Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN DATA SCIENCE

Application of document embedding for class name recommendation during UML class diagram creation

Capuano, Thibaut

Award date:
2020

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

FACULTÉ
D'INFORMATIQUE

Application of document embedding for class name recommendation during UML class diagram creation

Thibaut Capuano

Remerciements Acknowledgements

Je tiens tout d'abord à remercier toutes les personnes qui ont contribué de loin ou de près à la réalisation de mon projet de recherche, à la rédaction de mon mémoire et qui m'ont permis de réaliser mon stage à l'université Montréal.

Je voudrais exprimer ma gratitude envers mon maitre de stage, Pr. Sahraoui, qui m'a guidé et conseillé durant la réalisation de mon projet de recherche. Il m'a permis de comprendre la rigueur nécessaire pour mener à bien un projet de recherche.

Je suis reconnaissant envers mes promoteurs, Pr. Frenay et Pr. Vanderose, qui m'ont offert la possibilité d'effectuer mon stage à l'université de Montréal. Cela m'a permis de rencontrer de nombreuses personnes qualifiées dans le domaine de ma recherche ainsi que de découvrir une nouvelle culture. Je tiens aussi à les remercier pour les feedbacks apportés, tant pour la réalisation du projet que pour la rédaction de mon mémoire.

Surtout, je voudrais remercier ma famille, dont mes parents et mon frère pour le support et l'encouragement qu'ils m'ont apporté durant la rédaction de mon mémoire.

Résumé and abstract

System quality is an important aspect during development. But, while code quality has an important place during system development, system engineering techniques are generally not fully exploited. This research propose a new approach in order to promote system engineering. It contributes to system engineering by proposing to apply information gathered in source code to help users during class diagram creation. This work uses of machine learning to recommend class names to the user. Few approaches use machine learning with class diagrams even though it has shown to be useful for recommendation systems in similar fields. Document embedding is used on the sequences of relations contained in code. Based on the partial diagram already drawn by the user, the embedding suggests similar sequences of relations from which tokens are extracted and then suggested to the user. As a next step, the system also suggests entire class names to the user based on those tokens. The class names are selected from all class names presents in the train set using a full text index. Those class names aims at guiding the user during its reflection in the class diagram creation process.

La qualité des sytèmes est un aspect important lors du développement. Mais, même si la qualité du code prend une place importante dans le développement de logiciels, les techniques d'ingénierie du système ne sont généralement pas pleinement exploitées. Cette recherche propose une nouvelle approche afin de promouvoir l'ingénierie des systèmes. Elle contribue à l'ingénierie des systèmes en proposant d'appliquer les informations recueillies sur du code source pour aider les utilisateurs pendant la création des diagrammes de classe. Ce travail propose d'utiliser du machine learning pour recommander des noms de classe à l'utilisateur. Peu d'approches utilisent du machine learning avec des diagrammes de classe, même si cela s'est avéré utile pour les systèmes de recommandation dans des domaines similaires. Le document embedding est utilisé pour modéliser les séquences de relations contenues dans le code. Et, sur base du diagramme partiel déjà dessiné par l'utilisateur, l'embedding suggère des séquences de relations similaires desquelles des tokens sont extraits et ensuite suggérés à l'utilisateur. Ensuite, des noms de classe sont également suggérés à l'utilisateur sur base de ces tokens. Les noms de classe sont sélectionnés parmi tous les noms de classe présents dans le jeu de données en utilisant un full text index. Ces noms de classe visent à guider l'utilisateur lors de sa réflexion dans le processus de création d'un diagramme de classe.

Contents

1 **Introduction** **1**

2 **Related works** **4**
- 2.1 UML class diagram completion 4
- 2.2 Code completion 5
- 2.3 Graph completion 7
- 2.4 Research questions. 8

3 **Background** **10**
- 3.1 Embedding and intuition 10
- 3.2 Artificial neural network 12
- 3.3 Word embedding 13
- 3.4 Document embedding 14
- 3.5 Visualization 16

4 **Proposed approach** **17**
- 4.1 Training and library choice 18
- 4.2 Full text indexing 19

5 **Datasets and preprocessing** **22**
- 5.1 Choice of datasets 22
- 5.2 Representing a diagram as a document 23
- 5.3 Preprocessing 26

6 **Validation of the initial assumption** **28**

7 **Evaluation** **30**
- 7.1 Experimental settings 30
  - 7.1.1 Evaluation process 30
  - 7.1.2 Formatting and preprocessing 31
  - 7.1.3 Evaluation metrics 33
- 7.2 Results 34
  - 7.2.1 RQ1: Can document embedding be applied to structure diagram? 34
  - 7.2.2 RQ2: How does document embedding performs when used to generate suggestions? 34
  - 7.2.3 RQ3: How can the suggested concepts quality be evaluated? 39
- 7.3 Discussion 39

8 **Conclusion** **46**

Appendices **52**

Chapter 1

Introduction

Observations

Code quality and system engineering are very important during the creation of software. They both play an important role in the creation of quality systems. But, during software development, code quality is receiving most attention while system engineering techniques like design and modelling are, when undertaken, generally not fully exploited [25]. The literature shows in [30] [25] and [32] that such a gap is due to the great number of tools that are present to help with code design as opposed to the relatively few tools that can help with UML diagram creation. Nevertheless, model driven engineering (MDE) brings many benefits to software creation, such as better design support, better documentation, better maintenance and better program quality [30]. There is therefore a real interest in developing tools to promote the use of those methods. For that reason, research focuses on the exploration of new techniques that facilitate UML class diagram creation. A UML class diagram is a static structure diagram that represents the structure of a program. It contains the program classes and the relations between those classes. Each class from the diagram has attributes and operations that correspond to variables and methods from the corresponding class in the program. A resource that can be used to facilitate UML class diagram creation is historical program data. Indeed, software is more and more present in everyday life and the source code of those software is easily accessible thanks to the open source community and technologies like GitHub. Source code could be used by new tools to help users during diagram creation.

How to help users during software modelling

The objective of this work can be described in two phases. First, it is to abstract knowledge from available source code. And second,it is to use this knowledge to suggest information and lighten the tasks of software modelling and model maintenance by guiding the user during those processes. A prevalent tool used in code development that has shown to be useful is completion. The same idea is chosen to be applied to software modelling. The aim is to develop a technique allowing completion during the design of UML class diagrams. In other words, to offer a software designer, a series of suggested feature based on the UML class diagram he/she is building. In order to abstract knowledge from

source code and to generate suggestions, a machine learning technique named document embedding is evaluated. During a modelling task, the idea is to start from the few elements already defined and to infer new elements to suggest based on knowledge extracted from a corpus of projects. This idea is depicted in Figure 2.1. The focus is set on suggesting concepts (class names) in order to guide engineers during their reasoning. This new tool should ease the design of systems, and promote modelling driven engineering.

Thesis structure

Chapter 2 of this work explores different works in order to evaluate if completion can be done for class diagrams and how it can be done. Those studies approach completion on three fields of computer sciences: Class diagrams, code and graphs. Code and graphs are fields that can easily be related to class diagram. Thus, works done in those fields can be applied to class diagram completion. For class diagram completion, different methodologies are presented such as: completion using series of editing operations or completion by creating clusters of similar classes. For code completion, techniques such as bi-grams and deep recurrent neural network are already used. They make it possible to suggest tokens to complete a line of code and to create an embedding that can suggest class and method names. For graph completion, embeddings are used in some work to suggest links in Knowledge graphs and another work uses graph grammar to suggest correction to the user. As seen in those works, embeddings are used in both code completion and graph completion, in addition to many other fields such as text classification ([18] and [33]). Studies on class diagram completion points out that, even though class diagram completion has already been studied, no research has been done on applying embedding to it. Therefore this research focuses on exploring the use of embeddings to support the creation, of class diagram with concepts suggestions. The first research question is then to see if document embedding can be applied to class diagrams. The second one is to evaluate how document embedding performs when generating suggestions for the users. Finally, the third research question explores how the quality of the suggested concept can be evaluated.

In Chapter 3, the concept of embedding is introduced and two variants are presented in detail: word embedding and document embedding. Embeddings are used to represent complex information in a way that can be used by computers. During the creation of an embedding, features are extracted from the data in an unsupervised manner to form a multidimensional vector space. Each concept is represented by a vector of numbers, making it easier to compute the similarity between concepts. Various implementations of document embedding are available and the choice of one of them is explained in this chapter.

Chapter 4 explains the choices made for the implementation with all their important characteristics. It is explained how class names are composed from those tokens using a full text index. Then a library is chosen from multiple libraries implementing document embedding and the meta-parameters of the model are optimized. Finally, full text indexing is introduced. It is used to compose class names based on tokens.

The various datasets that are used to train the embedding are presented in Chapter 5. It presents two ways of embedding the data. In the first one, the whole class diagram is represented as a document. In the second one, a class diagram is represented using multiple sequence of class relations. And each sequence is given as a document. The different pre-processing steps such as tokenization and lemmatization are required to improve the suggestion quality and are then explained.

The evaluation of the system takes place in Chapter 6. First, the experimental settings are presented, where the methodology followed used to evaluate the system is justified. Here are some decisions that are explained: the selection of projects used to evaluate the system, how those projects are used and which metrics are used. Second, results are presented where data is presented to help answer the research questions. Finally, obtained results are discussed and the research questions are answered.

The 7th chapter concludes the work by synthesizing the research that is done and by explaining how it helps software engineering.

Chapter 2

Related works

In this chapter, different works related to class diagram completion are presented. Different completion methods in three fields are discussed: UML class diagram completion, code completion and graph completion. Each work approaches a concept similar to class diagram completion and therefore relevant to preemptively study. The research gap is then discussed, the path explored in this work is explained and the research questions are presented.

2.1 UML class diagram completion

Completion in the field of UML class diagram has already been studied by different works using different approaches.

Kuschke et al. [15] address completion during modelling by completing the **modelling activity** that the user is currently working on. The objective is to ease the creation of class diagrams using suggestions. The concept of modelling activity used in that work can be described as the creation of a sub-part of the diagram, based on the user intention. Here are some examples of modelling activities given in [15]:

- replacing an association between two classes by an interface realization;
- extracting an attribute into an associated class;
- specializing an element inheriting to a sub element.

To complete a modelling activity, Kuschke et al. propose to detect partly performed activities and to suggest a series of editing operations that result into a complete modelling activity. Editing operations are atomic actions that can be performed by the user such as a class creation or the creation of a relation between two classes. To find the most relevant modelling activity the partly performed modelling activity is compared with similar sequences of modelling activities. The most similar modelling activities are then suggested to the user who can select the most appropriate that will be used to complete the diagram. But this approach does not suggest concepts to guide the user during his reflection. Instead, it helps the user during the creation of the diagram by suggesting

operations to finish the current modelling activity. One of the limitations of that approach is that it requires many predefined activity sequences in order to be applicable to a broad field of subjects. Kuschke et al. cited "traceMaintainer" [20], a tool that can create such sequences by detecting modelling activities and recording the **editing operations sequence**. But, in order to have a large panel of sequences, a large number of class diagram creations need to be recorded. This process is not usually done and thus a large dataset of modelling activity does not exist.

Another approach is suggested by Elkamel et al. [7] to suggest classes during UML class diagram design. The user is proposed with a **list of similar classes** to the last class created. Each class is proposed with all its components, which are the class name, attributes, and operations. The user is able to select the components to keep when selecting the class that needs to be added to the diagram or, he/she can accept the whole class immediately. All classes that can be suggested are stored in a clustered database where similar classes are grouped in the same cluster. In order to select which classes to suggest to the user, a similarity metric is computed using all the classes components. As class components are concepts, a similarity metric is needed to compare those concepts. Therefore, to calculate the similarity between classes, a weighted average is used giving more importance to the class name, then to the attributes, then to the operations. Then, to create the clusters with the classes, the Communicating Ants for Clustering with Backtracking strategy (CACB) algorithm [8] is used. This clustering algorithm creates **clusters** with a small number of classes per cluster which is good for this application as the number of suggestions should not be too high. The approach is evaluated with student projects which makes it difficult to estimate as those projects may not contains best and/or common practices.

2.2 Code completion

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.

The work of Raychev et al. [29] introduces a code completion approach that uses statistical language models. They address code completion in programs that uses API's, so that suggestions can be made while taking into account methods and classes that are present in the libraries imported in the code. The proposed method works by suggesting a line of code. And, in order to produce the suggestions, a statistical language model is proposed. It models regularities in methods call sequences by building a probability distribution of possible sequences of words. In order to complete a line of code, first, a **bi-gram model** is used to reduce the number of candidate words. That bi-gram model is a probability distribution of all tuples of successive words present in

the training dataset. Second, a **statistical language model** or a **recurrent neural network** (RNN) is used to find the most likely words that are missing. Their approach computes the probability of each sequence yielded by the bi-gram in order to suggest missing words. The RNN and the statistical language model being of the same nature can be combined by averaging the probability of both models in order to give better results. All proposed models are trained on a large corpus coming from Github. From that corpus the method call sequences are extracted.

N-gram models suffer from some defaults that affect suggestions quality. White et al. explain in their work [31] that such model does not take into account the general context as they are limited to a sequence of N words. Moreover *N-gram* models lack information such as semantic similarity between words. Other models like two-layer feedforward neural networks also lack the ability to model the general context of the information they are modelling. In order to face these problems, they suggest a new technique to model information that takes into account the general context of a program. Different architectures of deep recurrent neural network are explored. They then apply the model on code suggestion as a case study and conclude that **deep recurrent neural networks** outperforms n-gram for the code suggestion task.

Allamanis et al. [3] tackle methods and classes name suggestion in source code. A neural probabilistic language model is used to learn an **embedding** that suggests method and class names based on the subtokens that compose them. Subtokens are used to train the model and therefore the names have to be recreated based on the suggested subtokens. Indeed, a class name, also called a token, such as *ModuleLoader* is composed of two words, $Module$ and $Loader$. Those two words are called subtokens and each of them carry some meaning. It is therefore interesting if the model can make suggestions based on both those suggestions as the full class name may be too specific. Moreover, as explained, class and method names often are neologism, which means that those names do not yet exist. Allamanis et al. point out the difficulty to suggest names that are functionally descriptive for both methods and classes. Thus, in order to suggest relevant names, they use the context and a set of features of methods and classes. Two models are involved in the generation of suggestions. First, a **neural probabilistic language model** is used to predict tokens based on different features such as the other tokens contained in the class or method. This model is a **logbilinear context model** which is called an embedding. In an embedding, each vector represents a token and is computed by using both the token context which is the surrounding tokens, and a set of features such as the variable type, return type, number of arguments, superclass ... The second model used for suggesting class name is a **subtoken context model**. This is a logbilinear model that is used to generate words that did not appear in the training set. To compute the likelihood of a token, a first set of vectors representing the subtokens are used. Then, to generate a token, the first subtoken is predicted based on the context, followed by next subtokens based on both the context and the previous subtokens. In order to make those predictions, each subtoken is assigned a second vector that helps compute its influence on the following subtokens.

In code completion, the suggestion of class name is less frequent, nevertheless, the approach proposed by Allamanis et al. can suggest class names. So the use of an embedding and subtokens is interesting to explore for class diagram.

2.3 Graph completion

As an UML class diagram can be represented by a graph, graph completion can be used to complete such diagrams. The task of concepts suggestion in diagrams can be compared with the task of nodes label suggestion in graphs.

Mazanek et al. [22] use that similarity between graphs and class diagrams to complete class diagrams. They use **graph grammar** to define the class diagram and then give suggestions to the user that aim at correcting mistakes in the diagram based on its meta-model. To do so, first, a spacial relationship hypergraph (SRHG) is generated based on the diagram. Then this hypergraph is simplified to reduce later timer complexity. Based on the graph model, a syntactical analysis is performed to compute the possible completions from which the user can choose. When selected by the user, the completion is embedded in the SRGH. The resulting SRGH can then be used to recreate the corrected diagram.

Lin et al. [19] propose TransR, a model for suggesting links in knowledge graphs. A knowledge graph is a centralized source of knowledge used to access information coming from different sources in a structured manner. In their approach the graphs are represented by two **embeddings**. The first one embed the entities, and the second one embed the relations between those entities. A property of those embeddings is that when two entities are projected from the entities embedding to the relations embedding, if it exists a relation r between those two entities h and t, then the vector resulting from the addition of the vector of the entity h and the vector of the relation r is close to the vector of the entity t, such as

$$\boldsymbol{V}h + \boldsymbol{V}r \simeq \boldsymbol{V}t. \tag{2.1}$$

Where $\boldsymbol{V}$ e is a vector representing the entity e. They apply this technique on three evaluation tasks: link prediction, triple classification and relational fact extraction. For link prediction, they use a triplet (h, r, t) where t is replaced by each possible entity from the knowledge graph. The model is then asked to evaluate each generated triplet and rank them.

Both works have different objectives than nodes label suggestion. The work of Mazanek et al. aims at correcting mistakes in the diagram while the work of Lin et al. focuses the suggestion of edges in a graph. They are therefore not applicable to concept suggestion in class diagrams.

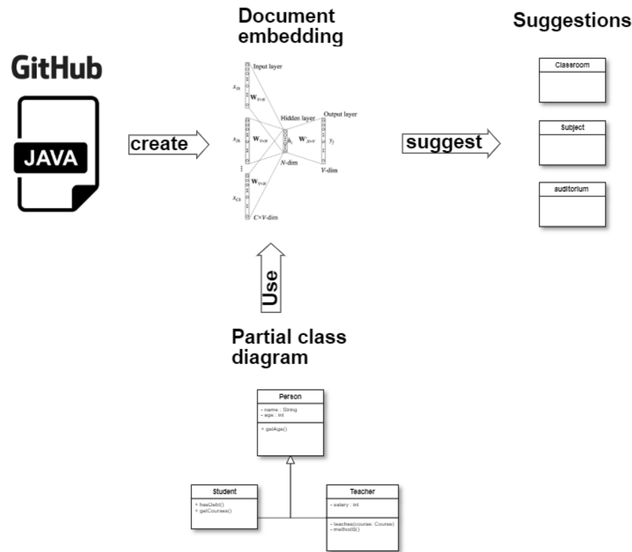


Figure 2.1: This diagram presents the intention of the proposed approach. Code available online is used to generate an embedding. The embedding can then be used to offer completion suggestions on a partially built diagrams.

2.4 Research questions.

As shown by the various works presented previously, different approaches have already been applied to class diagram completion. In their work, Kürsh et al. [15] aim at helping the user during the creation of the diagram by suggesting operations to finish the current modelling activity. Elkamel et al. suggest classes similar to the last drawn class using a clustered database of classes. Nevertheless, despite the existence of some works in diagram completion, the use of machine learning for this purpose has not yet been the subject of a comprehensive study. Machine learning is "the automatic discovery of regularities in data through the use of computer algorithms and with the use of these regularities to take actions such as classifying the data into different categories"[4]. In this particular case, the action taken is the suggestion of words, and the regularities the algorithm is learning are the co-occurrences of words in a text. The use of machine learning in other fields such as code completion and graph completion is not new as shown in [3] and [19] where embeddings are used. In order to evaluate if class diagram completion is possible using the large amount of data coming from open source software, **document embedding** is used in this work. Document embedding is a machine learning technique, to extract high-level concepts and the relationships that bind them from a large corpus of UML class diagrams. Word embedding is mainly used for text classification ([18] and [33]) and has been adapted for other purposes such as document embedding [21] and DNA embedding [26]. Since word embedding is also used for code completion [6], this work will evaluate whether it is possible to apply this technique to

the completion of UML class diagrams. Figure 2.1 shows the main idea of the approach that is explored in this work. The research questions intended to be answered are the following:

- **RQ 1: Can document embedding be applied to structure diagram?**
- **RQ 2: How does document embedding performs when used to generate suggestions?**
- **RQ 3: How can the suggested concepts quality be evaluated?**

This chapter introduces multiple techniques that can be used for class diagram completion coming from different fields. Some aspect of class diagram completion has not yet been studied such as the use of many methods of machine learning to suggest concepts that will guide the user. Therefore, the research questions about this aspect of the matter has been stated to better define the work that is done during this research.

Chapter 3

Background

In this chapter, the model used for computing similarity between concepts is presented in detail. First the concept of embedding is introduced. Second, artificial neural networks (ANN) are presented in order to then explain word embedding and document embedding in more details. Those technologies are important to understand the approach presented in this work. Third, a dimensionality reduction technique is introduced. It helps to visualize the produce embedding as it cannot be visualize directly.

3.1 Embedding and intuition

For computers to work with complex information, this information has to be **represented** in a way that can be used by computers. And, if some sources of information can be used directly such as tables and lists, others cannot be used directly in an optimum way, such as text. For example, in order to work with words from a text, it is interesting to be able to encode information about the usage of those words in the text or more generally in the whole language. The same problem is faced for all complex objects such as class diagrams. During class diagrams representation, it is interesting to take into account information such as class names and the relation between the words that compose them, the structure of the diagram and the components of each class (attributes and methods). It exists different techniques to represent complex information like one-hot encoding and embeddings.

One-hot encoding is a technique that represents each concept with a unique vector. Each of those vectors contains a single value set to one and is filled with zeros. An example of one-hot encoding is presented at table 3.1. This technique is useful to represent data with no ordering and works well with classification, but possesses some drawbacks. First, the representation is sparse, which means that the vector contains mainly zeros. Second, if it used to represent a great number of concepts, it will reach a high dimensionality. Which is the case when trying to represent words from large texts. It means that the number of values (zeros and ones) contained in a vector representing a word is as large as the number of elements to encode. Moreover, this representation will not contain

any information about the words. It is therefore not good to represent a great number of words and to make suggestions.

Hello	world	cat	...
0	0	1	
0	1	0	...
1	0	0	
	⋮		

Table 3.1: Example of words encoded using one-hot encoding.

Embedding is a method to represent any concept such as words or diagrams. Each element is represented by a vector of numbers similarly to one-hot encoding. But the values in this vector are not zeros and ones, they are numbers initialized during the creation of the embedding. Those numbers indicate the coordinates of the vector and all together form a multidimensional vector space. The embedding algorithm extracts features from the data in an unsupervised manner in order to create a vector for each element. This representation is dense when compared with one-hot encoding as the vector size (dimension) is smaller. An example of embedding is presented at table 3.2. An improvement compared to one-hot encoding is that the vector size is smaller than the number of elements to encode.

An embedding can be used in different ways: It can be used by another model to train using the embedding. It can be visualized to give the user an insight on the information it contains, such as clusters. And it can be used to calculate the similarity between two elements. Each element being represented by a vector, their similarity is computed using the cosine similarity between their vectors. The cosine similarity is a similarity metric computed by dividing the dot product of the vectors by the product of their magnitude such as

$$\cos\theta = \frac{A \cdot B}{||A|| \cdot ||B||}. \tag{3.1}$$

Embedding are chosen to be used as it allows to find elements similar to one that is presented. It is thus possible to present the embedding with the partial diagram by the user and to suggest the similar elements computed with the cosine similarity. In order to suggest relevant class names during class diagram creation, an embedding is created where each vector represents information from a diagram of the train set. As presented in section 5.2 each diagram is represented using the relations between its classes. This makes it possible to find similar relations based on the partial model the user has already drawn. Then, interesting concept can be found inside those similar relations and be used to form class names that will be suggested.

Hello	world	cat	...
0.4	0.2	0.6	
0.01	0.9	0.1	...
0.8	00.2	0.3	

Table 3.2: Example of words encoded using an embedding.

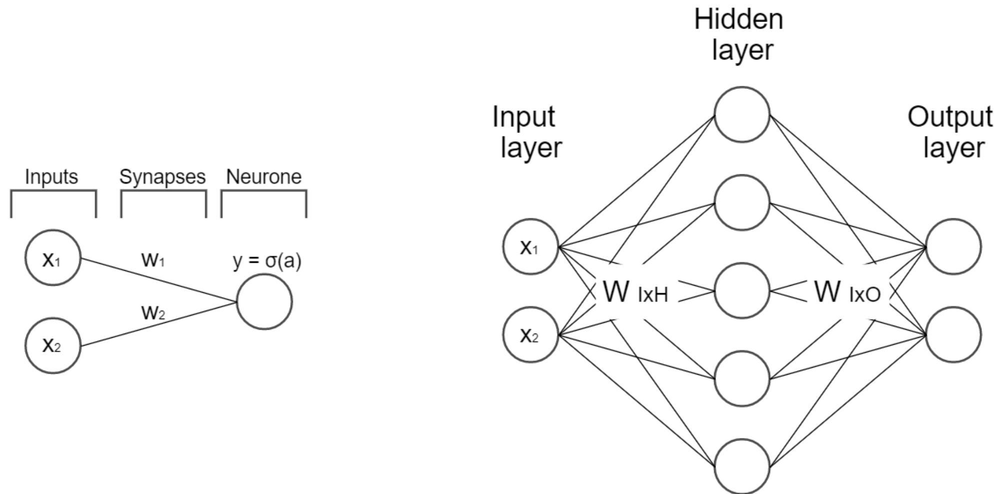


Figure 3.1: The left part of the diagram shows how the different components of the neural network are represented. And on the right a basic example of neural network with one hidden layer is presented.

3.2 Artificial neural network

In order to better understand how embeddings are created it is important to understand what is an artificial neural network (ANN) and how it works. An artificial neural network is a machine learning technique used for different tasks such as regression, clustering, density or estimation. It is inspired from the biological functioning of neurons to create a network of neurons linked by synapses, most often ordered in layers. The first layer is the input layer where inputs take their value from the different feature in object instance. The input layer is linked to the next layer by the synapses. Those synapses posses a weight that influences the values received by the next layer. The weight of the synapses are an important part of the neural network as they are updated during training to fit a pattern in the training set. Then comes the hidden layers, the right part of Figure 3.1 shows a neural network with only one hidden layer, but multiple layers can be added. Each neuron from the hidden layers posses an activation function which takes as argument a, the weighted sum of values from the previous layer. The result y of the activation function is used as input for the next layer. Some activation functions that are often used are:

-the **Sigmoid** that is computed with

$$\sigma(a) = 1/(1 + exp(-a)), \tag{3.2}$$

-the **identity**:

$$\sigma(a) = a, \tag{3.3}$$

-or the **Rectified linear unit ReLU**:

$$\sigma(a) = \begin{cases} 0 & \text{for } a \leq 0 \\ a & \text{for } a > 0 \end{cases} \tag{3.4}$$

Where σ represents the activation function, and a the input. It exists many other activation functions that will affect the result differently. The last layer is the output layer and is composed of one or many neurons depending on the objective. One neuron can be used for regression or multiple for categorisation and representation. For example to output a vector, multiple neurons represents the different values of the vector.

3.3 Word embedding

Word embedding is a Natural Language Processing (NLP) technique that is used to model the information contained in a text. It creates a multidimensional vector space where each vector represents a word from the training text. It is developed by Mikolov et al. in [24]. It models information such as similarity of words and their usage in the training corpus, so that, the distance between two vectors indicated how related those words are. The distance between two vectors in a multidimensional vector space is computed with the cosine similarity. The result of the cosine similarity varies between -1 and 1. A cosine similarity close to 1 means that the two vectors are similar and that the two words they represent are used in similar contexts. A value close to 0 means the two vectors are independent. And a value close to -1 means the two vector are opposed.

The model used to create the embedding is a log-linear model with a single hidden layer and using one-hot encoding as input to represent the word as a vector. On figure 3.2, each vector X_C represents a word. Each of them contains V values where V is the size of the vocabulary used to train the model. A single of those value is equal to 1 and all others are equal to 0. When a matrix multiplication is performed between a single vector (representing a single word) that has been transposed and the weight matrix W such as

$$h = x^T W, \tag{3.5}$$

it results in the vector h representing the word in the embedding. The weight matrix W is trained using stochastic gradient descent and back propagation.

Two variants of this technique exist:

Continuous Bag-of-Words (CBOW) In this variant, a missing central word is being predicted based on the surrounding words called the context. Such as illustrated in figure 3.2, the weight matrix W is the same for all words, the prediction is thus not influenced by word order.

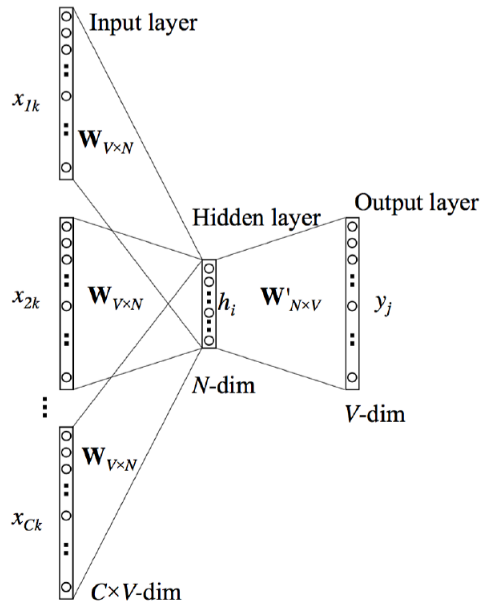


Figure 3.2: Shape of the cbow model. [23]

Continuous Skip-gram model In this variant, the model receives a single word as input and produces multiple words representing the context of the input words, that is, the surrounding words. Here, in a similar way to CBOW, the matrix W' is the same for all output vectors. Figure 3.3 show a representation of the skip-gram model.

The main drawback of this method, whatever variant is used, is that it ignores any input word that is not present in the vocabulary. Indeed, as each word from the vocabulary is associated with a vector, any word not present in the vocabulary is not associated to any vector and thus unusable. Moreover, the model always yields words from the vocabulary as it is unable to create neologisms. Another drawback of this method is that it has a $\mathcal{O}(|V|^2)$ complexity if classical softmax is used. Which is very bad as its quality will also depend on the number of words in the vocabulary it is trained on.

3.4 Document embedding

Document embedding, also named Doc2Vec or paragraph vector, is a variant of word embedding where a vector represents any piece of text ranging from a sentence to a full document. It is introduced by Mikolov et al. in [17] and works similarly to word embedding. The similarity between two documents is represented by the distance between the vectors that represent them and it can be processed with the cosine similarity. It can be used to infer documents that could be useful to suggest. Such as word embedding, it has the advantage

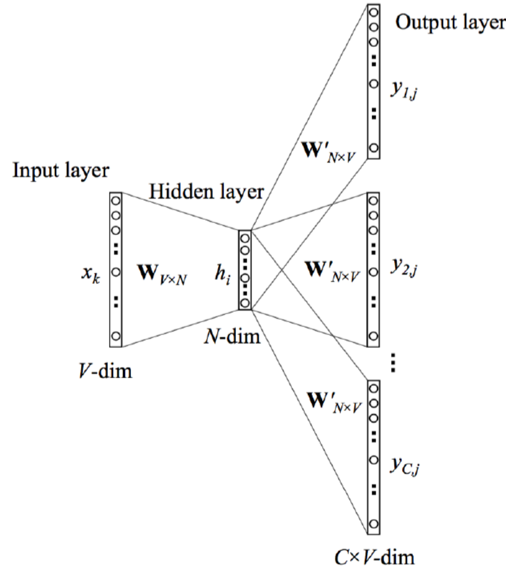


Figure 3.3: Shape of the skipgram model. [23]

that data do not need to be labelled. During the training of the document embedding, a embedding of word is created and depending of the variant of the algorithm chosen, the word vectors can be trained simultaneously with the document vectors and may improve the embedding quality. Document embedding is a log-linear model trained using stochastic gradient descent and back propagation such as word embedding. But in addition to word embedding, it uses a vector representing a paragraph as additional input.

Two variants of document embedding exist and are based on the one existing in word embedding:

Paragraph Vector: Distributed memory (PV-DM) This variant behaves similarly to CBOW from word embedding and a paragraph vector is concatenated with the word vectors as the input. It is considered to be similar to a word vector.

Paragraph Vector: Distributed bag of words (PV-DBOW) This variant predicts words from a small window. In this variant, the only input is the vector representing a document. Therefore, as opposed to PV_DM, there is no need to train the word vector.

As this work focuses on diagrams and not on text, document embedding seems to be more appropriate to represent those. Thus the analogy between a document and a diagram is made. Thanks to that representation, each diagram

is considered as independent from the other which would not have been possible with word embedding.

3.5 Visualization

t-SNE (T-distributed Stochastic Neighbor Embedding): t-SNE is a unsupervised machine learning algorithm used for dimensionality reduction. It allows the reduction of high dimensional vector space into vector space of 2 dimensions or higher, making them much easier to visualize. It is particularly useful with embeddings as they often have high dimensions. In the case of this work, the embeddings are composed of several hundred of dimensions which makes dimensionality reduction a necessary step before visualization. t-SNE does not preserve distances nor density, it only preserves the nearest neighbours which is sufficient for the application used in this work.

In this chapter the concept of embedding along with the concept of neural network are introduced. Then two techniques to apply embedding on words and documents are presented. The description of word embedding is important to understand how document embedding works. Moreover, the implementation details of the two versions of document embedding will be useful to choose the most relevant to be applied on diagrams. Lastly, dimensionality reduction algorithm is introduced to allow the visualization of the embedding.

Chapter 4

Proposed approach

This chapter explains how the model is used to suggest tokens and how class names are composed from those tokens using a full text index. The choice between different libraries implementing document embedding, the different hyper-parameters used and their selection are then explained. Finally, full text indexing is introduced.

In order to suggest a concept in the form of a class name to the user, multiple steps have to be taken. For that, the context need to be take into account. The context consist of all concepts already modelled by the user. It is the partial diagram the user has created. So, the first step is to format the partial diagram so it can be used by the model. The partial diagram needs to be represented in a document and preprocessed, both for training and for the suggestion generation. This representation is detailed later in Chapter 5.2. As the class names need to be pre-processed the same way it is done for the training set, each class name has to be tokenized, each resulting token is then lemmatized and the stop words are removed. This process is explained in Chapter 5.3. Once the context is given to the model, the model will infer multiple diagrams that are often used in similar contexts than the one it is given. Each diagram contains many tokens, possibly too much for all of them to be used. Those tokens need to be ordered based on their importance in a decreasing order and only take the first n are used. Different methods are evaluated later in this work to find. Those tokens cannot be consider as context as the model is only trained on token. So, a full-text search is performed on the index created with the unprocessed training set. This step results in a list of concepts that can be suggested to the user. The full-text search is an efficient way to search through a large corpus of text and it results in a list of concepts that are ordered, tarting with the one that better match the tokens suggested by the model. This list has to be trimmed down as it has been proven that a suggestion list that is too long will negatively affect the position of the first correct suggested element. Thus making it longer for the user to find a useful suggestion [12]. Figure 4.2 illustrate the different steps composing the recommendation process.

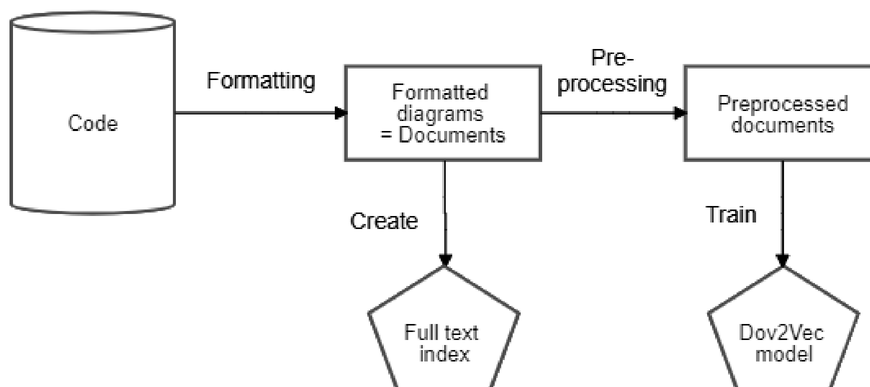


Figure 4.1: Diagram showing the different steps required to train the embedding and the full text index.

4.1 Training and library choice

This section presents the creation of the embedding and of the full text index.

The first step towards the creation of an embedding is is to choose which library to use. Many open source libraries implementing document embedding are available online, such as: Gensim, developed by RaRe-Technologies, doc2vec, by Github user hiyijian or also PVDM, by Github user JonathanRaiman. Gensim is used for many reasons: It supports multithreading; it exist since more than 9 years and have had many stable release; it has an active community, which is very helpful when facing problems; it is well documented; and it easy to use.

Then, the model has to be configured and the meta parameters selected. Indeed, in order for the model to give the best results, the best value for meta-parameters were chosen using grid search. Grid search is a hyperparameters optimization technique that consist in trying combinations of hyperparameters from a grid to find the ones that produce an optimal result. Here are the different meta-parameters for which different values were tested:

- **dm**: is used to select which variant of the algorithm have to be used, PV_DM or PV_DBOW.
- **vector_size**: allows to choose how many dimensions will the vector space have.
- **window**: which is the size of the window used to train the model. The window correspond to the number of neighbours of a word that is used to update the vector value.
- **min_count**: specifies the minimum number of occurrence a word should have in the training set to be taken into account during training.

- **epochs**: represents how many times the training iterates over the training set.
- **hs**: allows to choose between hierarchical softmax and negative sampling. Those are alternatives to objective function that reduce the model time complexity.
- **negative**: is the number of negative sampling to be used, only if negative sampling is used.
- **dbow_words**: Force the training of word-vector simultaneously with document-vector when PV_DBOW is used. PV_DM trains it by default.
- **Other**: other hyperparameters can be tweaked but the default values were used.

To find the best metaparameters for this project, the model is trained and evaluated multiple times with different values. This allows to find which values work the best to work with class diagrams. This process emphasizes that using PV_DBOW together with a word vector give better results than using PV_DM. And that, the embedding better models the data when ”vector_size” is set to 400 dimensions. As explained later in Chapter 5.2 two different representations of the data are used. Therefore, the size of the document vary depending on representation. One has longer document where the other has very short documents. Models for both of them are trained with a windows size of 20. Which gave the best result.Though, the representation with the shorter documents contains less than 20 tokens. Nevertheless, the windows size is kept high to take into account all words. Another consequence of this choice is that variation in windows size does not impact the results. The value for min_count is fixed at 3 as it does not seems to have much of an impact on the quality when set to a small number. The model is trained over 30 iterations and uses Hierarchical softmax over negative sampling as it gives better results.

4.2 Full text indexing

As explained before, when the context is given to the trained model, this one will yield a list of similar documents. Each of these documents contains a list of tokenized class names. Even if those could be useful to the user, this work aims at suggesting a list of class names. It is thus required to compose class names based on the subtokens present in the suggested documents. For that reason, full text indexing is used to find class names from the training set that contain the suggested subtokens. Full text indexing is a technology used to search efficiently through a text database. It allows to find a text containing multiples specified words. Whoosh is used as a library implementing this technology. This library is written in python witch eases the implementation with genism as it is also written in python. In order to create the index, each class name is indexed individually after being preprocessed. This allow the class name to be retrieved based on each token from the suggested diagram that is present in it. And for all retrieved class name a score is assigned so that they can be ordered in the suggestion list. The score is computed based on the BM25F algorithm (Okapi BM25).

This chapter explains how the model is used, What information it needs to be given and what information it yields. Then all the steps necessary to transform the model output into concepts are explained. After that, the library implementing document embedding that is used is introduced and its hyper parameter explained. Finally, full text indexing is described. It is used to generate concepts based on the tokens suggested by the embedding.

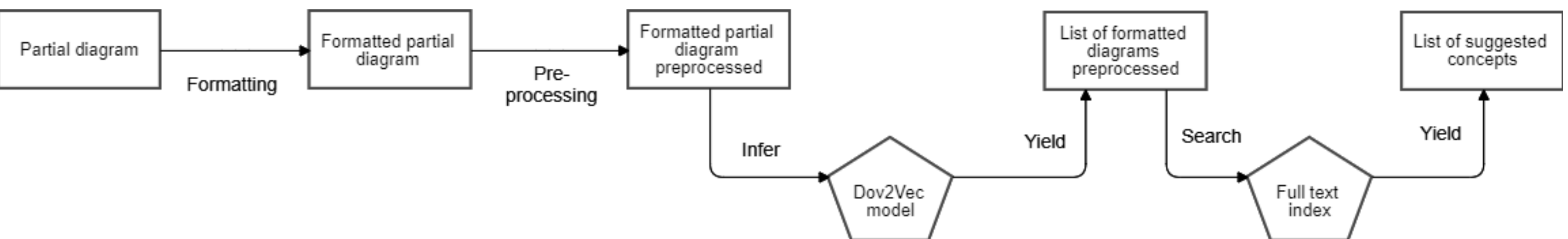


Figure 4.2: Diagram showing the steps used to generate a recommendation.

Chapter 5

Datasets and preprocessing

In this chapter different sources of data are presented and evaluated with regard to the requirements of the model. Then, two ways of representing a diagram into a format that can be used by the model are proposed. Lastly all the processes used to extract and clean the features from the dataset are described.

5.1 Choice of datasets

Few datasets containing class diagrams are documented. Three of them are "ReMoDD" [10], "IMG2UML" [13] and "The Lindholmen Dataset" [11]. But, to train the model, data coming from code can also be used, as code has many similarities with class diagrams, and as a lot of code is open source and can easily be accessed. The "Learning from Big Code" project makes available many code datasets in different languages such as JavaScript, Python and Java. For this work, the "Java GitHub corpus" [2] is evaluated along with ReMoDD, IMG2UML and the Lindholmen Dataset.

ReMoDD is a repository of resources to support the education and research in the field of model driven development. It contains many different kinds of information such as case studies, examples of good and bad practices, descriptions of modeling techniques, generic models and so forth.

IMG2UML is a repository of UML diagrams. It is populated with UML diagrams found online where most of the diagrams are generated from images. Similarly to ReMoDD it is intended to be used in education, research and in the industry. This repository is offered as a database which has the advantage that it can be searchable easily. Both ReMoDD and IMG2UML are not available online anymore.

The Lindholmen Dataset in contrast can be downloaded easily. It is a MySQL database containing information about diagrams and their origin. The diagrams it contains are gathered on more than 12 million GitHub repository and are converted from different file formats such as .xmi, .uml and images. The database structure is shown in the appendix 8.

The Java GitHub corpus is not a corpus of UML diagrams but a corpus of Java code. It is a repository of more than 14.000 Java projects gathered on github. It has the advantage that those projects have been filtered to guarantee a certain quality. Plus, this repository can easily be downloaded online.

All the UML diagram datasets are quite small and document embedding require quite a lot of data during training in order to give significant results. Therefore the Java GitHub corpus is more interesting for this project as it contains more than 2.000.000 Java files in more than 14.000 projects. The Lindholmen Dataset actually contains more projects, but after more in depth analysis, few can be used due to a bad quality and very small project size. This quality problem does not occur with the "Java GitHub corpus" as to create this dataset, only projects with a good quality were collected. To do that, Allamanis and Sutton filtered all project by they number of forks and only projects forked at least once were kept. Allamanis and Sutton explains that this characteristic is a good indicator for the projects overall quality. Then approximately one thousand projects were removed because they were present multiple times in the dataset. Allamanis and Sutton manually compared project sharing the same commit SHAs to detect and delete the duplocates. The Java GitHub corpus is thus chosen to be used for this work.

5.2 Representing a diagram as a document

As Java code contains much more information than a class diagram, this code had to be preprocessed to extract the same features from code than the features that would have been extracted from class diagrams. As explained before, document embedding trains an embedding using data shaped as a document, a document being a string of words. Gensim, the library used to train the embedding, requires to be given a file of any size where each line is considered as a document. Two ideas are explored, as what a document should be composed of:

A list of concepts: The underlying idea is to have an embedding capable of suggesting diagrams similar to the one being drawn by the user. It is then possible to suggest classes from those diagrams. Therefore, for the model training, only class names were extracted from each project. Those class names are stored in a single file where each line is called a document and contains all class names from a same diagram. Each class name is separated from the others with a single space. This is the format required by the gensim library to train the model that is going to be used to generate the suggestion. A basic example of this representation is shown in figure 5.1

A relation: The embedding can suggest relations based on the classes already drawn by the user. This idea could be applied to only part of a diagram. The idea is that a relation looks more like a sequence and thus should work well with document embedding. Indeed, a drawback of the previous idea is that the structure of the diagram is not represented in the document. Figure 5.2 shows an basic example of this representation.

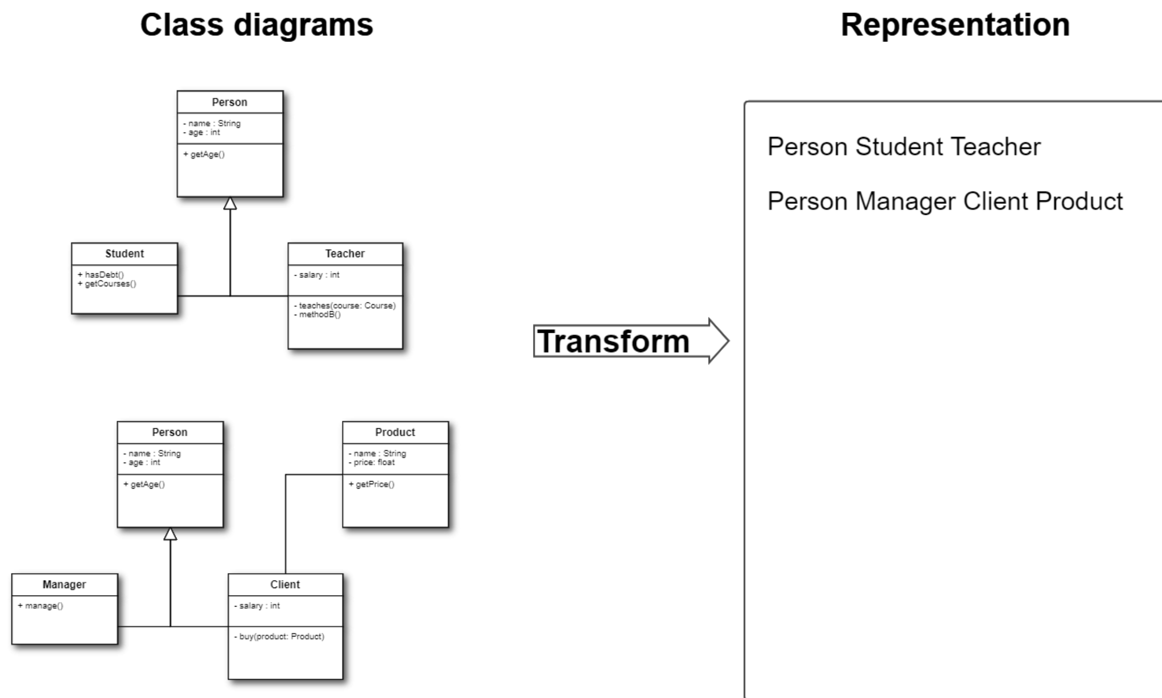
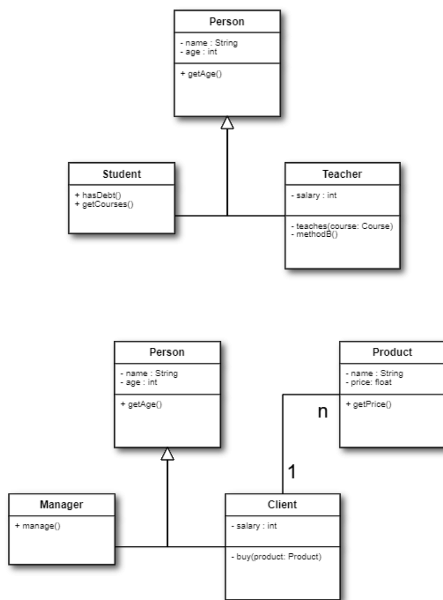


Figure 5.1: Basic example of the representation of a diagram where all class names from one diagram are listed with no particular order. The resulting String is considered as document.

Class diagrams **Representation**



Transform

Student extends Person
Teacher extends Person
Client reln Product
Manager extends Person
Client extends Person

Figure 5.2: Basic example of the representation of a diagram where each relation is considered as a document.

For both ways to represent the projects as documents, the abstract syntax tree (AST) is generated. The AST is built for each file and is used to retrieve the class names and the relations between the classes. This method is chosen as it can easily retrieve different information from the code which makes it easier to try different representations. As only java projects are used to train the embedding, the "JavaParser" library is used.

For the evaluation, another representation of the projects is needed. A user creates a diagram in a certain manner, it is interesting to evaluate the proposed approach similarly to how the user creates that diagram. The user usually constructs a sub-graph and expand it by adding neighbouring classes. In order to simulate the user behavior, such a sub-graph is constructed. Each diagram is thus represented as a graph in the DOT language, a popular language used to store graphs. As the proposed approach does not use other information than the classes name and their relations, it is possible to simplify the diagrams into graphs. The DOT language is chosen as files in this format can easily be generated and as a lot of libraries can work with it such as "networkX" used in Python. The only purpose of this representation is to correctly evaluate the proposed approach and is not used to generate documents.

5.3 Preprocessing

Class names can represent quite complex concepts as they are composed of multiple words, each bringing a new meaning to the class name and influencing the global concept. Therefore it is useful to split all class names into its composing words that are also called sub-tokens. This process is called **tokenization**. As the model is trained on those sub-tokens, suggestions can be made where only a part of the initial class name is related to the suggestions, which allows the system to not only suggest class names that are directly related, but also those that are composed of related sub-tokens.

In the English language, a same word can have plenty of endings. It is therefore important to find the root word that constitute each word from the dataset as it could affect the quality of the suggestions. For example, "split" and "splitted" could be considered two different concepts even though they are very closely related. This process of finding the root of a word is called **lemmatization** and the root of a word is called its lemma. Without lemmatization, the model could suggest both "split" and "splitted" as two different concepts which is a problem as only one real concept is suggested, preventing another maybe important concept to be proposed.

As explained earlier, class names are composed of many words/sub-tokens. Some of those words does not contain any useful information for the model such as **stop words** like "at", "by", "in", ... And, during manual evaluation it appears that class names containing stop words could not be correctly evaluated with quantitative evaluation. Class names such as *RotationAtModifier* and *RotationByModifier* are not detected as equal if stop words are not ignored. They are thus removed from the dataset and ignored during evaluation. An example

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

Table 5.1: Effect of the different preprocessing techniques on an example class name.

of the different pre-processing steps is given in table 5.1.

In this chapter, different dataset are introduced and the selection of one of them has been made. The dataset is used to train the model and has been chosen as it contains more data and is of better quality. Then, as the code cannot directly be used by the model, two different paths are explored to represent a project. Finally, different steps used to pre-process the data are explained. Those are important for the data to be used by the model in an efficient way. They are tokenization, lemmatization and stop words removal.

Chapter 6

Validation of the initial assumption

This Chapter evaluates whether the embedding successfully models patterns present in the code. To achieve this, t-SNE is applied to the embedding and a sample of documents are visualized.

RQ0: Can patterns be extracted from code into an embedding ? The intuition this research bases itself on is that it is possible to abstract a common knowledge about different features of class diagrams. This assumption is considered as the Research question number 0 (RQ0). And to see if it can be validated, it is interesting to see if some concepts from the projects form groups inside the embedding. Indeed, similar concepts from many different projects should share similar feature and therefore be closer to each other in the embedding. One way to do that is by visualizing the embedding, but as the concepts are represented by vectors in a high dimensional vector space it cannot be visualize directly. The first step is thus to reduce the number of dimension and t-SNE can be used for that purpose. Figure 6.1 displays a sample of 200 randomly selected documents from different projects. Each document represents a relation between two classes. From this visualization, two groups are selected and the class names they contain are retrieved. Then the class names from a same group are analysed to see if a general topic emerges from the group. On Figure 6.1, the top right one contains the following relations:

- AMQPNativeOutboundTransformer extend OutboundTransformer
- ActiveMQTempDestination → ActiveMQConnection
- DestinationMap → DestinationMapNode
- DestinationBaseMessageList → MessageBuff
- TempDestLoadTest → Session
- TempDestLoadTest → TempSession
- JMSTopicSelectorTest → Destination

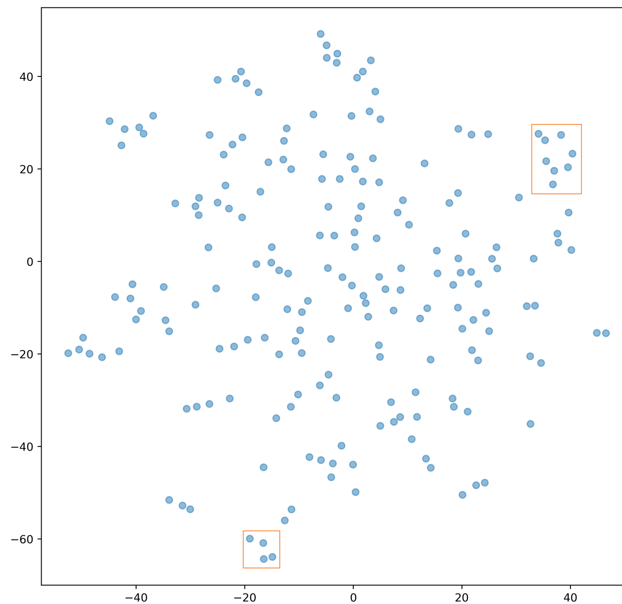


Figure 6.1: Visualization of the trained embedding using the encoding of relations, where t-SNE is used first to reduce the number of dimensions to 2.

(The class names have been recomposed based on their subtokens as the documents are preprocessed.)

The class names contained in this group seem to refer to communication. Indeed, class names such as "destination", "connection", "message", "session" and "topic" are all related to communication. Another group, the bottom one contains:

- AbstractGeneratorGest → VirtualMachineGenerator
- VirtualImageConversion → VirtualMachineTemplate
- NodeVirtualImageDAOTest → VirtualMachineTemplate
- VirtualBoxCollector → AIMCollector

Class names such as "virtual", "virtual box", "virtual image" and "virtual machine" all refers to virtual machines. Both groups contain class names referring to a main topic. This shows that the documents are placed in the embedding based on a certain logic. It thus makes it possible to use the similarity between those documents to suggest concepts to the user.

In this Chapter the embedding is evaluated to see if it successfully models patterns present in the code. The sample of documents visualized shows that the neighbours document share a similar topic. This proves that the embedding is able to extract and model knowledge from the dataset of code.

Chapter 7

Evaluation

This chapter evaluates the different parts of the approach. First the experimental settings are described and are used to explain different choices made during the evaluation. This section describe the followed methodology and the configuration used to evaluate the proposed approach. Second, the data gather about the suggestion quality are presented in graphs to help answer the three research questions. For the second one, three sub-questions are opened to be able to answer it fully. Finally, the three research questions opened earlier in this work are answered. The qualities and drawbacks of the approach are discussed and then different improvement suggestions are given.

7.1 Experimental settings

7.1.1 Evaluation process

As explained in Chapter 4, the partial diagram is first transformed into one or multiple documents depending on the representation. Then comes the preprocessing step which cleans the class names from their stop words, tokenizes and lemmatizes them. The resulting documents are a representation of the partial diagram drawn by the user. Those documents are given to the model which yields a list of similar documents. From those documents are extracted many tokens which are used to retrieve the most related class names from the full text index. But first, before suggesting class names, only the best tokens need to be selected as the number of token can be too high to select useful class names. So, the list tokens is ordered and trimmed down using different methods.

In order to evaluate this system, it is possible to use some projects from the same dataset used for training. Therefore, those projects were excluded from the dataset during the training and are selected using the ranking proposed by Allamanis and Sutton on the web page of "Java GitHub corpus" [2]. This ranking orders the different projects according to their popularity on GitHub and it can be supposed that the most popular projects have a better quality. To evaluate the approach as correctly as possible, the behavior of the user was mimicked, which means that the project is split in two part. The first one is

considered as the part of the diagram that is already drawn by the user and thus is given to the model. The second part of the project is considered as the part of the diagram that has not yet been drawn by the user and that needs to be suggested. The first part of the project are called "the context" in the rest of this work and contains the class names that have already been added to the diagram. The whole evaluation process is depicted in figure 7.1.

In order to split a project in two parts, a connected sub-graph is extracted from each project used during the evaluation. This sub-graph is considered as the part of the diagram already drawn by the user. And all classes from the diagram that have not been selected in this sub-graph are part of the diagram not yet drawn by the user. Those classes will be called "missing classes" in the rest of this work, and the tokens of those class names will be called "missing tokens". To select a sub-graph, the behaviour of the user during the design of a class diagram is simulate. An initial node is randomly selected and considered as the sub-graph. Then another node is randomly selected amongst all direct neighbours of the sub-graph to be added to the sub-graph. This last operation is repeated until the desired size is obtained. Selecting the classes in this manner, ensures that relations will be present in case the representation of a diagram uses a relation as a document.

As explained in Chapter 4, the class names are not directly suggested by the document embedding model. Instead, tokens are suggested and used to retrieve class names using full text indexing. Thus to evaluate the system more completely, it is evaluated at two stages, it is firstly evaluated on the token suggestion and then on the concept suggestion. This gives an insight in the different part of the system and shows what part perform well and what part performs more poorly.

7.1.2 Formatting and preprocessing

Formatting differs between training and evaluation. During training, all the projects can be formatted directly, whatever formatting method is used. But, for the evaluation, more steps are required as all project need to be split in two part, in a specific manner. Each project need to be represented as a graph in order to be able to extract a sub-graph and to use it as the already drawn diagram. Therefore, during evaluation, each diagram is represented as a graph in the DOT language, a language used to describe graphs. As the proposed approach does not use other information than the classes name and their relations, it is possible to simplify the diagrams into graphs. The DOT language is chosen because files in this format can easily be generated and because a lot of libraries can work with it, such as "networkX", written in Python. The next step is pre-processing where all class names contained are pre-processed directly in the graphs. The pre-processing applied is the same as the one applied during training explained in Chapter 5.3. This means that all class names are tokenized and, therefore, that the model is trained on tokens.

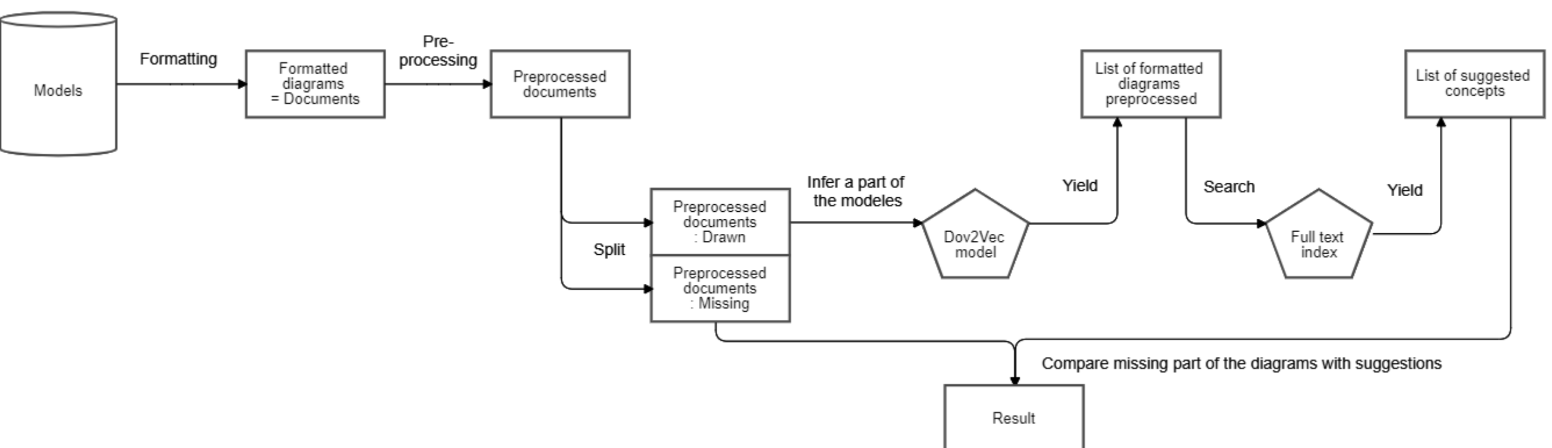


Figure 7.1: Diagram presenting the evaluation process.

7.1.3 Evaluation metrics

The evaluation consists in assessing the quality of the suggested tokens and concepts. A quantitative evaluation is applied to the suggested tokens. To this end, an automatic evaluation is conducted on 30 projects and is repeated 10 times for all projects, every time with a different randomly selected context. By using the average (arithmetic mean) value for every metric used, a representative value of the system quality can be computed. Moreover, when evaluating the suggestions on different context sizes, a growing context is simulated, once again to mimic the behavior of the user. The same idea is used when trimming the tokens list.

In order to evaluate different aspect of the system three metrics are used:

Precision at K : Evaluation metric for recommendation systems used in order to calculate the precision at cutoff k. It is calculated by counting the number of relevant items considering only the top k elements from the recommended items, divided by k so that

$$P@k = \frac{|True\ positive\ at\ k|}{k}. \tag{7.1}$$

K being the number of suggestions to evaluate and arbitrarily selected. This method is useful to evaluate recommendation systems. Indeed, even if many suggestions could be proposed to the user, only the K firsts can effectively be displayed in order not to affect the usability of the system. Thus, the evaluation is only done on the K firsts suggested elements.

Relevance : Evaluation metric for recommendation systems estimating the relevance of a list of suggestions. It calculates whether at least one suggestion is useful or none of them are. It value is either 1 or 0. If the precision at K has already been computed, the relevance R can be compute as

$$R = \begin{cases} 1, & \text{if } P@k > 0 \\ 0, & \text{otherwise} \end{cases}. \tag{7.2}$$

As opposed to Precision at K, it permits to which fraction of the list of suggestions are useful or not. And so, how often the suggestions can be used.

Rank : The rank is the position of the first correct element in the suggestions. As the user will read the suggestion from top to bottom, it is important that the suggestions are ranked to display the most relevant one first. Therefore the rank of the first correct element is a good indicator of the ranking system used to order the suggestions.

7.2 Results

7.2.1 RQ1: Can document embedding be applied to structure diagram?

Document embedding can be apply to a broad range of subjects from text to DNA. The most important part in applying document embedding to a subject like class diagram is the representation of class diagram into a document. Class diagrams are complex objects that contain many information, and they need to be transformed into one or many documents. A document being a single line of text, it is thus important to find a representation that contains the most relevant information to help in achieving the final objective. In this approach, the objective of the model is to suggest concepts neighbouring a given partial diagram. To that end, the structure of the diagram and its relations may be important. Two representation are explored in this approach:

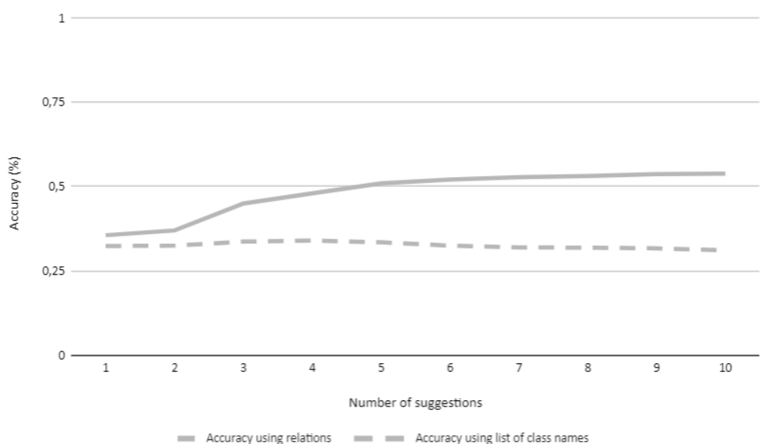
- List of class names embedding: Where each diagram is represented as a document.
- Sequence of relation embedding: Where a document represents a relation between two classes.

Those two representation are explained in much more details in Chapter 5.2. Figure 7.2 presents the quality of the suggested tokens when the model is trained with the different approaches. The graph show how the quality vary depending on the number of tokens suggested.

7.2.2 RQ2: How does document embedding performs when used to generate suggestions?

This research question can be answered with the help of three sub-questions as three decisions impact the quality. First, the quality will vary depending on what is being suggested. Trying to suggest concepts only from the neighbouring classes restricts the chance to guess correctly. Second, the tokens present in the suggested documents are randomly ordered and need to be ordered. A good ordering method will impact the quality of the suggested tokens as only the best tokens are suggested. Third, the number of those tokens to be suggested also has an impact on the quality, therefore, the suggestion list need to be trimmed if it is too long.

Suggesting all missing classes or only neighbours: To correctly evaluate the quality of the suggested tokens, it is evaluated in a similar way than it is used. It is supposed that the user starts by creating a primary class and gradually add neighbouring classes to create a connected graph. Nevertheless, it narrows the possible classes to suggest. And as the goal is to guide the user towards a solution he/she might not have thought about, suggesting class names that are not directly related to that connected sub-graph can also be helpful. Thus both methods are used to evaluate the suggestion. Figure 7.3 presents the impact of the objective of suggestions on their quality. The fact that the number of class names that must be guessed is narrower makes it more difficult to suggest



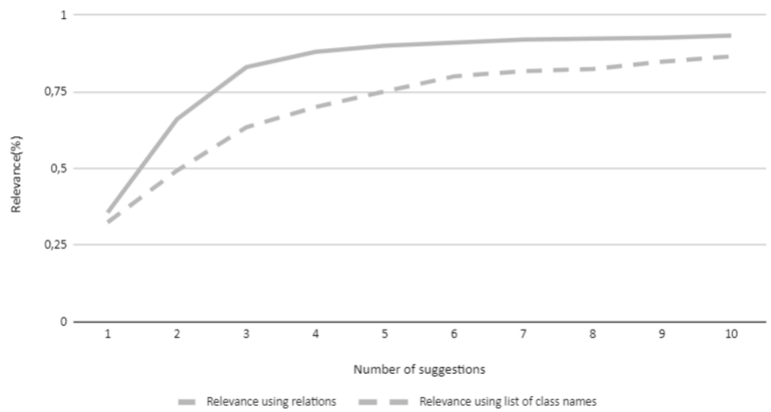


Figure 7.2: Evaluation of the suggestions quality comparing both diagram representations.

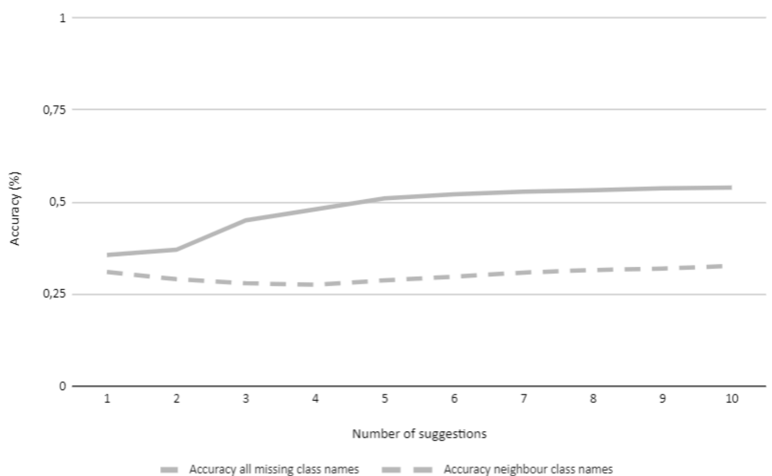
something correct. For the rest of the evaluation, the elements are suggested from the whole missing part of the diagram and not only from the neighbouring elements.

Ordering tokens by number of occurrences or with TF-IDF : As the model is trained with tokenized class names, it suggests documents containing tokens. But before those tokens can be used to form class names, their number need to be trimmed down. As the more token there are, the more class names there are. The list of suggestions presented to the user needs to not be too long to still be usable. But before to trim down the list of suggestions, it needs to be ordered to keep a maximum of good suggestions. Two metric are used to order the tokens: their number of occurrences and TF-IDF. TF-IDF is an acronym for *term frequency-inverse document frequency* and calculate the relevance of a word in a document using its frequency in that document and its frequency in a corpus of documents. Figure 7.4 presents that accuracy of the suggestions for different lengths of list of suggestions. It shows the quality of the ordering. As shown on the graph, with TF-IDF, the accuracy does not vary when the size of the list of suggestions increase. Which means that the good suggestions are distributed over the list of suggestions. Figure 7.5 presents the accuracy and relevance of each token individually. When ordered by number of occurrence, the accuracy of the first and second token gives poorer results than the rest of the tokens. This impact negatively the suggestion list as the last tokens might be excluded from the list even though they are more accurate. Those results show that even though those tokens are more present in the suggestion, they are not useful user. TF-IDF thus offers a improvement on the suggestion list for small number of tokens. During the rest of the evaluation process, a list of suggestions order with the number of occurrences is used as they have similar quality once more tokens are suggested.

Different methods to choose the number of tokens to keep : When trimming the suggestions list down, one must know how long to keep it. Different methods are used:

- Threshold on the number of elements: Only the top n elements are kept. This method was used for the previous evaluation. Figure 7.6 presents how this method behave with different threshold values.
- Threshold on the number of occurrences: Only tokens suggested more than n times are kept. This method behavior is presented in Figure 7.7.
- Threshold on the frequency of occurrences: Only the top n percent of the tokens are kept. Figure 7.8 shows the results of using this method.

In order to evaluate how the three methods behave, the accuracy is computed using all three methods while the context size varies. For each method a few values are evaluated to see how it impacts the results. The first method, "a fix threshold on the number of elements" is used for all evaluations.



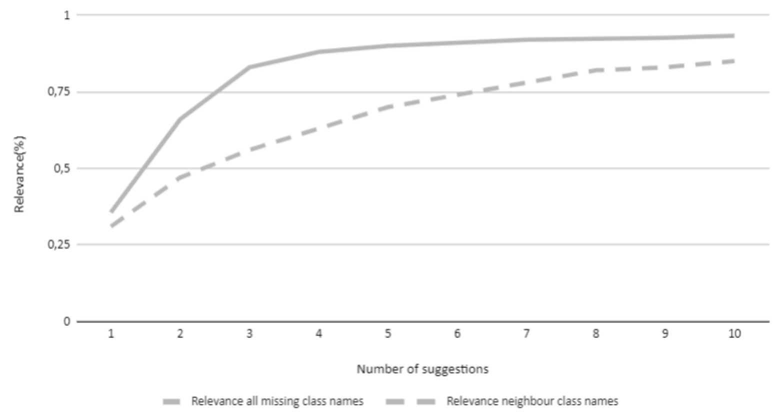
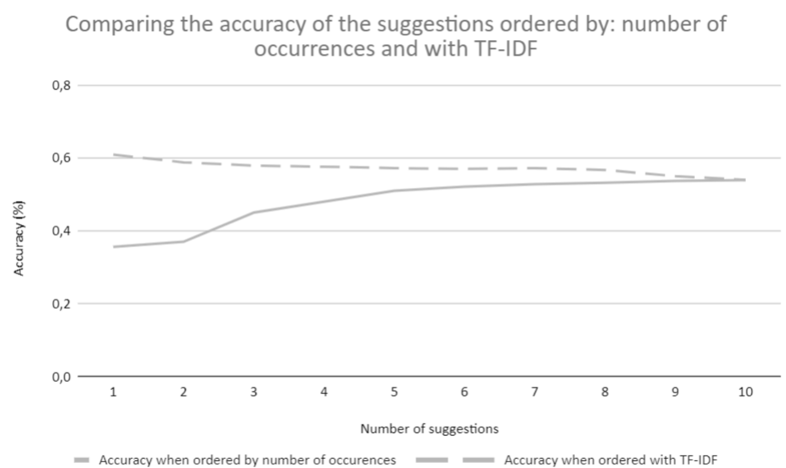


Figure 7.3: Evaluation of the suggestions quality comparing both objective of suggestion.



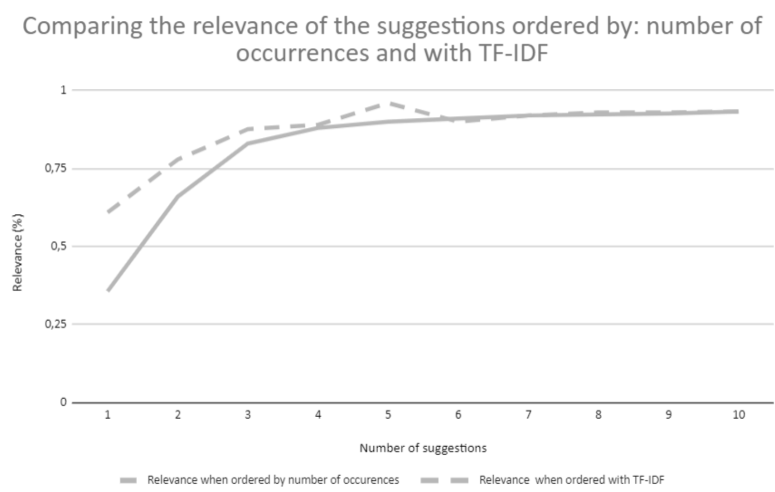


Figure 7.4: Evaluation of the suggestions quality comparing different ordering of suggestion.

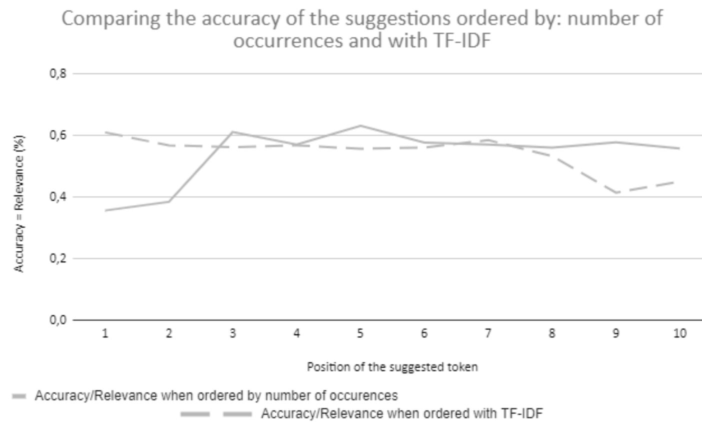


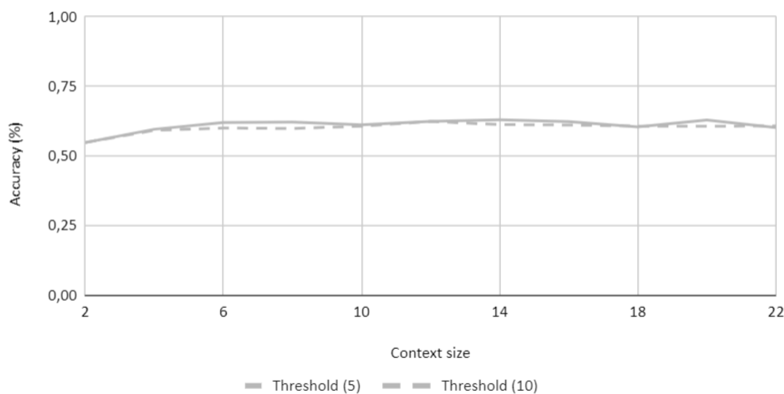
Figure 7.5: Evaluation of the suggestions quality comparing different ordering of suggestion.

7.2.3 RQ3: How can the suggested concepts quality be evaluated?

Automatically evaluating the quality of suggested concepts is quite difficult. Indeed, it requires to be able to compare the similarity of advanced concepts. Elkamel et al. [7] developed a class comparison system in order to evaluate the similarity between two classes. Part of that work could be reused to compare concepts. Though the concepts are manually evaluated as implementing that solution would be time consuming. In order to evaluate the quality of the concept, Figure 7.9 compares the evolution of suggested tokens quality and the suggested concepts quality at different number of suggested tokens. This shows the correlation between those two steps and how tokens suggestion quality impacts concept suggestion quality.

7.3 Discussion

In order to respond to **RQ1: Can document embedding be applied to structure diagram?**, two ways of representing class diagrams as documents were evaluated. Those have shown that document embedding can indeed be applied to structure diagrams. Never the less, the representation chosen greatly affects the quality of the embedding. As shown in the figure 7.2, the two representations give quite different results proving the importance in the selection of the feature to represent. Here, the use of relations between classes impact the quality of the suggested tokens and gives better results. Compared with this approach, the approach representing the diagram as one document containing all the class names loses the relation between classes. Even worse, the proximity between unrelated class names in the document may harm the quality of the suggestions. The approach using sequence of relation embedding is used for the rest of the evaluations. So, even though document embedding can be applied



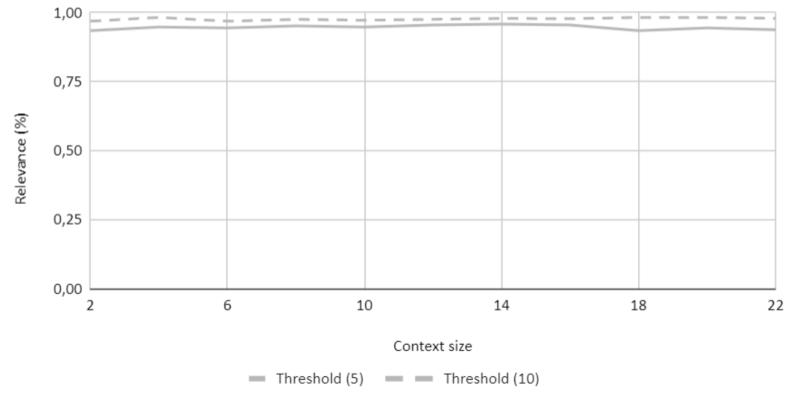
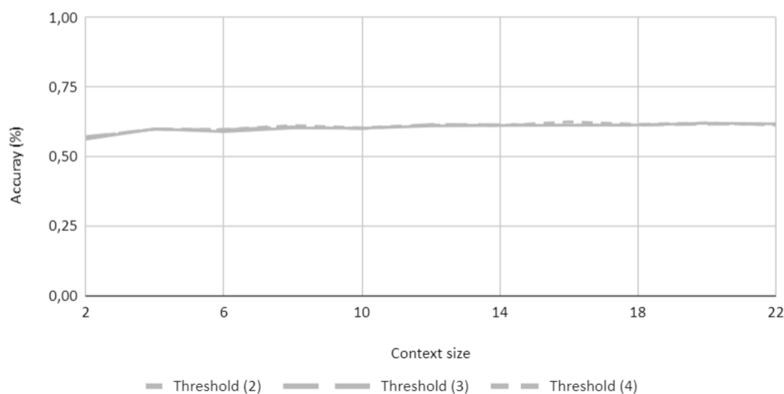


Figure 7.6: Evaluation of the suggestions quality comparing the first trimming method.



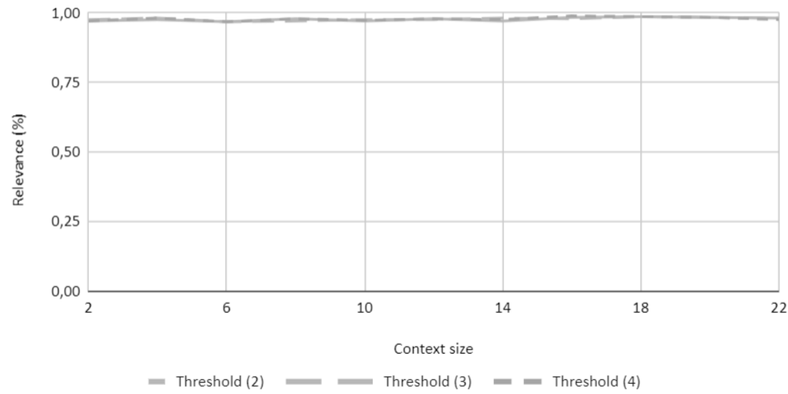
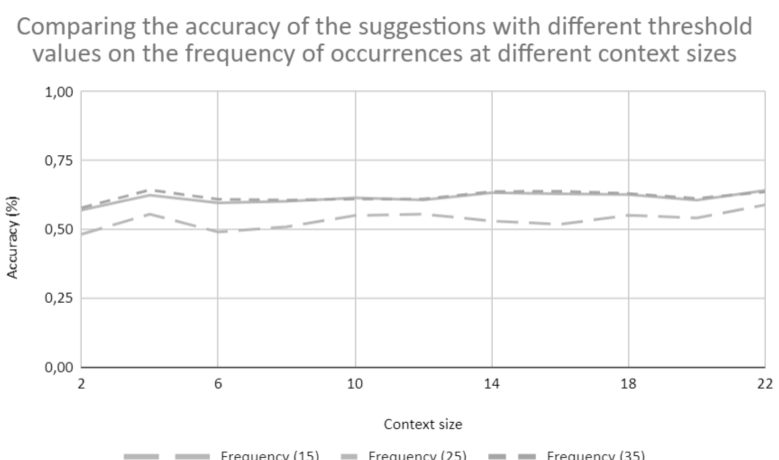


Figure 7.7: Evaluation of the suggestions quality comparing the second trimming method.



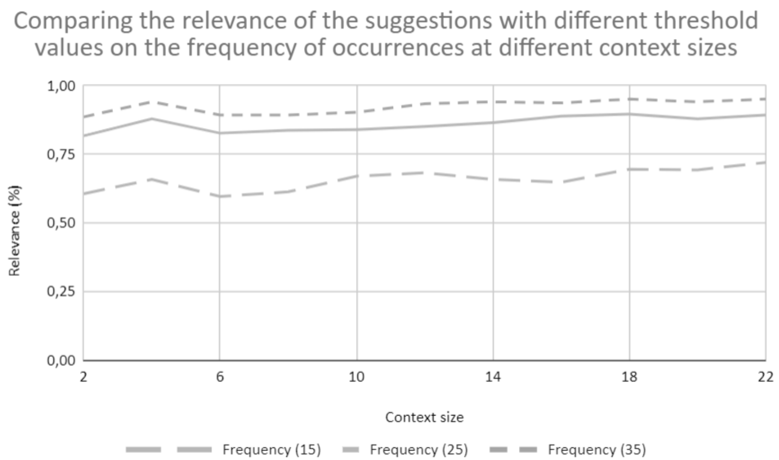
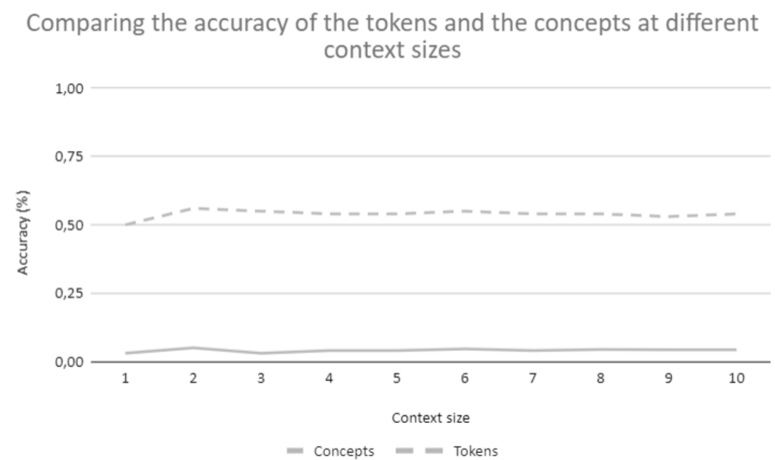


Figure 7.8: Evaluation of the suggestions quality comparing the third trimming method.



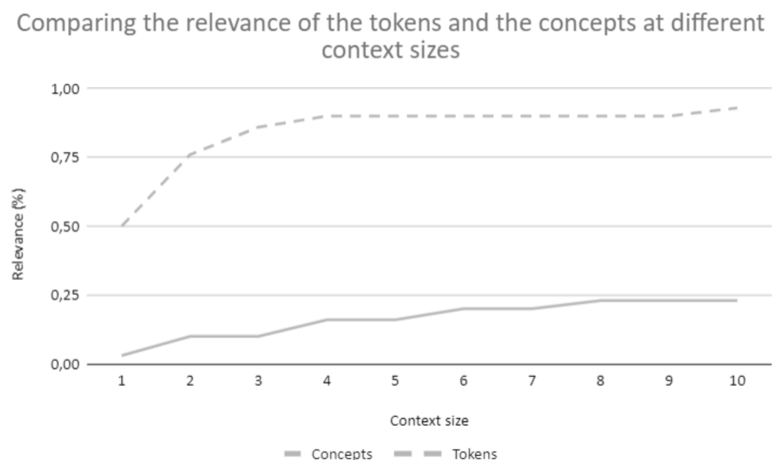


Figure 7.9: Evaluation of the suggested concepts quality comparing the suggested tokens quality.

to structure diagrams, the representation needs to be chosen carefully as it has a great impact on the results.

To answer **RQ2: How does document embedding performs when used to generate suggestions?**, four points of divergence are explored. **First**, it is observed how the objective of the suggestion impacts the accuracy. And even though the suggestions are more restricted when trying to suggest only neighbouring class names, both show promising results. The restriction applied when only evaluating with neighbours classes impact negatively the results compared to the other approach. This causes the number of class names that must be guessed to be smaller, which in turns makes it more difficult to suggest something correct. Never the less, it is an expected impact and they might be other features to train the model with, that could improve results for this use case. **Second**, as the number of tokens have to be trimmed down, the ordering of the tokens is particularly important to keep the best tokens. Two ideas are explored to order the tokens by placing the best one at the top of the list. It can be seen on Figure 7.4 that ordering the tokens using TF-IDF yield better result for small numbers of suggestions. Indeed, ordering tokens by their number of occurrence on a small number of suggestion gives bad results. This shows that the even though some tokens occur a lot in the suggestions, they do not help the user. **Third**, different methods have been used to determine the size of the suggestions list. Using a threshold on the number of elements as show in Figure 7.6, or using a threshold on the number of occurrences as show in Figure 7.7, results in similar quality of suggestion. But using a threshold on the frequency of occurrences as show in Figure 7.8, lowers the quality of the suggestion. This method is thus to avoid. **Fourth**, the size of the context does not alter the quality of the suggestion. Indeed it can be seen in figure 7.6 that the suggestion quality is not altered by the size of the context. Therefore, the quality of the suggestion does not vary during the creation of the diagram by the user. Though, depending on the representation used to train the model, at least two classes are required to produce an output as at least one relation is needed.

Document embedding can thus be used to suggest good quality tokens. The four points of divergence introduced emphases that TF-IDF should be used to order the tokens and that the trimming of the list of tokens too long should not be done based on the number of occurrences of those tokens. Also, applying restriction on the suggested tokens will negatively impact the results.

To answer **RQ3: How can the suggested concepts quality be evaluated?**, different works were explored. Elkamel et al. [7] created a method to compare classes. This method could be used, though a manual evaluation is done in this approach as the approach proposed in [7] is more complex. Figure 7.9 shows that suggesting class names is difficult even though in the tokens from the intermediary step are suggested with a good accuracy. Therefore, tokens are suggested to the user to guide him/her during the diagram creation. This solution does not allow for completion as opposed to class names suggestions but still provide assistance to the user.

This chapter assesses the quality of tokens and concepts suggested. First, the experimental settings are introduced to explain the configuration used to assess

the proposed approach. Decisions like, the choice of the testing dataset, the format used for the diagrams during evaluation, the methodology used and the metrics used are explained. Second, the approach is evaluated to help answering the three research questions. For each question different alternatives are evaluated such as the choice between "Diagram embedding" and "Sequence of relation embedding", the used of TF-IDF compared with the number of occurrences and proves to give much better results for small number of suggestions. Finally, the results are discussed and the three research questions are answered.

Chapter 8

Conclusion

The goal of this work is to provide support to the user while he/she is creating a class diagram. For this purpose, document embedding is used to guide the user using suggestions. Historical diagram data with the knowledge they contain are used to create those suggestions. To that end, a lot of data is required and existing datasets of class diagram suffer from different issues. Thus, a novel approach was proposed: the use of code as a source of knowledge to suggest concepts to the user. Features from Java code are therefore extracted and used to train the embedding. The approach described in this work demonstrates that it is possible to extract abstract knowledge that are contained in a corpus of code, in such a way that it can be used to help the user thereafter.

Works on suggestions in three fields have been studied: UML class diagram completion, code completion and graph completion. It emerged that machine learning has rarely been used for class diagrams completion. But document embedding has already been used in other areas and can be used for class suggestion in the area of class diagrams.

For computers to work with complex information, this information has to be represented in a way that can be used by computers. Embeddings are a good manner to represent information such as diagrams. They have already been used for suggestion in different domains such as code and graphs. Document embedding is used as it allows to embed many different kinds of data from DNA to graphs. Being based on word embedding, word embedding is presented in detail before presenting document embedding.

In order to suggest class names, multiple steps have to be taken. First, when the context (classes already drawn by the user) is given to the model, it infers multiple similar documents that could be suggested. Then tokens are extracted from those documents, tokens are all words composing the class names inside a document. Those tokens are ordered by importance and the best tokens are used to retrieve class names from a full-text index. An index that contains all class names the model is trained on.

Even though some datasets of UML class diagrams exists, none of them are big enough or good enough to train a document embedding model. Instead, the "Java GitHub corpus" is used, a dataset of code with more than 14.000 Java projects. Those projects have to be reverse engineered to extract information and to build a representation of their class diagram. As a class diagrams cannot directly be used as a document to train the embedding, they had to be formatted. Two formats were evaluated: the first one uses an unordered list of all class names as a document and the second one uses the relation between two classes including their name. The results obtained with the second format yields better results than the first one. Those results prove that class diagram can be modeled and used with document embedding. And even though the second approach shows good results other unexplored formats could be interesting to evaluate depending on the objective of the work. To implement the embedding, the Gensim library is used, but other libraries exist and may offer different input formats. More over the use of Gensim restricted the representation to a basic format. A class diagram is a complex representation of a system components and its relations and being able to compute the position in the embedding based on a more complete representation would be useful. For example, methods, attributes and their types could be used to bring more information in the embedding but all these information are complex to represent using only documents.

Then a pre-processing step is done in order for the system to create more relevant suggestions. It consists in tokenization, lemmatization and stop words removal.

The approach has been evaluated using 30 projects from the "Java GitHub corpus". Those 30 projects were therefore not used for training. From each project, a part of the diagram have been considered already drawn by the user and the rest of the diagram contains the missing classes that needs to be suggested. Both the tokens generated as an intermediary step and the class names are evaluated. They are evaluated using 3 metrics: precision at K, satisfaction and rank. This work clearly proves that document embedding can be applied to class diagrams even though the quality of the embedding will vary depending on the format of the diagrams. Then, the embedding will suggest similar documents from which tokens are extracted. The selection of the tokens and the number of tokens selected is important as it will directly affect the suggestion of tokens and the suggestion of class names. The obtained results demonstrate that tokens suggestion is feasible when trying to guide the used during class diagram creating. More over, it is shown that the suggestions quality is quite stable throughout the creation of the diagram as the results are not influenced by the size of the diagram. Another objective tested during this research is the suggestion of neighbouring classes only, in contrast with the suggestion of all missing classes. This objective being more complex, it yields poorer results even though it can still be used and be useful. The suggestion of tokens is much more accurate than the suggestion of class names as the latter is more complex. A drawback of the method used to suggest the class names is that it cannot create neologisms. It is possible to replace the full text indexing which retrieve class names based on the tokens with other methods such as, for example, statistical language model, but this would require more research. The suggestion

of concepts is difficult, so is their evaluation. The evaluation of the concepts has to be done manually as it exists many synonyms for the same idea and that a system evaluating the similarity between concept specific to the domain of model driven engineering does not exist.

This work acts as a first step towards UML class diagram completion. Even though entire class names cannot yet be suggested with high accuracy, the tokens that constitute those class names are relevant and can be suggested with a good accuracy. Corpus of data start appearing and start to be used in the industry and to support research and education. Though those corpus lack quantity, quality and/or ease of access. This approach proposes a novel approach by using the code of open source projects which takes advantage of the large amount of projects on Github. This method has two main advantages for this project. First, the code projects is very similar to class diagrams. And second, that the large amount of available projects allows data intensive model such as document embedding to be trained while still being able to filter in good quality projects.

Bibliography

[1] Julian Aijal. *What is a knowledge graph and how does one work?* https://thenextweb.com/podium/2019/06/11/what-is-a-knowledge-graph-and-how-does-one-work/. Accessed: 2020-04-26.

[2] Miltiadis Allamanis and Charles Sutton. "Mining Source Code Repositories at Massive Scale using Language Modeling". In: *The 10th Working Conference on Mining Software Repositories*. IEEE. 2013, pp. 207–216.

[3] Miltiadis Allamanis et al. "Suggesting accurate method and class names". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. the 2015 10th Joint Meeting. Bergamo, Italy: ACM Press, 2015, pp. 38–49. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786849. URL: http://dl.acm.org/citation.cfm?doid=2786805.2786849 (visited on 02/21/2020).

[4] Christopher M. Bishop. *Pattern recognition and machine learning*. Information science and statistics. page 1. New York: Springer, 2006. 738 pp. ISBN: 978-0-387-31073-2.

[5] *Cosine similarity*. https://en.wikipedia.org/wiki/Cosine_similarity. Accessed: 2020-02-15.

[6] Li Dongfang and Masuhara Hidehiko. "ASTToken2Vec: An Embedding Method for Neural Code Completion". In: (), p. 9.

[7] Akil Elkamel, Mariem Gzara, and Hanene Ben-Abdallah. "An UML class recommender system for software design". In: *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*. 2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA). Agadir, Morocco: IEEE, Nov. 2016, pp. 1–8. ISBN: 978-1-5090-4320-0. DOI: 10.1109/AICCSA.2016.7945659. URL: http://ieeexplore.ieee.org/document/7945659/ (visited on 06/06/2020).

[8] Akil Elkamel, Mariem Gzara, and Hanêne Ben-Abdallah. "A bio-inspired hierarchical clustering algorithm with backtracking strategy". In: *Applied Intelligence* 42.2 (Mar. 1, 2015), pp. 174–194. ISSN: 0924-669X. DOI: 10.1007/s10489-014-0573-6. URL: https://doi.org/10.1007/s10489-014-0573-6 (visited on 07/08/2020).

[9] Quentin Fily. *TF-IDF : Déterminer un score de pertinence*. https://www.quentinfily.fr/tf-idf-pertinence-lexicale/. Accessed:2020-06-25.

[10] Robert B. France et al. "Repository for Model Driven Development (ReMoDD)". In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012 34th International Conference on Software Engineering (ICSE). ISSN: 1558-1225. June 2012, pp. 1471–1472. DOI: 10.1109/ICSE.2012.6227059.

[11] Regina Hebig et al. "The Quest for Open Source Projects that use UML". In: (), p. 12.

[12] Xianhao Jin and Francisco Servant. “The hidden cost of code completion: understanding the impact of the recommendation-list length on its efficiency”. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR ’18. Gothenburg, Sweden: Association for Computing Machinery, May 28, 2018, pp. 70–73. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196474. URL: https://doi.org/10.1145/3196398.3196474 (visited on 02/19/2020).

[13] B. Karasneh and Michel Chaudron. “Online Img2UML Repository: An Online Repository for UML Models”. In: Oct. 1, 2013.

[14] Will Koehrsen. *Neural Network Embeddings Explained*. https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526. Accessed:2020-02-01.

[15] Tobias Kuschke, Patrick Mäder, and Patrick Rempel. “Recommending Auto-completions for Software Modeling Activities”. In: *Model-Driven Engineering Languages and Systems*. Ed. by Ana Moreira et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 170–186. ISBN: 978-3-642-41533-3. DOI: 10.1007/978-3-642-41533-3_11.

[16] Natasha Latysheva. *Why do we use word embeddings in NLP?* https://towardsdatascience.com/why-do-we-use-embeddings-in-nlp-2f20e1b632d2. Accessed:2020-02-01.

[17] Quoc Le and Tomas Mikolov. “Distributed Representations of Sentences and Documents”. In: (2014), p. 9.

[18] Joseph Lilleberg, Yun Zhu, and Yanqing Zhang. “Support vector machines and Word2vec for text classification with semantic features”. In: *2015 IEEE 14th International Conference on Cognitive Informatics Cognitive Computing (ICCI*CC)*. 2015 IEEE 14th International Conference on Cognitive Informatics Cognitive Computing (ICCI*CC). ISSN: null. July 2015, pp. 136–140. DOI: 10.1109/ICCI-CC.2015.7259377.

[19] Yankai Lin et al. “Learning Entity and Relation Embeddings for Knowledge Graph Completion”. In: (Jan. 25, 2015), p. 7.

[20] Patrick Mäder et al. “traceMaintainer - Automated Traceability Maintenance”. In: *2008 16th IEEE International Requirements Engineering Conference*. 2008 16th IEEE International Requirements Engineering Conference. ISSN: 2332-6441. Sept. 2008, pp. 329–330. DOI: 10.1109/RE.2008.25.

[21] Ilia Markov et al. “Author Profiling with doc2vec Neural Network-Based Document Embeddings Preprint version”. In: (2017), p. 16.

[22] Steffen Mazanek, Sonja Maier, and Mark Minas. “Auto-completion for diagram editors based on graph grammars”. In: *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. 2008 IEEE Symposium on Visual Languages and Human-Centric Computing. ISSN: 1943-6106. Sept. 2008, pp. 242–245. DOI: 10.1109/VLHCC.2008.4639094.

[23] David Meyer. “How exactly does word2vec work?” In: (2016), p. 18.

[24] Tomas Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *arXiv:1301.3781 [cs]* (Sept. 6, 2013). arXiv: 1301.3781. URL: http://arxiv.org/abs/1301.3781 (visited on 01/27/2020).

[25] Gunter Mussbacher et al. "The Relevance of Model-Driven Engineering Thirty Years from Now". In: *Model-Driven Engineering Languages and Systems*. Ed. by Juergen Dingel et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 183–200. ISBN: 978-3-319-11653-2. DOI: 10.1007/978-3-319-11653-2_12.

[26] Patrick Ng. "dna2vec: Consistent vector representations of variable-length k-mers". In: *arXiv:1701.06279 [cs, q-bio, stat]* (Jan. 23, 2017). arXiv: 1701.06279. URL: http://arxiv.org/abs/1701.06279 (visited on 01/23/2020).

[27] Selva Prabhakaran. *Cosine Similarity – Understanding the math and how it works (with python codes)*. https://www.machinelearningplus.com/nlp/cosine-similarity/. Accessed: 2020-02-15.

[28] *Présentation : Extraction et diffusion de représentations vectorielles continues de caractéristiques pour le machine learning*. https://cloud.google.com/solutions/machine-learning/overview-extracting-and-serving-feature-embeddings-for-machine-learning. Accessed:2020-02-01.

[29] Veselin Raychev, Martin Vechev, and Eran Yahav. "Code completion with statistical language models". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*. the 35th ACM SIGPLAN Conference. Edinburgh, United Kingdom: ACM Press, 2013, pp. 419–428. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594321. URL: http://dl.acm.org/citation.cfm?doid=2594291.2594321 (visited on 01/28/2020).

[30] Marco Torchiano et al. "Relevance, benefits, and problems of software modelling and model driven techniques—A survey in the Italian industry". In: *Journal of Systems and Software* 86.8 (Aug. 1, 2013), pp. 2110–2126. ISSN: 0164-1212. DOI: 10.1016/j.jss.2013.03.084. URL: http://www.sciencedirect.com/science/article/pii/S0164121213000824 (visited on 02/25/2020).

[31] Martin White et al. "Toward Deep Learning Software Repositories". In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. ISSN: 2160-1860. May 2015, pp. 334–345. DOI: 10.1109/MSR.2015.38.

[32] Jon Whittle et al. "Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?" In: *Model-Driven Engineering Languages and Systems*. Ed. by Ana Moreira et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 1–17. ISBN: 978-3-642-41533-3. DOI: 10.1007/978-3-642-41533-3_1.

[33] Dongwen Zhang et al. "Chinese comments sentiment classification based on word2vec and SVMperf". In: *Expert Systems with Applications* 42.4 (Mar. 1, 2015), pp. 1857–1863. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2014.09.011. URL: http://www.sciencedirect.com/science/article/pii/S0957417414005508 (visited on 01/23/2020).

Appendices

Appendix A: Database structure of the Lindholmen dataset