi-class (i.e., each FQN corresponds to a class) machine learning algorithm on the resulting vectors. RESICO supports using any supervised machine learning classifier to train on the vectorised dataset and considers FQNs as the label to predict. At resolution time, RESICO returns the most likely FQNs for all simple names in a given code snippet, regardless of whether the snippet is incomplete or syntactically incorrect.

Like StatType and COSTER, RESICO is applicable in contexts where the crowd's wisdom is to be preferred over the deductive reasoning of Baker, e.g., when snippets are small and contain few expressions that impose semantic constraints on ambiguous API names. Although RESICO is a machine learning approach such as StatType, it features a different approach to solving the problem. StatType considers API type resolution as a *sequence-to-sequence* task. In contrast, RESICO relies on a classification procedure where a learned context influences the class prediction (i.e., the FQN of the API reference). On the other hand, COSTER considers the surroundings of API references as RESICO, but it does not perform any learning process to improve context comparisons further. We chose COSTER as the baseline to compare to as previous studies showed that it is similar or superior in performance to StatType. Furthermore, despite repeated attempts, we could not obtain an implementation of StatType from the authors.

We evaluate RESICO and COSTER extensively on four datasets: one gathered from a corpus of 50K compilable GitHub projects and three datasets that serve as external validators for the trained models. Our approach is more complex than COSTER since it involves training several machine learning models; hence, it consumes more computational resources during training. Despite being slower to train, RESICO outperforms COSTER in all experiments we conducted. We also performed a root cause analysis of the type resolution failures of the two approaches. More specifically, we measured how many of the failures were due to simple names being ambiguous.

This paper makes the following contributions:

- We propose RESICO, a learning-based text classification approach to the problem of resolving the simple names of API types within a code snippet to the most likely corresponding FQNs. Incomplete and syntactically incorrect code snippets, common on SO, are supported by design. It is sufficient to train RESICO on compilable Java programs or snippets with import statements that qualify the simple names. We instantiate RESICO with four machine learning classifiers previously used in the literature for text classification tasks.

- As a complement to the labelled *COSTER-SO* (401 complete snippets) and *StatType-SO* (245 complete snippets) datasets on which COSTER was evaluated, we contribute a more diverse and more complex labelled dataset (i.e., *RESICO-SO*). The dataset consists of 371 syntactically correct and complete snippets referencing the same libraries considered in previous datasets. The *RESICO-SO* dataset has more unique FQNs to

predict, reinforcing the number of third-party references to predict, which makes it a more challenging and complete dataset than previous proposals. *RESICO-SO* could also be employed for future benchmarking on the topic. We used all these datasets in our evaluation.

- We share the code of RESICO, the datasets, and the RESICO-trained models to resolve the types in incomplete code snippets.[1] The model could be helpful to compare future models and be integrated into tools that need to complete the API information of SO code snippets.

The remainder of this paper is structured as follows. Section 2 motivates the problem of resolving the simple names of API types in Stack Overflow code snippets. Section 3 surveys the learning-based text classification techniques and machine learning classifiers upon which we build. We detail our approach in Section 4, and evaluate its prototype instantiations in Section 5. Section 6 and Section 7 discuss the results and limitations of the proposed approach and its evaluation. We detail the related work in Section 8 before concluding the paper in Section 9.

2. Incomplete Code Snippets on Stack Overflow

```
Objects.toString(gearBox, "")
Objects.toString(id, "")
```

```
Ordering<Map.Entry<Key, Value>> entryOrdering =
  ↪ Ordering.from(valueComparator)
    .onResultOf(new Function<Entry<Key, Value>, Value>() {
    public Value apply(Entry<Key, Value> entry) {
        return entry.getValue();
    }
}).reverse();
```

```
Iterator<?> i = queue.iterator();
...
Object next = i.next();
i.remove();
```

Listing 1: Incomplete code snippets containing different issues related to API resolution.

The code snippets in Listing 1 will serve to motivate the design of RESICO.

1. https://github.com/softwarelanguageslab/resico-paper

The snippets with a black,[2] blue,[3] and olive[4] frame will be referred to, in the order of their appearance, as Code Snippets 1, 2, and 3 respectively. These snippets stem from real-world answers on SO, the source of which is indicated in their respective footnotes.

Incomplete Structures. Java compilers expect compilation units that group class declarations together with import statements. On Q&A platforms, in contrast, users frequently post sequences of standalone Java statements or expressions that do not form a compilation unit. This is the case for all code snippets in Listing 1, which is an issue for compilers and most code analysis tools alike.

Missing Variable Declarations. Another source of incompleteness are references to undeclared variables. This is the case for `gearBox` and `id` in Snippet 1, `valueComparator` in Snippet 2, and `queue` in Snippet 3. The natural language semantics of their names or the Q&A text around them could provide hints about their types, but the information itself is missing from the code.

Missing Import Statements. Similarly, library types might be referenced by their simple name while an explicit `import` statement for the corresponding declaration is missing. `Ordering`, `Map`, `Function`, and `Entry` in the second snippet are examples of library types (e.g., from Google Guava) that are referenced in the snippet without their declaration being imported.

Name Ambiguities. Missing import statements cause another problem typical of incomplete code snippets: name ambiguities. Many libraries could share the same simple name for types and methods, encumbering the resolution of simple name to a fully qualified one. For example, the `Function` name in snippet 2 could be a reference to either of the interfaces `java.util.function.Function` or `com.google.common.base.Function`. This is a problem for both users and tools needing to reuse or reason about the snippets.

Despite their incompleteness, code snippets on Q&A platforms are an important source of information for both developers and tools. To fully realise their potential, a reliable and effective approach to resolving the simple name of API types in a code snippet to their fully qualified names is needed. This paper proposes such an approach, capable of predicting the most likely FQN for incomplete and ambiguous names by leveraging similarities in name usage contexts among other snippets on the Q&A platform.

2. `https://stackoverflow.com/questions/21936577`
3. `https://stackoverflow.com/questions/8897384`
4. `https://stackoverflow.com/questions/2319126`

3. Background

3.1. Text Classification

Text classification [6] is a natural language processing problem that requires assigning a correct label y_i to each set $X_1, X_2, \ldots, X_n$ of tokens that corresponds to a text from a given corpus. For example, a paragraph describing movies, actors, directors and scenes might be tagged with the label "cinema". In contrast, a text about a forecast with snowfall, wind direction, and expected humidity could be labelled as "weather". Supervised machine learning approaches for text classification train a learning model by extracting the relations between the tokens and the labels of a smaller representative set (i.e., the training set) to assign a correct label y'_i to a new set of tokens $X'_1, X'_2, \ldots, X'_n$ (i.e., the test set). Several approaches to training and predicting these labels have been proposed over the years (e.g., [7, 8, 9, 10, 11]).

3.2. Text Transformation

Generally, text classification approaches convert each input text into a structured representation through feature extraction. Representations such as Bag-of-Words (BoW) [7], Term Frequency-Inverse Document Frequency (TF-IDF) [8], and Word Embeddings [12, 13] have been proposed to this end. In all these cases, the outcome is an n-dimensional matrix representing the frequency of words (for BoW and TF-IDF) or the context-embedded vectors (for Word Embeddings). The use of Word Embeddings (e.g., Word2Vec) [12, 13] has become popular due to their ability to capture the semantic relations between words based on the surrounding context. For instance, the words "dog" and "pet" may be related because of the similarity of their contexts.

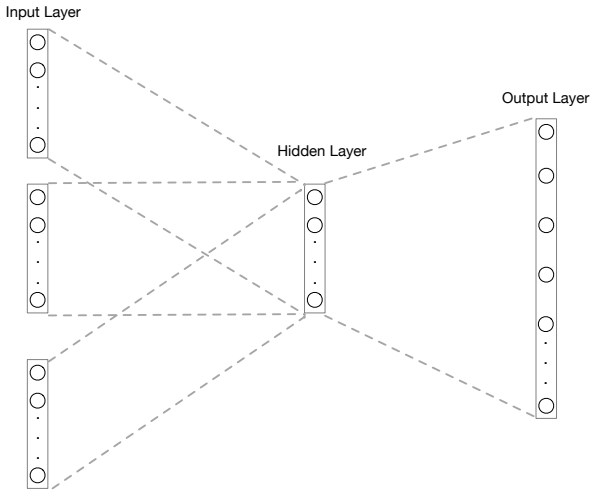

Figure 1: An overview of the CBOW architecture to extract word embeddings from a text corpus.

Two architectures characterise Word2Vec: Skip-gram and Continuous Bag Of Words (CBOW) [12]. The former predicts the context around a word, whereas the latter tries to predict a word given its surrounding context. We use CBOW as it is faster than the alternative Skip-gram [13] while resulting in similar efficiency. Figure 1 depicts an overview of the CBOW model architecture. CBOW starts with a one-hot encoded vector for every target word (Input layer) based on the words that surround it, and trains a single-layer neural network (Hidden layer) to try to infer the target word's vector (Output layer). Once the goal is achieved, or the maximum number of iterations has been reached, the resulting weight vector or embedding for the target word consists of the learnt weights of the hidden layer in the neural network. This vector captures the typical context in which the target word appears. Section 8 discusses recent applications of word embeddings within source code analysis as related work (e.g., [14, 15, 16, 17, 18, 19]).

3.3. ML-based Classification Algorithms

In classification problems, the feature vector resulting from feature extraction is given to a classification algorithm for either training or class prediction. We briefly discuss the ML-based classification algorithms our approach has been instantiated with.

3.3.1. K-Nearest Neighbours

K-Nearest Neighbours (KNN) [20] classifies a new instance by analysing its K closest neighbours in the space of the independent features. If these neighbouring instances belong to different classes, the assigned class will be the most common among the neighbours. KNN is popular because of its fast convergence and good results. Previous work [21, 22, 23] has applied KNN to text classification problems.

3.3.2. Random Forest

Random Forest (RF) [24, 25] belongs to the family of machine learning classifiers categorised as ensemble algorithms. Ensemble classifiers group several machine learning algorithms that are considered efficient yet lightweight. In the particular case of Random Forests, the ensemble consists of several Decision Tree classifiers, hence the term Forest. Each tree is trained with a random selection of features, hence the term random, which turns RF into an unbiased algorithm which can avoid overfitting, especially on imbalanced datasets. At prediction time for a classification task, each of the trees performs a vote on the decision to be taken, and the majority vote will represent the final prediction of the model. The number of decision trees comprising the forest is set using a hyperparameter. Like KNN, Random Forest has already been used with success for text classification tasks [23, 26].

3.3.3. Ridge Linear Classifier

Linear Regression classifiers fit a line to the data that optimally splits the data into two categories or labels. To use them on multi-class datasets, many techniques have been proposed including One-vs-Rest, One-vs-One, and regularised linear classification. The latter has been used with success for text problems [27]. We will therefore instantiate our approach with the Ridge Linear (RL) regressor [28], which first transforms the multi-class data into a multi-output regression problem and then fits one regressor per target label. RL regressors feature a regularisation parameter *alpha* that determines the variance of the estimated weights for the model.[5]

3.3.4. Support Vector Machines

The Support Vector Machines (SVM) classifier [29] searches for an optimal hyperplane that can effectively distinguish the classes based on their features. For example, when the number of features in a dataset is two, SVM will try to find the optimal line dividing the data in a 2D space. If the number of features is three, the division becomes a plane trying to separate a 3D feature space. This concept can be generalised to datasets with n feature dimensions and a hyperplane with $n-1$ dimensions, which require considerable computational resources if n is high. SVMs too have been used with success for text classification problems [30, 31].

4. The RESICO Approach to API Type Resolution

We present RESICO, a learning-based text classification approach to resolve API references in incomplete code snippets to FQNs. Figure 3, Figure 4, and Figure 5 provide a detailed overview of the steps involved in RESICO. Letters denote data resulting from (or given to) a step in a process, while numbers denote the actual steps.

Figure 2 shows examples of the output that can be expected. Lines of code from the input snippet are depicted in black. For this snippet, we removed all import statements from which the FQNs could be resolved. We annotated the API references to be resolved by RESICO with a rectangle. The line in red and italics depicts the computed FQN for each API reference.

In all cases but one, the predicted FQN was correct compared to previously removed import statements. RESICO could precisely resolve the FQN for most API references despite multiple candidates with the same simple name in our dataset. For example, the simple names `Configuration`, `SessionFactory`, `Session` and `Transaction` ambiguously denote 44, 4, 67 and 14 different FQNs respectively in one of our datasets.

For the simple name not annotated (i.e., a dash appears instead), RESICO could not correctly predict the FQN of the variable `c`, which was correctly

5. `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RidgeClassifier.html`

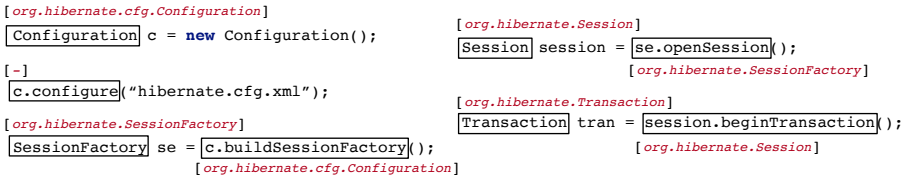

Figure 2: Example type resolutions computed by RESICO.

predicted for the line above (i.e., `Configuration`). However, if all predicted FQNs are considered import statements, the missing FQN will not represent an issue. Before detailing how RESICO resolves simple names to their FQN, we briefly describe the Eclipse JDT as it features prominently in the remainder of the section.

4.1. An Overview of Eclipse JDT for Fact Extraction

Eclipse Java Development Tools (JDT)[6] is a set of plugins developed for the Eclipse platform that enable lightweight static analysis of Java programs. The Eclipse JDT comprises five components: APT, Core, Debug, Text, and UI. Each component is independent, and specialised in a different purpose. For example, COSTER and RESICO use the Eclipse JDT Core component for their fact extraction from compilable Java programs (e.g., information about method invocations and variable declarations). It can be used headless without the Eclipse IDE and provides, among others, Abstract Syntax Trees (ASTs) and symbol and type hierarchy information for Java programs. As we use the Eclipse JDT Core component frequently and none of the other components, we will refer to the former as Eclipse JDT from now onwards.

4.2. Training Process

Figure 3 depicts the steps in the training process. This section will use the snippet depicted in Listing 2 as a running example.

In the first step of the training process, depicted as action node 1, the Eclipse JDT is used to i) compile the Java programs in the training corpus, and ii) extract the simple names of API references and their FQNs as follows:

- for variable declarations, the simple name and FQN of the declared type.
- for variable and field accesses, the simple name of the accessed type, the name of the field and the FQN of the accessed type.
- for method invocations, the simple name and FQN of the statically-declared type of the receiver expression, and the identifier of the invoked method.

6. `https://www.eclipse.org/jdt/`

```
1  import org.apache.cordova.DroidGap;
2  import android.context.Context;
3  import android.telephony.TelephonyManager;
4  import android.webkit.WebView;
5  import android.webkit.JavascriptInterface;
6
7  public class GetNativeTelephonyManager {
8      private WebView mAppView;
9      private DroidGap mGap;
10
11     public GetNativeTelephonyManager(DroidGap gap, WebView view) {
12         mGap = gap;
13         mAppView = view;
14     }
15     @JavascriptInterface
16     public String getIMEI() {
17         TelephonyManager tm = (TelephonyManager)
           ↪  mGap.getSystemService(Context.SERVICE);
18         String imeiID = tm.getDeviceId();
19         return imeiID;
20     }
21 }
```

Listing 2: Running example for explaining RESICO.

- the line numbers for each previous construct.

The information provided by the JDT is used to perform this extraction step. For example, for the method invocation on Line 18 of Listing 2, RESICO first collects `TelephonyManager` (the simple name or type of the receiver variable `tm`), `getDeviceId` (the method identifier), and `android.telephony.TelephonyManager` (the FQN of the receiver type). The latter represents the label to predict by RESICO, whereas the former is part of the training data.

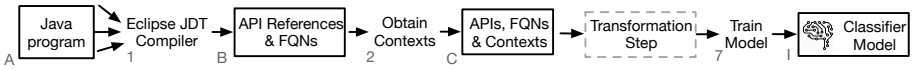

Figure 3: Training based on a corpus of programs.

Figure 3 shows the information collected for each program as data node *B*. The gathered data so far is further augmented with the context surrounding the API reference. We define the context of an API reference as the information extracted for all other API references in the same method body but without their FQNs. In this way, RESICO can extract the same information from complete and incomplete code snippets, since the FQNs are used neither in the API reference nor in the contexts. Finally, RESICO collects the vocabularies for both API references and contexts which will serve as input to train Word2Vec models later in the process. The vocabulary for contexts is obtained by adding all contexts into a single set of words. The vocabulary for the API references comprises the set of API references themselves.

To continue with our running example, from the API reference `tm.getDeviceID` on Line 18, its surrounding context is the following:

`TelephonyManager, DroidGap, getSystemService, Context, SERVICE, String, String`

Within Figure 3, the context extracted for each API reference is depicted as data node C in addition to the previously collected information. Table 1 shows for every API reference (column *API Ref.*), the line numbers where they are located (column #), their contexts (column *Context*) and their respective FQNs (column *FQN*) that RESICO is able to extract from the method `getIMEI()` from Listing 2. As observed, many API references could be present on the same line, enlarging the collected dataset even in small methods such as `getIMEI()`.

#	API Ref.	Context	FQN
17	`TelephonyManager`	`DroidGap,getSystemService,Context,SERVICE, String,TelephonyManager,getDeviceID,String`	`android.telephony.TelephonyManager`
17	`DroidGap,getSystemService`	`TelephonyManager,Context,SERVICE, String,TelephonyManager,getDeviceID,String`	`org.apache.cordova.DroidGap`
17	`Context,SERVICE`	`TelephonyManager,DroidGap,getSystemService, String,TelephonyManager,getDeviceID,String`	`org.android.Context`
18	`String`	`TelephonyManager,DroidGap,getSystemService,Context, SERVICE,TelephonyManager,getDeviceID,String`	`java.lang.String`
18	`TelephonyManager,getDeviceID`	`TelephonyManager,DroidGap,getSystemService, Context,SERVICE,String,String`	`android.telephony.TelephonyManager`
19	`String`	`TelephonyManager,DroidGap,getSystemService,Context, SERVICE,String,TelephonyManager,getDeviceID`	`java.lang.String`

Table 1: Extracted information from the method `getIMEI()` in Listing 2 by RESICO.

Figure 4 zooms in on the following transformation step. We use Word2Vec [12, 13] to vectorise API references and contexts, whereas label categorisation is used to convert FQNs to label numbers.

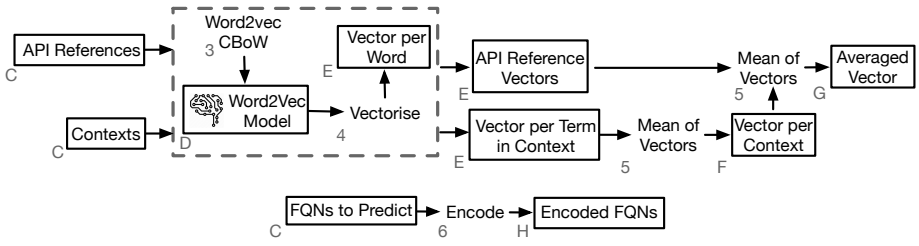

Figure 4: Transformation step used by the training process.

Word2Vec is used to learn the best possible vector representation of APIs and contexts (columns *API Ref.* and *Context* of Table 1) and obtain a word embedding per word in both cases. Each API reference is considered as a single word. API reference records with two words (e.g., `DroidGap`, `getSystemService` on line 17) are concatenated with a dot (e.g., '.') as it is done for a field or method call. However, words in context records are kept separate and considered individually in the training phase. This decision was made because API references

can take the form of variable declarations as well as method calls and field accesses, and the latter only comprise a single relevant word (i.e., the name of the API type) while the latter comprise two (i.e., the field or method's name along with the name of the API type). Contexts, in contrast, comprise the simple names stemming from surrounding API references and each word is therefore considered equally relevant.

Step 3 iterates once over all extracted API references and contexts while using the Word2Vec CBOW neural network architecture to train the algorithm (cf. Section 3.2). We used the implementation of the Word2Vec model in the Golang programming language[7] to take advantage of its Goroutines. We configured the Word2Vec training for the APIs and contexts similarly, relying on the CBOW architecture, a batch size of 1000 words, 20 generated features, negative sampling as the optimiser, five words of window size, a minimum occurrence of a word equals to 1 (i.e., considering all words), and five iterations for the training of the model. The number of features was set to 20 to enable us to perform the following steps (e.g., hyperparameter optimisation) more efficiently without reducing the ability to capture the semantic differences between the labels to predict. The number of iterations for the learning was set to five to keep the training process as simple as possible. The batch size considers numerous words per training batch with a window size of five to take short segments around the word to learn. The latter is also the default window size for the Word2Vec implementation in the Python library Gensim as it is the selection of negative sampling as optimiser.[8] No other changes specific to the domain were made.

The resulting models or word embeddings were saved in external files and are depicted as the data node D, one for the API references and another one for the contexts. Once the word embeddings have been learned, step 4 transforms the input data into the learned vectors. The vectors obtained for each word (data node E) are kept in the case of the API references and further averaged in the case of contexts in step 5. Data node F depicts the averaged vectors per context.[9]

To illustrate the transformation described above, Table 2 depicts the vector representation of the information in Table 1. In this case, column *API Vector* denotes a transformed API reference, whereas column *Cont. Vect.* denotes the corresponding vector for each word in its context. The averaged vector of all vectors in a context is shown in column *Av. Cont. Vect.*. This vector is stored in data node F, which will be used in the following steps. The column *FQN* remains the same, and it will be transformed in a subsequent step.

A final averaged vector is obtained in data node G by considering each vector of API references and its corresponding averaged context vector. In parallel to the vectorisation, step 6 encodes the FQNs collected in data node B. A natural number is assigned to each FQN, denoting the label to be predicted by the

7. `https://github.com/ynqa/wego`
8. `https://radimrehurek.com/gensim/models/word2vec.html`
9. Please remember that contexts are composed of several words

#	API Vector	Cont. Vect.	Av. Cont. Vect.	FQN
17	[0.684, ..., 0.457]	[-0.158, ..., -0.378], ..., [-0.728, ..., -0.629]	[-0.332, ..., 0.643]	`android.telephony.TelephonyManager`
17	[-0.154, ..., 0.254]	[0.029, ..., -0.916], ..., [0.904, ..., 0.601]	[0.378, ..., -0.173]	`org.apache.cordova.DroidGap`
17	[-0.804, ..., 0.915]	[0.807, ..., 0.092] ..., [0.277, ..., 0.592]	[0.155, ..., 0.791]	`org.android.Context`
18	[0.218, ..., 0.613]	[0.167, ..., -0.805], ..., [0.767, ..., 0.466]	[-0.464, ..., 0.608]	`java.lang.String`
18	[0.349, ..., 0.505]	[0.092, ..., 0.397], ..., [-0.336, ..., 0.719]	[0.047, ..., 0.927]	`android.telephony.TelephonyManager`
19	[0.218, ..., 0.613]	[0.167, ..., -0.805], ..., [0.767, ..., 0.466]	[-0.464, ..., 0.608]	`java.lang.String`

Table 2: Transformed API references and contexts from Table 1 by the Word2Vec models.

classifier. Table 3 exemplifies this last transformation step for our dataset. Column *Av. Embedding Vector* contains the final averaged vector, and column *FQN* contains the transformed FQNs as numbered labels. In machine learning concepts, the former is the training dataset, while the latter corresponds to the class to predict.

#	Av. Embedding Vector	FQN
17	[-0.756, ..., 0.628]	0
17	[0.118, ..., 0.112]	1
17	[0.860, ..., -0.145]	2
18	[0.253, ..., 0.183]	3
18	[0.092, ..., -0.792]	0
19	[0.253, ..., 0.183]	3

Table 3: The last transformation step in the RESICO process. The previous context vector is further averaged with the API vector, and FQNs are converted into numbers.

The last step in the training process (action node 7) is responsible for training a supervised machine learning classifier (data node I) and for saving the resulting model into a file. RESICO supports the use of any supervised machine learning classifier with the data extracted at this point. For our experiments, we selected four machine learning classifiers previously used in text classification tasks: KNN, Random Forests, Ridge Linear Classifier, and Support Vector Machines (cf. Section 3). The outcome of the training process is a model to predict a numbered FQN given an averaged vector. The averaged vector is the mean of a vector A for an API reference and a vector B which is the mean vector of all vectors in the surrounding context of the mentioned API reference.

4.3. Resolution Process

The resolution process for code snippets depicted in Figure 5 resolves all API type references within a code snippet to their FQNs. The process has to

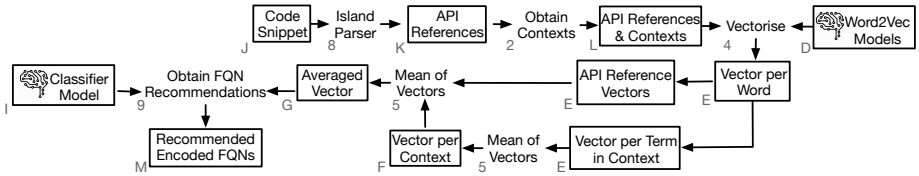

Figure 5: Resolution process for API type references in code snippets.

parse a snippet in a *non-standard* way due to the many issues a SO code might have (cf. Section 2). RESICO uses a custom Island Parser [32] (action node 8) configured to parse SO code snippets regardless of their syntactical correctness. Such parsers focus on the constructs of interest (i.e., API references in our case) and consider the remainder of the code as water. It is implemented using the framework *Parboiled*[10] for Java.

The statements are parsed using this parser. API references are gathered in data node K, for which our supervised machine learning models will predict missing FQNs. The contexts of API references are also collected similarly as previously described for a complete code example.

After gathering API references and their respective contexts in an incomplete code snippet, RESICO will transform the input to make it suitable for the resolution phase. The vectorisation of API references and their contexts relies on the formerly trained Word2Vec models saved in files for APIs and contexts (data node D), respectively. The availability of each API reference word and its context words are checked in the trained Word2Vec models. If the API reference word cannot be found (i.e., the API Word2Vec model was not trained with it), the process stops since API references are fundamental for the prediction. However, if one or multiple context words cannot be found (i.e., the context Word2Vec model was not trained with them), the resolution process continues with other context words as this scenario is more likely to happen, and contexts are usually composed of several words. Nonetheless, if there are no context words, RESICO will also likely fail since in this case, it only depends on the API reference word.

Each vector is processed to obtain an averaged vector that will serve as input to the trained models in data node I. These trained models are loaded from the previously saved files. Classifiers have different ways of predicting the class for an unseen input; for example, KNN calculates the closest neighbours and takes the majority class among them. The outcome of the machine learning model therefore depends on how the classifier calculates the most likely class for its final prediction. In general, their output takes the form of a number that corresponds to a particular FQN and needs to be mapped back to the original FQN.

10. `https://github.com/sirthias/parboiled`

4.3.1. Providing Top-K Recommendations

Most supervised machine learning classifiers predict the class with the highest probability for a certain input vector. This probability is calculated based on the similarity of the input vector to already trained vectors in the model. By default, trained models make a Top-1 prediction, returning the class that achieved the highest probability for a certain input vector. However, it is common for implementations to provide access to the internal probabilities for all of the predicteded classes. For example, in the Python Scikit-Learn framework, method `predict_proba`[11] is provided by most implementations. RESICO will therefore return a Top-K with $1 \leq K \leq N\_Classes$ for its resolutions of API type references. If the actual FQN of the reference is among the Top-K resolutions returned by RESICO, the resolution is considered successful; otherwise, the resolution represents a failure of the trained model.

5. Evaluation

We describe the design and results of the empirical evaluation conducted to assess the RESICO machine learning classifiers (from now onwards, RESICO classifiers) for API type resolution. The evaluation compares our approach to COSTER [5], an information retrieval-based approach, which outperforms in several circumstances earlier methods such as StatType [4] and BAKER [3]. **Please note that we contacted COSTER's authors to verify that we configured and used the tool correctly before conducting our evaluation.**[12]
Our study aims to answer the following research questions:

RQ$_1$ What are the best hyperparameter combinations for the classifiers used within RESICO?

RQ$_2$ How well do COSTER and the RESICO classifiers perform on instances extracted from the dataset used for training?

RQ$_3$ What are the performance of COSTER and RESICO classifiers when evaluated on unseen datasets?

RQ$_4$ How much time is needed to train COSTER and the RESICO classifiers?

RQ$_5$ To what extent do ambiguities in simple names influence the performance of the approaches?

Figure 6 depicts a graphical overview of the steps we took to answer these research questions. Our evaluation started by gathering the datasets needed to train and evaluate RESICO and COSTER. We collected one dataset to train

11. `https://github.com/scikit-learn/scikit-learn/blob/f3f51f9b6/sklearn/neighbors/_classification.py#L256`
12. Boldface added for the reviewing process only.

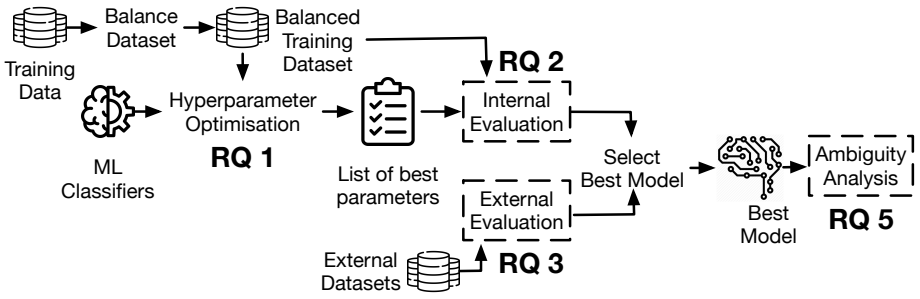

Figure 6: Overview of our evaluation approach.

and evaluate the models, and three additional datasets to analyse the prediction capabilities of the trained models. Given the imbalanced nature of the dataset used to train the models, we applied data balancing. The same balanced dataset is also used to optimise the hyperparameters of the RESICO classifiers in *RQ1*. In *RQ2*, we use the best configuration of hyperparameters for each RESICO classifier to conduct an internal evaluation with the balanced dataset and compare our approach against COSTER. We also perform an external evaluation in *RQ3* to assess the performance of the two approaches on other datasets. For *RQ4*, we present the performance results in terms of training time required by each approach. Finally, we perform an ambiguity analysis in *RQ5* on the best RESICO and COSTER models.

5.1. Datasets Collection

In this section, we report on the datasets used in our empirical study: the dataset used to train and tune models and to evaluate their performance on similar data in *RQ2*, and the three datasets used to evaluate the obtained models on different data in *RQ3*.

5.1.1. Internal Dataset

We relied on the 50K-C dataset to answer *RQ1*, *RQ2* and train the models (i.e., COSTER and all RESICO classifiers) that are further evaluated on external datasets in *RQ3*. The dataset was extracted from a collection of 50K compilable Java projects mined from GitHub [33]. We followed the same extraction process previously used by Saifullah et al. [5] for COSTER, using the same Eclipse JDT extractor configuration for the internal datasets of the two approaches. We did it this way to ensure that the internal datasets where COSTER and RESICO are trained and evaluated were constructed with the same API references and to avoid bias in the evaluation towards either approach. After the API references are extracted, we build the surrounding contexts for each and store the API reference, its context and its corresponding FQN.

We consider not only the 100 most frequent libraries as reported in Saifullah et al. [5] but all 5,356 libraries provided by the 50K-C dataset. From an initial pool of 50,000 projects, we extracted the API references, contexts, and FQNs

for 48,951 (i.e., 98%). The resulting dataset contains 19,088,813 records, each representing an API reference.

Data Balancing. The dataset gathered in the previous step is highly imbalanced. As can be expected, it contains more instances related to some commonly used FQNs (e.g., `java.lang.String` with more than 2M occurrences) and lacks instances of some rarely used FQNs (e.g., `java.sql.Statement[]` with only one occurrence). Figure 7 shows a fragment of the data distribution for the three most and three least frequent types out of the 39,643 unique FQNs in the dataset.

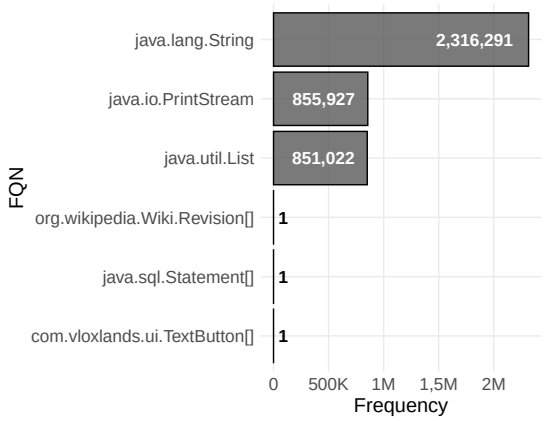

Figure 7: The three most and least frequent FQNs in the gathered dataset.

In this highly unbalanced setting, we decided to sample the previously extracted dataset to balance the training data and, thus, not to introduce a bias towards any particular FQN to be predicted. We selected a threshold of 50 occurrences, as this threshold was previously selected in COSTER [5]. The FQNs with fewer occurrences than the defined threshold are not considered for the training phase and are therefore excluded. Those records with more FQNs than the threshold are randomly sampled into 50 instances. The resulting balanced dataset consists of 4,860 unique FQNs with 50 instances per FQN, amounting to 243,000 records. The new balanced dataset constitutes the internal training dataset, and it will be also used for the internal evaluation of both approaches.

5.1.2. External Datasets

Three external datasets are used to increase the generalisability of the results and to answer *RQ2*. Table 4 shows the characteristics of the datasets considered for the evaluation. Eclipse JDT can parse and compile the snippets in these datasets to (i) extract the referenced API types and their surrounding contexts and (ii) use their FQNs as ground truth.

Two of the datasets *COSTER-SO* [5] and *StatType-SO* [4] contain 401 and 245 code snippets respectively (col. *Snippets*), which have been previously used

Dataset	Snippets	Fs	I-Fs	E-Fs	U-Fs	UI-Fs	UE-Fs
COSTER-SO	401	1,373	1,330	43	30	20	10
StatType-SO	245	1,827	431	1,396	167	39	128
RESICO-SO	371	1,741	596	1,145	215	47	168

Table 4: Datasets used for the external evaluation of COSTER and RESICO.

to assess COSTER.[13] After processing the code snippets, the number of records extracted from *COSTER-SO* and *StatType-SO* were 1,373 and 1,827 respectively (col.*Fs*), including API references, their contexts and FQNs.

A closer look at the FQNs of *COSTER-SO* raises concerns about their distribution in the dataset. 1,330 out of the 1,373 references (96.7%) belong to the Java standard library (col.*I-Fs*) (i.e., the default Java Runtime Environment). Simple names with FQN prefixes starting with `java.lang` or `java.io` are considered as always observable[14] and thus, less significant for the potential users of the approaches. Therefore, a more diverse dataset with a prevalent number of non-default FQNs is desirable.

Alternatively, the *StatType-SO* dataset contains an increased number of external FQNs (76.4%) (col.*E-Fs*) w.r.t *COSTER-SO*. Such a behaviour is also reflected in the number of unique external FQNs (col. *UE-Fs*) with only 10 for *COSTER-SO* and 128 for *StatType-SO*. To further increase the evaluation setup, we created another dataset (*RESICO-SO*) that replicates the distribution of FQNs in the *StatType-SO* dataset and possibly improves the number of unique external FQNs compared to previous datasets. This new dataset represents the third dataset considered to verify the generalisability of the results.

We randomly selected 371 code snippets from Stack Overflow referencing the same 11 libraries as *COSTER-SO* and *StatType-SO*. The 371 code snippets represent a statistically significant sample from the 11,047 Java code snippets with import statements in the SOTorrent dataset [34] dated March 15^{th}, 2020 (95% Confidence Level and 5% Confidence Interval). We manually ensured that the snippets could be parsed and compiled using Eclipse JDT. The new dataset, named *RESICO-SO*, comprises 1,741 API references, their contexts, and FQNs. Most records (1,145 i.e., 65.8%) are references to external FQNs. At the same time, *RESICO-SO* exhibits a more extensive number of unique FQNs (215) with a larger number of unique internal (47) and external (168) FQNs. These aspects make the new dataset more challenging to predict than previous ones and might reinforce the results obtained for the *StatType-SO* with which it shares a similar FQN distribution.

13. `https://zenodo.org/record/7244690`
14. `https://docs.oracle.com/javase/specs/jls/se8/html/jls-7.html#jls-7.4.3`

5.2. *RQ1. What are the best hyperparameter combinations for the classifiers used within RESICO?*

Design. This research question investigates the best hyperparameter configuration on the selected machine learning classifiers for the RESICO approach (cf. Section 3).

All considered classifiers have default hyperparameters in their implementation. These default parameters should have an average performance in various applications and are not optimised for any particular task or dataset. Each classifier has a different set of hyperparameters with a range of possible values to choose from, representing a *search space*.

The search space for some hyperparameters is strictly limited to a group of options in a list, e.g., the hyperparameter *weights* for the KNN classifier restricts the possibilities to *'uniform'* and *'distance'* to calculate the space between neighbours.[15] In other cases, this limitation does not exist; therefore, the search space for such hyperparameters could be infinite. For instance, the parameter *alpha* of the Ridge linear classifier[16] can take any possible float as value. Additionally, a set of hyperparameters usually consists of more than one parameter, rendering searching for the best parameters a multi-objective optimisation problem.

We relied on the optimisation libraries Optuna[17] and HyperOpt[18] to perform the multi-objective search. Optuna was used for those machine learning algorithms that do not heavily demand computer resources, such as KNN, the Ridge linear classifier (RL) and the linear Support Vector Classifier (SVC). We took advantage of the distributed hyperparameter optimisation of HyperOpt to search for the best parameters for the Random Forest classifier without incurring memory overflow issues. The setup of the two libraries was similar, with 200 trials for each classifier and using the Tree of Parzen Estimators (TPE) [35]. Using TPE is recommended over other search space algorithms such as Random Search [36].

In addition to the similar setup, we also defined a similar goal for each search. More specifically, the goal was to minimise a loss function defined as $1-F1$. In other words, the search for the best hyperparameter tried to minimise the difference between the maximum $F1$ score (e.g., 1) and the obtained score. When the difference reached a minimum, the optimal parameters were found.

Table 5 shows the selected machine learning classifiers, their hyperparameters, a brief description, and their configured search space.

Note that we had to bound some hyperparameters with unbounded limits to sufficiently large limits when tuning the classifiers. Also, note that some intervals are floating point ranges while others consist of integer numbers. When

15. `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html`
16. `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RidgeClassifier.html`
17. `https://optuna.org/`
18. `https://hyperopt.github.io/hyperopt/`

Classifier	Hyperparameter	Brief description	Search Space
K-Nearest Neighbors (KNN)	*n_neighbours*	Number of neighbours.	$2 \leq x \leq 1E3$
	weights	Weight function.	['uniform', 'distance']
	algorithm	Algorithm to compute the nearest neighbours.	['ball_tree', 'kd_tree', 'brute']
	leaf_size	Leaf size passed to BallTree or KDTree.	$2 \leq x \leq 1E3$
Random Forest (RF)	*n_estimators*	Number of trees in the forest.	$10 \leq x \leq 200$
	criterion	Function to measure the quality of a split.	['gini', 'entropy']
	min_samples_leaf	Minimum number of samples to be at a leaf node.	$1 \leq x \leq 20$
	min_samples_split	Minimum number of samples to split an internal node.	$0.0 \leq x \leq 1.0$
Ridge Linear (RL)	*alpha*	Regularization strength.	$1.0 \leq x \leq 1E10$
	solver	Solver to use to compute the Ridge coefficients.	['auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga']
Support Vector Classifier (SVC)	*C*	Regularization parameter.	$1.0 \leq x \leq 1E10$

Table 5: Hyperparameters of the classifiers and their search space configuration

the limits are shown as floats in Table 5, the range of possible values belongs to the former case, whereas integer values indicate the latter case.

Results. The overview of the hyperparameter optimisation process for the classifiers is shown in Figure 8. Categorical parameters are encoded as numbers.

For example, for KNN, the values of the hyperparameter *algorithm* are transformed as follows $ball\_tree \rightarrow 0, brute \rightarrow 1, kd\_tree \rightarrow 2$ and the values of the hyperparameter *weights* are converted as $distance \rightarrow 0, uniform \rightarrow 1$. For the RidgeLinear classifier the values of *solver* are changed to $auto \rightarrow 0, cholesky \rightarrow 1, lsqr \rightarrow 2, sag \rightarrow 3, saga \rightarrow 4, sparse\_cg \rightarrow 5, svd \rightarrow 6$. Lastly, the values of the RF hyperparameter *criterion* are transformed to $gini \rightarrow 0, entropy \rightarrow 1$.

Figure 8 highlights in blue those parameter values resulting in the lowest and therefore most optimal loss after 200 trials. In some cases, the figure indicates a convergence towards a particular parameter for the best result, such as *weights* for the value 0 (i.e.,'distance') in the K-Nearest Neighbors classifier. Other best hyperparameters have a majority indicating the likely best selection as is for the value 2 (i.e., kd_tree) for the parameter *algorithm* in KNN. Nonetheless, there are some cases where value changes in the parameter do not seem to influence the loss. Examples of the former are *solver* in the Ridge Linear classifier, where

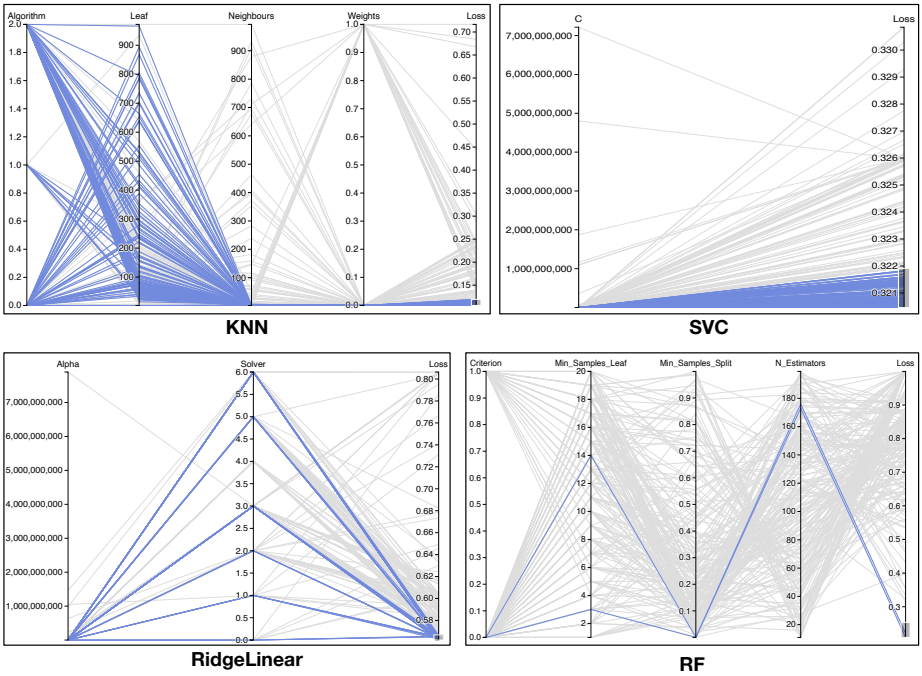

Figure 8: Hyperparameter optimisation for the classifiers considered in RESICO.

all optimal losses contain values from all possible solvers and C in the Support Vector Classifier, where the loss does not improve significantly regardless of the selected hyperparameter value.

The best hyperparameter configuration overall for each classifier is:

KNN $n\_neighbours = 2$, $weights = distance$, $algorithm = kd\_tree$, $leaf\_size = 63$.

RF $n\_estimators = 173$, $criterion = gini$, $min\_samples\_leaf = 14$, $min\_samples\_split = 3E-4$.

RL $alpha = 6473.18$, $solver = sag$.

SVC $C = 11.14$.

We will use the optimal values for each hyperparameter for training the models that will be evaluated in the next research questions. The optimised parameters allow us to obtain models tailored to predict FQNs from incomplete code snippets.

The machine learning algorithms employed within RESICO provide default values for hyperparameters that might be suboptimal for the problem at hand. However, several parameters can be tuned effectively, while the values for others do not influence the loss minimisation. The optimal values resulting from the hyperparameter optimisation are used to train the models used in the remaining research questions.

5.3. RQ2. How well do COSTER and the RESICO classifiers perform on instances extracted from the dataset used for training?

Design. This research question investigates the performance of COSTER and each of the RESICO classifiers on the internal dataset described in Section 5.1.

To this end, we trained and evaluated the approaches using a 10-fold cross-validation technique [37], where nine folds are considered for training, and the remaining fold is considered for evaluation. This process iterates over each of the folds until they are all evaluated. Furthermore, we adopted a more reliable partition technique called stratified cross-fold validation [38]. This technique improves the folding partition of the data by ensuring that each fold contains approximately the same distribution of labels to predict. In such a way, the training and evaluation processes avoid (i) training a label without an evaluation and (ii) evaluating a label not part of the training data. We use the implementation of the stratified cross-validation technique provided by the Scikit-learn library of Python.[19]

In the case of COSTER, we use its implementation for training and evaluation. We kept the configuration of COSTER's parameters as provided, only disabling the parameter concerning the minimum number of required contexts (named `fqnThreshold` in the command-line options). This parameter is set to 50 by default and heavily influences the selection of FQNs, establishing a high mark many FQNs cannot reach. These instances would be consequently excluded from the training and evaluation processes; therefore, we set COSTER to consider all FQNs in the balanced dataset described in Section 5.1. We configure RESICO to use the machine learning classifiers described in previous sections with the optimised hyperparameters obtained in Section 5.2.

For each evaluated fold, we queried the Top-K predictions related to the most likely FQN with K equals to 1, 3, and 5. If the actual value is among the Top-K predictions, we counted the prediction as a success; otherwise, as a failure. This process allowed us to build sets of actual and predicted instances for all top predictions per fold.

Once the prediction data is gathered for each fold, we evaluate the performance of COSTER and each RESICO-trained model using Precision, Recall and F1-Score for each Top-K. For each FQN_A, we define precision as the number of correctly predicted FQNs out of the number of predicted FQN_A (Equation (5)).

19. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html`

Similarly, we define recall as the number of correctly predicted FQN_A out of the number of actual FQN_A (Equation (2)). Lastly, for each FQN_A, its F1-Score is the harmonic mean of both Precision and Recall defined in Equation (3).

$$Precision = \frac{\#\text{ correctly predicted } FQN_A}{\text{total }\#\text{ predicted } FQN_A} \tag{1}$$

$$Recall = \frac{\#\text{ correctly predicted } FQN_A}{\text{total }\#\text{ actual } FQN_A} \tag{2}$$

$$F1\text{-}Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{3}$$

Since our labels are multi-class, we adopted the "micro" averaging approach, which is recommended for multi-class problems when the focus is on the general performance instead of rare classes [39]. Micro-average precision and recall are computed as follows, whereas the micro-average F1-Score is computed similarly to Equation (3) but taking into account micro-average precision and recall.

$$\text{Micro-average Precision} = \frac{\sum_{i=1}^{k} \#\text{ correctly predicted } FQN_i}{\text{total }\#\text{ predicted } FQNs} \tag{4}$$

$$\text{Micro-average Recall} = \frac{\sum_{i=1}^{k} \#\text{ correctly predicted } FQN_i}{\text{total }\#\text{ actual } FQNs} \tag{5}$$

Finally, we average the metrics for all folds and report them.

Results. Figure 9 shows the averaged results for the trained models on the internal dataset for the Top-1, 3, and 5 predictions. Each row of bar charts represents the Top-K evaluation on the internal dataset for a particular value of K. The individual bars within a bar chart depict the performance of the particular classifier denoted on the horizontal axis. A cell (i, j) on the intersection of row i and column j denotes the performance of the classifier j considering the Top-K in i. We selected F1-Scores as performance comparison metric as they represent the harmonic mean between precision and recall; thus, we can observe the average performance of classifiers.

The performance of the two approaches increases along with the number of provided recommendations (i.e., K). In the specific case of COSTER, it starts with a reported Top-1 performance of 58% according to the F1-Score. This performance increases substantially (11% more) for the Top-3, whereas it grows only a 4% from the Top-3 to the Top-5 recommendations. Nonetheless, we note that overall, COSTER performs well, predicting most of the FQNs for this internal dataset.

Most RESICO classifiers (3 out of 4) outperform COSTER in all Top-K configurations. The KNN classifier performs best in all three Top-K, starting with an excellent performance of 90% for the Top-1 and slightly improving 2% in the remaining cases. The RF classifier ranked second with good scores, especially in Top-3 and Top-5, where it can correctly predict 88% and 90%

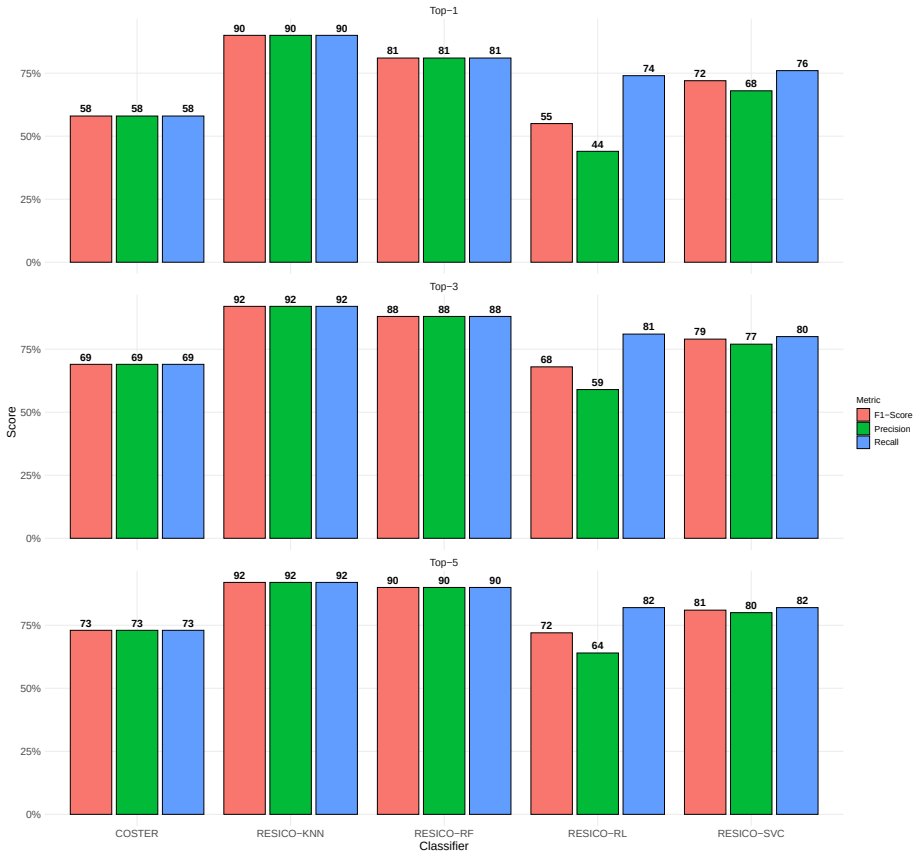

Figure 9: Performance of the models on the internal dataset.

of the FQNs, respectively. In the case of SVC, lower scores were achieved w.r.t. the previous classifiers but still superior to COSTER's scores. Lastly, the RL classifier was the least performing classifier of all considered approaches. However, it is worth noting that the slight F1-Score differences compared to COSTER are due to the low precision of the RL since it performs similarly to SVC in terms of recall.

We conclude that the best classifier of our approach (i.e., KNN) is more effective than COSTER on this dataset. The API references and their surrounding contexts were correctly captured and learned by the Word2Vec and classification processes allowing to improve the prediction results of the RESICO classifiers.

COSTER achieves good performance on the balanced internal dataset with a maximum of 73% for the Top-5 recommendations. RESICO presents excellent results, with the best Top-1 recommendations of 90% and Top-5 of 92% for the KNN classifier, thus, outperforming COSTER on the internal dataset.

5.4. RQ3. What are the performance of COSTER and RESICO classifiers when evaluated on unseen datasets?

Design. This research question investigates the performance of COSTER and all RESICO models previously trained on the internal dataset and applied to external and unseen datasets.

We trained the approaches on the full version of the balanced internal dataset to answer this research question. Once the models are trained on the data, we use them to predict the extracted types from the external datasets *COSTER-SO*, *StatType-SO*, and *RESICO-SO*. Please note that the snippets in the datasets must be compilable because a ground truth of FQNs is needed to verify the effectiveness of the trained models. FQNs of API references are challenging to determine in incomplete code snippets, hindering obtaining the ground truth needed for an accurate evaluation of the models.

All snippets in the three datasets are compilable; hence, information about API references, their surrounding contexts, and FQNs can be extracted using Eclipse JDT. For each API reference, their specific context is extracted either using the configuration of COSTER (cf. Saifullah et al. [5]) or RESICO (cf. Section 4). When leveraging RESICO, the extracted information has to be transformed into a vector (cf. Section 4.3) before sending it as input to the trained model.

We match the predicted FQNs to the true FQNs, similarly to the previous research question. For each API reference and context (e.g., 1, 3, or 5), the likely FQNs are predicted and checked against the true FQN. The prediction is successful when the actual value is present; otherwise, it represents a failure. Finally, we compute the Top-K micro Precision, Recall, and F1-Score for each external dataset and report them.

Results. Figure 10 depicts the results of COSTER and the RESICO trained models on the external datasets. Here, each row corresponds to one external dataset. Each bar chart within a row corresponds to the particular Top-K being considered denoted at the top. Each bar within a bar chart corresponds to the precision, recall, or F1-score for the approach denoted on the horizontal axis.

RESICO outperforms COSTER on all three datasets. The difference in F1-Score is considerable for some configurations. For example, there is a 71% performance difference between the F1-Scores of COSTER and RESICO-KNN when analysing the first predictions (i.e., Top-1) on the *COSTER-SO* dataset.

The best performance overall was RESICO-KNN, as in the previous research question (cf. Figure 9). This model showed a relevant performance starting with 87% F1-Score for the Top-1 recommendations in the *COSTER-SO* dataset.

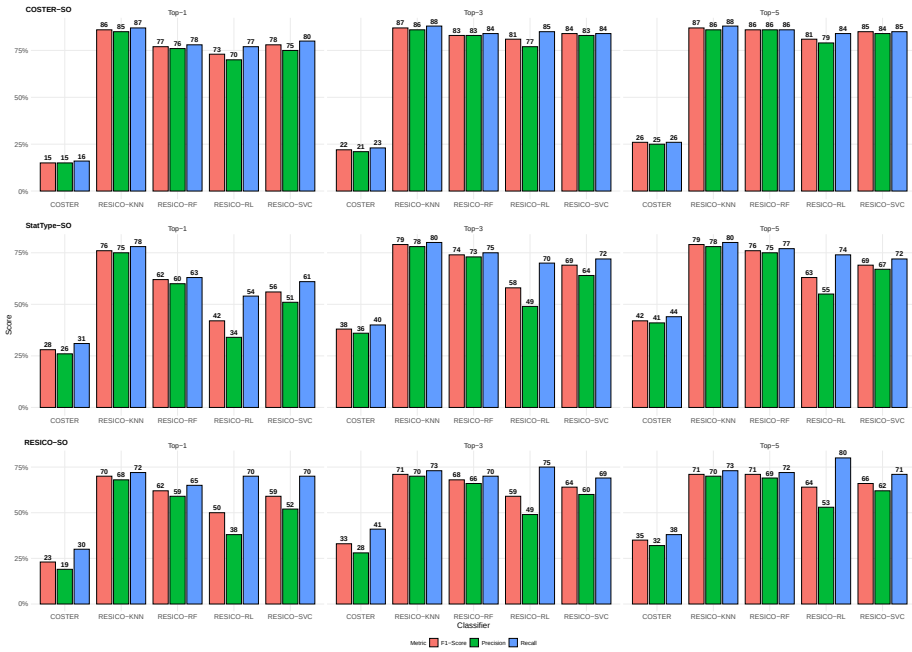

Figure 10: Performance of the models on the three external datasets.

Although inferior to the *COSTER-SO* performance, RESICO-KNN still predicts with good accuracy the FQNs in the *StatType-SO* and *RESICO-SO* datasets. The F1-Score metrics are above 70% for the last two datasets considering all Top-Ks.

The performance difference of the models across the three external datasets can be explained by the number of records per FQN (e.g., 50) and the balancing of the training dataset. In the data balancing step (cf. Section 5.1), we limited the number of records per FQN to 50. The results in Figure 10 show that COSTER might need more occurrences to improve its performance. On the other hand, most of the RESICO classifiers have good generalisability with this limited number of examples per FQN. Additionally, data balancing allowed us to harmonise the importance of each FQN, hence, widespread types such as `java.lang.String` have the same relevance as other less frequent FQNs. In such a way, the trained models are not biased towards any FQN, enabling better performance than other methods, such as COSTER, without this feature.

In the three external datasets considered to evaluate the generalisability of the performance, RESICO-trained models outperform COSTER with a notable difference in some cases. Some decisions made when designing the machine learning models, such as data balancing and hyperparameter optimisation, allow RESICO to achieve better predictive capabilities than the COSTER approach.

5.5. RQ4. How much time is needed to train COSTER and the RESICO classifiers?

Design. This research question investigates the computational cost of COSTER and RESICO. The experiments were conducted on a Dell PowerEdge R730 with 2 Intel Xeon 2637 CPUs, each with four cores at 3.5 GHz with HyperThreading and 256 GB of RAM.

For each approach, we measured the time employed to extract the information from the projects in the initial corpus [33] and the time to train the model in the balanced internal dataset (cf. *RQ2*). Additionally, for RESICO, we measure the embedding time, i.e., the time required to transform the API references and contexts into vectors suitable for a machine learning algorithm. We do not consider the encoding time of FQNs to label numbers since it is a simple mapping whose execution time is negligible.

Results. Table 6 depicts the time measurements for all experiments.

Approach	**Token Extraction**	**Context Embedding**	**Model Training**	**Total**
COSTER	11h 49m 4s	-	25s	11h 49m 29s
RESICO-KNN		3m 43s	671ms	11h 52m 48s
RESICO-RF			29m 6s	12h 21m 53s
RESICO-RL			52m 16s	12h 45m 3s
RESICO-SVC			4m 43s	11h 57m 30s

Table 6: Computational cost of the approaches. Time is measured in hours (h), minutes (m), seconds (s) and milliseconds (ms).

The extraction time (col. *Extraction*) is the same for the two approaches. As COSTER and RESICO rely on the same extraction procedure built on top of Eclipse JDT with slight differences related to the handling of contexts, for every processed FQN, two outputs were written to two different datasets.

The total time (col. *Total*) required by COSTER is lower than that required by RESICO. In Table 6, we show the time needed to extract the tokens, the time needed to embed the contexts, if any, and the time needed to train the models using the sampled data. COSTER does not need embeddings, saving the time needed to craft them.

Fact extraction takes the longest, with nearly 12 hours for processing the data of 50K projects. After sampling and balancing the dataset, the remaining training data consists of 243,000 records. Please consider that embedding the API references takes less than a second, whereas embedding their contexts takes

almost 4 minutes. These running times can be considered efficient for the specified dataset, however, with a more complex dataset, embedding times could increase drastically. The learned embeddings for the API references and their surrounding contexts allowed most RESICO classifiers to outperform COSTER both on the internal dataset (cf. Figure 9) and the three external datasets (cf. Figure 10).

Finally, concerning the training times for the RESICO classifiers and the COSTER approach, the fastest overall is RESICO-KNN with less than a second used for its training on the balanced dataset. COSTER also trained quickly with only 25 seconds, followed by RESICO-SVC, requiring almost 5 minutes. The slowest approaches are RESICO-SVC, with nearly 30 minutes and RESICO-RL, with more than 50 minutes to complete their respective training. For SVC, its training time agrees with its design since it is not an easy task to search for an optimal hyperplane in high-dimensional data. A hypothesis for why the RL model takes the longest to train might be the challenges in linearly differentiating the classes in a multi-class data scenario.

> The fact extraction times of COSTER and RESICO are the same as they both rely on the same Eclipse JDT Core extension. On the dataset considered for training, embedding the tokens does not take considerable time while it does improve the predictions, as shown in the results of RQ2 and RQ3. The classifiers that take the longest to train are those of which characteristics of our internal dataset pose challenges to their design and implementation (e.g., multi-class and high-dimensional data).

5.6. RQ5. To what extent do ambiguities in simple names influence the performance of the approaches?

Design. Name ambiguities affect the type resolution made by COSTER and RESICO. This research question analyses the resolution failures and how ambiguities could have impacted them. Consider an incomplete code snippet having the simple name `Element`. It could resolve to `org.jdom.Element` or `org.jsoup.nodes.Element` within an incomplete code snippet. There are 17 FQN candidates in the internal balanced dataset for this simple name alone. The more ambiguous a simple name is, the more challenging it is for a resolver to predict its exact FQN effectively.

We analyse the erroneous type resolutions made by the COSTER and RESICO models, which were trained on the internal balanced dataset, by investigating whether the root cause for their incorrect resolutions produced on the external datasets is an ambiguous simple name. Specifically, we consider the models of COSTER and RESICO-KNN for this research question, the former being the approach to compare with and the latter the best RESICO model overall. We only consider the Top-1 predictions from the models on the external datasets since they have the majority of failures compared to the other two (i.e., Top-3 and Top-5).

Model	Dataset	# Uniq. Mis.	# Mis.	# Amb.	% Amb.
COSTER	COSTER-SO	27	1,173	4	0.34
	StatType-SO	155	1,355	37	2.73
	RESICO-SO	197	1,412	16	1.13
RESICO-KNN	COSTER-SO	18	205	31	15.12
	StatType-SO	96	458	108	23.58
	RESICO-SO	112	553	73	13.2

Table 7: Ambiguity analysis for COSTER and RESICO-KNN trained models on the external datasets.

Results. The results of the ambiguity analysis are depicted in Table 7. The number of unique misclassifications (col. *Uniq. Mis.*) for the *COSTER-SO* dataset is lower compared to the other datasets. However, those numbers increase when counting the total number of misclassifications regardless of their uniqueness (col. *# Mis.*). The COSTER model produces more misclassifications than RESICO-KNN on all datasets, as previously reported in *RQ3*.

Interestingly, only a minority of the type resolution failures are due to ambiguous simple names. Column *# Amb.* of Table 7 demonstrates that in all datasets but one, less than 100 misclassifications correspond to such cases. Only a tiny percentage (col. *% Amb.*) of FQNs could have been misclassified because other FQNs share the same simple name. The *StatType-SO* dataset with the RESICO-KNN model combination has the highest percentage of ambiguous misclassifications compared to other combinations. A closer look indicates that only 31 unique misclassifications occurred but were repeated multiple times. For example, the FQN `org.hibernate.Session` is predicted as `org.hibernate.classic.Session` and as `javax.websocket.Session`, 35 and 3 times respectively.

These findings indicate that the main reason for failures from the models is not the presence of ambiguous simple names. A likely reason for the mispredictions might be the close similarity of the contexts around different API references. The closer the contexts around API references are, the more prone the models might be to recommend distinct FQNs as similar.

> The number of unique mispredicted FQNs is the highest for both models on the *StatType-SO* and the *RESICO-SO* datasets, illustrating their challenging nature. Only a tiny percentage of incorrect type resolutions is due to ambiguous simple names. The main reason for the mispredictions might not be ambiguous simple names, but closer contexts could mislead the trained models towards failures.

6. Discussion

This section discusses the results, limitations, and potential impact of RESICO.

6.1. Context-based Approaches to API Type Resolution

The approaches considered in the evaluation section (cf. Section 5) are based on the contexts surrounding API references to resolve API types. Indeed, both RESICO and COSTER capture the contexts around the usage of APIs. Then, they recommend a FQN based on similar contexts in incomplete code snippets. Despite this similarity between the approaches, they have many differences, such as their context definitions and the usage of machine learning versus information retrieval techniques.

COSTER captures the API references, their surrounding contexts, and the FQNs of the referenced API element, and stores them into a Lucene[20] database to be queried later on. RESICO, in contrast, starts three learning processes after a similar extraction process to improve its final prediction (cf. Section 4). After the learning processes, a classifier uses the learned embeddings in the form of vectors to learn to distinguish different contexts corresponding to FQNs. The embeddings and classifier models are stored in files for posterior use at resolution time.

We noticed that even though our approach takes slightly more time (around 3 minutes) and thus more computational resources than COSTER (cf. Table 6), it can resolve API types more effectively (cf. Figure 9, Figure 10).

Figure 11: Similar FQNs by their context vectors.

Concerning type resolution failures, our experiments found that most are due to the contexts in which the same or different simple names occur being

20. `https://lucene.apache.org/`

overly similar. For example, Figure 11 depicts different FQNs close to each other because of the similarity of their contexts. The figure was built by extracting the averaged vectors between API references and contexts of 14 randomly selected misclassified FQNs by RESICO. Afterwards, we reduced the dimensions of these vectors to 2 using TSNE[21] to facilitate their plotting. Figure 11 shows the context similarity of different FQNs. The colours and the text on the points identify each FQN in the 2-dimensional cartesian space. The points corresponding to the vectors of `org.joda.time.DateTimeZone` and `org.joda.time.DateTime` are close to each other since they might be used in a similar environment. The same happens to the points of the Apache and Java collections, such as `MultiMap` from the former and `List` and `Map` from the latter. Lastly, `PeriodFormatterBuilder`, `PeriodFormatter`, and `PeriodType` from the `joda` library share context similarities and the trained models struggle to distinguish them in some cases effectively.

In addition, we made an analysis based on the effectiveness of the approaches at the library level for the external datasets evaluated on *RQ3*. Table 8 shows how COSTER and RESICO-KNN (our best model) perform per library for the *StatType-SO* dataset. The analysis for the remaining datasets can be found in our online repository.[1]

Library	**Total**	**COSTER**		**RESICO-KNN**	
		Success	Failure	Success	Failure
JDK	665	46	619	453	212
GWT	350	149	201	303	47
Hibernate	319	65	254	242	77
Joda-Time	250	56	194	170	80
XStream	173	139	34	144	29
Apache-Http	29	1	28	26	3
Apache-Commons	15	10	5	14	1
Apache-Struts	7	0	7	7	0
Org-JSON	7	0	7	0	7
Dom4J	5	1	4	4	1
KSoap2	5	5	0	5	0
Apache-Log4J	2	0	2	1	1

Table 8: The accuracy of the best models per approach shown per library in the *StatType-SO* dataset. Highlighted in green and red are the largest successes and failures, respectively.

For a total of 12 libraries in the *StatType-SO* dataset, the green entries in Table 8 show that RESICO-KNN outperforms COSTER on 10 libraries. For only 2 libraries, the approaches achieved the same score. In the case of the `Org-JSON` library, no approach could correctly resolve any of the seven API

21. `https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html`

types. In the case of `KSoap2`, the approaches successfully predicted all five types.

Table 8 also shows RESICO-KNN scores more success than failure cases in all but two libraries (`Org-JSON` and `Apache-Log4J`). COSTER, in contrast, has a majority of failures instead of successes except for `XStream`, `Apache-Commons` and `KSoap2`. Interestingly, most failures for both approaches are located in the JDK library, with a very low success rate for COSTER (7%) and a relatively low one for RESICO (32%).

6.2. Limitations

RESICO cannot handle multiple versions of the same library, just like previous approaches (e.g., COSTER). They recommend the old and the new FQNs when a class is moved to another package.

The approach should be equally applicable to other languages with explicit import statements (e.g., Python). Some relatively small extensions will be required to support import statements with wildcards, which import several API types simultaneously without analysing the corresponding library files.

To support languages featuring import statements that load a library definition at run time will require more extensive work. For such languages, we envision run-time analysis of the libraries. Note that we are not gauging the modifications required based on whether the language is dynamically or statically typed but on the type of import statements that would occur in its snippets.

Finally, RESICO is based on word embeddings. Like other approaches relying on this technique, it suffers from *Out-Of-Vocabulary* errors when the model must predict a term that is not in the training set. RESICO will fail to resolve the API type for such a term. Nonetheless, the NLP community has proposed mitigation strategies such as enriching the vectors with subword information [40, 41], which are not currently included in the implementation and are part of our future agenda.

6.3. Potential Impact

As revealed by our experiments, RESICO scores high on all considered datasets. Researchers and practitioners can use our approach to design and implement tools that analyse SO posts, e.g., API usages and the natural language around the code. Our approach and trained model provide type resolution to client analyses and tools requiring type information, even for syntactically incorrect and incomplete code snippets. One example tool could, for instance, help developers who rely on SO solutions and frequently copy their code into an IDE by providing import statements for all referenced library types.

7. Threats to Validity

We now discuss the threats that might affect the validity of our study.

7.1. Threats to Construct Validity

RESICO relies upon a Word2Vec implementation provided by the Golang programming language to obtain the word embeddings and four classifiers from the Scikit-Learn framework for building and evaluating the machine learning models. RESICO uses these libraries extensively, so their inclusion constitutes a threat to validity. Nevertheless, they are among the most popular open-source machine-learning libraries.

7.2. Threats to Internal Validity

We chose the *Continuous Bag of Words* (CBOW) architecture for the Word2-Vec algorithm instead of the alternative Skip-gram. Our selection was motivated by the faster processing times of the former approach compared to the latter. CBOW tries to infer a word given its context, whereas Skip-gram attempts to infer a context given a word. Skip-gram might produce slightly different results; however, as reported by Mikolov et al. [13], CBOW can also produce reliable vector representations while reducing the learning time.

Data balancing was used to equalise internal dataset extracted from the 50K-C corpus, thereby avoiding bias training towards imbalanced classes. Data balancing should in general be applied to the training dataset and not to the testing dataset. In *RQ2*, we balanced the full dataset before the stratification and posterior training and testing, to verify whether our model could detect FQNs from a dataset similar to the training one. The possible implications of this decision are minimal for a two-fold reason. On the one hand, all evaluated methods (e.g., COSTER and all RESICO classifiers) use the same balanced dataset for training and testing, making the evaluation fair. On the other hand, we conducted a second evaluation on external datasets, showing similar results.

7.3. Threats to External Validity

An external threat might be due to the comparisons between both approaches, COSTER [5] and RESICO. For an unbiased comparison, we first extracted information from the same API references in the 50K-C dataset [33]. Second, the training dataset is balanced to avoid bias towards any particular label, and the approaches are trained on this balanced dataset. Third, we evaluated both approaches on the datasets initially employed for assessing the quality of COSTER (e.g., *COSTER-SO* and *StatType-SO*). Lastly, we created a third external dataset (e.g., *RESICO-SO*) of code snippets that reference the same set of libraries present in the previous datasets.

Another external threat concerns the classifiers and the word embedding algorithm (i.e., Word2Vec) used in RESICO. We chose four different classifiers as instantiations of RESICO which have been used in ML-based software engineering solutions [42, 43] as well as in previous text classification works [23, 26, 27, 31]. Likewise, Word2Vec has been successfully used in other studies [14, 17, 44].

7.4. Threats to Conclusion Validity

The metrics used to evaluate our approach (i.e., Precision, Recall, and F1-Score) are widely used among recommenders, including the reference work (i.e., COSTER). To evaluate to what extent programs in the 50K-C dataset can be used to predict fully-qualified names with similar context, we adopted a 10-fold stratified cross-validation [37] after a previous balancing of the dataset. For datasets which are not completely balanced, the number of instances per fold will correspond to a similar distribution of the full dataset. Since we have a balanced dataset, it is not only ensured that each fold follows the general distribution but also that each of them contains the same number of instances per class.

8. Related Work

We discuss tools that leverage Stack Overflow, program analysis methods concerning the code snippets available there, approaches that employ type inference to improve certain aspects of programming languages, and other program analysis applications of word embeddings.

8.1. Development Tools Incorporating SO Information

Several tools have been proposed that use SO posts as a source of information. RecoDoc [45], for instance, identifies API references within textual information in sources such as documentation and Q&A websites. Prompter [46] proactively retrieves posts related to developers' context within the IDE. SISE [1] augments library documentation with insightful sentences found in posts. GitSearch [47] consults Q&A posts to enrich free-from code search queries against code repositories with API-specific terminology. CodeTube [48] enables querying the contents of software development video tutorials and complements the results with relevant discussions. BIKER [49] takes a natural language query as input and returns related posts and example API methods. POME [2] analyses SO posts to synthesise the community's opinion about a library. PostFinder [50] recommends SO posts based on the development context in an IDE (e.g., Eclipse) by constructing improved queries to a formerly constructed Lucene database.

To link code snippets to libraries and their API elements, several of the mentioned works use regular expressions (e.g., Treude and Robillard [1] and Lin et al. [2]) or a sequence of candidate eliminating conditions (e.g., Linares-Vásquez et al. [51]). Approaches to resolving incomplete declarations have also been proposed. In particular, PostFinder [50] tries to resolve API types by parsing a code snippet using Eclipse JDT and then querying likely FQNs from the Maven Dependency Graph [52] based on the unresolved simple names of declarations. For those cases with multiple candidates sharing the same simple name (i.e., ambiguous cases), PostFinder computes the Levenshtein distance between the candidates and the SO post title, the question body and the answer. However, this heuristic does not ensure a near-accurate resolution for an ambiguous simple name for the following reasons. First, The Levenshtein distance calculates

the number of changes needed to transform a text sequence into another at the character level.[22]. Second, PostFinder takes the maximum distance (cf. Line 163 - Line 178 in its GitHub source code[23]), meaning that the selected FQN is the most dissimilar to the post text corpus. We did not find a rationale behind this decision. Finally, SO code snippets contain API types that are not related to the topic under discussion, yet are entangled with those types that are.

Therefore, our trained machine learning models could be used to perform API-type resolution and improve the mentioned approaches and tools.

8.2. Program Analyses for SO Code Snippets

Algorithms dedicated to API type resolution of code snippets have been proposed. Baker [53, 3] traverses a best-effort Abstract Syntax Tree (AST) constructed by the Eclipse JDT parser or the Esprima parser for JavaScript to collect type information at variable declaration nodes. It then associates a list of candidate FQNs for these nodes by consulting a database populated from the JAR files of candidate libraries in the case of Java code snippets. These lists are iteratively refined for every method invocation of which the receiver expression is a known variable reference or method invocation. The computed lists satisfy the semantic constraints imposed by using the API-related variables in the snippet. The deductive reasoning approach taken by Baker is therefore limited by (i) the syntactical correctness of the snippet, (ii) the extent to which it contains candidate-reducing API usages, and (iii) the library implementations for which its database has been populated. RESICO also has limitations concerning the vocabulary sizes for which the Word2Vec models were trained (i.e., similar to (iii)). However, RESICO does not rely on the correctness of the code snippet (i.e., (i)), nor has a candidate-reducing API list (i.e., (ii)), but it relies on the wisdom of the crowd to resolve API types.

Yang et al. [54] investigate how usable code snippets are in Stack Overflow for four programming languages: C#, Java, Python and JavaScript. Usability, in this case, is defined as whether a code snippet passes the steps of parsing, compiling and running from standard tools in the mentioned programming languages. Their findings indicate that few snippets are usable as they are in the posts, especially in the case of Java, with less than 4% parsing and 1% compiling success rates. Including heuristics to repair code snippets (e.g., addition of wrapping classes, methods and semicolons) slightly improves previous scores, but still, a large percentage (more than 80%) remains unusable. As pointed out by Yang et al. [54], almost 63% of errors are due to unfound symbols, which reinforces the need for tools such as RESICO to deal with incomplete code snippets.

CSnippEx [55] is an Eclipse plugin to repair Java code snippets and convert them into compilable source code files. The design of CSnippEx is mainly based

22. `https://commons.apache.org/proper/commons-text/apidocs/org/apache/commons/text/similarity/LevenshteinDistance.html`
23. `https://github.com/MDEGroup/PostFinder/blob/master/tools/src/main/java/soRec/Utils/Jdt.java`

on a feedback approach where the Java compiler is continuously queried for possible errors. First, the tool discovers the likely merging of multiple code snippets in a post. Second, it resolves the import dependencies by relying on extracted pairs of FQNs to their latest version of JARs and exploiting their *clustering hypothesis*. Third, CSnippEx considers the compilation errors of the Eclipse Quick Fix tool to identify the solutions that could finally convert the code into a compilable source. Terragni et al. [55] reported a successful synthesis of 40,410 code snippets from a total of 242,175 for a success rate of around 17%, which is still testament to the difficulty of the problem. RESICO does not go beyond API-type resolution to form a compilable unit, although this is part of our future work. Additionally, our approach is based on the context around simple names and not on heuristics, as is the case for CSnippEx.

StatType [4] uses statistical machine translation to resolve FQNs. The approach translates sentences from a source language (i.e., of partially-qualified API names) to a target language (i.e., of FQNs). It requires training a language model and a separate mapping model on a corpus of projects that use the API of the targeted libraries. These projects are simultaneously translated into StatType's source and target language by traversing their methods' ASTs. Sentences in the source language capture information about the simple name of each referenced API element. In contrast, sentences in the target language convey the corresponding FQN and the syntactic construct through which the API element was referenced. Extracting this information requires the input projects to compile. However, the approach is entirely data-driven because it is agnostic to program semantics. Resolutions learned by combining the language and mapping model are refined based on the local context surrounding the name to be resolved. StatType achieved higher accuracy than Baker [53, 3]; however, training its models may be computationally expensive. In contrast to StatType, RESICO handles the incompleteness problem as a classification procedure where a learned context will influence the label to be predicted (i.e., the FQN of the API reference in analysis).

Saifullah et al. [5] presented COSTER, which takes an information retrieval approach to the problem of resolving FQNs in incomplete code snippets. As StatType, it relies on the obtained wisdom of the crowd from a corpus of compilable projects. Extracted information from the corpus is used to compute the likelihood that the textual tokens surrounding the non-qualified API reference co-occur in the project corpus when the reference resolves to the corresponding FQN. The immediately surrounding tokens are referred to as the local context of the API reference. The local context is extended with its global context. It relies on the names of methods called on the variable to which the reference is assigned or the names of methods to which this variable is passed as an argument. In a snippet, context and name similarities are subsequently used to refine the candidates returned from this so-called occurrence likelihood dictionary for the queried name and context. COSTER has outperformed StatType and Baker in many aspects of the evaluation conducted in Saifullah et al. [5]. Therefore, we compared our approach against COSTER, which constitutes the state-of-the-art method. RESICO leverages the same datasets and the contexts

surrounding the API references as COSTER, allowing RESICO to analyse many libraries. The training phase represents a distinctive feature of RESICO compared to previous approaches (e.g., Baker, StatType and COSTER). After the learning phase, a classification algorithm (e.g., Random Forest in this case) tries to associate FQNs to the learned contexts.

Dong et al. [56] propose SnR as an approach to infer FQNs based on the constraints in SO code snippets. SnR constructs a knowledge base from libraries consisting of relations between types belonging to constructs, such as fields and methods. As in the case of CSnippEx [55], SnR tries first to repair a code snippet and form a compilation unit to extract an AST from which to infer API types. The type inference uses Datalog, given the current code snippet constraints and the knowledge base, to provide reachable nodes in a dependency graph. Lastly, SnR ranks type candidates and selects those on the top to finally create import statements and include them in the transformed code solution. SnR is compared against COSTER [5] using the *StatType-SO* dataset as we did in this work. There are some relevant differences between SnR and RESICO:

1. SnR is constraint-based, whereas RESICO is context-based.

2. To evaluate COSTER against SnR, Dong et al. [56] did not consider that COSTER was trained on a highly imbalanced dataset and used that trained model; we did consider it and retrained COSTER on balanced data.

3. SnR has only been evaluated on the external *StatType-SO* dataset. RESICO, additionally, has, in this paper, been evaluated on the same balanced internal dataset as COSTER, and on two more external datasets (e.g., *COSTER-SO* and *RESICO-SO*), outperforming COSTER in all cases.

A more in-depth comparison between the tools in particular and context-based and constraint-based approaches in general is required to evaluate the impact of these differences. Moreover, a hybrid technique combining the strong points of each could lead to a very effective and improved approach overall.

8.3. Type Inference for Programs

Type inference for programs written in dynamically-typed languages (e.g., Python and JavaScript) has enjoyed attention in the programming language research community. Raychev et al. [57] propose an approach for predicting names of identifiers and type annotations of variables in JavaScript programs. Their tool, JSNice, can indicate the mentioned properties for unseen programs based on a trained Conditional Random Field model. Xu et al. [58] rely on incomplete type hints (e.g., the names and usages of variables) to make a probabilistic inference on the types of Python programs. Hellendoorn et al. [59] proposed to use Deep Learning (DL) to tackle this problem with DeepTyper, a model that infers types in JavaScript based on the learning capabilities of the Recurrent Neural Networks. They laid the foundation for using DL techniques for type inference. Malik et al. [60] leverage natural language information in the

source code (e.g., comments and function names) and train an LSTM neural network to assist in the type inference of JavaScript programs. Wei et al. [61] propose LambdaNet as a Graph Neural Network approach to probabilistically infer types in TypeScript. LambdaNet relies upon a type dependency graph (TDG) which includes additional hints such as variable names and usages in the graph itself. Peng et al. [62] leverage models from previous approaches and developed HiTyper, a hybrid approach that condenses static and AI-based type inferences. Type inference was also applied to recover information from binaries by Lehmann and Pradel [63] and Wang et al. [64] for WebAssembly and C++ binaries, respectively.

RESICO aims not to infer types for programs implemented in dynamically-typed languages but to resolve the FQNs of API references for which the declaration or import statement is missing from an incomplete and possibly syntactically incorrect SO code snippet. Furthermore, RESICO resolves the simple names in a snippet without ensuring that all of the produced resolutions for the snippet together satisfy the programming language's type constraints. RESICO is agnostic of the semantics of the programming language. In the semantic sense, our approach could therefore be considered unsound, although our evaluation (cf. Section 5) demonstrates its effectiveness in practice. Type constraint checks on RESICO's predictions for a snippet could be added as a post-processing step, which is part of our future research agenda.

8.4. Word Embeddings for Source Code Analysis

Our approach uses word embeddings [12, 13] to transform API references and their contexts into a format suitable as input to machine learning algorithms. Such applications are increasingly prevalent in the literature. code2vec [19] is a machine learning approach to predicting source code properties (e.g., method names provided their bodies). It uses a neural network for learning embeddings that effectively model the correspondence between the code in the method body and its associated label. The networks learn to aggregate syntactic paths from the AST of the snippet into a vector. Similarly, code2seq [16] uses AST paths and an encoder-decoder architecture to learn code snippet representations which are subsequently used in tasks such as code summarisation and code captioning. Our approach relies on the embeddings of the API references and terms in their surrounding context. These references are extracted from the complete programs rather than the paths in the ASTs.

The machine learning approach proposed by Henkel et al. [17] attempts to learn trace-oriented embeddings for source code. These are obtained by applying abstracting transformations to a lightweight form of intra-procedural symbolic execution. The resulting embeddings capture behavioural aspects of the code and outperform more syntactic embeddings on a series of tasks. Similarly, Wang and Su [65] propose LiGer, a neural network that can learn from a mixture of symbolic and concrete traces. The mixture (or blended) trace represents the input to an encoder-decoder neural network architecture which outperforms code2seq in the task of predicting method names. We consider the use of sym-

bolic execution for the embedding part of our approach infeasible given the sheer amount of libraries and library usages on SO.

The import2vec [18] framework takes a word embedding approach to learning library representations based on the co-occurrence of the corresponding import statements in client projects. The learned embeddings are suitable for recommending imports with usage context similar to the queried one and have been shown to capture meaningful semantic relations, such as which library packages are often used together in particular problem domains. RESICO does not focus on the used import statements alone, but it goes one step further and learns the similarities of the API usages and their surrounding contexts.

9. Conclusion

This paper proposes a new learning-based approach to resolving the fully-qualified name of API types referenced by their simple name in code snippets from online Q&A platforms such as Stack Overflow. The approach, called RESICO, extracts API references, their surrounding contexts and their associated FQNs from a dataset of 50K compilable GitHub projects. A data balancing technique is applied to balance the dataset from where COSTER and RESICO train and learn to distinguish API types in incomplete code. RESICO uses Word2Vec to transform API references and their contexts into vector representations and class categorisation to convert FQNs into numbers. The vectors of the API references and contexts are combined into the input of machine learning classifiers, whereas the numbered FQNs are the labels to predict. The approach is instantiated with four machine learning classifiers, namely KNN, Random Forests, Ridge Linear, and Support Vector Machines. Before evaluating the resulting models, hyperparameter optimisation is applied to find an optimal configuration for these classifiers.

We have compared the RESICO machine learning models in depth to the state-of-the-art approach, COSTER, which is based on information retrieval. The setup of our approach is more computationally intensive than that of COSTER, as it involves training two Word2Vec models to learn embeddings and one classification algorithm per RESICO instantiation. Our best classifier (KNN) is slightly slower to train than COSTER. However, once deployed, most of the RESICO-trained models outperform COSTER in the 10-fold cross-validation on an internal dataset and on three external datasets. Finally, we showed that incorrect type resolutions produced by RESICO and COSTER might not be due to ambiguous simple names but mainly to similar contexts around usages.

Acknowledgements

We would like to thank the authors of COSTER [5] for sharing their tool and data. This research was partially funded by the Excellence of Science project EOS 30446992 *SECO-ASSIST* financed by FWO-Vlaanderen and F.R.S.-FNRS.

References

[1] C. Treude, M. P. Robillard, Augmenting API Documentation with Insights from Stack Overflow, in: Proceedings of the 38th International Conference on Software Engineering (ICSE16), 2016, pp. 392–403.

[2] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, Pattern-based Mining of Opinions in Q&A Websites, in: Proceedings of the 41st International Conference on Software Engineering (ICSE19), 2019, pp. 548–559.

[3] S. Subramanian, L. Inozemtseva, R. Holmes, Live API Documentation, in: Proceedings of the 36th International Conference on Software Engineering (ICSE14), 2014, pp. 643–652. doi:10.1145/2568225.2568313.

[4] H. Phan, H. A. Nguyen, N. M. Tran, L. H. Truong, A. T. Nguyen, T. N. Nguyen, Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums, in: Proceedings of the 40th International Conference on Software Engineering (ICSE18), IEEE, 2018, pp. 632–642.

[5] C. K. Saifullah, M. Asaduzzaman, C. K. Roy, Learning from Examples to Find Fully Qualified Names of API Elements in Code Snippets, in: Proceedings of the 34th International Conference on Automated Software Engineering (ASE19), 2019, pp. 243–254.

[6] M. Bates, Models of natural language understanding, Proceedings of the National Academy of Sciences 92 (22) (1995) 9977–9982.

[7] Z. S. Harris, Distributional structure, Word 10 (2-3) (1954) 146–162.

[8] K. Sparck Jones, A statistical interpretation of term specificity and its application in retrieval, Journal of documentation 28 (1) (1972) 11–21.

[9] Y. Sakakibara, K. Misue, T. Koshiba, Text classification and keyword extraction by learning decision trees, in: Proceedings of 9th Conference on Artificial Intelligence for Applications (AIAI93), 1993, p. 466.

[10] A. Bouaziz, C. Dartigues-Pallez, C. da Costa Pereira, F. Precioso, P. Lloret, Short Text Classification Using Semantic Random Forest, in: International Conference on Data Warehousing and Knowledge Discovery, 2014, pp. 288–299.

[11] B. Xu, X. Guo, Y. Ye, J. Cheng, An Improved Random Forest Classifier for Text Categorization, JCP 7 (12) (2012) 2913–2920.

[12] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed Representations of Words and Phrases and their Compositionality, in: Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NIPS13), 2013.

[13] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient Estimation of Word Representations in Vector Space, arXiv preprint arXiv:1301.3781 (2013).

[14] T. D. Nguyen, A. T. Nguyen, T. N. Nguyen, Mapping API Elements for Code Migration with Vector Representations, in: Companion to the Proceedings of the 38th International Conference on Software Engineering (ICSE-C16), IEEE, 2016, pp. 756–758.

[15] T. D. Nguyen, A. T. Nguyen, H. D. Phan, T. N. Nguyen, Exploring API Embedding for API Usages and Applications, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017, pp. 438–449.

[16] U. Alon, S. Brody, O. Levy, E. Yahav, code2seq: Generating Sequences from Structured Representations of Code, arXiv preprint arXiv:1808.01400 (2018).

[17] J. Henkel, S. K. Lahiri, B. Liblit, T. Reps, Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces, in: Proceedings of the 26th Joint Meeting of European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE18), 2018, pp. 163–174.

[18] B. Theeten, F. Vandeputte, T. Van Cutsem, Import2vec Learning Embeddings for Software Libraries, in: Proceedings of the 16th International Conference on Mining Software Repositories (MSR19), 2019, pp. 18–28.

[19] U. Alon, M. Zilberstein, O. Levy, E. Yahav, code2vec: Learning Distributed Representations of Code, Proceedings of the ACM on Programming Languages 3 (POPL) (2019) 40.

[20] B. V. Dasarathy, Nearest neighbor (NN) norms: NN pattern classification techniques, IEEE Computer Society Tutorial (1991).

[21] P. Soucy, G. W. Mineau, A Simple KNN Algorithm for Text Categorization, Proceedings - IEEE International Conference on Data Mining, ICDM (2001) 647–648doi:10.1109/icdm.2001.989592.

[22] V. Bijalwan, V. Kumar, P. Kumari, J. Pascual, KNN based Machine Learning Approach for Text and Document Mining, International Journal of Database Theory and Application 7 (1) (2014) 61–70.

[23] K. Shah, H. Patel, D. Sanghvi, M. Shah, A Comparative Analysis of Logistic Regression, Random Forest and KNN Models for the Text Classification, Augmented Human Research 5 (1) (2020) 1–16.

[24] T. K. Ho, Random Decision Forests, in: Proceedings of 3rd international conference on document analysis and recognition, Vol. 1, IEEE, 1995, pp. 278–282.

[25] L. Breiman, Random Forests, Machine learning 45 (1) (2001) 5–32.

[26] H. Chen, L. Wu, J. Chen, W. Lu, J. Ding, A Comparative Study of Automated Legal Text Classification using Random Forests and Deep Learning, Information Processing & Management 59 (2) (2022) 102798.

[27] T. Zhang, F. J. Oles, Text Categorization based on Regularized Linear Classification Methods, Information retrieval 4 (1) (2001) 5–31.

[28] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Numerical Recipes 3rd edition: The Art of Scientific Computing, Cambridge University Press, 2007.

[29] C. Cortes, V. Vapnik, Support-vector Networks, Machine learning 20 (3) (1995) 273–297.

[30] T. Joachims, Learning to Classify Text using Support Vector Machines, Vol. 668, Springer Science & Business Media, 2002.

[31] J. Lilleberg, Y. Zhu, Y. Zhang, Support Vector Machines and Word2vec for Text Classification with Semantic Features, in: 2015 IEEE 14th International Conference on Cognitive Informatics & Cognitive Computing (ICCI* CC), IEEE, 2015, pp. 136–140.

[32] L. Moonen, Generating Robust Parsers using Island Grammars, Proceedings of the Eighth Working Conference on Reverse Engineering (2001) 13–22doi:10.1109/wcre.2001.957806.

[33] P. Martins, R. Achar, C. V. Lopes, 50K-C: A dataset of compilable, and compiled, Java projects, in: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), IEEE, 2018, pp. 1–5.

[34] S. Baltes, L. Dumani, C. Treude, S. Diehl, SOTorrent: Reconstructing and Analyzing the Evolution of Stack Overflow Posts, in: Proceedings of the 15th international conference on Mining Software Repositories (MSR), 2018, pp. 319–330. arXiv:1803.07311, doi:10.1145/3196398.3196430.

[35] J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, Algorithms for Hyper-Parameter Optimization, Advances in neural information processing systems 24 (2011).

[36] J. Bergstra, D. Yamins, D. Cox, Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures, in: International Conference on Machine Learning, PMLR, 2013, pp. 115–123.

[37] M. Stone, Cross-validatory Choice and Assessment of Statistical Predictions, Journal of the royal statistical society. Series B (Methodological) (1974) 111–147.

[38] K. Sechidis, G. Tsoumakas, I. Vlahavas, On the Stratification of Multi-label Data, Machine Learning and Knowledge Discovery in Databases (2011) 145–158.

[39] Z. C. Lipton, C. Elkan, B. Naryanaswamy, Optimal Thresholding of Classifiers to Maximize F1 Measure, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2014, pp. 225–239.

[40] M.-T. Luong, C. D. Manning, Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models, arXiv preprint arXiv:1604.00788 (2016).

[41] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov, Enriching Word Vectors with Subword Information, Transactions of the Association for Computational Linguistics 5 (2017) 135–146.

[42] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, A. De Lucia, Detecting Code Smells using Machine Learning Techniques: Are We There Yet?, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2018, pp. 612–621.

[43] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Transactions on Software Engineering 38 (6) (2011) 1276–1304.

[44] X. Ye, H. Shen, X. Ma, R. Bunescu, C. Liu, From Word Embeddings To Document Similarities for Improved Information Retrieval in Software Engineering, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 404–415.

[45] B. Dagenais, M. P. Robillard, Recovering Traceability Links between an API and Its Learning Resources, in: Proceedings - International Conference on Software Engineering, 2012, pp. 47–57. doi:10.1109/ICSE.2012.6227207.

[46] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, M. Lanza, Mining StackOverflow to Turn the IDE into a Self-Confident Programming Prompter, 11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings (2014) 102–111doi:10.1145/2597073.2597077.

[47] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, Y. L. Traon, Augmenting and structuring user queries to support efficient free-form code search, Empirical Software Engineering (EMSE) 23 (5) (2018).

[48] L. Ponzanelli, G. Bavota, A. Mocci, R. Oliveto, M. D. Penta, S. Haiduc, B. Russo, M. Lanza, Automatic Identification and Classification of Software Development Video Tutorial Fragments, IEEE Transactions on Software Engineering 45 (5) (2019) 464–488. doi:10.1109/TSE.2017.2779479.

[49] L. Cai, H. Wang, Q. Huang, X. Xia, Z. Xing, D. Lo, BIKER: A Tool for Bi-information Source Based API Method Recommendation, in: Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE19), 2019, pp. 1075–1079.

[50] R. Rubei, C. Di Sipio, P. T. Nguyen, J. Di Rocco, D. Di Ruscio, PostFinder: Mining Stack Overflow posts to support software developers, Information and Software Technology 127 (2020) 106367.

[51] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK, 22nd International Conference on Program Comprehension, ICPC 2014 - Proceedings (2014) 83–94doi:10.1145/2597008.2597155.

[52] A. Benelallam, N. Harrand, C. S. Valero, B. Baudry, O. Barais, Maven central dependency graph, The Maven dependency graph is the fruit of a collaboration between the DiverSE team (Inria Rennes, France) and CASTOR project (KTH, Sweden). Instructions on how to use and reproduce the dataset can be found in the dataset's repository on [Github](https://github.com/diverse-project /maven-miner). A complete description of the dataset and usages can be found in the accompanying [paper] (https://arxiv.org/abs/1901.05392). (Nov. 2018). doi:10.5281/zenodo.1489120.
URL `https://doi.org/10.5281/zenodo.1489120`

[53] S. Subramanian, R. Holmes, Making Sense of Online Code Snippets, IEEE International Working Conference on Mining Software Repositories (2013) 85–88doi:10.1109/MSR.2013.6624012.

[54] D. Yang, A. Hussain, C. V. Lopes, From Query to Usable Code: An Analysis of Stack Overflow Code Snippets, in: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), IEEE, 2016, pp. 391–401.

[55] V. Terragni, Y. Liu, S.-C. Cheung, CSnippEx: Automated Synthesis of Compilable Code Snippets from Q&A Sites, in: Proceedings of the 25th international symposium on software testing and analysis, 2016, pp. 118–129.

[56] Y. Dong, T. Gu, Y. Tian, C. Sun, SnR: Constraint-Based Type Inference for Incomplete Java Code Snippets, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1982–1993.

[57] V. Raychev, M. Vechev, A. Krause, Predicting Program Properties from "Big Code", ACM SIGPLAN Notices 50 (1) (2015) 111–124. doi:10.1145/2775051.2677009.

[58] Z. Xu, X. Zhang, L. Chen, K. Pei, B. Xu, Python Probabilistic Type Inference with Natural Language Support, in: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, 2016, pp. 607–618.

[59] V. J. Hellendoorn, C. Bird, E. T. Barr, M. Allamanis, Deep Learning Type Inference, Proceedings of the 2018 26th ACM Joint Meeting on European

Software Engineering Conference and Symposium on the Foundations of Software Engineering (2018) 152–162doi:10.1145/3236024.3236051.

[60] R. S. Malik, J. Patra, M. Pradel, NL2Type: Inferring JavaScript Function Types from Natural Language Information, 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) 00 (2019) 304–315. doi:10.1109/icse.2019.00045.

[61] J. Wei, M. Goyal, G. Durrett, I. Dillig, LambdaNet: Probabilistic Type Inference using Graph Neural Networks, 2020 8th International Conference on Learning Representations (2020). arXiv:2005.02161.

[62] Y. Peng, C. Gao, Z. Li, B. Gao, D. Lo, Q. Zhang, M. Lyu, Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python, 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE) (2022). arXiv:2105.03595, doi:10.1145/3510003.3510038.

[63] D. Lehmann, M. Pradel, Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries, 2022 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2022).

[64] X. Wang, X. Xu, Q. Li, M. Yuan, J. Xue, Recovering Container Class Types in C++ Binaries, 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) 00 (2022) 131–143. doi:10.1109/cgo53902.2022.9741274.

[65] K. Wang, Z. Su, Blended, Precise Semantic Program Embeddings, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020, pp. 121–134.