JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Learning from Code Repositories to Recommend Model Classes

Thibaut Capuano[*], **Houari Sahraoui**[†], **Benoit Frenay**[‡], **and Benoit Vanderose**[‡]
[*]Haulogy, Belgium
[†]DIRO, Université de Montréal, Canada
[‡]Faculté d'informatique, Université de Namur, Belgium

ABSTRACT With the growing popularity of machine learning algorithms, dramatic advances have been made for code completion, and specifically method-call completion. These advances were also possible thanks to the availability of large code repositories to learn from and to the well-defined boundaries of the method-call completion problem. This is, however, not the case for design completion, where model repositories are scarce and the space of possibilities for design completion is theoretically infinite. We propose in this paper an approach that learns numeric representations of domain concepts and their relations from code repositories in order to recommend classes for UML class diagrams.

KEYWORDS Model completion, document embedding, Doc2Vec.

1. Introduction

The Model-Driven Engineering (MDE) paradigm is changing the way we create software. Models are becoming first-class citizens that are exploited to automatically generate various software artifacts, such as code (Loniewski et al. 2010) and test cases (Gutiérrez et al. 2015). Creating domain models such as UML class diagrams remains a complex task that requires a large spectrum of knowledge and expertise, i.e., modeling environments, languages, and more importantly domain knowledge. Offering assistance for such a task is important to alleviate the burden of software designers, especially novice ones.

Design completion is a good way to offer such an assistance. However, unlike for method-call completion, design completion is more complex for mainly two reasons. Firstly, this activity must rely on various data sources from which domain knowledge can be inferred. Although a significant number of data sources are available at the code level thanks to open source initiatives, such sources are scarce at the design level (Rocco et al. 2021). The second reason lies in the space of completion possibilities. Unlike for method-call completion, the set of possibilities is infinite as any concept is a potential element for completion. Indeed, the completion for method calls consists in predicting the next method to call from a finite set of possibilities, in general. The boundaries of this set are defined by the context of the completion site, the typing system, the code already written and the linked libraries.

To circumvent the constraints of the availability of data sources and the infinite space of possibilities, we propose, in this paper, an approach for UML Class diagram completion based on the hypotheses: (1) the huge mass of code available in open repositories contains elements that cover a large part of domain concepts, and (2) such domain concepts can be abstracted from the code.

Our approach uses a document embedding algorithm to learn domain concepts and their relationships from identifiers used in diagrams reverse engineered from the code. Then these concepts are recommended to the designers according to the similarities between the design context at a given stage and the relationships between the learned concepts. In the use case scenario we consider, a potential designer starts by defining some classes and their relations in a partial diagram. Then, our recommending tool suggests other domain concepts that can be potentially related to the already defined classes. When one or more concepts are accepted by the designer and added to the diagram, the recommendation process is repeated until the dia-

JOT reference format:
Thibaut Capuano, Houari Sahraoui, Benoit Frenay, and Benoit Vanderose. *Learning from Code Repositories to Recommend Model Classes*. Journal of Object Technology. Vol. 21, No. 3, 2022.
http://dx.doi.org/10.5381/jot.2022.21.3.a4

gram is considered complete. Our goal is therefore, to produce an assistant for completing the design itself and not just to help represent domain concepts in UML notations. We implemented our approach and evaluated its performance by learning completion models from 14,000 Java projects and testing them on 30 projects. Our results show that we are able to suggest relevant concepts in an iterative process of diagram construction, with an average relevance higher than 90% with as few as 5 suggestions at each iteration. Among other findings, the results reveal that learning from diagram fragments is more efficient than from whole diagrams. Additionally, using sophisticated ranking with TF-IDF improves the quality of the suggestions as it avoids favoring frequent general terms.

The remainder of this paper is organised as follows. Section 2 provides a motivating example and some background about text embedding techniques. Section 3 describes the training and completion phases of the proposed approach. The evaluation setup is explained in Section 4 including the addressed research questions, the implementation information, the data set, and the evaluation procedure. The evaluation results are presented in Section 5. Section 6 discusses the related work. Finally, Section 7 provides concluding remarks and outlines the future work.

2. Motivation and Background

2.1. Motivating example

To illustrate the difference between method-call and design completion, let us start by considering the following Java method, which returns the size of a file.

```
1  // ...
2  public long size() throws IOException {
3      if (!file.isFile()) {
4          throw new FileNotFoundException(
5              file.toString());
6      }
7      return file.? // prediction
8  }
```

Listing 1 Example of code context for suggestion.

After typing `file.` in line 7, a basic completion system could exploit a static analysis system (imports, language typing, etc.) to suggest a list of around 50 methods that can be invoked from this calling site (e.g., `canRead()`, `delete()`, `getName()`, `getPath()`, `isFile()`, `length()`, `toString`, etc.). This list can be refined and ranked, for example, by exploiting the similarity of the method sequence `size()` (line 2), `isFile()` (line 3), `toString()` (line 5) and `file.?` (line 7) with sequences involving the methods found by static analysis in large repositories of source code (Weyssow et al. 2020). This may give a higher ranking to method `length()`, which is the right option.

Now consider the completion of the class diagram shown in Fig. 1. Suppose that only the classes in black are defined by the designer, namely, `Car Model` and `Car`. Unlike for the example of call completion, any concept can potentially be suggested, e.g., `Company`, `Person`, `House`, or `Stadium`. In the absence of any form of knowledge, the existing context with two classes does not allow to define a finite set of suggestions from which the designer can choose.

The knowledge that helps reducing the infinite set of possibilities to a limited number of suggestions can be provided, for example, as a specification from which can be inferred concepts related semantically to those already defined by the designer (Saini et al. 2020). Having complete specifications is not always possible. As an alternative, semantic relations can also be abstracted from an existing repository of models and reused for model completion (Weyssow et al. 2021). However model repositories are scarce and do not provide a data volume at the level of those expected for efficient machine learning algorithms. For example, with a small model set, the likelihood to abstract a strong relation between the context `Car Model - Car` and the concept `Engine` is low.

We conjecture, in this paper, that code repositories can be good surrogate sources to model repositories for abstracting accurate semantic relations between concepts. For example, the likelihood to find a strong relation between `Car Model - Car` and `Engine` is higher considering the thousands of code projects that can be used.

2.2. Background

To explore the idea of abstracting high-level concepts and their relations from code repositories, for model completion, we exploit the document embedding technique. In the remainder of this section we briefly introduce this technique.

Embedding is one of the different techniques that exist to represent complex information. Each concept (word, diagram, etc.) is represented by a vector of numbers. The embeddings can be exploited in different ways. They can be used to train another model that requires numerical data, empowering it to learn on complex data. The embedding can also be visualized to give the user an insight on the information it contains, such as clusters. One can also calculate the similarity between each pair of vectors, e.g., using cosine similarity. This can be used by other algorithms, e.g., to find the closest concept to another concept.

Figure 2-left illustrates the concept of embedding for words (notice that the dimension of the embedding is in practice much larger). Word2Vec (Mikolov, Chen, et al. 2013; Mikolov, Sutskever, et al. 2013) is the most well-known algorithm to embed words. It is built on a neural architecture (Bengio et al. 2006) that learns an internal, numerical representation for each word in the considered vocabulary. In the continuous bag-of-words (CBOW) architecture, the model learns to predict the word at each position in a text, based on the words in a given window. Each word in the window is mapped to its internal representation, which are then averaged or concatenated to predict the target word. The resulting numerical representations of the words (e.g., the embedding) contains semantic information about them. For example, Figure 2-right shows an example of semantic relationships that can be preserved in the representation.

While Word2Vec provides interesting results for isolated

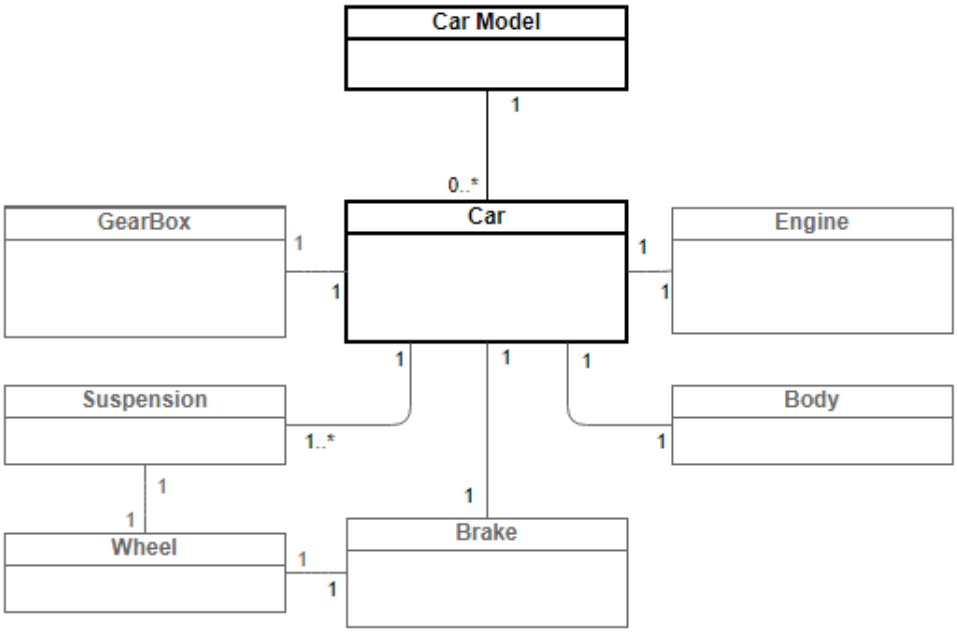


Figure 1 An excerpt of the Car class diagram. In the context of completion, black classes are already defined by the designer, and gray classes are those that should be suggested.

concepts, it is unable to characterise entire documents. Paragraph vectors (Le & Mikolov 2014) have been introduced for that purpose. When the neural architecture learns to predict a word based on close words, it is now allowed to build an additional vector for each paragraph. This paragraph vector embeds the semantic context of the whole paragraph and helps to use that context to better predict individual words. Interestingly, paragraph vectors can then be used as an embedding of the paragraphs themselves to represent them and to understand their relationship. The same can be done for entire documents, resulting in document vectors (Doc2Vec). This numerical representation can be used for example to find documents that are similar to a new one.

As this work focuses on diagrams that contain complex information (e.g., structure), document embedding seems to be appropriate to represent those. Thus the analogy between a document and a diagram is made.

3. Approach

The proposed approach for design completion has two phases: learning and completion. The learning phase consists in abstracting domain concepts and their relations from a code dataset using the document embedding technique. The abstracted concepts are then used in the second phase to assist the designer during modeling activities.

3.1. Learning Phase

The learning phase is shown in Figure 3. The first step is to reverse engineer the class diagrams from Java projects (T1). As we are interested in high-level concepts, we limit the reverse engineering to the classes and their relations. To limit the size of the derived diagrams, we consider one diagram per main package. So a Java project may result in many diagrams according to the packages it contains. This decision is also motivated by the fact that classes within the same package tend to implement semantically-related concepts. This helps reducing the noise that can result from very large diagrams containing hundreds of classes. We provide, in Section 4, more information on the data set used in our experimentation.

The second step of the learning phase is to encode the extracted diagrams into documents (T2). As we use the document embedding technique, i.e., define vector representations of documents, the choice of this mapping is important. Identifiers within the same document increase the probability of semantic proximity between them. In this respect, we explore two assumptions: (1) a diagram as a document and (2) a basic related fragment as a document.

In the first case, all the identifiers that appear in a diagram are included in the corresponding document. Consider that the diagram in Fig. 1 is part of the learning dataset. `Car Model`, `Car`, `GearBox`, `Body`, etc. are all part of the same document. This assumption, call it global, represents the idea that being part of the same diagram, i.e., package, is an indication of a semantic relation with the other elements of the diagram.

For our second assumption, call it local, we consider basic related fragments in a diagram as documents. As our diagram is a graph, a fragment of interest is two nodes related by a link, i.e., two related classes, or three for association classes. For the diagram in Fig. 1, we will have separate documents for `Car` and `Engine`, `Car` and `Brake`, `Brake` and `Wheel`, etc. The rationale behind this assumption is that explicitly-related concepts have strong semantic relations.

For the local assumption, we use the relation direction to

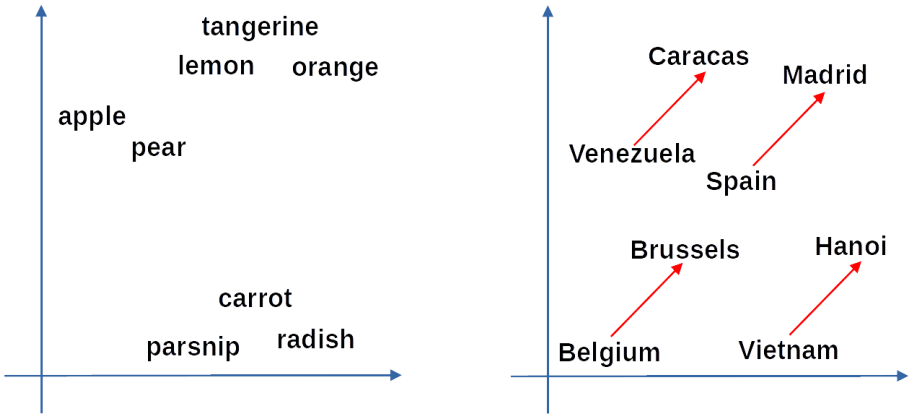


Figure 2 Two-dimensional representation of an embedding of words. The left graph depicts how similar words are represented by vectors closer to each other. The right graph shows that the relation between words is consistently encoded in the position of the vectors.

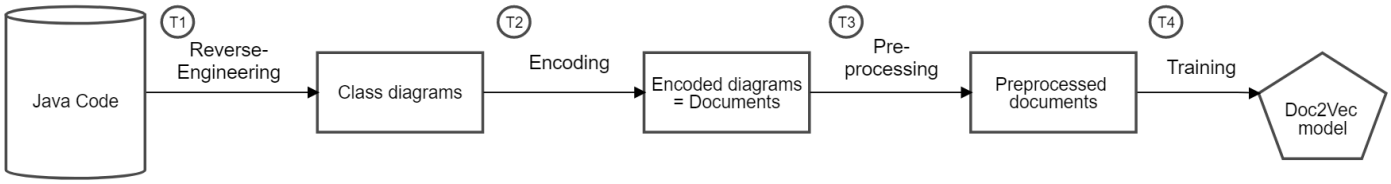


Figure 3 Illustration of the process used to train a Doc2Vec model using Java code.

define the order in which the identifiers in a fragment are added to the document. For the global encoding, we first identify the classes without incoming relations (roots) as starting points. Then, we use a deep-first traversal of the diagram to add the identifiers. For a node in the diagram, i.e., a class, the sub-nodes, i.e., related classes, do not have a specific order. Then, when traversing these related classes, a random order is chosen. Although different orders may affect differently the learned representations, we do believe the large amount of code we use for the training limits the order effect. Moreover, for both assumptions, this encoding process simulates somehow the way a diagram is created step by step by a designer.

After encoding the diagrams into documents, the next step is to pre-process the documents, i.e., to clean the identifiers in order to harmonise the vocabulary between documents (T3). As illustrated in Table 1, for each identifier included in a document, we successively apply three operations: (1) tokenization to split the identifiers into single words, (2) lemmatization to map the inflected forms of a word to a single root form called lemma, and (3) stop words removal to delete common words that have little lexical meaning such as *the*, *to*, *by*, etc.

The cleaned documents are fed into the Doc2Vec learning algorithm to generate a descriptive vector for each document in order to predict the closest document for a given document based on these vectors (T4).

3.2. Completion Phase

The completion process is shown in Figure 4. When a designer has defined a partial diagram, this is converted into one or more documents (C1) and preprocessed in the same way as in the learning phase (C2), i.e., tokenization, lemmatization, and stop words removal. Then, the resulting documents are given as input to the learned Doc2Vec model to infer the most similar documents (C3). Documents with a similarity higher than a predefined threshold determine the set from which the concepts to suggest should be extracted.

Extracting suggestions from the similar documents is done in two steps (C4). Firstly, similar documents may contain a long list of terms. Presenting such a long list to the designer is not useful if the terms are not ranked by their estimated relevance. In our approach, we use two methods to rank the terms in the similar documents. The first technique relies on the frequency. The more a term appears in the similar documents, the higher its relevance is. The rationale behind this option is that the strength of the relation between the defined model fragment and a given term depends on the frequency with which both elements are associated in the learning corpus. The drawback however is that generic terms used in many domains tend to be suggested systematically on top of the lists because of their frequency.

To cope with this limitation, we explore an alternative ranking technique, namely TF-IDF. TF-IDF (short for term frequency-inverse document frequency) is a measure that aims to mitigate the term frequency with the specificity of the term to a limited number of documents (Schütze et al. 2008). In other words, the frequency of a term is multiplied by a factor that decreases when the number of documents in which the term appears increases.

In the second suggestion step, we can directly present the

Preprocessing technique	Class name
None	RotatedByModifier
Tokenization	Rotated By Modifier
Tokenization + lemmatization	Rotate By Modifier
Tokenization + lemmatization + stop words removal	Rotate Modifier

Table 1 Effect of the successive preprocessing operations on a class-name example.

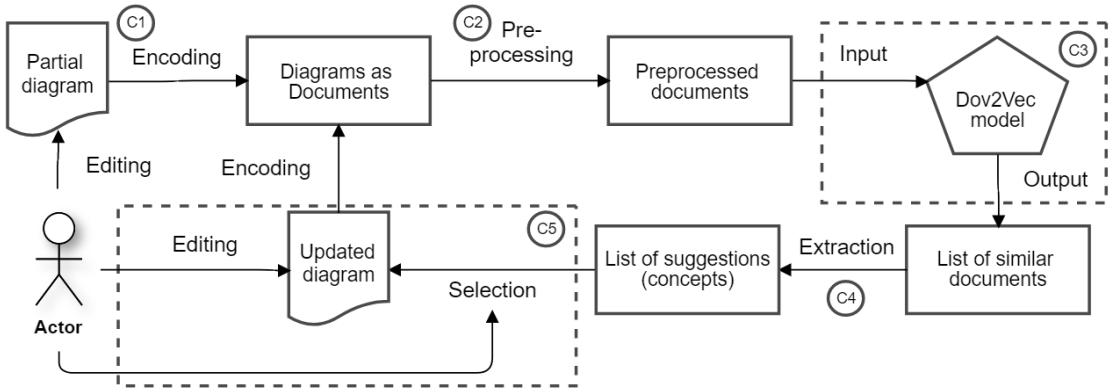


Figure 4 Illustration of the process used to suggest concepts for diagram completion.

ranked list of terms to the designer. In addition to this option, we also consider suggesting concept names created from the ranked terms. To this end, we select the terms with relevance scores (frequency or TF-IDF) higher than a given threshold. Then, from the documents where these terms were extracted, we retrieve the identifiers containing the different forms of the terms. Each identifier is assigned a relevance score corresponding to the sum of the scores of frequent terms composing it.

In the final step of the completion process the designer selects the appropriate concepts and edits the diagram (C5). This starts a new completion cycle.

4. Evaluation Setup

In the proposed approach, we had to make many decisions such as the encoding of diagrams into documents or the ranking of the suggestions to present to the designer. In this section, we evaluate the decision alternatives.

4.1. Evaluation Procedure and Metrics

4.2. Research Questions

Our evaluation aims to answer the following research questions.

- **RQ1: What is the quality of the suggestions produced by our approach?** *To answer this question, we consider both options of encoding the diagrams presented in Section 3.1.*
- **RQ2: What is the effect of the completion context, i.e., the partial diagram already defined by the designer, on the relevance of the suggestions?** *For this question, we evaluate two aspects: (1) the ability of our approach to suggest relevant neighbouring classes, i.e., missing classes that should be connected to the already defined partial diagram, vs any missing class of that diagram, and (2) the evolution of the suggestion relevance with the size of the context, i.e., the number of classes already defined by the designer.*
- **RQ3: Does using term frequency or TF-IDF produce a better ranking of suggestions?**

4.3. Implementation and Hyperparameters

To implement our approach, we used the GENSIM library[1] with a Doc2Vec algorithm based on Paragraph Vector - Distributed Bag of Words (PV-DBOW). We used a grid search to define the best hyperparameters to use. Accordingly, the vectors to represent the documents are set to 400 dimensions. This vector size is the best trade-off we found between a better representation of the documents with a high number of dimensions and a reasonable size of the learned representations. For whole diagram encoding, we set the window in which the terms are considered as close to 20. For fragment encoding, a size of 20 was enough to have all the terms included within a unique window. We set to 3 the minimum occurrences of a term in the training data set to be considered. The model was trained over 30 iterations and uses hierarchical softmax to speed up the training process (Morin & Bengio 2005).

4.4. Dataset

To conduct our evaluation, we used an existing repository, the GitHub Java Corpus (Allamanis & Sutton 2013). This repos-

[1] https://radimrehurek.com/gensim/

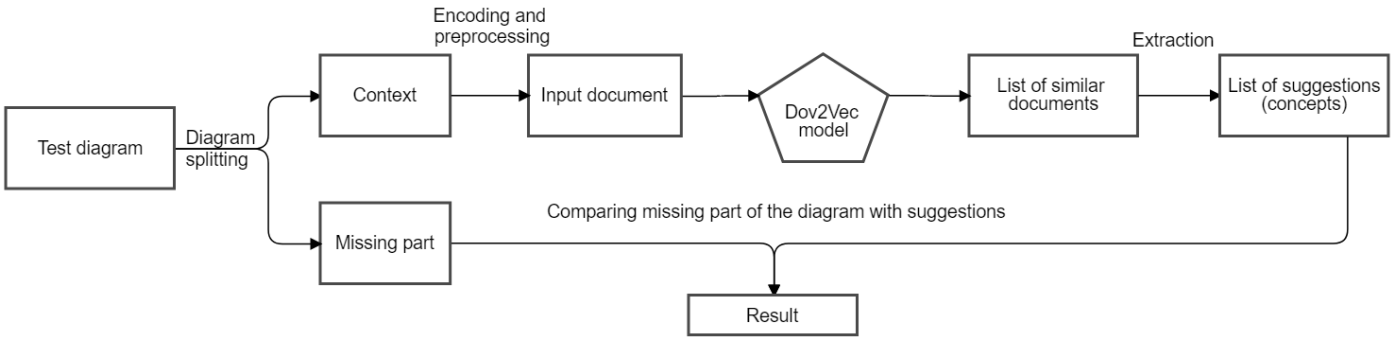


Figure 5 Evaluation procedure for a single diagram.

itory contains 14,000 projects totalling about 2,000,000 java files. From this corpus, we extracted 30 projects to test our completion approach. To have a consistent selection criterion, we took the 30 top-ranked projects according to Allamanis and Sutton (Allamanis & Sutton 2013). From these 30 projects, we manually defined 30 class diagrams representing the domain concepts used in the projects. The remaining projects were used to train the Doc2Vec models. From the training projects, the class diagrams are extracted. The diagrams are then mapped to documents and preprocessed according to the strategies described in Section 3.1.

After training the Doc2Vec model with the GitHub Java Corpus, we use the evaluation procedure summarized in Figure 5. Each of the diagrams, obtained from the 30 test projects, is split into two parts, one simulating the partial model defined by the designer (the completion context) and one representing the missing part containing the concepts that should be suggested by our completion approach. The context part is chosen randomly, but it should define a connected graph to simulate the progressive construction of a class diagram by a designer. Its size depends on the research question, a fixed random size for questions related to the number of suggestions, and a variable size from 2 to 22 classes for the question dealing with the context size effects. As for the training diagrams, the context parts of the test diagrams are converted to documents and preprocessed. Then the obtained documents are given to the learned Doc2Vec model to find similar diagrams from which the suggestions are extracted.

To evaluate the results by comparing the suggestions to the missing parts, we use two metrics: the average precision@k and the average relevance@k. These two metrics correspond to two use-case scenarios of the completion. In the first scenario, the designer defines a diagram and asks for potential additional concepts to include. In that case, the $precision@k \in [0,1]$ measures the proportion of k suggestions that are present in the missing part. For the second scenario, the designer starts by defining her diagram with very few classes (two in this experiment), then the completion system recommends one or more classes to add. If the list of suggestions contains a missing class, then this is integrated into the diagram. The designer can modify the diagram and trigger a new round of completion. In this case, the *relevance@k* equals 1 if the k suggestions include at least one missing class, 0 otherwise.

In our evaluation, we experimented with k between 1 and 10. In the results, *precision* (respectively *relevance*) refers to the average *precision@k* (respectively *relevance@k*) of the tested diagrams. Moreover, as the majority of class names contain a single term after removing the stop words, we decided to use the highly-ranked terms as suggestions. A suggestion is then considered relevant if the suggested term equals or is included in a missing class name. In addition to *precision@k* and *relevance@k*, we also report on the average rank (*rank*) of the first relevant concept in the suggestion list.

Note that in our evaluation, we do not use real designers to assess if the suggestions are correct or not. We alternatively use the missing parts of the diagrams to simulate what the designers could have added to the partial diagrams to complete them.

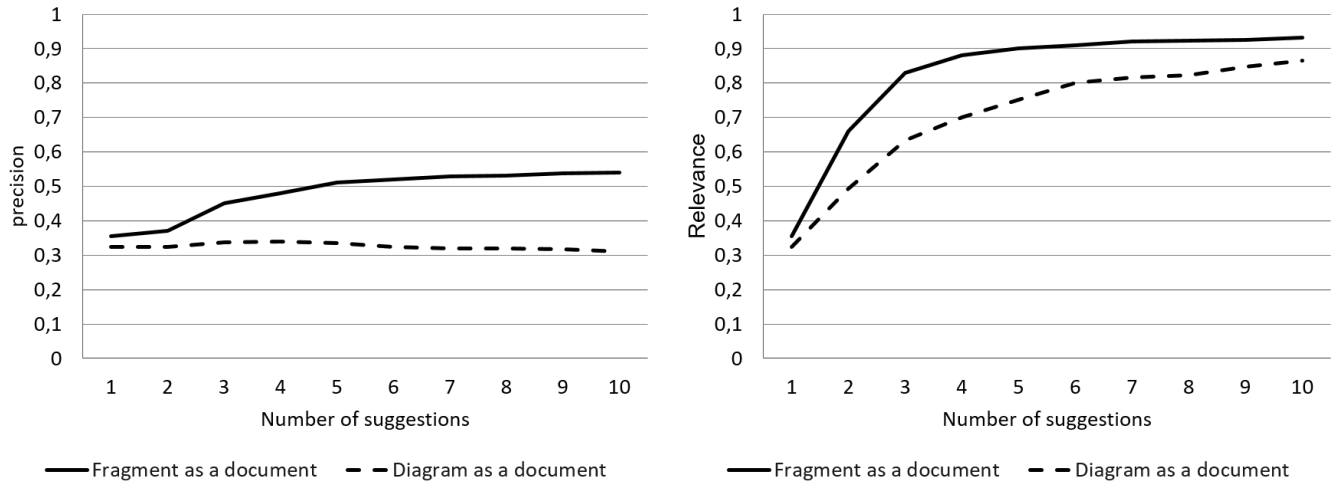
5. Results

5.1. Diagram Encoding Options (RQ1)

Figure 6 shows the variation of the performance of our approach w.r.t. the number of suggestions given to the designer for both encoding options (fragment vs diagram) of the training data. As the reader can see in Figure 6a, the average precision is better for the fragment-based encoding regardless of the number of suggestions. Starting from 5 suggestions, on average more than half of the suggested concepts are relevant (precision > 0.5). The superiority of fragment-based encoding is also observed for the average relevance as shown in Figure 6b. In average, a relevant concept is present in more than 80% if the completion tests with only 3 suggestions. This average increases to 90% or more with 5 suggestion or more. The better performance of the fragment-based encoding is also confirmed by the average rank, $rank = 2.2$ as opposed to $rank = 2.8$ for the whole-diagram encoding option.

> To answer **RQ1**, based on our sample of completion tests, we can state that using fragment-based encoding of diagrams into documents allows generating relevant concept suggestions in a majority of cases. This is particularly striking in a setting where the designer seeks for one suggestion at a time in an iterative diagram construction process (*relevance* metric).

For the rest of the study we will consider only the fragment-based encoding.



(a) Accuracy of the suggestions.

(b) Relevance of the suggestions.

Figure 6 Quality of the suggested concepts comparing two methods of encoding diagrams.

5.2. Effect of the Context Size (RQ2)

We study the effect of the completion context from two perspectives. Firstly, we want to assess whether starting from a partial diagram (context), our approach is able to suggest relevant concepts explicitly related to ones in the context, or any concepts pertinent to the whole targeted diagram. The second studied aspect is the effect of the context size, i.e., the size of the partial diagram given as completion context, on the relevance of the suggestions.

Figures 7a and 7b show respectively the average precision and the average relevance when we consider as correct suggestions (1) only the concepts matching missing classes directly related to those in the partial diagram, and (2) concepts matching any missing class. For the average precision, the neighbouring classes represent the larger part of the correct suggestions, 80% for the first 2 suggestions and more or less 60% for 3-to-10 suggestions. From the average relevance perspective, for a striking majority of completion cases, our approach suggests at least one correct neighbouring class, i.e., the area bounded by the neighbouring classes curve spans over almost all the area bounded by the all missing class curve.

In the second analysis, we look at the average precision and relevance (@10) when we vary the size of the completion context from 2 classes to 22. As shown in Figure 8, both metrics are stable regardless of the size of the partial diagram already defined by the designer.

To answer **RQ2**, we can state that our approach suggests mostly concepts that are directly related to the completion context, but also other relevant concepts that are not directly related to the latter. Additionally, the average precision and relevance of the suggestions are independent from the completion context size.

5.3. Ranking the Suggestions (RQ3)

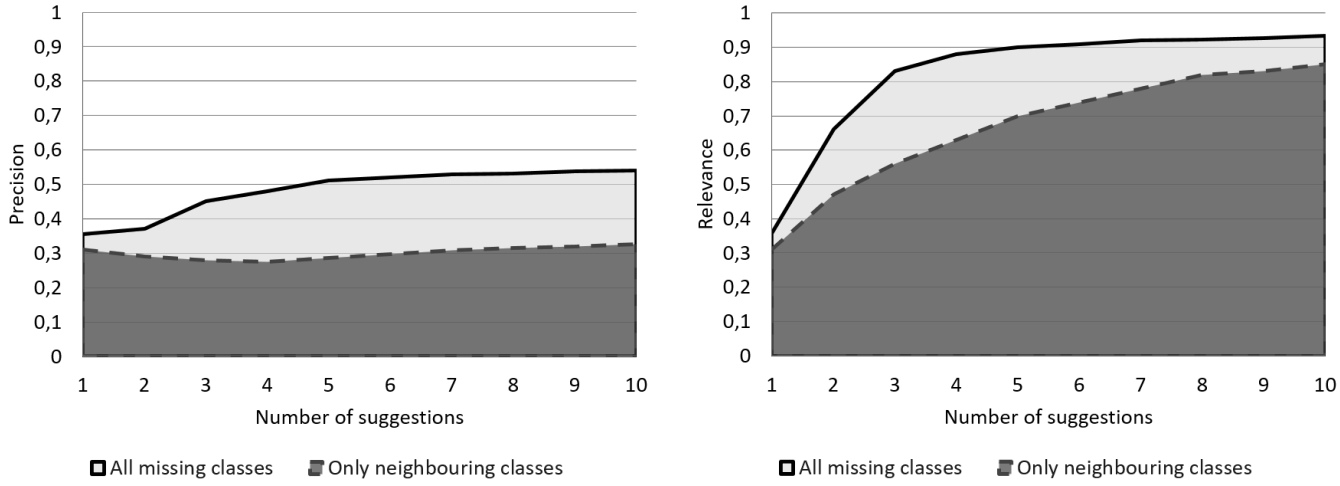
For a given completion context, when the learned model gives the most similar documents, one has to decide what ranking metric to select terms to extract to form the suggestions. To this end, we compare the frequency of terms in the similar documents with TF-IDF to rank the terms when varying the number of suggestions.

Figures 9a and 9b show respectively the average precision and the average relevance when raking the terms with the frequency and with TF-IDF. For both metrics, TF-IDF allows selecting more relevant terms especially for the top-ranked suggestions. Moreover, the average rank of the first relevant suggestion is 1.6 for TF-IDF ranking whereas it is 2 for the frequency-based ranking. We conjecture that these results are due to the fact that domain-specific terms are favored by TF-IDF over general terms that appear in many projects.

To answer **RQ3**, we can state that using TF-IDF to select the terms from the Doc2Vec model output is more effective than relying on term frequencies.

5.4. Threats to Validity and Discussion

Although our evaluation showed promising results, several threats can limit its validity. The first threat concerns the class diagrams we used to test our completion approach. These diagrams where defined based on our comprehension of the selected projects. For the training data, the class diagrams were reverse engineered using a custom tool that parses the java code and extracts classes, inheritance relations and basic associations. Association extraction is based on attribute types. It does not consider the multiplicities nor the refinement into associations vs aggregations or compositions. We acknowledge that this simple reverse-engineering method may have an impact on the training results. We are currently exploring the use of more powerful tools such as MoDisco (Bruneliere et al. 2010). An-



(a) Precision for neighbouring vs all missing classes.

(b) Relevance for neighbouring vs all missing classes.

Figure 7 Evaluation of the suggestions quality comparing both objective of suggestion.

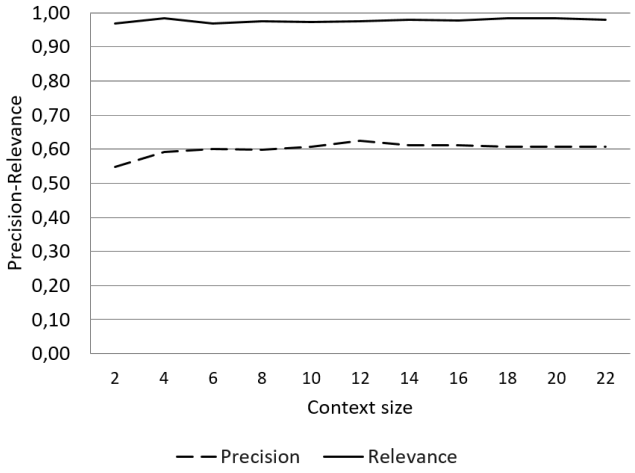


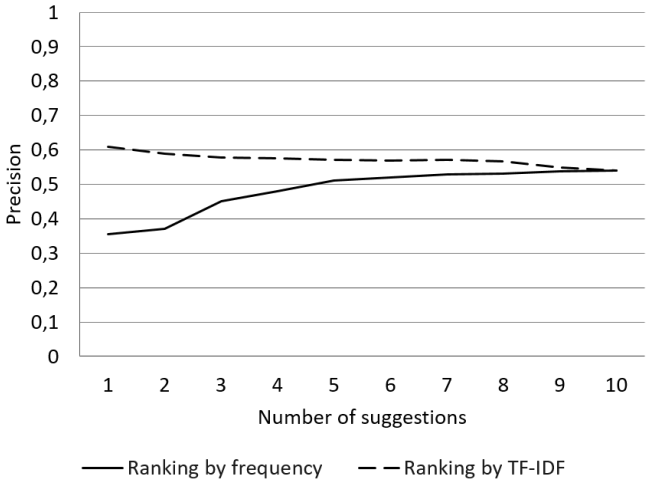
Figure 8 Suggestions quality with a varying context size.

other threat is related to the use of a single term to suggest a concept, e.g., ”Session", rather than a group of terms, e.g., ”Remote Session". In our evaluation, we consider that ”Session" is relevant if this term is included in one of the missing-class names, e.g., ”RemoteSession". This threat has a limited impact on the results for many reasons. First, the majority of classes involved in the evaluation have names with a single term after removing the stop words. Additionally, as discussed in RQ3, using TF-IDF gives more weight to terms that are specific as opposed to general terms that can appear in many names such as ”Abstract", ”Basic", ”Default", etc. Using the term exact match when evaluating the relevance of the suggestions is another threat. Indeed, if the term ”Teacher" is suggested and none of the missing classes contain this term, then it is tagged as non-relevant, even if there is a class ”Lecturer" in the suggestions. The exact match tends to under-estimate the quality of the suggestions. In the future, we plan to use semantic similarity measures as alternatives to exact match. This also helps mitigating the single-term threat discussed above.

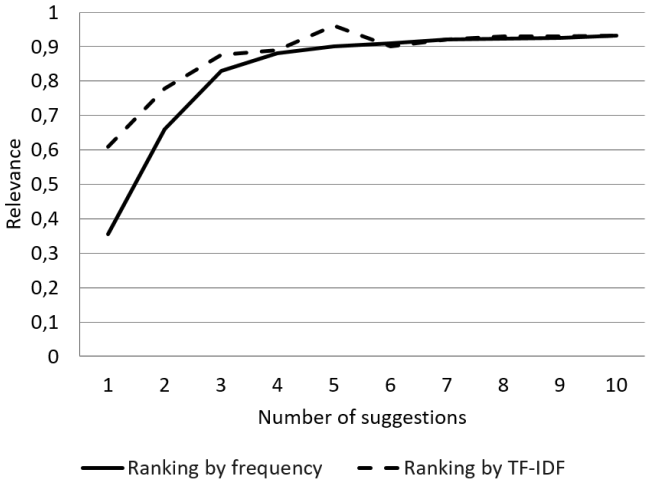
For the approach itself, for the global assumption, we used a depth-first strategy to encode the diagrams, with a random traversal of the sub-nodes. A different encoding may have led to different performance for the global assumption. As future work, we can explore other encoding strategies and study their effects. Finally, some concepts derived from the code can be related to the implementation, and may not be relevant to the design level. This is evidenced by the relatively low accuracy in Figure 6a. However, Figure 6b shows that within a limited set of suggestion, there is always at least a concept relevant to the design level.

6. Related work

Using completion for UML class diagram has been studied by different research teams in the last decade. This work can be classified into two families, structure/well-formedness completion and content completion. For the first family, an early work on model completion was proposed by Sen et al. (Sen et al. 2010). They used the metamodel specification to guide the modeling activity through the editor. Later different approaches used the same idea by suggesting editing operations for partially-defined model fragments (e.g., (Steimann & Ulke 2013) and (Kuschke et al. 2013)). The second family targeted the recommendation of model elements starting from an existing model fragment. An interesting work in that direction is the one by Elkamel et al. (Elkamel et al. 2016). In this work, a set of classes is extracted from existing models and clustered using a similarity metric. When a user defines a new class, their tool compares this class to those existing in the repository and suggest similar classes. Although very interesting, this work has two limitations. First, the limited coverage of the available class diagrams, as pointed out in Section 2.2, lower the likelihood to suggest relevant classes. More importantly, the completion approach does not suggest new concepts, but just concepts similar



(a) Average precision for frequency vs TF-IDF.



(b) Average relevance for frequency vs TF-IDF.

Figure 9 Comparing the two ranking strategies for terms selection.

to one already introduced. The work that is more closely related to ours is one by Burgueño et al. (Burgueño et al. 2021). In this paper, they propose a general embedding-based approach to suggest concepts in design models. There are two main differences compared to our work. The first difference is related to the sources from which the learning data is extracted. They use general textual documents such as Wikipedia and Google news. This may introduce noise in the learned models because of the ambiguity of the general language. The second difference comes from the embedding technique they use, word embedding. Indeed, in our approach we use document embedding which allows to exploit the similarity among the diagrams in addition to one between terms. More recently, Weyssow et al. (Weyssow et al. 2021), proposed an approach to suggest concepts in metamodel activities. Their approach consists in retraining an existing deep learning model RoBERTa on a limited set of metamodels. In an extensive evaluation, they found that their approach is good at renaming existing concepts, but has less compelling results for suggesting new concepts in an iterative metamodel construction process. Finally, Di Rocco et al. (Rocco et al. 2021) used graph neural networks to lean recommenders for model and metamodel completion. Like for our work, they trained the model recommender with data extracted from Java projects.

Our approach uses source code as training data to derive high-level concepts to use for the completion. This school of thought comes from the code completion community based on the idea of code naturalness and its exploitation using advanced natural language processing and machine learning techniques (Allamanis et al. 2018). The existing work targets many completion objectives. For example Nguyen et al. (Nguyen et al. 2021) propose an approach for the completion of API function calls and code snippets suggestions. A similar embedding-based approach was proposed by Weyssow et al. (Weyssow et al. 2020) for recommending function calls not specifically for API. An interesting related work was proposed by Allamanis et al. (Allamanis et al. 2015) to recommend class and method names in the code. Although close to our problem, this work does not suggest new concepts, but rather names to already defined classes. UML class diagram is closely related to object oriented code. Thus similar techniques may be used to work with both of these data. In code completion, the closest thing that can be related to concepts suggestion in class diagram is class names suggestion.

7. Conclusion

The goal of the work described in this paper is to provide support to designers when creating UML class diagrams. The rationale behind our approach is to exploit existing software artefacts to abstract high-level concepts that can be used as suggestions for a given context in a modeling activity. However, because of the limited number of publicly-available class diagrams, they cannot be used as data from which completion models can be learned. Alternatively, we use, in this work, the large amount of available source code to reverse engineer class diagrams to serve as training data.

The novel approach we propose exploits these data through a document embedding algorithm to create a multidimensional space that encodes both diagrams and enclosed concepts. The resulting model allows suggesting new concepts that are semantically related to those already-defined by the designer. We evaluated our approach by learning from 14,000 Java projects and testing on 30 diagrams. Our results show that we are able to suggest relevant concepts in an iterative process of diagram construction, with an average relevance higher than 90% with as few as 5 suggestions. Among other findings, the results also reveal that learning from diagram fragments is more efficient than from whole diagrams. Additionally, using sophisticated ranking with TF-IDF improves the quality of the suggestions as it avoids favoring frequent general terms.

The good results obtained for class diagrams encourage us to generalize the idea of code-based completion to other kinds of

diagrams. Many diagrams can be reverse-engineered from the code to create learning data. In addition to this future investigation, we plan to conduct other studies to address the limitations of the proposed approach. More specifically, the data used in this work can be completed by fine-grained data about attribute and method parameter identifiers and types. Additionally, we will explore transfer learning techniques to retrain code-based models with limited sets of diagrams. We also envision closing the loop by actually generating diagram fragments based on accepted concept suggestions. Finally, we plan to compare our approach with code-completion tools that suggest domain concepts such as GitHub Copilot[2].

Acknowledgments

The authors would like to thank Martin Weyssow for his valuable comments on this work. This work was partially funded by the MITACS-Université de Montréal partnership program.

References

Allamanis, M., Barr, E. T., Bird, C., & Sutton, C. (2015). Suggesting accurate method and class names. In *Foundations of software engineering - ESEC/FSE* (pp. 38–49).

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, *51*(4), 1–37.

Allamanis, M., & Sutton, C. (2013). Mining Source Code Repositories at Massive Scale using Language Modeling. In *Work. conf. on mining software repositories* (pp. 207–216).

Bengio, Y., Schwenk, H., Senécal, J.-S., Morin, F., & Gauvain, J.-L. (2006). Neural probabilistic language models. In D. E. Holmes & L. C. Jain (Eds.), *Innovations in machine learning: Theory and applications* (pp. 137–186). Springer Berlin Heidelberg.

Bruneliere, H., Cabot, J., Jouault, F., & Madiot, F. (2010). Modisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the ieee/acm international conference on automated software engineering* (pp. 173–174).

Burgueño, L., Clarisó, R., Gérard, S., Li, S., & Cabot, J. (2021). An NLP-based architecture for the autocompletion of partial domain models. In M. La Rosa, S. Sadiq, & E. Teniente (Eds.), *Advanced information systems engineering* (pp. 91–106).

Elkamel, A., Gzara, M., & Ben-Abdallah, H. (2016). An UML class recommender system for software design. In *Int. conf. of computer systems and applications.*

Gutiérrez, J., Escalona, M., & Mejías, M. (2015). A model-driven approach for functional test case generation. *Journal of Systems and Software*, *109*, 214–228.

Kuschke, T., Mäder, P., & Rempel, P. (2013). Recommending auto-completions for software modeling activities. In *Int. conf. on model driven engineering languages and systems* (pp. 170–186).

Le, Q., & Mikolov, T. (2014). Distributed representations of sentences and documents. In *Int. conf. on machine learning* (pp. 1188–1196).

Loniewski, G., Insfran, E., & Abrahão, S. (2010). A systematic review of the use of requirements engineering techniques in model-driven development. In *Int. conf. on model driven engineering languages and systems* (pp. 213–227).

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. In *Int. conf. on learning representations.*

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Int. conf. on neural information processing systems* (p. 3111–3119).

Morin, F., & Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In *International workshop on artificial intelligence and statistics* (pp. 246–252).

Nguyen, P. T., Di Rocco, J., Di Sipio, C., Di Ruscio, D., & Di Penta, M. (2021). Recommending api function calls and code snippets to support software development. *IEEE Transactions on Software Engineering*.

Rocco, J. D., Sipio, C. D., Ruscio, D. D., & Nguyen, P. T. (2021). A GNN-based recommender system to assist the specification of metamodels and models. In *Int. conf. on model driven engineering languages and systems* (pp. 70–81).

Saini, R., Mussbacher, G., Guo, J. L. C., & Kienzle, J. (2020). Domobot: A bot for automated and interactive domain modelling. In *Int. conf. on model driven engineering languages and systems: Companion proceedings.*

Schütze, H., Manning, C. D., & Raghavan, P. (2008). *Introduction to information retrieval* (Vol. 39). Cambridge University Press Cambridge.

Sen, S., Baudry, B., & Vangheluwe, H. (2010). Towards domain-specific model editors with automatic model completion. *Simulation*, *86*(2), 109–126.

Steimann, F., & Ulke, B. (2013). Generic model assist. In *Int. conf. on model driven engineering languages and systems* (pp. 18–34).

Weyssow, M., Sahraoui, H., & Syriani, E. (2021). Recommending metamodel concepts during modeling activities with pre-trained language models. *arXiv preprint arXiv:2104.01642*.

Weyssow, M., Sahraoui, H., Vanderose, B., & Frénay, B. (2020). Function completion in the time of massive data: A code embedding perspective. *arXiv preprint arXiv:2008.03731*.

About the authors

Thibaut Capuano is a Software Engineer and Architect at Haulogy working on the combination of data science and software engineering in the field of smart energy management. He graduated from Université de Namur in computer science with a specialization in Data Science. This work was done during a research stay at Université de Montréal. You can contact the author at Thibaut-c@hotmail.com.

[2] https://copilot.github.com/

Houari Sahraoui is a professor at the department of Computer Science and Operations Research (GEODES Group) of the Université de Montréal. His research interests include the application of AI techniques to software engineering, automated software engineering, search-based software engineering and model-driven engineering. You can contact the author at sahraouh@iro.umontreal.ca .

Benoit Frenay is an associate professor with the Université de Namur, Namur, Belgium. His research interests in machine learning include interpretability, interactive machine learning, dimensionality reduction, label noise, robust inference, and feature selection. He was the recipient of the Scientific Prize IBM Belgium for Informatics in 2014 for his Ph.D. thesis on uncertainty and label noise in machine learning. You can contact the author at benoit.frenay@unamur.be.

Benoit Vanderose is an assistant professor in software engineering at the Faculty of Computer Science of the University of Namur, Belgium. His research interests include Agile methods customization, software processes and products quality and digital transformation in public administrations. You can contact the author at benoit.vanderose@unamur.be.