OneSpace: Detecting cross-language clones by learning a common embedding space

Mohammed El Arnaoty[a,b], Francisco Servant[c,1,*]

[a]Department of Computer Science, Virginia Tech, United States of America
[b]Department of computer science, Faculty of computers and artificial intelligence, Cairo university, Giza, Egypt
[c]ITIS Software, Universidad de Málaga, Málaga, Spain

Abstract

Identifying clone code fragments across different languages can enhance the productivity of software developers in several ways. However, the clone detection task is often studied in the context of a single language and less explored for code snippets spanning different languages. In this paper, we present OneSpace, a new cross-language clone detection approach. OneSpace projects different programming languages to the same embedding space using both code and API data. OneSpace, hence, leverages a Siamese Network to infer the similarity of the embedded programs. We evaluate OneSpace by detecting clones across three language pairs; JAVA-Python, Java-C++ and Java-C. We compared OneSpace with the other state-of-art techniques, SupLearn and CLCDSA. In our evaluation, OneSpace provided higher effectiveness than the state of the art. Our ablation study validated some of our intuitions in designing OneSpace, particularly that using a single embedding space (as opposed to separate ones) provides higher effectiveness. Additionally, we designed a variant of OneSpace that uses Word-Mover-Distance Algorithm and provides lower effectiveness, but is much more efficient. We also found that OneSpace provides higher effectiveness than the state of the art, even for: complex implementations, single-method implementations, varying ratios of positive to negative clones in training, varying amounts of training data, and for additional programming languages.

Keywords: clone detection, Siamese Neural Networks, word vector, embedding

1. Introduction

When developers copy and paste the same code fragments (with or without minor adaptations), they produce *code clones* [100]. Code clones are very common in software — multiple studies found that between 5% and 50% of the code of software systems can be cloned [13, 16, 35, 50, 62, 66, 72, 76, 84]. Code clones can be introduced under diverse situations, *e.g.,* when multiple algorithms are very similar [76], due to coding style [16], by accident [1], when porting code to another programming language [6], or for many other reasons [13, 16, 58, 61, 68, 84]

Code clones may exist within the same programming language (*i.e., single-language code clones*) or in different ones (*i.e., cross-language code clones*) [69].

Modern software systems are developed using multiple programming languages [38, 82, 83]. For example, Netflix recently discontinued their monolingual development in Java to move to a multilingual services ecosystem [47]. More broadly, recent studies found that software projects now use between 2 – 5 languages in open source [82, 118], and an average of 7 languages in industry [83].

Some software projects contain clones for some (or all) of their functionality in different programming languages, to support multiple platforms. For example, the same micro-services are often implemented in multiple languages [44, 113]. Also Antlr, a widely used parser generator, has versions implemented in Java, C#, JavaScript, and Python [10]. Similarly, Lucene, a renowned text search engine, offers implementations in Java and C# [8].

Unfortunately, software systems with code clones may be harder to maintain [13, 57, 119]. This is primarily because, when developers change one code clone, they often have to propagate those changes to several other clones (in the same or different programming languages) [41].

To reduce the maintainability cost of code clones, *automatic clone detection* research targets automatically supporting developers in finding and tracking code clones, *e.g.,* [65, 77, 105, 124, 128]. In practice, developers run automatic clone detection over their codebase as input, and obtain as output the code pairs that the technique estimates to be clones [28, 29]. This saves developers the effort of identifying the clones manually.

Automatic (single or cross language) clone detection can be beneficial in multiple software engineering tasks, *e.g.,* reducing development time and costs [100], reducing the proliferation of bugs and security vulnerabilities [100], efficiently identifying reusable code as candidates for refactoring [100], and reducing the time and cost of code porting

*Corresponding author
Email addresses: marnaoty@vt.edu (Mohammed El Arnaoty), fservant@uma.es (Francisco Servant)

[1]Some work performed while at Virginia Tech, U.S.A. and at Universidad Rey Juan Carlos, Spain.

and migration across programming languages [81].

Automatic (single or cross language) clone detection reduces development time by easing software maintenance. When developers change one clone, automatic clone detection saves them the effort of manually finding all its other clones, to consider changing them too [58, 84]. Performing that process manually may be time-consuming and also error-prone, since it may require understanding many diverse components of the software, possibly in many programming languages.

Automatic (single or cross language) clone detection also reduces the proliferation of bugs and security vulnerabilities. When some code is found to be buggy [98] or vulnerable [29, 122], automatic clone detection is often used to find its clones (sometimes in code ported to other languages [98]), to be able to resolve the issue in all the affected clones. This saves developers the effort of manually inspecting other areas of the code to find out if the same bug or vulnerability also exists in other parts of the system. This may also help developers to find additional potential instances of the bug or vulnerability, reducing its proliferation. This approach to vulnerability detection is already applied by software companies, *e.g.,* at Microsoft [28, 29].

Automatic (single or cross language) clone detection also helps developers when they decide to refactor a set of clones into a single library, to reduce their maintenance cost [19, 31, 128]. Automatic clone detection saves developers the effort of manually identifying all the code clones that could be refactored into the library (or, *e.g.,* into a micro-service, when multiple languages will access it).

Automatic cross language clone detection may also reduce development time when porting or migrating code to another programming language [6, 81]. For example, during the porting of the video game "Fez" from Microsoft Xbox to Sony PlayStation, the most significant challenge was converting the original C# code to C++, due to language differences [6]. Automatic clone detection may help developers when porting code sections for which clones of them already exist in the target language (*e.g.,* they were already ported for other projects, or they exist in online repositories [77, 103]), by automatically finding such clones for them. This may save developers the effort and error-proneness of manually porting the code, or of manually finding its existing ported clones.

Unfortunately, the majority of existing clone detection techniques support a single programming language, *e.g.,* [65, 77, 105, 124]. Our work focuses on a more challenging instance of the clone-detection problem: *cross-language clone detection*, *i.e.,* the detection of equivalent code across different programming languages.

A few techniques have been proposed for cross-language clone detection. Some work only under specific circumstances, detecting cross-language code clones that, *e.g.,* mostly evolved in parallel in their corresponding code repositories [23, 24], have very similar size and structure [120], were implemented within the .NET framework [2, 69], or can be executed within a time threshold. [80, 81].

Finally, two existing approaches have a more general purpose, *i.e.,* they do not impose such restrictions. These are SupLearn [95] and CLCDSA [92]. SupLearn and CLCDSA use machine learning (Siamese Neural Networks), to be able to apply to cross-language clones of any programming language, irrespective of whether they were developed in parallel or have similar structure, and without requiring execution of the code under analysis.

In this paper, we present OneSpace, a cross-language clone detection technique that achieves high effectiveness by jointly training a single vector space for different programming languages and leverages a Siamese network to assess cross-lingual clones projected to this common space. The main insight behind OneSpace is that its single-space projection would assign close coordinates to semantically-similar code tokens and use them to match code clones across programming languages.

We performed eight experiments to evaluate OneSpace. First (Section 5), we measured OneSpace's effectiveness to detect cross-language code clones across the Java and Python languages, and we compared its results with the state-of-the-art general-purpose cross-language clone detection techniques, SupLearn [95] and CLCDSA [92]. Second (Section 6), we performed a sensitivity study to understand the separate impact of each of our design decisions in OneSpace. on its effectiveness. Next, we also investigated how some characteristics of our evaluation impact the effectiveness of OneSpace. So, in our third Experiment (Section 7), we studied the effectiveness of OneSpace and the state-of-the-art on an additional dataset to see how these techniques generalize to implementations of higher complexity. In our fourth experiment (Section 8), we studied the effectiveness of OneSpace in detecting single-method clones, as compared to the state of the art techniques. Fifth (Section 9), we performed an experiment to measure the efficiency of OneSpace for detecting cross-language code clones, which we also compared to the state-of-the-art. Next, we studied how some characteristics of the training data affect the results of our evaluation. In our sixth experiment (Section 10), we analyzed the impact of the ratio of positive examples (i.e., true clones) to negative examples (i.e., false clones) in the training set on the accuracy of OneSpace. In our seventh experiment (Section 11), we analyzed the impact of the amount of training data used on technique effectiveness. Finally, we conducted an eighth experiment (Section 12) to study the generalizability of our results to other programming languages. We evaluated the effectiveness of our studied techniques over the Java-C++ and Java-C language pairs.

In our experiments, OneSpace obtained 41.02% Fmeasure score for cross-language clone identification (RQ1). OneSpace strongly improved over the state-of-the-art techniques SupLearn [95] and CLCDSA [92], which obtained scores of 24.01% and 10.86% respectively. Our sensitivity analysis (RQ2) showed that the component that contributed most to OneSpace's effectiveness was its em-

bedding training step — embedding all languages into a single shared space. Our experiments with the additional test data (RQ3) showed that OneSpace still outperforms the state of the art when applied over more complex programs, and also when evaluated over single methods (RQ4).

Our efficiency analysis (RQ5) showed the trade-off between two OneSpace variants, where each OneSpace variant excelled in a different dimension: (1) using a Siamese Neural Network for similarity inference achieved higher f-measure effectiveness, but (2) using Word Mover Distance for inference highly increased OneSpace 's efficiency — reducing its f-measure. We also found that OneSpace still outperformed the state-of-the-art techniques for different ratios of clones to non-clones in the training data (RQ6). In RQ7, OneSpace provided high effectiveness for varying amounts of training data. As data grew, OneSpace generally increased its Precision and lowered its Recall. Finally, our experiments on the additional programming language pairs (RQ8) yielded comparable results to the Java-Python experiments. In them, OneSpace still outperformed the state of the art techniques in all our studied language pairs.

Finally (Section 13), we also analyzed OneSpace as well as the baselines in a different context, (in a ranking context), to see how well they would perform for clone search. This context offers an interesting perspective on the effectiveness of the techniques since it measures not only whether clones are detected, but also how accurately they are ranked. This additional evaluation also allows us to gain a more comprehensive understanding of the strengths and weaknesses of our approach. In this new evaluation, we measure the performance of the techniques in terms of normalized discounted cumulative gain at ranks (NDCG@k) In this context, OneSpace obtained a median NDCG@100 score of 63.28%, outperforming the other techniques, which achieved scores of 30.38% and 22.47% for SupLearn [95] and CLCDSA [92], respectively. This suggests that OneSpace not only effectively detects cross-language clones but also ranks them better than the state of the art techniques.

This work provides the following contributions:

- A cross-language clone detection technique, OneSpace, that outperforms the state-of-the-art in terms of effectiveness for detecting cross-language code clones.

  The technique is evaluated on different test sets of varying complexity as well as on different programming language pairs and outperforms the state of the art in both clone detection and clone ranking contexts.

- An analysis of the components of OneSpace, showing that the technique is most effective when using the shared space idea with the Siamese Neural Network.

- A study of the efficiency of cross-language clone detection techniques, demonstrating a trade-off between effectiveness and efficiency of cross-language clone detection: OneSpace is most effective when using a Siamese Neural Network while OneSpace provided its highest efficiency (but lower effectiveness) when using the word mover distance metric for assessing similarity.

- A thorough evaluation of the impact on the effectiveness provided by OneSpace of: complex implementations, single-method implementations, ratio or training clones, amount of training data, and additional programming languages.

2. Our Approach: OneSpace

OneSpace includes four main steps, which we show in Figure 1. OneSpace identifies how likely it is that a pair of code snippets in different programming languages are clones (*Prediction*) by employing a Siamese Neural Network.

The Siamese Neural Network learns from a Shared Embedding Space that assigns close coordinates to words with similar semantic meanings in different programming languages. The Shared Embedding captures the semantic meaning of a word by observing its context (*i.e.,* from the words preceding and following it), ignoring the content of the word itself, *i.e.,* ignoring its syntax. This way, different syntax constructs are assigned similar semantic meanings when used in similar contexts, and the same syntax construct is assigned different semantic meanings when it is used in different contexts.

OneSpace obtains this Shared Embedding Space (*Embedding Space Training*) by training it with examples of code in different programming languages. To increase the chances of functionally similar keywords in different programming languages obtaining similar coordinates in the Shared Embedding Space, OneSpace pre-processes the training data (*Data Preparation*), adding the documentation of the API of the programming languages involved. OneSpace trains its Siamese Neural Network by feeding it examples of true and false cross language clones (*Similarity Network Training*). Next, we describe these steps in more detail.

2.1. Data Preparation

In this first step, OneSpace prepares the data to train its Shared Embedding Space. As training data, OneSpace uses examples of code in multiple programming languages and the API documentation of those programming languages.

The goal of training with API documentation in addition to source code is to further help the embedding algorithm bridge the lexical gap between the different programming languages. Different programming languages

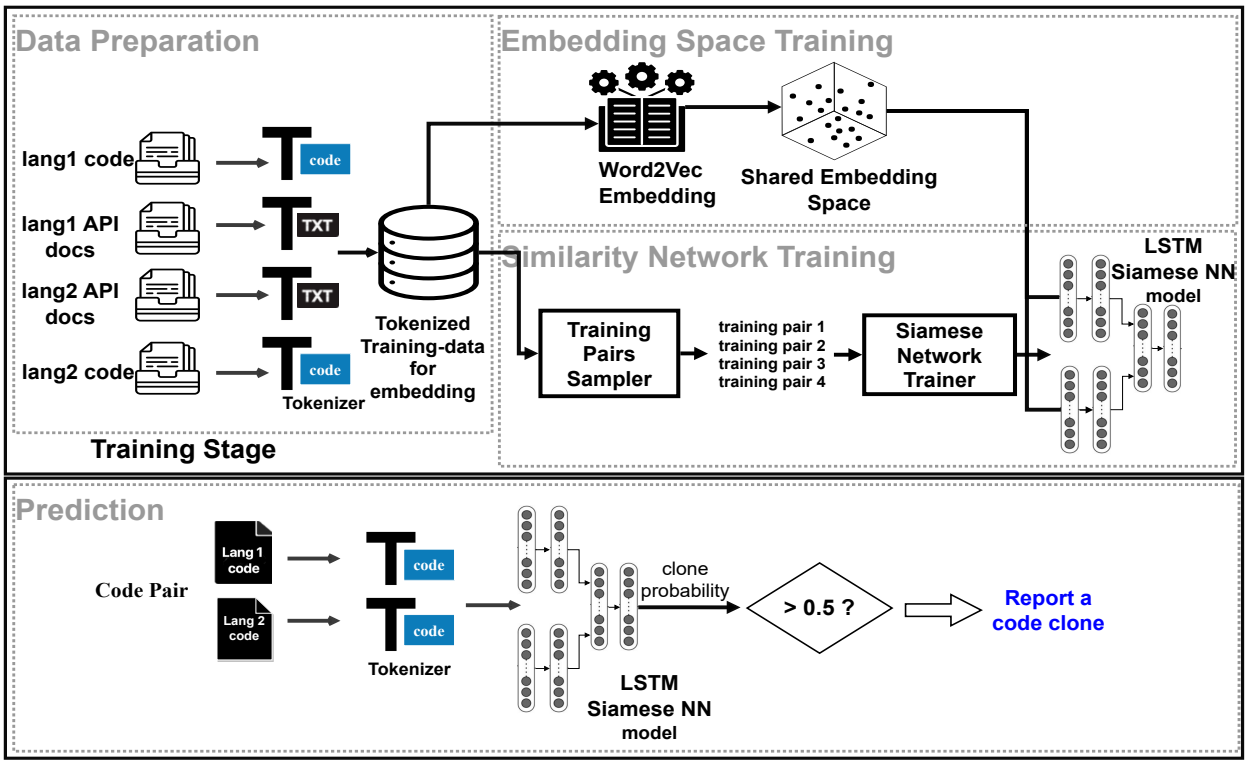

Figure 1: OneSpace flow diagram, in four steps; Data Preparation, Embedding Space Training, Similarity Network Training and Prediction

sometimes use different names for keywords, methods, classes and libraries that express the same concepts, *e.g.,* Java Hashtable vs. Python dictionary. The embedding training can assign similar coordinates to these different (but semantically similar) keywords if it observes that they are often surrounded by similar textual contexts. Thus, the goal of using API documentation (in addition to the code examples) to train OneSpace's embedding space is to provide it with a larger number of samples of such different (but semantically similar) keywords being used in similar contexts: (1) in the source code, being surrounded by similar code keywords, and (2) in the API documentation, being surrounded by similarly worded explanations.

Given a collection of code examples in different programming languages, and the API documentation of their programming languages, OneSpace pre-processes the data by tokenizing the code and documentation files and unifying the naming style of all identifiers. This helps to ensure that the data is in a consistent format and can be used effectively by our search algorithm.

2.2. Embedding Space Training

In its second step, OneSpace uses the tokens obtained in the Data Preparation step to train a Shared Embedding Space. It will later use this Shared Embedding Space to identify code clones in its Similarity Network Training step.

The goal of OneSpace's Shared Embedding Space is to capture different keywords in different programming languages that have similar meanings, by assigning them similar coordinates in a vector space. To achieve this, OneSpace trains a single, Shared Embedding Space for all the desired programming languages. In contrast, the previous work that applied embedding to cross-language clone detection [95] used separate spaces for different programming languages, which it then had to align to each other.

We believe that training a single Shared Embedding Space for different programming languages will allow OneSpace to provide higher effectiveness than if it aligned separate embedding spaces that belong to different programming languages. Different programming languages often have some lexical overlap — some language constructs use similar names and developers sometimes use similar identifier names across languages. Our intuition is that this lexical overlap would guide the embedding step as "anchors" in its space, and it would assign similar coordinates to different words that are used with similar meaning around them.

Similarly, we believe that training separate embedding spaces would make it harder to assign similar coordinates to words with similar meanings. In separate embedding spaces for separate programming languages, we expect that analogous *semantic clusters* of words would form in both spaces — *e.g.,* clusters forming with file processing methods in each language's space. However, finding a transformation that aligns *all semantic clusters* between spaces may often not be possible — a transformation that brings one set of semantic clusters closer together may set other set of semantic clusters further apart [123].

Furthermore, shared embedding spaces have been applied successfully to other, different problems. For instance, in the field of natural language processing, shared embedding spaces improved performance in tasks such as translation between natural languages [73] and natural language understanding [34]. Therefore, it is possible that, also in the domain of cross-language clone detection, a unified embedding space would also be more successful at capturing the same concepts (*i.e.,* program specifications) with different manifestations (*i.e.,* implementations) into a common representation (*i.e.,* embedding). We study the extent to which these intuitions are valid with an ablation study in which we measure the success of our approach with a shared vs. separate embedding spaces (see RQ2).

To train its Shared Embedding Space, OneSpace uses the tokenized documents obtained from the previous step (*Data Preparation*). Word embedding training takes as input a textual corpus of documents containing tokens. In OneSpace's context, a document in the corpus is a source code file or an API documentation webpage.

OneSpace's Embedding Space Training step returns a trained embedding space — a lookup table for all words in the corpus — where words that have similar contexts are located in closer coordinates to each other.

2.3. Similarity Network Training

After training its Embedding Space, OneSpace trains its Siamese Neural Network. In this training, the Siamese Neural Network learns to predict code clones based on the coordinates of their tokens in the embedding space as well as the order of such tokens.

OneSpace uses a Deep Siamese Neural Network [18]: two identical sub-networks that use the same weights, joined at their output layer. Each sub-network consists of an embedding layer that turns the code tokens into vector representations, followed by two recurrent bidirectional LSTM (Long-Short Term Memory) layers [45]. LSTM layers are a type of Recurrent Neural Network (RNN) architecture that are specifically designed to handle sequential data, such as code, by allowing the network to selectively remember important information and forget irrelevant information, which allows the model to effectively capture the long-term dependencies in the code. The embedding layers for the two sub-networks contain the pre-trained shared embedding space that OneSpace obtained in its Embedding Space Training step (Section 2.2).

Using a single set of weights for the two sub-networks has the benefit of likely reducing overfitting [42], which we expect would also improve effectiveness.

OneSpace trains its Siamese Neural Network using a collection of positive (*i.e.,* clone) and negative (*i.e.,* not-clone) pairs of implementations in different languages. For each pair, OneSpace tokenizes the two implementations, embeds them, feeds them to the Siamese network's last layer, observes their ground-truth class (*i.e.,* clone or not-clone), and propagates the error back to update the learned weights.

2.4. Prediction

In the final step, OneSpace is used to detect clone pairs within a dataset of implementations in different languages. Given a dataset of Lang1-Lang2 implementation pairs, OneSpace compares each Lang1 implementation to its corresponding Lang2 implementation, and estimates how likely it is that they are clones. Then, it predicts as clones those with a likelihood above a certain threshold, *e.g.,* 0.5.

To assess the similarity between the two implementations of a given pair, OneSpace first tokenizes both implementations as described in Section 2.1. Then, it applies its identical embedding layers to transform the sequence of tokens into sequence vector embeddings. Then, the LSTM layers of the Siamese Network process the sequence vector embeddings and the final layer calculates a probability of the two sequences being clones.

3. Implementation Details

In this section, we describe the implementation details of our approach, OneSpace, described in Section 2.

For its *Data Preparation* step (see Section 2.1), We downloaded the official API documentation for Java SE-7 [94], Python 3.7.4 [97]from their official websites. OneSpace strips the HTML from the API documentation, and then tokenizes the code data using the Python "sctokenizer" library [2] and the documentation data using a simple text tokenizer. OneSpace also converts camel case tokens to lowercase and remove underscore characters to unify the naming style of all identifiers across different programming languages. This is important because different programming languages use different conventions for naming, such as camel case for Java or snake case for Python.

For the *Embedding Space Training* step (see Section 2.2), we utilize a widely used technique called Continuous Bag of Words (CBOW) Word2Vec algorithm [90, 91]. CBOW is a popular method for word embedding, which captures contextual information by predicting a target word based on its surrounding words within a given context window. In our approach, we set the context window size to 10, which determines the number of words considered on both sides of the target word. We also specify a word vector dimension of 100, which determines the size of the vector representation for each word in the embedding space. Additionally, we set a minimum frequency threshold of 10 for including words in the training process, which ensures that only words occurring frequently enough in the data are considered during the embedding training.

We chose to use Word2vec rather than the newer CodeBERT model [12] because CodeBERT imposes a limit of 512 words per input (in our case, per processed implementation), which would have severely limited the practical

[2] `https://pypi.org/project/sctokenizer/`.

usage of our approach. In fact, 30% of the tokenized implementations in our experiments were longer than 512 words. Word2vec does not impose such a limit.

For the *Similarity Network Training* step, (see Section 2.3) Our Siamese Neural Network consists of two identical sub-networks that use the same weights, connected at their output layers. Each sub-network consists of an embedding layer and two recurrent bidirectional LSTM layers, with the embedding layers containing the same pre-trained shared embedding space. We use a binary cross-entropy loss function as the objective function, an Adam optimizer with Nesterov momentum and an early stop option. Each sub-network comprises two layers of 50 LSTM cores with relu activation and a 0.25 drop-out rate.

For *Prediction* (Section 2.4), we flag as clones those pairs with an estimated clone likelihood that is higher than 0.5.

4. Research Questions

We evaluate OneSpace with experiments that answer eight research questions.

***RQ1*: How effective is OneSpace for detecting code clones across programming languages?**
***RQ2*: What is the impact of each individual component of OneSpace on its effectiveness?**
***RQ3*: How sensitive are the findings of *RQ1* and *RQ2* to the complexity of clone implementations?**
***RQ4*: How does effectiveness change in cross-language clone detection for single methods?**
***RQ5*: How efficiently does OneSpace provide its cross-language code clone recommendations?**
***RQ6*: How does the clone ratio in the training data affect the effectiveness of the studied techniques?**
***RQ7*: What is the impact of the amount of training data on technique effectiveness?**
***RQ8*: How effective is OneSpace for other programming language pairs?**

5. *RQ1*: How effective is OneSpace for detecting code clones across programming languages?

First, we evaluate OneSpace 's effectiveness by using it to detect software clones in a dataset of cross-language code clones across Java and Python. We study Java and Python clones, as was done in previous studies, *e.g.,* [92, 95]. To provide a reference point for assessing OneSpace 's effectiveness, we also executed two state-of-the-art cross-language clone-detection techniques, SupLearn [95] and CLCDSA [92], over the same dataset. We compared the effectiveness provided by OneSpace, SupLearn and CLCDSA using three different metrics: Precision, Recall, and Fmeasure.

5.1. Evaluation Dataset

In its typical usage, developers apply automatic clone detection using their codebase as input [28, 29]. The input codebase may contain any combination of modules and projects (one or many, partial or complete) and programming languages (in the case of cross-language clone detection) [28, 29]. As output, automatic clone detection techniques return an assessment of which code pairs in the codebase are clones [28, 29, 100].

Clone detection techniques that are based on machine learning also require a set of pre-labeled code clones in the input codebase, for their training process. This is the case for all our evaluated techniques: OneSpace (our proposed approach), SupLearn [95], and CLCDSA [92]. If a codebase has no pre-labeled clones yet (*e.g.,* in the very first run of the technique), developers may label some clones manually, or they may add to the training set some pre-labeled clones from other codebases.

To evaluate our studied techniques in a representative setting, we needed a codebase with pre-labeled code clone pairs in multiple programming languages. Building such a codebase is inefficient and error-prone, since it requires manually inspecting a large number of code pairs in the codebase to assess and label which ones are clones. Therefore, we used a codebase that was used in the original evaluation of all our studied techniques: the AtCoder dataset [92, 95].

AtCoder is an independent, publicly-available codebase with cross-language clone and non-clone code pairs. It was compiled from a competitive programming website [11], in which developers create implementations for given software specifications. It only contains correct implementations, *i.e.,* that were accepted by the website judging system as correctly fulfilling the specification [11, 95]. AtCoder provides an objective pre-labeled assessment for which code pairs are clones *i.e.,* those fulfilling the same specification.

AtCoder contains 576 different software specifications, divided into:

1. **AtCoder-b** which comprises 300 "beginner" problem specifications — as specified in the AtCoder website, and
2. **AtCoder-r** which comprises 276 "regular" problem specifications — as specified in the AtCoder website.

Each specification has a number of implementations in Java and a number of implementations in Python that are clones of each other. In total, it contains 50,091 implementations (20,828 in Java and 29,263 in Python). We did not investigate the characteristics of this dataset before designing our technique, and thus the dataset did not influence the design of our technique.

There are multiple reasons why we believe that the results obtained by our evaluated techniques in the AtCoder codebase can be representative of the results that they would obtain in other codebases.

First, AtCoder contains a diverse set of software specifications, of varying purpose and complexity, written by diverse developers (with varying coding and problem-solving styles). This helps represent the diversity of codebases that are multi-lingual (and the diversity of developers that contribute to them) — more than 15,000 projects in Github are multi-lingual [118].

Second, AtCoder contains implementations of relatively limited size — the median implementation in it contains only one method. This helps represent the length of the code fragments for which developers typically apply clone detection (*i.e.,* single methods) [80, 81].

Third, codebases from competitive programming (*e.g.,* AtCoder) sometimes contain bugs and vulnerabilities [30]. This helps our experiments represent one of the most common usages of automatic clone detection — finding the clones of code that is known to be buggy or vulnerable [28, 29, 98, 122].

Fourth, past work obtained results on competitive programming codebases that generalized to other codebases, *e.g.,* for code de-anonymization [21]. This supports our expectation that results obtained on competitive programming codebases would generalize to other codebases also for other code analysis tasks, such as clone detection.

These may be some of the reasons why past work also evaluated their cross-language clone detection techniques with AtCoder, *e.g.,* [92, 95].

5.2. Training and Testing Process

For *RQ1*, we followed the experiment design of SupLearn [95] by evaluating our studied techniques over the dataset's 300 "beginner" specifications. We study "regular" specifications separately in *RQ3* (see Section 7). We query our studied techniques with Java-Python pairs of implementations and ask them to predict which ones are cross-language clones.

5.2.1. Training

We used 10-fold cross-validation, *i.e.,* we randomly divided the 300 specifications into 10 folds of 30 specifications each. We then tested our studied techniques separately for each fold. For each fold, we trained the techniques with the 270 specifications of the remaining 9 folds, and with the 276 specifications for "regular" problems — to take advantage of as much data as we had available.

Our studied techniques train their Siamese network with positive (*i.e.,* clones) and negative (*i.e.,* not clones) examples of implementation pairs in different languages. Training our studied techniques with every possible Java-Python implementation pair in our training data would take too long, given the combinatorial explosion of all possible pairs. Thus, we created a random sample of positive and negative examples from our dataset (*i.e.,* sampling from both "beginner" and "regular specifications") to train our studied techniques. For positive examples, we randomly sampled 25 Java-Python clones from each specification. If a specification had fewer clones, we collected all of them for it. This amounted to 13,435 positive examples, sampled from all the 576 specifications. For negative examples, we randomly sampled 26,565 negative examples, also from all specifications. We sampled approximately double for the number of negative examples, since we expect them to be more common in the real world. Our sampled positive and negative examples add up to 40,000 total examples for training.

For each testing fold, we trained our studied techniques with our sampled positive and negative implementation pairs for which both their Java and Python implementations addressed specifications in the remaining 9 folds of "beginner" problems or specifications of "regular" problems. This resulted in a median of 37,239 training pairs per testing fold.

This process represents how we expect the developers would train our studied techniques to apply them to their own codebases. We also describe in Sections 2.1, 2.2 and 3 how developers can run OneSpace's *Data Preparation* step, processing the multi-language implementations in their codebase.

To train our studied techniques, developers can use a sample of the pre-labeled clones and non-clones in their codebase, as described earlier in this section (and in: Sections 2.3 and 3 for OneSpace; Section 5.3.2 for SupLearn; and Section 5.3.3 for CLCDSA). Developers often have some pre-labeled clones in their codebase (*e.g.,* manually labeled clones are a common manifestation of self-admitted technical debt [15]), whereas non-clones can be easily generated randomly if needed.

However, there may be some cases in which developers may not have pre-labeled clones and non-clones in their codebase, *e.g.,* in the very first run of a clone detection technique over a codebase. How to best address this situation is a known challenge that is currently being investigated in the clone detection research field [129] — and is thus beyond the scope of our paper.

There are various approaches that developers could take to obtain some code clones for training in this initial execution: manually labeling a limited number of clones in their codebase, using pre-labeled clones from other codebases, synthesizing clones in their codebase using mutation operators [99, 36], or using more complex program transformations [127]. Our experiments resemble the approach of using pre-labeled clones from other codebases, since we trained our studied techniques with implementations for specifications that were not implemented in the test set, *i.e.,* we kept the specifications (and implementations) in the training and test sets separate.

Finally, for subsequent executions, developers can iteratively extend their training set with the clones identified in the previous execution — extending the training set in this way is known in machine learning as *fine-tuning*, and it typically improves effectiveness [25, 46, 70, 34].

5.2.2. Testing

For each testing fold, we used sampling again to address the combinatorial explosion of implementation pairs that techniques would have to evaluate, as is common in the evaluation of clone-detection techniques [95, 117]. Our testing folds had a median number of 1,589 Java implementations and 2,326 Python implementations. Taking all combinations of these would have resulted in approximately 4 million Java-Python pairs that techniques would have had to evaluate for each fold. Thus, for each testing fold, we randomly sampled 1,000 implementations — ensuring that they covered all specifications, with both Java and Python files. We repeatedly sampled until this condition was true. Sampling brought down the median number of implementations per fold to 397 Java ones and 603 Python ones.

We evaluated our studied techniques over each possible Java-Python implementation pair from these sampled implementations, resulting in a median of 239,375 Java-Python implementation pairs evaluated per fold. In total for all folds, we evaluated 2,391,351 implementation pairs, belonging to 300 specifications, 130,473 of which were clones. The ratio of clones to non-clones is $\approx$ 1:17.

5.3. Studied Techniques

We evaluated three techniques over the same dataset: our proposed approach OneSpace, and the two state-of-the-art ones, SupLearn [95] and CLCDSA [92]. We obtained the code for both the SupLearn and CLCDSA techniques from their respective GitHub repositories [116, 27] and used them as baselines for comparison with our proposed approach.

5.3.1. OneSpace

Our proposed approach is described in Section 2 and its implementation details are explained in Section 3.

5.3.2. Suplearn

SupLearn [95] uses a Siamese Neural Network to predict whether a pair of implementations in different languages are clones. Given a pair of implementations, SupLearn first extracts their ASTs and parses each AST into a sequence of nodes. It then feeds the two node sequences to two separate embedding layers, one for each language, to generate the embeddings of the program tokens. The two embedded spaces are connected to a multi-layer Siamese Neural Network that predicts whether the pair of implementations are clones or not.

For training, SupLearn uses as input a collection of positive (*i.e.,* clones) and negative (*i.e.,* not clones) examples of implementation pairs in different languages. This process informs the weights in its Siamese Neural Network.

Then, users can query SupLearn with a new pair of implementations and it predicts whether they are clones.

There are multiple differences between the design of SupLearn and our proposed technique OneSpace. First, OneSpace uses a single embedding space, as opposed to SupLearn's two embedding spaces. We posit that this design decision will allow OneSpace to provide higher effectiveness, because OneSpace will not have to align two spaces that have been trained separately — such a process can often be inaccurate [102, 123].

Also, using a single embedding space allows OneSpace to use the typical Siamese Network weight sharing regularization method. This is different to SupLearn, which uses different weights for the Siamese Network sub-networks. This weight sharing typically reduces the possibility of overfitting and therefore is expected to achieve better generalization [42].

Finally, OneSpace models implementations extracting the words used in them, instead of parsing their code to obtain their AST (as in SupLearn). We introduced this decision to make OneSpace easier to generalize to many programming languages, *i.e.,* it does not need to implement a new parser to obtain ASTs for each new programming language.

5.3.3. CLCDSA

CLCDSA [92] applies two steps: action filter and prediction. First, CLCDSA's action filter discards improbable clones. Given a pair of implementations, the action filter extracts all API calls in them, obtains the documentation for each API call, and projects the documentation into a word2vec text embedding space that was pre-trained by Google [89]. The action filter then calculates all the pairwise cosine similarities between the coordinates of all the possible pairings of projected API-call-documentations between the two implementations. Next, the action filter sorts all pairs by cosine similarity and applies a greedy approach to match individual API calls in one program to (estimated corresponding) API-calls in the other program. The action filter then computes the average cosine similarity of the matched pairs. If the average cosine similarity is lower than 0.5, it marks the two programs as an improbable clone pair.

Second, for the implementation pairs that remain after applying the action filter, CLCDSA applies its prediction step. This step predicts whether a pair of implementations are clones using a Siamese Neural Network with statistical features as input. CLCDSA extracts 9 statistical features for each implementation (18 features, 9 per implementation), *e.g.,* its number of variables, its number of operators, and its Cyclomatic complexity [85]. CLCDSA uses these feature vectors to train its Siamese network. To predict if a new implementation pair are clones or not, CLCDSA queries its trained Siamese network.

The design of CLCDSA is very different form OneSpace. CLCDSA measures similarity between projected embeddings, but it does so in a broad-strokes manner, only to filter improbable clones. Also it compares very different things than OneSpace does: CLCDSA compares the projection of the documentation of the APIs used in the implementations on a pre-trained model, and OneSpace

compares the projection of the tokens used in the implementations using a model trained with cross-language implementations and the complete API documentation of the programming languages. Then, CLCDSA predicts clones based on their similarity in terms of statistical features. OneSpace predicts clones based on the semantic similarity of the tokens used in their implementations as well as their order. We anticipate that CLCDSA will successfully predict clones that have very similar statistical features, but will find it harder to predict clones with different statistical features, even if they use language that is semantically very similar.

In our experiments, for a given test fold, CLCDSA first applied its action filter to filter out less probable clone pairs. Then, it used its Siamese network to predict whether each of the remaining pairs are clones.

5.4. Evaluation Metrics

We evaluate cross-language clone detection using the Precision, Recall, and Fmeasure metrics.

Precision: measures, within the pairs classified as clones, the proportion of them that are true clones (as in eq. (1)).

$$\text{Precision} = \frac{|\{\text{True clone pairs}\} \cap \{\text{positively classified pairs}\}|}{|\{\text{positively classified pairs}\}|} \cdot 100 \quad (1)$$

Recall: measures the proportion of predicted true clones out of all existing true clones in the test data (as in eq. (2)).

$$\text{Recall} = \frac{|\{\text{True clone pairs}\} \cap \{\text{positively classified pairs}\}|}{|\{\text{True clone pairs}\}|} \cdot 100 \quad (2)$$

Fmeasure: The harmonic mean of Precision and Recall.

$$\text{Fmeasure} = \frac{2*\text{Precision}*\text{Recall}}{\text{Precision}+\text{Recall}} \quad (3)$$

5.5. RQ1 Results

We report the results of our experiment to answer *RQ1* in Table 1. To calculate the results, we constructed a confusion matrix with all the implementation pairs within the 10 Java-Python testing folds, for each studied technique. We then measured Recall, Precision, and Fmeasure on each matrix, for each studied technique.

Our first observation from the results is that OneSpace provided higher effectiveness (Fmeasure) than the state of the art techniques. OneSpace provided a higher f-measure than SupLearn. It provided much higher Precision (by about 20 points) and higher Recall than SupLearn (by about 9 points), leading to a high difference in f-measure.

OneSpace also provided a higher f-measure than CLCDSA. Although CLCDSA provided higher Recall than OneSpace, it also provided very low Precision, resulting in low f-measure.

Table 1: (*RQ1*) Effectiveness of OneSpace in terms of Recall, Precision and Fmeasure metrics compared to the baselines.

Approach	Recall	Precision	Fmeasure
OneSpace	45.07	37.63	41.02
SupLearn	36.55	17.87	24.01
CLCDSA	71.11	05.88	10.86

Our second observation is the relatively low Precision of all techniques. This may be because our studied testing data contained a low ratio of clones to non-clones ($\approx$ 1:17), making it difficult to achieve high Precision.

Finally, It's worth noting that in our experiments SupLearn outperformed CLCDSA, even though CLCDSA obtained higher f-measure than SupLearn in its original evaluation [92]. This may be because of our evaluation setup. In our evaluation, we applied cross-validation separating the training and testing sets in terms of problem specifications. That is, the implementation pairs in the testing set followed problem specifications that were different from the problem specifications that the implementation pairs of the training set followed. For example, in the original evaluation of CLCDSA, it may have trained its model using some implementation pairs for, *e.g.,* quicksort, and then may have been asked to predict for other implementation pairs of quick-sort. This may have improved the effectiveness of CLCDSA in its original evaluation.

6. *RQ2*: What is the impact of each individual component of OneSpace on its effectiveness?

In this research question, we evaluate the relative importance of each of OneSpace's components on its effectiveness. For that, we created three variants of OneSpace, substituting each of its components by an alternative one. We evaluated these variants with the same evaluation design that we used for *RQ1* (see Section 5).

6.1. Studied Variants of OneSpace

We created three variants of OneSpace. Every variant substitutes one of its components with a common alternative for the same purpose, *i.e.,* data preparation, embedding space training, and prediction. We depict these variants in Figure 2, and we describe them below.

6.1.1. OneSpace-Code

We used this variant to study the importance of our choice of including API specifications in OneSpace's training. This variant changes OneSpace's data preparation step to train its shared embedding space only with code, without analyzing the programming languages' API specifications (Figure 2.a). Analyzing only source code is a common choice in clone-detection techniques *e.g.,* [24, 95, 124].

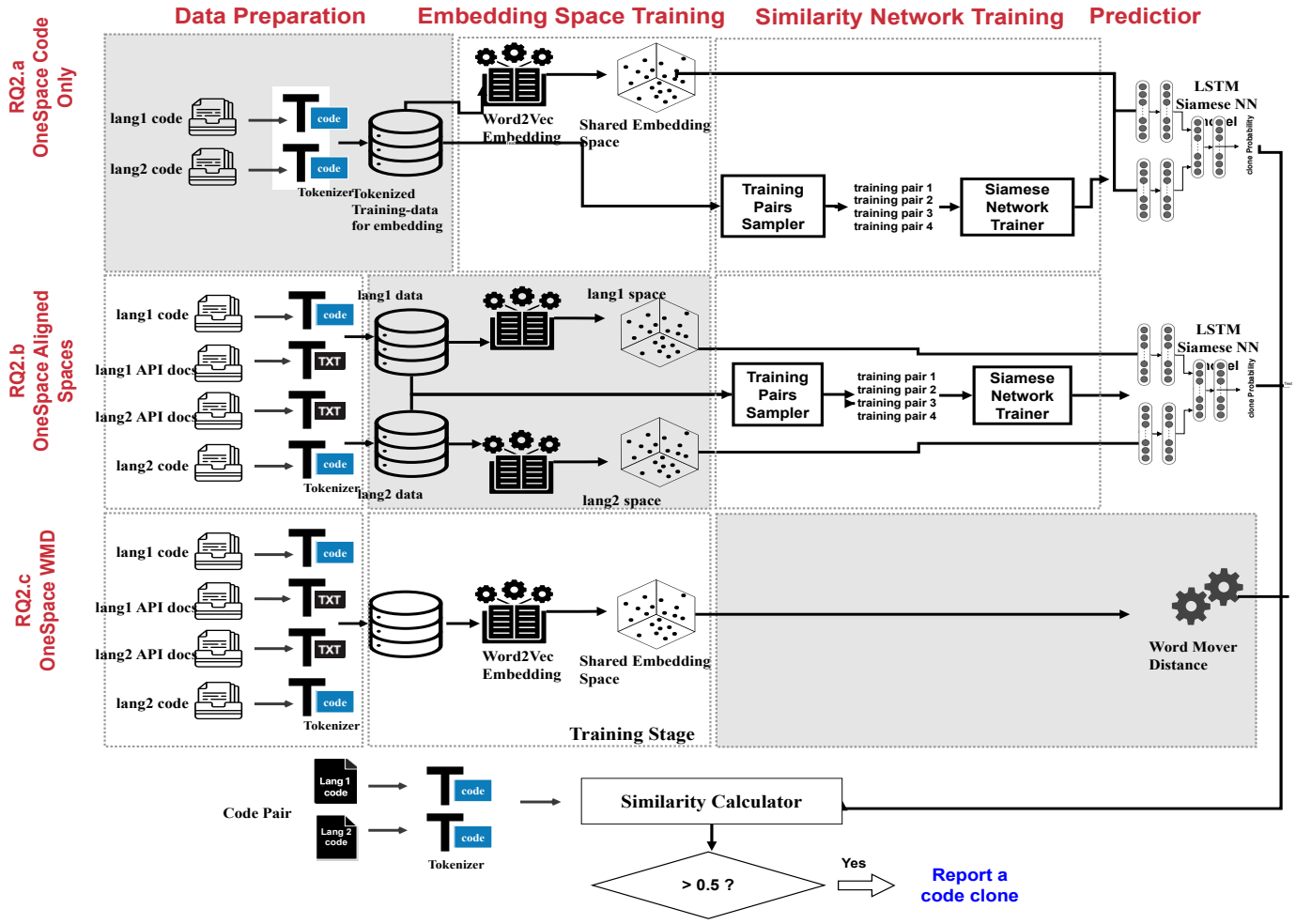

Figure 2: Sensitivity analysis. We studied four variants of OneSpace. Each one (each row) changes one of OneSpace's components.

6.1.2. OneSpace-AlignedSpaces

We created this variant to study the importance of training OneSpace's embedding as a single embedding space. As we discussed in Section 2.2, we expect that training a single embedding space (original OneSpace) would provide better alignments of semantically related concepts than training separate spaces and aligning them afterwards with a Neural Network (this variant).

This variant changes OneSpace's language embedding step to an alternative design: it trains two separate language embeddings instead of one — one trained with Java code and API docs, and the other with Python code and API docs (Figure 2.b). This alternative embedding design is common for aligning natural languages [102] and has been also used to map API usages across programming languages [93]. The design of this variant is also similar to SupLearn's, in that it uses two separate embedding spaces that are later aligned by a Siamese network. However, it is different in that it trains its embedding spaces from the tokens in the source code, while SupLearn trains them from the AST nodes of the source code.

6.1.3. OneSpace-WMD

The goal of this variant is to understand the importance of assessing similarity with a Siamese Neural Network. This variant changes OneSpace's prediction step to use a different similarity assessment method: Word Mover's Distance (WMD) [71] (Figure 2.c). WMD measures the minimum cumulative distance that needs to be traveled from each word in a first document to match the cloud of words in a second one. Measuring document similarity with embedded spaces using WMD produced strong results in other domains, *e.g.,* natural language processing [60, 71, 75, 104].

OneSpace-WMD predicts an implementation pair as clones when they are among the most similar within their prediction batch. In practice, machine learning predictions are often requested in batches, since it enables faster predictions [9, 67, 126]. Selecting a static similarity threshold could have biased prediction, making it more likely to predict shorter implementations as clones. Therefore, OneSpace-WMD uses a threshold for each implementation pair that is relative to the other pairs in their batch — aiming to connect it to the larger context in which the prediction

Table 2: (*RQ2*) Sensitivity Analysis Results.

Approach	Recall	Precision	Fmeasure
OneSpace	45.07	37.63	41.02
OneSpace-Code	48.16	34.57	40.25
OneSpace-AlignedSpaces	36.88	17.89	24.10
OneSpace-WMD	28.70	22.62	25.30

is being requested. OneSpace-WMD batches together all the implementation pairs that have the same Java implementation. Then, it predicts as clones those implementation pairs with the 7% lowest similarity of their batch. We chose this value based on the lowest reported number for percentage of clones in codebases in the literature, which is estimated to be around 7-23% [17, 101].

6.2. RQ2 Results

We report in Table 2 the scores obtained by the three variants of OneSpace that we studied, for all our studied metrics.

OneSpace-Code obtained very close results to OneSpace. OneSpace-Code achieved slightly higher Recall and lower Precision than the original variant, ending with a slightly lower f-measure. This shows that using API documentation to train the Shared Embedding Space did have a slightly positive impact on OneSpace's effectiveness, which validates our intuition when designing OneSpace (see section 2.1), and also confirms its positive impact reported in related work [92, 93].

OneSpace-AlignedSpaces achieved much lower effectiveness than OneSpace for all metrics. This means that OneSpace highly benefited from the decision of training its embedding in a single Shared Embedding Space. This observation validates the expectations that we described when designing our approach: that a single embedding space would achieve higher effectiveness than two embedding spaces that are later aligned (see section 2.2).

This may also be the main reason why OneSpace achieved much higher effectiveness than SupLearn in our experiments to answer *RQ1*. SupLearn obtained very similar effectiveness for all metrics than the OneSpace-AlignedSpaces variant, and they both use separate embedding spaces that are later joint with a Siamese Neural Network.

OneSpace-WMD achieved much lower effectiveness than OneSpace for all metrics. This validates our design choice of assessing similarity with a Siamese Neural Network (see section 2.3), since it greatly benefited OneSpace's effectiveness.

Summing up, OneSpace strongly benefited from two of its design choices: training its embedding space as a single space, and assessing similarity using a Siamese Neural Network. It benefited more strongly from training a shared embedding space, but it still benefited very strongly from both choices.

7. *RQ3*: How sensitive are the findings of *RQ1* and *RQ2* to the complexity of clone implementations?

In *RQ3*, we study how much the findings in *RQ1* and *RQ2* would change if we apply our studied techniques to more complex implementations. The AtCoder dataset [11] contains code clones from 300 "beginner" problem specifications and 276 "regular" problem specifications. The state-of-the-art cross-language clone-detection techniques were evaluated only with "beginner" specifications [116] or with a random sample from the mix of both "beginner" and "regular" specifications [27]. In this paper, we present the first study to evaluate cross-language clone-detection techniques separately for the two kinds of specifications, to understand the extent to which they perform differently for each category.

7.1. Evaluation Dataset

The "regular" specifications in our dataset comprise our AtCoder-r dataset. These specifications contain more lines of code in their implementations (with a mean of 100.5 Java and 19.7 Python lines) than the "beginner" specifications (with mean 44.1 and 12.2 Python lines). They also have higher Cyclomatic Complexity in their implementations (mean 16.4 in Java and 4.7 in Python) than "beginner" specifications do (median 6.1 in Java and 3.5 in Python). The dataset creators [95] characterize the "beginner" implementations as less likely to follow completely different algorithms to each other and thus being closer to type III clones, while the "regular" specifications are considered closer to type IV.

7.2. Experiment Design

For this RQ, we repeated our experiments for *RQ1* and *RQ2*, using the AtCoder-r dataset for testing. We used a similar training and testing process as we did for *RQ1* and *RQ2*.

First, we curated the dataset, discarding specifications which had implementations in only one programming language. Out of the 276 "regular" specifications in our dataset, we kept the 210 ones that contained implementations for both Java and Python.

Then, we divided the specifications into folds, for training and testing. We randomly divided the 210 "regular" specifications into 7 folds, to maintain the same number of specifications per fold (30) as in *RQ1* and *RQ2*. We then tested our studied techniques separately for each fold.

7.2.1. Training Process

For each fold, we trained the techniques with the 180 specifications of the remaining 6 folds, and with the 300 specifications for "beginner" problems — to take advantage of as much data as we had available (as we did for *RQ1* and *RQ2*). That is, we used the same training data as in *RQ1* and *RQ2* (see Section 5.2.1). This training data originally contained positive and negative examples from

Table 3: (*RQ3*) Results of Studied techniques on AtCoder-r regular specifications.

Approach	Recall	Precision	Fmeasure
OneSpace	50.11	20.17	28.76
OneSpace-Code	51.35	19.23	27.98
OneSpace-AlignedSpaces	38.73	11.13	17.29
OneSpace-WMD	22.80	21.26	22.00
SupLearn	37.74	10.72	16.69
CLCDSA	67.31	07.23	13.05

both beginner and regular specifications. For each regular specification fold we tested on, we trained our studied techniques with our sampled positive and negative examples for which both their Java and Python implementations addressed specifications in the remaining 6 folds of AtCoder-r dataset or specifications of AtCoder-b dataset. This resulted in a median of 37,660 training pairs per fold.

7.2.2. Testing Process

For testing, we also sampled implementations from our testing fold, to limit the combinatorial explosion of our evaluated Java-Python pairs. As we did in *RQ1* and *RQ2*, we randomly sampled 1,000 implementations from each fold, ensuring that all sampled specifications had both Java and Python implementations. After sampling, our tested folds had a median of 486 Java implementations and 514 Python implementations. We evaluated our studied techniques over each possible Java-Python implementation pair from these sampled implementations, resulting in a median of 249,775 Java-Python implementation pairs evaluated per fold. In total for all folds, we evaluated 1,561,465 implementation pairs, 100,237 of which were clones. The ratio of clones to non-clones is $\approx$ 1:15.

7.3. RQ3 Results

Tables 3 shows the scores obtained by all techniques in this experiment. We applied metrics as we did for *RQ1* in Table 1.

We found that most techniques obtained lower effectiveness for "regular" specifications than for "beginner" ones — except CLCDSA. However, the relative differences of effectiveness between techniques remained. This means that most of the observations that we made in *RQ2* are also valid for more complex specifications.

OneSpace still provided the highest effectiveness (f-measure), largely improving over every other technique. OneSpace-Code provided slightly lower (but very close) Precision, Recall, and f-measure to OneSpace, showing again that our choice of training with the API documentation had a limited (but positive) influence. Also again, OneSpace-AlignedSpaces and SupLearn provided similar results to each other, since they use similar designs. This again validates our expectation that using a single embedding space provides higher effectiveness (*i.e.,* OneSpace was more effective than OneSpace-AlignedSpaces).

OneSpace-WMD provided lower effectiveness (f-measure) than OneSpace, but not too much lower. This may be a beneficial property to make it a good technique to use in situations when the efficiency of the technique is very important, *e.g.,* when a large amount of implementations need to be requested in a large batch. Even though OneSpace-WMD provided lower effectiveness than OneSpace, the difference in effectiveness between them is lower for more complex specifications, and OneSpace-WMD is faster than OneSpace (we observe that in our experiment for *RQ5*).

For CLCDSA, while its effectiveness (f-measure) improved, it remained fairly low. It's possible that this slight improvement (particularly in terms of Precision) may be because larger programs allowed the action filter to more easily filter out non-clones (for larger programs, non-clones are less likely to use semantically related APIs, and thus are easier to identify as non-clones).

8. *RQ4*: How does effectiveness change in cross-language clone detection for single methods?

Clone detection techniques are typically applied for the granularity of methods, *i.e.,* to detect multiple methods implementing the same functionality. However, our previous research questions evaluated techniques over a dataset in which clones could be single-method or multi-method (as was the case in the original evaluations of our studied state-of-the-art techniques [92, 95]).

We also wanted to obtain a better understanding of technique effectiveness at the granularity of individual methods. So, we performed an additional experiment. We applied all our studied techniques and variants over implementations from our dataset that only included a single method.

8.1. Experiment Design

We used the same methodology as in RQ1, RQ2, and RQ3, but this time including only single-method implementations. That is, we evaluated OneSpace, all its variants, SupLearn, and CLCDSA over our studied implementation pairs from the AtCoder-b and AtCoder-r datasets that were single-method.

8.2. Evaluation Dataset

We identified single-method implementations in the following way: For Python, we wrapped all method-free code in a single method called "solve", which we called at the beginning of the main method. We created a main method if it didn't exist. Then, we counted the number of methods in the implementation, excluding the main method when it only contained a single call to "solve". For Java, we counted the number of methods within the class(es), excluding the main method when it only had a single call to one of the existing methods. For both Java and Python, we marked implementations as single-methods if their count of methods was 1.

8.3. Training and Testing

We used the same training data as in RQ1 and RQ2 for AtCoder-b (see Section 5.2.1), and in RQ3 (see Section 7.2.1) for AtCoder-r.

We tested techniques with the implementation pairs that we sampled in RQ1 and RQ2 for AtCoder-b (see Section 5.2.2), and in RQ3 (see Section 7.2.2) for AtCoder-r, but using only those pairs for which both implementations were single-method. We also performed cross-validation, keeping the same folds, but only including the testing pairs that contained a single method for both the Java and Python implementations within each fold.

For AtCoder-b, the resulting folds had a median 28 specifications, median 153,136 single-method Java-Python implementation pairs. In total for AtCoder-b, this experiment evaluated 1,478,665 single-method Java-Python implementation pairs, out of the 2,391,351 pairs that we evaluated for RQ1 and RQ2. 86,551 of these were clones. The ratio of clones to non-clones is $\approx$ 1:16.

For AtCoder-r, the resulting folds had a median 28 specifications, median 73,920 single-method Java-Python implementation pairs. In total for AtCoder-r, this experiment evaluated 492,970 single-method Java-Python implementation pairs, out of the 1,561,465 pairs that we evaluated for RQ1 and RQ2. 39,312 of these were clones. The ratio of clones to non-clones is $\approx$ 1:12.

8.4. Results.

We report the results of this experiment in Table 4. When considering only single-method implementations, our approach OneSpace still provided the highest effectiveness (f-measure). It provided higher effectiveness (f-measure) than its variants, and than the state-of-the-art techniques SupLearn and CLCDSA, for both the AtCoder-b and AtCoder-r datasets.

All our studied techniques obtained higher f-measure in this experiment than in previous experiments, for all metrics and datasets (particularly for AtCoder-r). In other words, all techniques were more successful when detecting single-method clones. Again, the relative differences of effectiveness between techniques remained as in RQ1, RQ2, and RQ3.

In terms of techniques, OneSpace again provided the highest f-measure in this experiment, followed by SupLearn, and CLCDSA, respectively (as in RQ1, RQ2 and RQ3). Also again, SupLearn provided lower Recall and Precision than OneSpace, and CLCDSA provided higher Recall than other techniques, but at the expense of providing very low Precision.

In terms of variants of OneSpace, their f-measure also compared to each other as it did in RQ1, RQ2 and RQ3. OneSpace-Code provided very similar f-measure to OneSpace, showing again that training with API documentation had limited (positive) impact. OneSpace-AlignedSpaces and OneSpace-WMD provided f-measure similar to each other, and a bit below OneSpace-Code. This again validates our expectation that using a single embedding space provides higher effectiveness.

9. *RQ5*: How efficiently does OneSpace provide its cross-language code clone recommendations?

In *RQ5*, we aim to understand the efficiency with which our studied techniques provide their recommendations. We measured the execution time of each of our studied techniques, executing them for a randomly selected testing fold out of the ten folds that we investigated (see section 5.2). We executed all the techniques over the same fold.

The randomly selected testing fold contained 242,079 Java-Python implementation pairs — all the possible combinations of 411 Java implementations and 589 Python implementations. To expedite the experiment, and as is often done in practice, we grouped these pairs into batches [67, 126]. Each batch contained all the pairs corresponding to each Java implementation, paired with each one of the Python implementations. We queried each technique for each of the 411 batches, and we recorded the time that they took to provide the results. This effort took a total of 32 hours. We ran all these experiments serially in a single machine: a 2-core Windows machine with 4.10 GHz processor speed and 16 GB of memory.

We also measured the one-time training time of each of our studied techniques. OneSpace, OneSpace-Code, OneSpace-AlignedSpaces, and SupLearn required approximately 12 hours of training each (using 10 epochs for training). OneSpace-WMD took around 7 minutes for training, since it only trains its embedding space (it does not use a Siamese Neural Network). CLCDSA took only 15 minutes, because its Neural Network uses a low number (18) of features.

Nevertheless, the focus of this study is the testing time of our studied techniques. In practice, developers only need to train their technique once (or once every few months, to refresh them), but they will have to run their testing step for each queried implementation pair. Next, we present the testing time of our studied techniques.

9.1. RQ5 Results

Figure 3 presents the timings that we collected for our studied techniques. Each box in the boxplot represents the distribution of durations that we measured for each batch for each technique. To compare the distribution of durations measured for OneSpace and for every other technique, we tested for statistical significance with a Wilcoxon one-tailed paired test (since the distributions of data were not normal, and the scores were paired), and found that all such differences were statistically significant ($p < 0.05$).

OneSpace, OneSpace-Code, OneSpace-AlignedSpaces, and SupLearn took about 60 seconds to produce the results of a single batch. While this execution time is reasonable, these techniques were

Table 4: (*RQ4*) Detecting single method clones using various approaches on AtCoder-b and AtCoder-r datasets.

Approach	AtCoder-b results			AtCoder-r results		
	Recall	Precision	Fmeasure	Recall	Precision	Fmeasure
OneSpace	48.85	48.96	48.91	52.08	40.74	45.72
OneSpace-Code	51.61	45.77	48.52	51.11	39.61	44.63
OneSpace-AlignedSpaces	43.51	22.36	29.54	47.47	20.07	28.21
OneSpace-WMD	31.11	26.27	28.49	25.06	29.32	27.03
SupLearn	43.73	22.29	29.53	50.23	17.50	25.95
CLCDSA	89.76	06.07	11.36	94.50	08.22	15.12

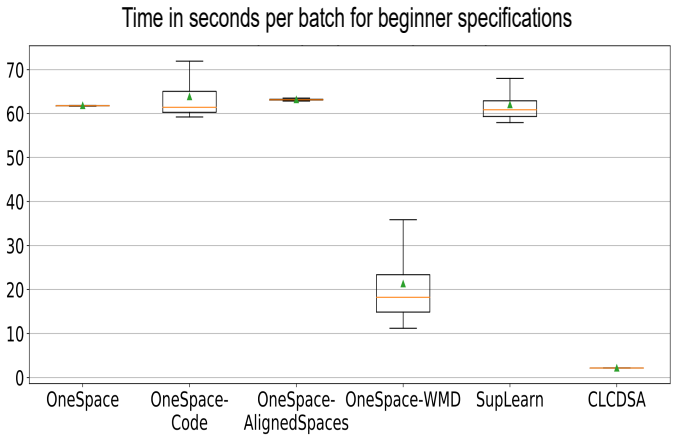

Figure 3: Box plot for the inference times per batch of all techniques .

slower than the other studied techniques. This is because they apply a Siamese Neural Network with recurrent LSTM layers that recursively process the program tokens.

OneSpace-WMD took a median of just under 20 seconds to produce the results. This execution time was faster than the majority of our studied techniques, with the exception of CLCDSA. The main reason why this technique was faster than most is that it does not use a recurrent Siamese Neural Network. It instead uses a similarity metric (Word Mover Distance), which is often fast to apply [71].

Finally, CLCDSA was the fastest technique, with an execution time of 2-3 seconds for most batches. CLCDSA uses a Siamese Neural Network, which should make its efficiency be within the range of OneSpace's and SupLearn's. However, it contains three characteristics that together make it highly efficient. First, its action filter quickly discards many of the candidate implementations, and thus avoids analyzing them in the Siamese Neural Network. Second, its Siamese Neural Network doesn't contain any recurrent layers. Third, its Siamese Neural Network uses a low number of features, making its prediction fast.

We also compared the results for *RQ5* with the results for *RQ1* and *RQ2* to observe trade-offs between effectiveness and efficiency. First, we observed that while OneSpace and OneSpace-code were among the least efficient techniques, they were by far the most effective ones (and it still provided acceptable efficiency). Next, we also observed that while CLCDSA was the most efficient technique, it provided very low effectiveness. Next, OneSpace-WMD provided a middle ground of effectiveness and efficiency. It was less effective than the best (OneSpace), but it was also faster. This variant may be a reasonable technique when its gain in efficiency (both to query and to train) may counterbalance its loss in effectiveness. Finally, OneSpace-AlignedSpaces and SupLearn provided an undesirable trade-off: they provided both low effectiveness and low efficiency.

10. *RQ6*: How does the clone ratio in the training data affect the effectiveness of the studied techniques?

In previous experiments, we trained our evaluated techniques with a ratio of 1:2 positive (clone) to negative (non-clone) pairs of implementations. This ratio in the training data influences how inclined a technique is to predict whether a new pair of implementations are clones or not. Therefore, it can impact its results.

In this research question, we study how different ratios of positive to negative clones impacts the effectiveness of our studied techniques.

10.1. Experiment Design

We used the same methodology as in RQ1 for AtCoder-b (see Section 5.2), and in RQ3 for AtCoder-r (see Section 7.2), but this time modifying the ratio of clones to non-clones in the training data.

We used the same testing data as in RQ1 for AtCoder-b (see Section 5.2.2), and in RQ3 (see Section 7.2.2) for AtCoder-r.

We trained techniques with the implementation pairs that we sampled in RQ1 for AtCoder-b (see Section 5.2.1), and in RQ3 (see Section 7.2.1) for AtCoder-r. We reused the same sampled positive (clone) and negative (non-clone) Java-Python implementation pairs that we sampled in those experiments. To produce higher than 1:2 positive to negative ratios, we randomly sampled additional negative implementation pairs. To produce the lower 1:1 positive to negative ratio, we randomly removed negative

Table 5: (*RQ6*) Impact of true and false clone tatio on detection accuracy for various techniques.

Approach	Clone Ratio	AtCoder-b results			AtCoder-r results		
		Recall	Precision	Fmeasure	Recall	Precision	Fmeasure
OneSpace	1:1	64.83	20.51	31.16	66.60	13.49	22.43
	1:2	45.07	37.63	41.02	50.11	20.17	28.76
	1:3	36.42	43.00	39.44	36.15	23.83	28.72
	1:4	29.97	53.96	38.54	29.66	26.12	27.78
SupLearn	1:1	67.93	09.45	16.60	53.41	08.31	14.38
	1:2	36.55	17.87	24.01	37.74	10.72	16.69
	1:3	36.85	10.08	15.83	50.37	09.60	16.12
	1:4	32.08	10.61	15.94	36.92	08.35	13.62
CLCDSA	1:1	96.47	05.60	10.58	81.59	06.02	11,21
	1:2	71.11	05.88	10.86	67.31	07.23	13.05
	1:3	61.66	06.24	11.33	44.70	06.51	11.36
	1:4	59.20	06.13	11.10	41.13	06.60	11.38

implementation pairs from the sample. For the 1:2 ratio, we directly report the results from RQ1 and RQ3.

10.2. Results

We report the results for this experiment in Table 5. Our studied techniques showed similar trends for AtCoder-b and AtCoder-r.

Our first observation in Table 5 is that, as in previous research questions, OneSpace provided a higher f-measure than the state-of-the-art SupLearn, CLCDSA (irrespective of the applied ratio of clones to non-clones in training). Then, we observed different trends in Recall and Precision for different techniques.

OneSpace showed an expected trend as the clone ratio increases: its Precision increased and its Recall decreased. As training contained more and more negative clones, OneSpace became more inclined to predict implementation pairs as non-clones, which made it achieve higher Precision and lower Recall.

SupLearn showed a different trend to OneSpace. When going from a 1:1 to 1:2 clone:non-clone ratio, Recall decreased and Precision increased, as expected. However, when going from 1:2 to 1:3 and 1:4, SupLearn Precision also decreased. It is possible that the larger amounts of non-clones in the training data influenced SupLearn so strongly that it ended up predicting so many non-clones that it lost both false positives (false clones) and true positives (true clones).

Finally, CLCDSA showed a very similar trend to SupLearn. As the number of non-clones increased in training, CLCDSA decreased its Recall, but very lightly increased its Precision. A possible explanation is that CLCDSA predicted more non-clones as the training data contained more non-clones, making it decrease Recall, but not increasing Precision as strongly, since it still kept its strong bias towards predicting clones. We can also observe that, as in previous research questions, that CLCDSA generally provided very low Precision.

11. *RQ7*: What is the impact of the amount of training data on technique effectiveness?

The effectiveness of machine learning techniques is influenced by the amount of data that is used to train them. Normally, adding training data increases the effectiveness of machine learning techniques, but it is not clear how much is enough to obtain accurate predictions.

In this research question, we study how different amounts of training data impact the effectiveness of our studied techniques.

11.1. Experiment Design

We used the same methodology as in RQ1 for AtCoder-b (see Section 5.2), but this time iteratively modifying the amount of data that we used for training. In this experiment, we only evaluated OneSpace.

For testing, we randomly selected one of the 10 folds that we used for testing in RQ1 (see Section 5.2.2). This fold had 242,079 implementation pairs, 11,704 of which were clones. The ratio of clones to non-clones is $\approx$ 1:20. We evaluated OneSpace 9 times on this same testing fold, changing the training data each time.

For training, we used the implementation pairs that we used in RQ1 for training when evaluating our randomly-selected testing fold (see Section 5.2.1). We randomly divided this training set into 9 equal subsets (*i.e.,* increments), and we used them incrementally for training. That is, we first trained using increment 1 and tested in our randomly selected fold, then trained using increments 1 and 2 and tested in our randomly selected fold, and so on. Each one of these training increments also kept the same positive to negative ratio as the original training set (1:2). Each increment had approximately 4,000 implementation pairs.

11.2. Results

We report the results for this research question in Table 6. The general trend that we can see in these results is

Table 6: (*RQ7*) Results of OneSpace on a random fold with different amount of training data. Each row represents the Recall, Precision and Fmeasure of the results obtained by training with n increments and testing using the implementations of the pre-selected random fold.

Training increments	#examples	Recall	Precision	Fmeasure
increment 1	≈ 4k	22.46	26.65	24.38
increments 1 and 2	≈ 8k	55.58	20.90	30.38
increments 1 to 3	≈ 12k	54.72	23.13	32.51
increments 1 to 4	≈ 16k	47.92	30.27	37.10
increments 1 to 5	≈ 20k	49.99	28.52	36.32
increments 1 to 6	≈ 24k	42.78	33.61	37.64
increments 1 to 7	≈ 28k	44.25	35.86	39.62
increments 1 to 8	≈ 32k	41.61	35.09	38.07
increments 1 to 9	≈ 36k	39.99	41.23	40.60

that, as the amount of training increased, Recall increased quickly and then decreased, and Precision and f-measure increased. This means that the gain in Precision was more substantial than the decrease in Recall. Also, there were some fluctuations results, particularly in terms of Recall for some increments, which also impacted the Fmeasure.

These results show that having more training data led to higher effectiveness for OneSpace in terms of Precision and Fmeasure. However, as a counterbalance, when increasing the training data, Recall also increases quickly, but then it also quickly peaks and starts slowly decreasing. While such decrease gets counterbalanced by a larger increase in benefit, this may not always be what the final user wants — they may prefer Recall to Precision. The results of this experiment reveal this trade-off.

12. *RQ8*: How effective is OneSpace for other programming language pairs?

In *RQ1*, we evaluated our cross-language clone detection approach using two programming languages: Java and Python. In this research question (*RQ8*), we study whether our findings generalize to other programming languages, by evaluating two additional programming languages: C and C++. Previous work in the field only analyzed up to three programming languages (Java, Python, and C#) [92, 95]. As in RQ1, we again compare the results of OneSpace with the state-of-the-art techniques SupLearn [95] and CLCDSA [92], using the Precision, Recall, and Fmeasure metrics.

12.1. Evaluation Dataset

For this experiment, we use a different dataset than in RQ1. We use the CodeChef competitive programming website dataset, available at the Kaggle website [59]. Instead of building our own dataset, we decided to use the independent, publicly-available CodeChef dataset (as when using AtCoder in RQ1). Also as in RQ1, we did not investigate the characteristics of the dataset, and thus it did not influence the design of our technique.

CodeChef contains program specifications with implementations for them in multiple languages. We only considered implementations that the dataset pre-labeled as "accepted", since it also contains implementations that were "rejected", *e.g.,* for not passing the test suite. We focused this evaluation in the language pairs that had the highest number of implementations available: Java-C++ and Java-C. We considered all the specifications and their implementations that our evaluated techniques could successfully parse: 210 specifications for Java and C++, and 150 specifications for Java and C. In total, this included 31,126 implementations in Java, 78,547 in C++, and 40,514 in C.

As ground truth, we considered as clones those implementations that fulfilled the same specification (as we did for the AtCoder dataset in previous research questions).

In sum, we used the following two data sets for RQ8:

1. **CodeChef-CPP:** This is a Java-C++ data set containing the 210 specifications of CodeChef that contain both Java and C++ implementations that parsed successfully. To be consistent with our previous experiments in RQ1, we randomly divided the 210 specifications into 7 folds of 30 specifications each. The total number of implementations in CodeChef-CPP is 94,237 (15,690 in Java and 78,547 in C++).

2. **CodeChef-C:** This is a Java-C data set containing the 150 specifications of CodeChef that contain both Java and C implementations that parsed successfully. We randomly divided them into 5 folds of 30 specifications each, to maintain the same number of specifications per fold as above and in RQ1. The total number of implementations in CodeChef-C is 55,950 (15,436 in Java and 40,514 in C).

12.2. Training and Testing Process

We used the same process described for *RQ1* in Section 5.2 to build our training and testing sets. We used cross-validation over the 7 randomly-divided folds of specifications of Java-C++ implementations. We also used cross-validation over the 5 randomly-divided folds of specifications of Java-C implementations. Each fold contained 30 specifications.

12.2.1. Training Process

For each dataset (CodeChef-CPP and CodeChef-C), we separately and randomly sampled positive and negative implementation pairs for training, following the method that we used for *RQ1* (see Section 5.2.1). As in *RQ1*, we sampled 40,000 implementation pairs for each dataset, covering all their specifications, with the same positive to negative (*i.e.,* clone to non-clone) ratio of 1:2. For CodeChef-CPP (Java-C++), we sampled 13,864 positive implementation pairs and 26,136 negative implementation pairs. For CodeChef-C (Java-C), we sampled 13,664 positive implementation pairs and 26,336 negative implementation pairs.

For each testing fold, we trained our studied techniques with our sampled positive and negative pairs for which both their implementations addressed specifications in the remaining folds. This resulted in a median of 32,463 training pairs per testing fold in CodeChef-CPP, and median 32027 training pairs per testing fold in CodeChef-C.

12.2.2. Testing Process

Also to stay consistent with the experiment design of *RQ1* (see Section 5.2.2), we randomly sampled 1,000 implementations from each testing fold, covering the 30 specifications of that fold.

After sampling, for CodeChef-CPP (Java-C++), the median number of Java and C++ implementations per fold was 475 and 525 respectively, resulting in a median of 249,375 Java-C++ implementation pairs evaluated per fold. The closer Java implementations to 500, the larger the number of pairs. So, we have the same order for Java implementations as the test pairs. Similarly, after sampling, for CodeChef-C (Java-C), the median number of Java and C implementations per fold was 451 and 549 respectively, resulting in a median of 247,599 Java-C implementation pairs evaluated per fold.

Finally, also as with *RQ1*, we evaluated our studied techniques over each possible implementation pair (Java-C++ for CodeChef-CPP, or Java-C for CodeChef-C) from these sampled implementations. In total for all folds for CodeChef-CPP, we evaluated 1,734,028 Java-C++ implementation pairs, belonging to 210 specifications, 146,035 of which were clones ($\approx$ 1:11 ratio). Similarly, for CodeChef-C, we evaluated a total of 1,237,941 Java-C implementation pairs, belonging to 150 specifications, 106,408 of which were clones ($\approx$ 1:11 ratio).

12.3. Studied Techniques

We adapted the implementation of our studied techniques (OneSpace, SupLearn [95], and CLCDSA [92]) to support the C and C++ languages.

SupLearn [95], and CLCDSA [92] require parsing implementations to obtain their abstract syntax tree (AST). The original version of SupLearn and CLCDSA can already parse Java and Python. We adapted them to generate ASTs for C and C++ code using the "pycparser" Python library. [3] For CLCDSA, we also adapted it to extract Cyclomatic Complexity from C and C++ using the "lizard" Python library. [4]. For OneSpace, we also downloaded Microsoft's documentation for C++ [87] and C [88] from Microsoft website since OneSpace is trained with both code and API documentation data, but the remaining implementation details remained unchanged.

[3] `https://pypi.org/project/pycparser/`.

[4] `https://pypi.org/project/lizard/`.

12.4. Evaluation Metrics

As in *RQ1*, to calculate the Java-CPP results, for each of the studied techniques, we constructed a confusion matrix with the prediction outcome of all the tested implementation pairs for all the Java-CPP folds. We then applied Recall, Precision, and Fmeasure on these matrices for each studied technique. We repeated the same process to calculate the Java-C results.

12.5. Results

We report the results of both experiments in Table 7, which also includes our previously obtained results for Java-Python in *RQ1*.

In summary, our evaluation of OneSpace on Java-C++ and Java-C shows that OneSpace was effective in detecting clones beyond the Java-Python language pair. OneSpace provided a higher f-measure than both state-of-the-art techniques (SupLearn and CLCDSA), for all our studied programming language pairs.

Next, we discuss how OneSpace provided different results for different programming language pairs.

In terms of Recall, OneSpace provided high Recall (71–76%) for Java-C and Java-C++, and it provided limited Recall (45–50%) for Java-Python. A possible reason for this is that C and C++ provided more similar tokens to Java than Python did, *i.e.,* the analyzed Java implementations may have used more similar tokens to C and C++, and may have use them surrounded by other tokens also in more similar ways. Python has more tokens that can serve to code the same specification (*e.g.,* in more or less "Pythonic" ways). That may have made it harder for OneSpace to identify clones that used different tokens than Java, surrounded by other tokens that are also different.

For Precision, OneSpace provided higher Precision for Java-Python in AtCoder-b (37%) than it did for the remaining programming language pairs (17–21%). It's possible that, for simpler specifications (as in AtCoder-b), if OneSpace identified that two implementations used similar tokens surrounded by similar other ones, they may have been more likely to be in fact clones (*i.e.,* there may be less room to implement non-clones that use similar tokens in similar contexts). However, for more complex specifications (as in AtCoder-r), and when the token vocabulary of a programming language is less diverse (as in C and C++), if OneSpace identified that two implementations used similar tokens surrounded by similar other ones, it does not necessarily mean that they are clones (*i.e.,* it may have been easier to write non-clones using similar tokens in similar contexts, because there were not many tokens to choose from, to begin with).

For f-measure, OneSpace provided different scores in different cases, which we believe can be explained by the same factors. OneSpace provided its highest f-measure for Java-Python in AtCoder-b: 41%. It may have been hard for it to flag clones (since it may not have caught

Table 7: (*RQ8*) Effectiveness of the studied techniques for various language pairs in terms of Recall, Precision and Fmeasure metrics. The Java-Python columns are copied from tables Table 1 and Table 8 for reference.

Approach	Java-C++ CodeChef-CPP			Java-C CodeChef-C			Java-Python AtCoder-b			Java-Python AtCoder-r		
	Rec	Pre	fmeasure	Rec	Pre	fmeasure	Rec	Pre	fmeasure	Rec	Pre	fmeasure
OneSpace	71.75	21.37	32.94	76.08	17.18	28.04	45.07	37.63	41.02	50.11	20.17	28.76
SupLearn	42.24	13.38	20.33	57.08	11.93	19.73	36.55	17.87	24.01	37.74	10.72	16.69
CLCDSA	45.05	14.99	22.49	89.03	11.80	20.84	71.11	05.88	10.86	67.31	07.23	13.05

clones implemented in Python with more diverse tokens), but when it did flag clones, it was often right (since specifications may have been simpler, and thus would have had less room for diverse implementations).

The next highest f-measure went to Java-C++: 33%. It may have been relatively easy for OneSpace to flag clones (since they would be implemented with similar tokens in similar contexts), but it may also have often wrongly flagged implementations as clones (developers may have implemented more different things with similar tokens, since there were fewer tokens available).

Finally, both Java-Python in AtCoder-r and Java-C obtained the lowest f-measure: 28%. Java-Python in AtCoder-r may have obtained lower Precision than in AtCoder-b because more complex specifications now had more room for implementing non-clones with similar tokens in similar contexts. Java-C may have obtained even lower Precision than Java-C++ because C has even less diverse tokens that C++.

13. Studying cross-language clone-detection as a recommendation task.

In this section, we study cross-language clone-detection as a recommendation task rather than a classification task, following the advice of past work [100, 121]. Studying cross-language clone-detection as a recommendation task has two major advantages.

First, it gives us more information about the effectiveness of each technique, since we can measure both the *correctness* and *rank* of their answers. The new *rank* information tells us how *confident* each technique is in its answers.

Second, we believe that modeling clone-detection answers as recommendations provides a more faithful representation of the experience of practitioners using these techniques, *i.e.,* they would assess a ranked list of recommended clones.

A common scenario in which cross-language clone ranking is useful is when a software project needs to be migrated to a different programming language to take advantage of its better features. For example, Microsoft migrating to Rust to take advantage of Rust's secure memory management capabilities [48, 74]. In such cases, developers could benefit from finding and reusing clones for their implemented code in the target language. In such a case, it would be better to recommend the candidates with the highest probability of being clones ahead of the retrieved list.

13.1. Evaluation method.

We envision users of these techniques in the following context: given an implementation in language A as a query, they would want to find all its clones within a given collection of implementations in other languages. A cross-language clone-detection technique would produce an ordered ranking of all the implementations in the other languages, in decreasing order of likelihood of them being clones of the given implementation in language A. In our evaluation, we assign a single effectiveness score for each recommendation — *i.e.,* each ranked list — obtained for a queried implementation. We assign higher scores to recommendations that rank the correct answers higher — by applying the learning-to-rank NDCG@k metric [49].

We repeat the evaluation of *RQ1*, but this time using the following metrics in this analysis: NDCG@k, Precision@k, and Recall@k. We measured these metrics for the recommendation that each technique produced for each of the queried Java implementations of our experiments .

NDCG@k [49] is a popular metric for scoring ranked recommendations. It accounts for both the correctness and rank of each answer in the recommendation.

In addition to NDCG@k, we also measured Precision@k and Recall@k for the techniques' recommendations. For all metrics in this experiment, k represents the top k positions that are considered inside a recommendation. We studied many values of k for all metrics: 1,5,10,20,40,60,80,100. These metrics produce scores in the range 0–100 and higher scores are better.

NDCG@k: We used Burges *et al.*'s [20] formula. NDCG is calculated as the Discounted Cumulative Gain (DCG) score for a recommendation, divided by the DCG score of an ideal recommendation — a recommendation in which the correct answers are contained in its top positions. In the DCG formula (Equation (5)), i represents the top i^{th} position inside a recommendation, and rel_i represents the actual, ground-truth relevance of the item recommended in position i. We assigned relevance 1 to recommended implementations that were clones and relevance 0 to those that were not.

$$\text{NDCG@}k = \frac{\text{DCG@}k\text{(recommendation)}}{\text{DCG@}k\text{(ideal recommendation)}} \cdot 100 \qquad (4)$$

$$\text{DCG@}k = \sum_{i=1}^{k} \frac{2^{\text{rel}_i} - 1}{log_2(i+1)} \cdot 100 \quad (5)$$

Precision@k: measures, within the top k items in a recommendation, the proportion of them that are true clones.

$$\text{Precision@}k = \frac{|\{\text{relevant items}\} \cap \{\text{retrieved items}\}@k|}{|\{\text{retrieved items}\}@k|} \cdot 100 \quad (6)$$

Recall@k: measures, within the top k items in a recommendation, the proportion of all true clones that are in the recommendation.

$$\text{Recall@}k = \frac{|\{\text{relevant items}\} \cap \{\text{retrieved items}\}@k|}{|\{\text{relevant items}\}|} \cdot 100 \quad (7)$$

13.2. Results in ranking metrics

We report *RQ1* results using ranking metrics in Table 8 and Figure 4. 'Table 8 shows the median values obtained by our studied techniques for all queries (including all folds), for all metrics and values of k. Figure 4 highlights the distribution of scores obtained by all techniques for k=10 (*i.e.,* for NDCG@10, Precision@10, and Recall@10). To compare the distribution of scores obtained by OneSpace and every other technique, we tested for statistical significance with a Wilcoxon one-tailed paired test (since the distributions of data were not normal, and the scores were paired).

Our results show that OneSpace provided much higher effectiveness than the state-of-the-art techniques in ranking context, for all our studied metrics and thresholds, with statistical significance ($p < 0.05$) for all of them.

Regarding NDCG@k scores, we highlight that OneSpace obtained a median NDCG@1 value of 100. This means that, in the median case, it recommended a clone in its top 1 result. As we consider more results in its recommendation, OneSpace lowers its NDCG@k score, since it also recommends some implementations that are not clones. Still, OneSpace's NDCG scores were very high, even for long recommendation lists. For example, its median NDCG@10 is 70.15, *e.g.,* it recommended clones in positions 2,3,4,5,7,8 and 9 within its top 10 recommendations, in the median case.

For long recommendations, *e.g.,* NDCG@100, OneSpace still provided highly effective results: it recommended clones in positions [3, 6, 7, 8, 9, 10, 12, 13, 14, 15, 17, 19, 20, 21, 22, 24, 29, 32, 37, 47, 51, 60, 76, 86, 97] within its top 100 recommendations, in the median case (Figure 5, bottom row).

In contrast, the state of the art techniques generally recommended much fewer clones, and in much lower positions in their recommendation lists. SupLearn obtained median 0 for NDCG@1, 9.1 for NDCG@10, and 38.17 for NDCG@100. This means that even when we consider a large number of recommended implementations (NDCG@100), SupLearn only recommended clones in positions [11, 22, 25, 26, 36, 38, 41, 45, 50, 59, 66, 75, 85, 95 , 98] in the median case (Figure 5, middle row). CLCDSA obtained even lower median NDCG@k scores: 0 for NDCG@1, 0 for NDCG@10, and 14.88 for NDCG@100. This means that even when we considered a large number of recommended implementations (NDCG@100), CLCDSA only recommended clones in positions 11, 30 , 56, 81 and 82 in the median case (Figure 5, top row).

For Precision@k, we observed a similar effect to NDCG@k. OneSpace obtained median 100% Precision@1 because it recommended a clone in its top 1 result in the median case. Then, as k increases, Precision@k decreases, due to the recommendation containing some non-clones. However, the drop in Precision@k is more acute than the drop in NDCG@k. This is because Precision@k does not account for the position of clones within the recommendation list —NDCG@k assigns higher scores to higher positions. These two observations together mean that OneSpace tends to recommend clones around the top positions of its recommendation —in the median case, it recommended 22 clones out of 100 recommendations.

The other studied techniques obtained much lower Precision@k values. For example, in the median case, SupLearn recommended only 1 clone within its top 10 recommendations (Precision@10) and 11 clones in its top 100 recommendations (Precision@100). In the median case of CLCDSA, it performed worse than SupLearn. It recommended 0 clones within its top 10 recommendations (Precision@10) and 6 clones in its top 100 recommendations (Precision@100). In contrast, OneSpace performed much better. It recommended 6 clones in its top 10 recommendations (Precision@10) and 22 clones in its top 100 recommendations (Precision@100). Summing up, SupLearn and CLCDSA recommended much fewer clones and in much lower positions in their recommendation than OneSpace.

Regarding Recall@k scores, OneSpace also obtained high scores: 2% for Recall@1, 18% for Recall@10, and 87.5% for Recall@100. These are high scores because they are close to the highest achievable Recall for each threshold. For our test data, the median number of true clones per query in the ground truth is 32. Thus, in the median case, the highest achievable Recall for thresholds 1, 10, and 100 were 3%, 31%, and 100%, respectively.

The other studied techniques obtained much lower values of Recall@k. For example, in the median case, for Recall@10, SupLearn achieved 4% and CLCDSA 0% (while OneSpace had 18% Recall@10). OneSpace outperformed SupLearn and CLCDSA also for larger recommendations, *e.g.,* for Recall@100, SupLearn achieved 68% and CLCDSA 23% (while OneSpace had 87 % Recall@10).

Our observation for all metrics show that, generally, OneSpace recommended more clones and in higher positions than the state of the art techniques. This is also

Table 8: Median Effectiveness. OneSpace provided higher effectiveness than the state-of-the-art for all ranking metrics, with statistical significance.

	OneSpace			SupLearn			CLCDSA		
k	Precision@k	Recall@k	NDCG@k	Precision@k	Recall@k	NDCG@k	Precision@k	Recall@k	NDCG@k
1	100	02.00	100	00.00	00.00	00.00	00.00	00.00	00.00
5	60.00	09.43	72.27	00.00	00.00	00.00	00.00	00.00	00.00
10	60.00	18.18	70.15	10.00	04.35	09.10	00.00	00.00	00.00
20	50.00	34.04	65.89	10.00	11.43	14.11	00.00	00.00	00.00
40	35.00	59.57	64.78	10.00	27.27	20.54	05.00	04.76	04.10
60	30.00	73.58	67.80	11.67	43.18	27.96	03.33	09.30	07.65
80	25.00	81.97	71.80	11.25	56.67	34.07	05.00	15.38	10.95
100	22.00	87.50	74.70	11.00	68.18	38.17	06.00	23.08	14.88

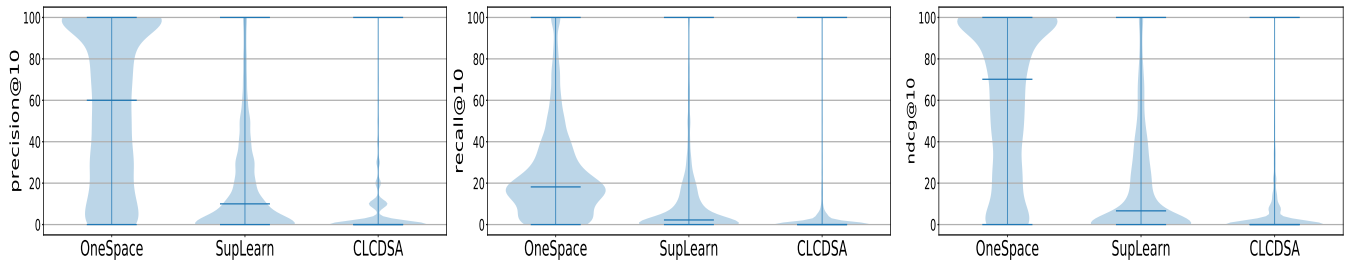

Figure 4: OneSpace provided higher scores than the state-of-the-art for all metrics.

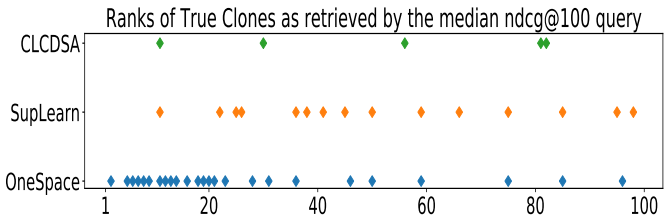

Figure 5: The positions of the true clones, as recommended for the median query, in terms of NDCG@100, for the three studied approaches, OneSpace, SupLearn, and CLCDSA. Lower values are better. We notice that OneSpace recommended true clones in earlier positions of the ranked list, whereas SupLearn and CLCDSA recommended lower positions for true clones.

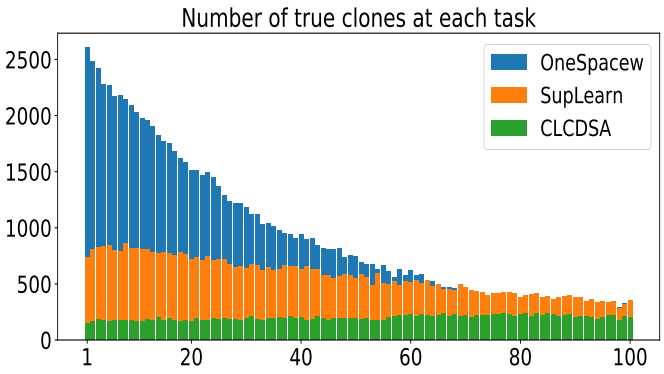

Figure 6: The number of times a true clone appeared at rank k for k in [0-100] for the three studied approaches: OneSpace, SupLearn, and CLCDSA. Lower ranks are better. The figure shows that OneSpace assigned lower ranks in its recommendation list than SupLearn and CLCDSA. That means it is better for ranking candidate clones than the state of the art techniques.

reflected in Figure 4: the most common NDCG@10 and Precision@10 scores obtained by OneSpace's recommendations were among the highest of the scale, while SupLearn's and CLCDSA's concentrated among the lowest values.

This conclusion can also be observed in Figure 6 , which plots the number of times that each technique recommends a true clone at position k where k is in range [0-100]. We can see in the figure that OneSpace recommend true clones earlier in its recommendation list, when compared to the other two state of the art techniques.

14. Discussion

14.1. When did OneSpace Succeed and When did It Not?

We found in *RQ1* (Section 5.5) that OneSpace improved over the state of the art in terms of the number of detected true clones that it recommends and in terms of their ranking (Section 13). We also found in *RQ2* (Section 6.2) that OneSpace achieved its highest effectiveness when it used a single shared embedding space for the two languages and a Siamese Neural Network. This means that OneSpace recommends in high rankings clone pairs that have one or two of these properties: (a) the clone pair has semantically related tokens that are placed close to each other in the shared space, and/or (b) it contains such similar tokens in a similar order.

As an example, Figure 7 shows two clone fragments in Java and Python. The implemented functionality checks

```
import java.util.Scanner;
public Class Main{
    public static void main(string[] args){
        Scanner scanner = new Scanner(System.in);
        String s = scanner.next();
        if(s.length()>=0 && s.matches("(ch|o|k|u)*")){
            system.out.println("YES");
        } else{
            system.out.println("NO")
        }
    }
}
```

```
A = input()
B=A
B=B.replace("ch","")
B=B.replace("o","")
B=B.replace("k","")
B=B.replace("u","")
if len(B)!=0:
    print('NO')
elif A[-2:-1]=='ch':
    print('YES')
elif A[-1]=='o':
    print('YES')
elif A[-1]=='k':
    print('YES')
elif A[-1]=='u':
    print('YES')
else: print('NO')
```

Figure 7: A true positive example for OneSpace

```
A, B, C = map(int, input().split())
if (A==5 and B==5 and C==7) or (A==5 and B==7 and C==5) or (A==7 and B==5 and
C==5):
    print("YES")
else:
    print("NO")
```

Figure 8: A CLCDSA's false positive example for the code in Figure 7

whether an input string is composed of certain substrings. For the Java query on the left part of the figure, OneSpace recommended 10 true clones on the top 10 positions of its recommendation list. Figure 7 shows the true Python clone on the top of the recommendation list. These the two implementations have the two properties we mentioned before. First, the two programs share many semantically related tokens such as input/output methods, string processing tokens, YES/NO literal strings, and constant character variables. Second, the order of the corresponding tokens in the two implementations is similar, even if not perfect. We believe that these two properties contributed for OneSpace to assign a higher probability of being clones to the clone pair in Figure 7.

In contrast, SupLearn recommended only 3 true clones within the top 10 positions of its recommendation for the same Java query (in positions 1, 7 and 9), which did not include the Python implementation in figure 7-right. SupLearn uses the AST of the considered implementations as features for its predictions, and the AST of the two implementations in Figure 7 are not very similar. This may have impacted SupLearn for recommending it in a lower position in its recommendation, *i.e.,* not within its top 10 positions.

CLCDSA failed to retrieve any correct clones in the top 10. CLCDSA instead recommended the code in Figure 8 on the top of its recommendation list, which solves a totally different problem with different input variable types. CLCDSA uses statistical features for its prediction, and the code in Figure 8 is more similar to the queried one (left side of Figure 7) in terms of statistical features like length and Cyclomatic Complexity than the true clone in the right side of Figure 7 is.

We also investigated when OneSpace did not succeed to detect true clones at the top 10 positions in its recommendation list. As an example, consider the false negative clone pair in Figure 9. This represents Java and Python implementations for finding the transitive closure of a set, and printing the number of edges at each vertex. The Java and Python implementations tackle the problem using different data structures. While the Java implementation uses a 2D array as an adjacency matrix, the Python implementation uses a dictionary as an adjacency list. Although the two programs have commonalities in terms of tokens and their order, OneSpace ranked multiple other (non-clone) implementations higher than this true clone. Most of those higher-ranked implementations had nested loops that use single character identifiers such as [i,n,m,j,k] with conditional statements inside the nested loop, which may have made OneSpace rank them higher. We plan to further explore this in the future to try to reduce the number of highly-ranked implementations that are not clones in OneSpace's recommendations, *e.g.,* by adding features to it based on dynamic analysis.

14.2. Shared Embedding Space vs. Separate Embedding Spaces

We observed in our results that the component of OneSpace that had the most impact on its effectiveness was the usage of a shared embedding space in its algorithm. The variant of OneSpace that provided the lowest effectiveness when compared to it was the one that used two separate embedding spaces. Similarly, the existing technique SupLearn, which also uses two separate embedding spaces, also provided low effectiveness when compared to OneSpace.

Our original intuition was that using a shared embedding space would provide better alignment of semantically related words in different programming languages than if we trained separate embedding spaces for each programming language and later aligned them. Such close positioning of semantically related concepts would aid the Neural Network to identify clones in different programming languages.

To better investigate our intuition, we performed an additional deep study of the embedding spaces that would be produced for one of the clones that OneSpace ranked highly, but OneSpace-AlignedSpaces and SupLearn ranked in lower positions. We again studied the clone pair in Figure 7 (as we did in Section 14.1). We wanted to know if training the same embedding space with this specific clone pair would position semantically related words closer in the space than if we trained two separate spaces and aligned them later.

```java
public static void main(String[] args) {
       Scanner sc = new Scanner(System.in);
       int N = sc.nextInt();
       int M = sc.nextInt();
       int friends[][] = new int[N][N];
       for(int i = 0;i < N;i++){
          for(int j = 0;j < N;j++){
              if(i == j)
                continue;
              friends[i][j] = 10000;
          }
       }
       for(int i = 0;i < M;i++){
           int a = sc.nextInt() - 1;
           int b = sc.nextInt() - 1;
           friends[a][b] = 1;
           friends[b][a] = 1;
       }
       for(int k = 1;k <= N;k++){
           for(int i = 0;i < N;i++){
              for(int j = 0;j < N;j++){
                  friends[i][j] = Math.min(friends[i][j],friends[i][k-1] + friends[k-1][j]);

              }
           }
       }
       for(int i = 0;i < N;i++){
           int ans = 0;
           for(int j = 0;j < N;j++){
               if(i == j)
                   continue;
               if(friends[i][j] == 2)
                   ans++;
           }
           System.out.println(ans);
       }
   }
```

```python
import collections

n, m = map(int, input().split())

r = collections.defaultdict(list)

for _ in range(m):
  a, b = map(int, input().split())
  r[a].append(b)
  r[b].append(a)

for i in range(1, n + 1):
  s = set()
  for f in r[i]:
    for ff in r[f]:
      if ff not in r[i] and ff != i:
        s.add(ff)
  print(len(s))
```

Figure 9: A false negative example for OneSpace

```python
A, B, C = map(int, input().split())
if (A==5 and B==5 and C==7) or (A==5 and B==7 and C==5) or (A==7 and B==5 and C==5):
    print("YES")
else:
    print("NO")
```

Figure 10: A CLCDSA's false positive example for the code in Figure 7

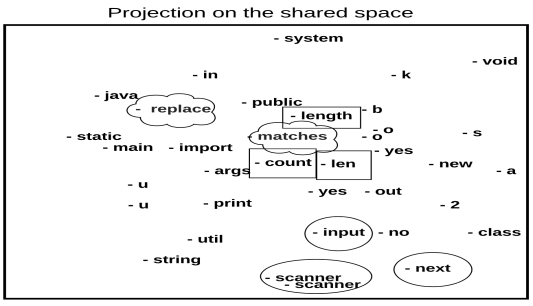

Figure 11: t-SNE visualization of the projection of the true positive example in Figure 7 on OneSpace's shared embedding

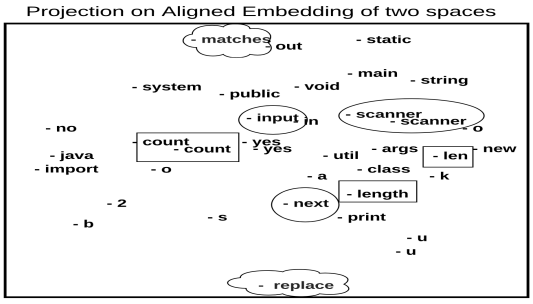

Figure 12: t-SNE visualization of the projection of the true positive example in Figure 7 on an aligned embedding space

We projected the tokens of the two programs to a shared embedding space and visualized the projection on a 2D space using the t-SNE algorithm [79] in Figure 11. In this figure, we see that related concepts (like the string processing methods "replace" and "matches") are very close to each other. Similarly, words like "length", "len" and "count" were also together and also close to the string methods cluster, since they also can be executed over strings. We manually marked these exemplified clusters on the t-SNE plot with different shapes, for illustration purposes.

Then, we also trained two separate spaces, for the Java and Python implementations, and aligned them using a widely recognized embedding alignment tool (used in *e.g.,*

```
public static void solve(int x, int[] a){
    String set = BigInteger.valueOf(x).toString(2);
    int sum = 0;
    for(int i=0;i<set.length();i++) {
        if(set.charAt(set.length()-i-1)=='1')
            sum+=a[i];
    }
    System.out.println(sum);
}
```

```
N, X = tuple(map(int, input().split()))
a = list(map(int, input().split()))
X = list(bin(X))
X.reverse()
del X[-2:]
while len(X) < N:
    X.append('0')
S = 0
for i in range(N):
    if X[i] == '1': S += a[i]
print(S)
```

Figure 13: A true positive example for OneSpace that OneSpace-WMD failed

```
def solve():
    li = [int(i) for i in input().split()]
    print("YES" if sum(li) == 17 and len(set(li)) == 2 else "NO")
```

Figure 14: A OneSpace-WMD's false positive example for the code in Figure 13-left

[14, 102, 114, 125]): Crosslingual-CCA [37]. We show the resulting T-SNE plot of the alignment in Figure 12. In it, we see that the string processing functions are now scattered, and the "count" and "len" functions are separated into two clusters. These observations support our intuition that training a single space achieves closer positioning of semantically related concepts in the space. While the alignment algorithm that we used in this investigation is not the same one used by OneSpace-AlignedSpaces or SupLearn, our experiment shows that aligning separate spaces is a challenging problem (which likely impacted the ranking that those two algorithms assigned to our studied clone pair).

14.3. Siamese Neural Network vs. Word Mover Distance?

To better understand why OneSpace recommended clones at higher positions than the OneSpace-WMD variant, we manually inspected multiple cases for which OneSpace ranked clones at higher positions than OneSpace-WMD did. We present one example next.

The Java listing in figure 13 (left) receives an array as input and produces the sum of elements whose indices correspond to 1 in a binary represented integer. OneSpace-WMD did not recommend any clones within its top-30 recommendations for this queried Java implementation. It instead recommended many implementations that were not clones, but shared many similar tokens (although not necessarily in a similar order). For example, OneSpace-WMD retrieved on top 5 the implementation at figure 14, which has multiple tokens that correspond to tokens in the with the query [Set, Len, Sum, solve, int, i].

In contrast, OneSpace recommended a true clone at the top rank of its recommendation (Figure 13 (right)). OneSpace was able to identify both the similarity of semantically similar tokens and the order of their usage, and thus assigned a high rank to this implementation.

15. Threats to Validity

15.1. Construct Validity

A threat to construct validity concerns whether we are basing our evaluation on appropriate data. To mitigate this threat, we used the AtCoder dataset, which is the same one that was used to evaluate SupLearn. AtCoder contains a large and diverse set of implementations for a diverse set of specifications, making our evaluation included diverse clone and non-clone pairs.

15.2. External Validity

A threat to external validity is whether the results of this study will generalize to other datasets. To mitigate this threat, we evaluated techniques separately for two kinds of problem specifications: beginner and regular (*RQ3*), and for a total of 4 programming languages: Java, Python, C, and C++ (*RQ8*). This provides evidence about how the accuracy of our proposed technique and existing techniques may vary as they face more complex sets of clones and non-clones, and different programming languages. Additionally, we are making all our code, data, fold splits information, and full experiment output available for researchers to replicate our experiments, re-execute under different experimental setup, compare against or apply to different data sets.

An additional threat to validity relates to whether OneSpace could detect clones across programming languages that are highly dissimilar, *e.g.,* each clone using a different paradigm, such as procedural (*e.g.,* Modula-2) or functional (*e.g.,* Scheme) vs. object oriented (*e.g.,* Java). Our evaluation addresses this question to some extent. We observed that OneSpace provided high Recall for Java and C and C++. This may be because the two programming languages involved tokens that are more

similar and that are more often used in similar token contexts. In contrast, OneSpace obtained lower Recall for Java-Python. A possible explanation is that it may be harder for it to flag clones that use more "Pythonic" implementations with more diverse tokens in more diverse contexts. This may also explain the opposite trend that we observed in terms of Precision. However, despite these differences in results for different programming languages, OneSpace still achieved higher results than the state of the art techniques for all of them.

Future research could focus on broadening the set of programming languages and programming paradigms in which cross-language clone detection techniques are evaluated. Such direction would require non-trivial effort of building new datasets of code clones across a wide variety of languages and paradigms (via repository mining or manual translation). To be able to better compare our new technique (OneSpace) with the state of the art techniques, we focused on evaluating them using the programming languages with which they were originally evaluated.

Another threat to external validity involves whether performing our evaluation over implementations taken from competitive programming websites is representative of the usage in practice of cross-language clone detection. The extent to which our findings generalize to other polyglot industrial software projects will depend on the characteristics of each project, *i.e.,* the problem will be differently easy or hard for different projects. For example, if a development team followed strict coding styles, it may be easier for cross-language clone detection techniques to identify clones in that software project. Conversely, it may be harder for techniques to detect clones in software projects that contain many similar-looking methods, *e.g.,* with many similar machine-learning pipelines applied to different goals.

To be able to evaluate our studied cross-language clone detection techniques under diverse scenarios, competitive programming websites provide a unique opportunity, *i.e.,* they allow us to gather a large number of diverse cross-language clones. Each specification was implemented independently by different programmers, coding in different programming languages and styles. We believe that such a dataset provides a good representation of the problem space, and therefore captures how cross-language techniques would perform under diverse situations. These code submissions provide a diverse set of algorithms, coding styles, data structures, and programming constructs. Furthermore, both of our studied state-of-the-art techniques were originally evaluated with datasets from competitive programming websites [92, 95].

Therefore, our evaluated datasets allow us to compare the effectiveness of our proposed technique OneSpace with the state-of-the-art techniques for a diverse set of scenarios.

16. Related Work

16.1. Cross-language Clone Detection

Some past approaches have been proposed to detect clones across programming languages.

Some of them apply static analysis techniques. For example, the techniques by Kraft *et al.* [69] and Al-Omari *et al.* [2] identify clones by looking for similarities in the Common Intermediate Language to which multiple programming languages get compiled within Microsoft's .NET framework. Later, Vislavski *et al.* [120] created LICCA, which uses the SSQSA framework to represent implementations as enriched Concrete Syntax Trees (eCST) [96]. It then identifies clones using a variant of the longest common subsequence algorithm (LCS) to find a common sub-tree between implementations. In contrast with these techniques, our proposed approach (OneSpace) does not require clones to be implemented for the .NET framework, or to have a common sub-tree in their syntax tree.

Other past approaches apply dynamic analysis. For example, Fang-Hsiang Su *et al.* [115] detect clones for JVM-based languages such as Java and Scala by executing methods and observing similarities in the outputs of the same inputs. Then, Mathew *et al.* [80] proposed a similar approach to detect clones between Java and Python, also based on comparing whether the same inputs produce the same outputs in implementations in different languages. Later, Mathew *et al.* [81] improved the effectiveness of their algorithm by complementing it with static analysis (measuring tree edit distance and Jaccard token similarity). In contrast to these algorithms, our proposed approach (OneSpace) provides multiple benefits. First, it does not require programs to be able to execute within a time window. Dynamic analysis approaches impose an execution time limit to be able to obtain and compare multiple outputs for each program within a reasonable amount of time. Second, OneSpace does not require any modification of the source code under analysis (*e.g.,* to make it not interactive), pre-processing its dependencies, or configuring its build environment. All these are requirements in dynamic analysis approaches, to be able to run the code under analysis.

Other approaches analyze the evolution of the source code under analysis. Cheng *et al.* [23][24] proposed CLCMiner, which assumes that clones spanning different languages evolve similarly on a software repository. CLCMiner mines the project's source code repository, looking for similar diffs in the candidate implementations. In contrast to these approaches, our proposed approach (OneSpace) does not require clones to have evolved in similar ways.

Finally, other approaches apply machine learning techniques over some representation of the source code under analysis. These are SupLearn [95] and CLCDSA [92]. We described them in detail in Sections 5.3.2 and 5.3.3, and we evaluated them in our experiments. The techniques that apply machine learning do not have any of the

limitations that are imposed by other approaches based on static analysis, dynamic analysis, or evolution analysis. Techniques that apply machine learning do not limit the programming languages to which they can be applied, the structure of the code under analysis, its execution time, or the way in which it evolved. Our experiments in this paper found that our proposed approach (OneSpace) provides much higher effectiveness than the other approaches based on machine learning.

16.2. Single-language Clone Detection

Multiple approaches have been proposed in the past to automatically detect implementation clones within a single programming language. Roy and Cordy [100] classified clone detection approaches by the way they represent the source code. Early approaches [58] leverage the *raw code representation* of the code and apply string matching algorithms to determine similarity. More recently, CodeXGlue [78] also uses a raw code representation and applies machine learning to obtain better results.

CCFinder [61] uses a different representation: it transforms the code text into a coarser-grained *token representation* and compares it to detect clones. SourcererCC [106] [77] also compares a coarser-grained token representation, and it uses optimized indexing to achieve high scalability.

Graph approaches compare static program representations, such as the AST and PDG. The most popular tool in this category is Deckard [50] which transforms the AST into a feature vector that it uses to measure implementation similarity. Other graph-representation techniques [22, 39] detect isomorphic graphs inside other static program representations, like the PDG or CFG. White *et al.* [124] compares graph representations using embedding and Recursive Neural Networks.

Finally, a recent family of clone detection approaches apply dynamic analysis. The basic intuition behind these approaches is that methods that provide the same output to the same inputs should be identified as clones [65, 115].

In contrast to all these approaches, our proposed approach (OneSpace) was designed and evaluated to detect implementation clones across different programming languages.

17. Conclusion and Future Work

In this paper, we presented OneSpace, a cross-language clone detection approach that provides much higher effectiveness than the state of art cross-language clone detection techniques. We analyzed the impact of the different components of OneSpace. We found that the component that contributed the most to its improved effectiveness was training a single embedding space with all the analyzed implementations in different languages. In the future, we plan to apply our design of having a common embedding space to other multi-programming-language software engineering tasks.

Also, we plan to complement the design of OneSpace with additional features that capture more characteristics of clones. For example, we will explore additional prediction features related to, *e.g.,* code-change history [108, 109, 107, 111, 112, 5], testing activity, *e.g.,* [64, 40, 63], decision-making metadata, *e.g.,* [86, 4, 3], developer expertise, *e.g.,* [110, 26], failure prediction *e.g.,* [52, 54, 53, 51, 55, 56], vulnerability prediction *e.g.,* [32, 43] or programming language similarities and differences, *e.g.,* [33].

18. Replication

We include a replication package for our paper [7].

Acknowledgements

We thank all reviewers for their valuable feedback. This work was partially funded by the United States National Science Foundation (NSF) under award CCF-2046403, by Virginia Tech under a startup grant, and by Universidad Rey Juan Carlos under the International Distinguished Researcher award C01INVESDIST. This work also served as foundation for award PID2022-142964OA-I00 by the Spanish Agencia Estatal de Investigación.

References

[1] Raihan Al-Ekram, Cory Kapser, Richard Holt, and Michael Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *2005 International Symposium on Empirical Software Engineering, 2005*, pages 10–pp. IEEE, 2005.

[2] Farouq Al-Omari, Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. Detecting clones across microsoft. net programming languages. In *the 19th Working Conference on Reverse Engineering*, pages 05–414. IEEE, 2012.

[3] Khadijah Al Safwan, Mohammed Elarnaoty, and Francisco Servant. Developers' need for the rationale of code commits: An in-breadth and in-depth study. *Journal of Systems and Software*, 2022.

[4] Khadijah Al Safwan and Francisco Servant. Decomposing the rationale of code commits: The software developer's perspective. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[5] Waad Aldndni, Na Meng, and Francisco Servant. Automatic prediction of developers' resolutions for software merge conflicts. *Journal of Systems and Software*, 206:111836, 2023.

[6] Alex Wawro. What exactly goes into porting a video game? blitworks explains. `http://tiny.cc/r5jqsz`, August, 2014. [Online; accessed September 2023].

[7] anonymous. Onespace, November 2020.

[8] Apache Inc. Lucene tool. `http://lucene.apache.org`. [Online; accessed September 2023].

[9] Kubilay Atasu and Thomas Mittelholzer. Linear-complexity data-parallel earth mover's distance approximations. In *International Conference on Machine Learning, PMLR*, pages 364–373, 2019.

[10] AtCoder Inc. Antlr tool. `http://www.antlr.org.` [Online; accessed September 2023].

[11] AtCoder Inc. Atcoder programming contests. `https://atcoder.jp/`, 2020. [Online; accessed February 2020].

[12] Feng. AZhangyin, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. A pre-trained model for programming and natural languages. In *arXiv preprint arXiv:2002.08155*, 2020.

[13] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95. IEEE, 1995.

[14] Maria Barrett, Ana Valeria González, Lea Frermann, and AnFders Søgaard. Unsupervised induction of linguistic categories with records of reading, speaking, and writing. In *NAACL-HLT*, 2018.

[15] Gabriele Bavota and Barbara Russo. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 315–326, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[17] Stefan Bellon, Rainer Koschke, and Giuliano Antoniol. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. *IEEE Transactions on Software Engineering*, 33(6):367–390, 2007.

[18] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a" siamese" time delay neural network. In *Advances in neural information processing systems*, pages 737–744, 1994.

[19] Elizabeth Burd and Malcolm Munro. Investigating the maintenance implications of the replication of code. In *1997 Proceedings International Conference on Software Maintenance,*, pages 322–329. IEEE, 1997.

[20] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to Rank using Gradient Descent. In *International Conference on Machine learning*, pages 89–96, 2005.

[21] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *arXiv preprint arXiv:1512.08546*, 2015.

[22] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186. ACM, 2014.

[23] Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. Mining revision histories to detect cross-language clones without intermediates. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 696–701. ACM, 2016.

[24] Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. Clcminer: Detecting cross-language clones without intermediates. In *IEICE TRANSACTIONS on Information and Systems 100, no. 2*, pages 273–284, 2017.

[25] Tincy Thomas Chungath, Athira M. Nambiar, and Anurag Mittal. Transfer learning and few-shot learning based deep neural network models for underwater sonar image classification with a few samples. *IEEE Journal of Oceanic Engineering*, pages 1–17, 2023.

[26] Lykes Claytor and Francisco Servant. Understanding and leveraging developer inexpertise. In *International Conference on Software Engineering: Companion Proceeedings*, 2018.

[27] CLCDSA. Clcdsa replication. `https://github.com/Kawser-nerd/CLCDSA`, 2019. [Online; accessed May 2020].

[28] Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie. Xiao: Tuning code clones at hands of engineers in practice. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, page 369–378, New York, NY, USA, 2012. Association for Computing Machinery.

[29] Yingnong Dang, Dongmei Zhang, Song Ge, Ray Huang, Chengyun Chu, and Tao Xie. Transferring code-clone detection and analysis to practice. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 53–62, 2017.

[30] Debeshee Das, Noble Saji Mathews, and Sridhar Chimalakonda. Exploring security vulnerabilities in competitive programming: An empirical study. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, EASE '22, page 110–119, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Neil Davey, Paul Barson, Simon Field, Ray Frank, and D. Tansley. he development of a software clone detector. In *International Journal of Applied Software Technology*, 1995.

[32] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.

[33] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[34] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *arXiv preprint arXiv:1810.04805*, 2018.

[35] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99)*, pages 109–118, 1999.

[36] Schaeffer Duncan, Andrew Walker, Caleb DeHaan, Stephanie Alvord, Tomas Cerny, and Pavel Tisnovsky. Pyclone: A python code clone test bank generator. In Hyuncheol Kim, Kuinam J. Kim, and Suhyun Park, editors, *Information Science and Applications*, pages 235–243, Singapore, 2021. Springer Singapore.

[37] Manaal Faruqui and Chris Dyer. Improving vector space word representations using multilingual correlation. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 462–471, 2014.

[38] Hans-Christian Fjeldberg. Polyglot programming. *Norwegian University of Science and Technology*, 2008.

[39] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330. ACM, 2008.

[40] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. An empirical study of activity, popularity, size, testing, and stability in continuous integration. In *International Conference on Mining Software Repositories*, 2017.

[41] Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger. Relation of code clones and change couplings. In *Fundamental Approaches to Software Engineering: 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006. Proceedings 9,*, pages 411–425. Springer, 2006.

[42] Shrikant Gupta, Rajat Gupta, Muneendra Ojha, and K. P. Singh. A comparative analysis of various regularization techniques to solve overfitting problem in artificial neural network. In *International Conference on Recent Developments in Science, Engineering and Technology, Singapore, 2017*, pages 363–371. Springer, 2017.

[43] Sk Adnan Hassan, Zainab Aamir, Dongyoon Lee, James C. Davis, and Francisco Servant. Improving developers' under-

standing of regex denial of service tools through anti-patterns and fix strategies. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1238–1255, 2023.

[44] Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg*, pages 243–246, 2017.

[45] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[46] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[47] Joab Jackson. Netflix builds a pipeline for polyglot programming. In *theNewStack blog: https://thenewstack.io/netflix-builds-pipeline-polyglot-programming/*, 2017.

[48] Jackson, Joab . Microsoft: Rust is the industry's 'best chance' at safe systems programming. `https://thenewstack.io/microsoft-rust-is-the-industrys-best-chance-at-safe-systems-programming/`, June, 2020. [Online; accessed November 2020].

[49] Kalervo Järvelin and Jaana Kekäläinen. IR Evaluation Methods for Retrieving Highly Relevant Documents. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 41–48, 2000.

[50] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE, 2007.

[51] Xianhao Jin. Reducing cost in continuous integration with a collection of build selection approaches. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[52] Xianhao Jin and Francisco Servant. A cost-efficient approach to building in continuous integration. In *International Conference on Software Engineering*, 2020.

[53] Xianhao Jin and Francisco Servant. Cibench: A dataset and collection of techniques for build and test selection and prioritization in continuous integration. In *International Conference on Software Engineering: Companion Proceedings*, 2021.

[54] Xianhao Jin and Francisco Servant. What helped, and what did not? an evaluation of the strategies to improve continuous integration. In *International Conference on Software Engineering*, 2021.

[55] Xianhao Jin and Francisco Servant. Which builds are really safe to skip? maximizing failure observation for build selection in continuous integration. *Journal of Systems and Software*, 2022.

[56] Xianhao Jin and Francisco Servant. Hybridcisave: A combined build and test selection approach in continuous integration. *ACM Transactions on Software Engineering and Methodology*, 32(4), may 2023.

[57] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pages 171–183, 1993.

[58] J. Howard Johnson. Substring matching for clone detection and change tracking. In *ICSM, vol. 94*, pages 120–126, 1994.

[59] Kaggle Inc. Kaggle codechef competetive programming dataset. `https://www.kaggle.com/datasets/arjoonn/codechef-competitive-programming?`, 2017. [Online; accessed October 2022].

[60] Tomoyuki Kajiwara and Mamoru Komachi. Building a monolingual parallel corpus for text simplification using sentence similarity based on alignment between word embeddings. In *COLING*, 2016.

[61] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. In *IEEE Transactions on Software Engineering 28, no. 7*, pages 654–670. IEEE, 2002.

[62] Cory J. Kapser and Michael W. Godfrey. Supporting the analysis of clones in software systems. In *Journal of Software Maintenance and Evolution: Research and Practice 18, no. 2*, pages 61–82, 2006.

[63] Ayaan M Kazerouni, James C Davis, Arinjoy Basak, Clifford A Shaffer, Francisco Servant, and Stephen H Edwards. Fast and accurate incremental feedback for students' software tests using selective mutation analysis. *Journal of Systems and Software*, 2021.

[64] Ayaan M. Kazerouni, Clifford A. Shaffer, Stephen H. Edwards, and Francisco Servant. Assessing incremental testing practices and their impact on project outcomes. 2019.

[65] Marcus Kessel and Colin Atkinson. On the efficacy of dynamic behavior comparison for judging functional equivalence. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, page 193–203. IEEE, 2019.

[66] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, 2005.

[67] Yuriy Kochura, Yuri Gordienko, Vlad Taran, Nikita Gordienko, Alexandr Rokovyi, Oleg Alienin, , and Sergii Stirenko. Batch size influence on performance of graphic and tensor processing units during training and inference phases. In *Advances in Computer Science for Engineering and Education II*, pages 658–668. Springer International Publishing, 2020.

[68] Kostas A. Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. Pattern matching for clone and concept detection. In *Automated Software Engineering 3, no. 1-2 (1996)*, pages 77–108, 1996.

[69] Nicholas A. Kraft, Brandon W. Bonds, and Randy K. Smith. Cross-language clone detection. In *SEKE 2008*, pages 45–59, 2008.

[70] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017.

[71] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In *International Conference on Machine Learning, ICML*, 2015.

[72] Bruno Laguë, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *1997 Proceedings International Conference on Software Maintenance*, pages 314–321. IEEE, 1997.

[73] Guillaume Lample, Myle Ott, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Phrase-based & neural unsupervised machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, page 5039–5049, 2018.

[74] Levick, Ryan. Microsoft security response center: Why rust for safe systems programming. `https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/`, July, 2019. [Online; accessed May 2020].

[75] Changchun Li, Jihong Ouyang, and Ximing Li. Classifying extremely short texts by exploiting semantic centroids in word mover's distance space. In *WWW '19*, 2019.

[76] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. In *IEEE Transactions on software Engineering 32, no. 3*, pages 176–192. IEEE, 2006.

[77] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on github. In *Proceedings of the ACM on Programming Languages 1, no. OOPSLA*, page 84. ACM, 2017.

[78] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie

Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. 2021.

[79] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. In *Journal of machine learning research 9, no. Nov 2008*, pages 2579–2605, 2008.

[80] George Mathew, Chris Parnin, and Kathryn T. Stolee. Slacc: Simion-based language agnostic code clones. In *ICSE*. ACM, 2020.

[81] George Mathew and Kathryn T. Stolee. Cross-language code search using static and dynamic analyses. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 205–217. ACM, 2021.

[82] Philip Mayer and Alexander Bauer. An empirical analysis of the utilization of multiple programming languages in open source projects. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10, 2015.

[83] Philip Mayer, Michael Kirsch, and Minh Anh Le. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. In *Journal of Software Engineering Research and Development 5, no. 1*, page 1, 2017.

[84] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *1996 Proceedings of International Conference on Software Maintenance*, pages 244–253. IEEE, 1996.

[85] T. J. McCabe. A complexity measure. In *in IEEE Transactions on Software Engineering, vol. SE-2, no. 4, Dec. 1976, doi: 10.1109/TSE.1976.233837*, pages 308–320. IEEE, 1976.

[86] Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *International Conference on Automated Software Engineering*, 2019.

[87] Microsoft Cooperation. Python 3.7.4 api documentation. `https://learn.microsoft.com/en-us/cpp/cpp/?view=msvc-170`, 2020. [Online; accessed November 2022].

[88] Microsoft Cooperation. Python 3.7.4 api documentation. `https://learn.microsoft.com/en-us/cpp/c-language/?view=msvc-170`, 2020. [Online; accessed November 2022].

[89] Mihált, Márton. Google w2v pre-trained model. `https://github.com/mmihaltz/word2vec-GoogleNews-vectors`, 2016. [Online; accessed May 2020].

[90] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *ICLR Workshop 2013*, pages 957–966, 2013.

[91] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[92] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. Clcdsa: Cross language code clone detection using syntactical features and api documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1026–1037. IEEE, 2019.

[93] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. Exploring api embedding for api usages and applications. In *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 438–449. IEEE, 2017.

[94] Oracle Systems. Java se-7 api documentation. `https://docs.oracle.com/javase/7/docs/api/`, 2020. [Online; accessed February 2020].

[95] Daniel Perez and Shigeru Chiba. Cross-language clone detection by learning over abstract syntax trees. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 518–528. IEEE, 2019.

[96] Petri Pulkkinen, Johannes Holvitie, Olli S Nevalainen, and Ville Leppänen. Reusability based program clone detection: case study on large scale healthcare software system. In *Proceedings of the 16th International Conference on Computer Systems and Technologies*, pages 90–97, 2015.

[97] Python Software Foundation. Python 3.7.4 api documentation. `https://docs.python.org/3/download.html`, 2020. [Online; accessed February 2020].

[98] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 367–377. IEEE, 2013.

[99] Chanchal K. Roy and James R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 157–166, 2009.

[100] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[101] Chanchal Kumar Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[102] Sebastian Ruder, Ivan Vulić, and Anders Søgaard. A survey of cross-lingual word embedding models. In *Journal of Artificial Intelligence Research 65*, pages 569–631, 2019.

[103] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 191–201, 2015.

[104] Naveen Saini, Sriparna Saha, and Pushpak Bhattacharyya. Multiobjective-based approach for microblog summarization. *IEEE Transactions on Computational Social Systems*, 6:1219–1231, 2019.

[105] Vaibhav Saini, Farima Farmahini farahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, pages 354–365. ACM, 2018.

[106] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168. ICSE, 2016.

[107] Francisco Servant. Supporting bug investigation using history analysis. In *International Conference on Automated Software Engineering*, 2013.

[108] Francisco Servant and James A Jones. History slicing. In *International Conference on Automated Software Engineering*. IEEE, 2011.

[109] Francisco Servant and James A Jones. History slicing: Assisting code-evolution tasks. In *International Symposium on the Foundations of Software Engineering*, 2012.

[110] Francisco Servant and James A Jones. WhoseFault: Automatic Developer-to-Fault Assignment through Fault Localization. In *International Conference on Software Engineering*, 2012.

[111] Francisco Servant and James A Jones. Chronos: Visualizing slices of source-code history. In *Working Conference on Software Visualization*, 2013.

[112] Francisco Servant and James A Jones. Fuzzy fine-grained code-history analysis. In *International Conference on Software Engineering*, 2017.

[113] Dharmendra Shadija, Mo Rezai, and Richard Hill. Towards an understanding of microservices. In *23rd International Conference on Automation and Computing (ICAC), Huddersfield*, pages 1–6, 2017.

[114] Samuel L. Smith, David HP Turban, Steven Hamblin, and Nils Y. Hammerla. Offline bilingual word vectors, orthogonal transformations and the inverted softmax. In *ICLR-2017*, 2017.

[115] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethu-

madhavan. Identifying functionally similar code in complex codebases. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 2016.

[116] SupLearn. Suplearn replication. `https://github.com/danhper/suplearn-clone-detection`, 2019. [Online; accessed February 2020].

[117] Jeff Thomas Svajlenko. Large-scale clone detection and benchmarking. *PhD diss., University of Saskatchewan*, 2018.

[118] Federico Tomassetti and Marco Torchiano. An empirical assessment of polyglotism in github. In *Proceedings of the 18th International Confer- ence on Evaluation and Assessment in Software Engineering (London, England, United Kingdom) (EASE '14). ACM, New York, NY, USA, Article 17, 4 pages. https://doi.org/10.1145/2601248.2601269*, 2014.

[119] Filip Van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques. In *Proc. Int'l Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*, pages 25–36, 2003.

[120] Tijana Vislavski, Gordana Rakić, Nicolás Cardozo, and Zoran Budimac. Licca: A tool for cross-language clone detection. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 512–516. IEEE, 2018.

[121] Andrew Walenstein and Arun Lakhotia. Clone detector evaluation can be improved: Ideas from information retrieval. In *Proceedings of the 2nd International Workshop on Detection of Software Clones (IWDSC'03)*, 2003.

[122] Andrew Walenstein and Arun Lakhotia. The software similarity problem in malware analysis. In *Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik*, 2007.

[123] Chang Wang and Sridhar Mahadevan. Manifold alignment preserving global geometry. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, IJCAI*, pages 1743–1749, 2013.

[124] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM, 2016.

[125] John Wieting, Mohit Bansal, Kevin Gimpel, and Karen Livescu. Towards universal paraphrastic sentence embeddings. In *ICLR-2016*, 2016.

[126] Wolfram Wingerath, Felix Gessert, Steffen Friedrich, and Norbert Ritter. Real-time stream processing for big data. In *IT-Information Technology 58, no. 4, Proceedings of Machine Learning Research, PMLR*, pages 186–194, 2016.

[127] Shiwen Yu, Ting Wang, and Ji Wang. Data augmentation by program transformation. *Journal of Systems and Software*, 190:111304, 2022.

[128] Ruru Yue, Zhe Gao, Na Meng, Yingfei Xiong, Xiaoyin Wang, and J. David Morgenthaler. Automatic clone recommendation for refactoring based on the present and the past. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 115–126. IEEE, 2018.

[129] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software*, 204:111796, 2023.

Mohammed El-Arnaoty is a Ph.D. student working with Dr. Francisco Servant at the Department of Computer Science, Virginia Tech. He earned his M.S. in Computer Science and his B.S. from Cairo University, Egypt. His interests include empirical software engineering, applied machine learning in software engineering, and automated software engineering.

Francisco Servant is Assistant Professor at the Institute for Software Engineering and Software Technology (ITIS) of the University of Málaga and Adjunct Faculty at Virginia Tech. He received a Ph.D. in Software Engineering from the University of California, Irvine, advised by James A. Jones. He also holds a M.S. in Information and Computer Sciences from the University of California, Irvine, supervised by André van der Hoek. Francisco obtained his B.S. in Computer Science from the University of Granada in Spain. In his research, Francisco uses software evolution analysis and program analysis to create practical, efficient, and human-friendly techniques and tools that provide automatic support for all stages of software development. His research interests include software development productivity, software quality, mining of software repositories, program comprehension, and software visualization. He has published articles in these areas at top software engineering conferences (e.g., ICSE, FSE, ASE) and he has performed research for large technology companies, such as Microsoft Research and DreamWorks Animation. Dr. Servant received the NSF CAREER award, an International Distinguished Researcher award, and was selected for the Ramón y Cajal award. His research is supported by the United States National Science Foundation (NSF), an International Distinguished Researcher award, and the Spanish Agencia Estatal de Investigación (AEI).