



# Universiteit Leiden

## Opleiding Informatica

A network-driven feature construction approach for  
labelling software classes using machine learning

Name: Xavyr Rademaker  
Date: 08/07/2018  
1st supervisor: Frank Takes  
2nd supervisor: Michel Chaudron

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)  
Leiden University  
Niels Bohrweg 1  
2333 CA Leiden  
The Netherlands

## Abstract

UML is a standard of modelling object-oriented software systems. Among others, it describes the relationships between classes within a software ecosystem. Each class in such a diagram can be labelled as one of six archetypes, according to its characteristics as defined in software engineering literature. Relationships between classes in the UML diagram can also be modelled as a network, which can then be analysed using methods and metrics from the field of network science. In this thesis, we use such network metrics to improve the accuracy of three different machine learning algorithms in the task of automatically labelling classes. In this process we observed that network metrics can also be used to determine the quality of both high and low level design decisions, some of which are: coupling, inter-package relationships and frequent small patterns, can be measured using these network metrics. We discovered that in the systems we analysed, a combination of both semantic and network-based features performs best. Next to this, we noticed that adding behavioural features to the feature set increases this accuracy even further.

## 1 Introduction

The Unified Modeling Language (UML) is the standard language software engineers use to document and model object-oriented software. The language provides a standard way of representing and visualizing how the system is structured. It does so by using various types of diagrams, each offering a different perspective on the system. Examples of these diagrams are class diagrams, which give an overview of the classes present, what methods/attributes each class has and which relationships exists between classes. Sequence diagrams, which shows the interaction between objects and use case diagrams, and use case diagrams, which show what functionalities can be used by certain actors.

Software systems tend to get big, meaning that the number of classes and the interaction between objects increases as the software is maturing. This growth results in large diagrams which take quite some time to understand for someone who was not involved in the development of the system. In order to increase the understandability of class diagrams, Wirfs-Brock proposed a way to simplify the diagram [35]. Their approach was to assign roles to classes. These roles represent archetypes of classes, which show the purpose of a class. In an ideal design, each class should fit exactly one role. The six roles she describes are: information holder, structurer, service provider, controller, coordinator and interfacier. A description of these roles can be found in Table 1.

Role	Description
Information holder	An object designed to know certain information and provide that information to other objects
Structurer	An object that maintains relationships between objects and information about those relationships
Service provider	An object that performs specific work and offers services to others on demand
Controller	An object designed to make decisions and control a complex task
Coordinator	An object that does not make many decisions but delegates work to other objects
Interfacier	An object that transforms information between distinct parts of the system

**Table 1: Description of the roles as defined in [35]**

Even though it is easier to understand the purpose of each class after labelling them, the labelling process is time consuming to do manually. To tackle this problem, this thesis will focus on an automated way of assigning labels to classes based on two types of features which will be explained later in this section. In order to accomplish this, several different machine learning classification algorithms will be compared on the task of labelling the classes. The algorithms of interest in this thesis can be divided in two categories: Tree based algorithms (decision trees and random forests) and logistic regression. These two classes of algorithms have been chosen for the following reasons: they are two of the most interpretable algorithms in the sense that after performing the algorithm, it is easy to understand what features the algorithm used to make its decisions. There is one exception to this rule, which is the random forest algorithm. At the time, this is the most powerful tree based algorithm but it also is the least interpretable.

When choosing which machine learning algorithm performs best, there are three factors that should be taken into account. These factors are: the complexity/interpretability (how easy is it to understand how the algorithm produced its results), performance of the algorithm for a given performance measure and

overfitting (explained in Section 2). In this thesis, we will choose the algorithm that produces the highest accuracy on data it did not see during the training of the algorithm. When two algorithms produce similar results (difference of up to one percent) we will choose the one with the best interpretability. Since our dataset is relatively small for more powerful algorithms, we will put less focus on overfitting (since more powerful algorithms tend to overfit faster on small datasets).

As mentioned before, we are going to use two types of features in our experiments. One of these types comes from the field of software engineering. In the field of software engineering, class properties can be divided in three categories, semantic properties (focus on semantics, for example: the number of methods of a class), structural properties (focus on relations between classes) and behavioural properties (focus on how a class performs a task). We will focus on the semantic properties and will refer to them as semantic software features in this thesis. The other type of features we will be using are going to be referred to as network features. Before explaining what these are, we have to start by introducing the term networks and the field of network science.

A network is a data structure in which the focus lies on the relationships between objects. A network has the underlying structure of a graph in graph theory. There are more extensive forms of these types of networks (directed, labelled, etc.). A more detailed explanation will be given in Section 2. Some examples of data that can be modelled as a network are: protein-protein interaction, employee interaction, computer interaction and social media friendships.

The field of network science focusses on the extraction of information from these networks. In this field, techniques and algorithms are used to extract information. Examples of questions which can be answered using network science are: what is the most important vertex (object) in this network, what patterns are frequent in different types of networks and which node has the most influence in this network. Modelling data as a network thus gives new insights into the data. It is worth to note that the algorithms of today are capable of handling very large networks (more than 100.000 nodes and more then 1.000.000 edges for example), meaning that very large datasets can be analysed.

Since a software system can be represented as a network of classes interacting with each other, it is possible to create a network of software class interactions or relations. In doing so, the structural properties from the field of software engineering are extended with metrics from network science. This can lead to new insights in the system. The second set of features (the network features) mentioned before will be acquired by performing network analysis on networks created from software systems. As we are using machine learning (which is a different research area) techniques for the classification task, we are combining three fields of study in this thesis. To the best of our knowledge, this combination (software engineering, network science and machine learning) has not been studied that much until now.

The focus of this thesis will be on the construction of a feature set consisting of network features and semantic software features. We will compare the performance of the classification algorithms in the previously mentioned classification task using three feature sets (semantic software features, network features and all features). The underlying assumptions are that: (1) constructing a feature set containing both network features and semantic software features will give the best results and (2) that the combination of features and a machine learning algorithm will perform better than individual features. The research question we will answer during this thesis is: *Does extending software class properties with metrics of network science improve the accuracy of machine learning algorithms in the task of labelling classes automatically?*

The rest of this thesis is structured as follows: Section 2 will give an overview of all definitions used throughout this thesis including examples wherever it is necessary. In Section 3 previous related work will be described. This section will discuss what was done in other papers, how it relates to this topic and what contribution we are making with this thesis. In Section 4 the approach used, and steps taken, to acquire the results will be explained. Section 5 will discuss the data used in the experiments including network and semantic software properties of the data. For the network properties, we will also see how to reflect to the software systems. Section 6 will discuss the results of the experiments and Section 7 provides the conclusion made based on our results and discusses assumptions. In Section 8 we propose possibilities for further research. Finally in Section 9 we present the tools we made during the thesis, where to find them and how to use them (input and output of the tools).

## 2 Definitions and notations

In this section we will describe the definitions and algorithms which shall be used throughout this thesis. Each definition has its own subsection in which a description shall be given.

### 2.1 UML diagrams

As mentioned in the introduction, UML diagrams provide insights into the way a system is structured. Because of this, UML diagrams are often used by people new to the system (for example the people maintaining the system) to get a quick understanding of what every class does, since the diagrams are often easier to understand than going through the actual code. There are several types of diagrams, each giving a different perspective on the system. In this thesis, the only diagram used is the class diagram. For this reason this subsection focusses on class diagrams only, followed by a brief description of design patterns.

It should be noted that UML diagrams are an abstraction of the system and are independent of the programming language used. However, the diagrams are limited to Object-Oriented software (software in which the code is divided in various classes) and thus this thesis is scoped to only focus on object-oriented systems. Even though the UML diagram is language independent, not all Object-oriented programming languages share the same terminology when implementing the code (for example, in Objective-C the interface is called a protocol).

#### 2.1.1 Class diagrams

A class diagram shows the existing classes within a system. It shows both structural and semantic information about each class. A class can be seen as a template from which objects originate. An object is an actual instance of a class. For example, if there is a class `human`, then an object could be `Xavyr Rademaker`. The idea is that each object of the same class has the same template (each human has two eyes, one nose etc.). There are four types of classes, which are:

- *Abstract class*: a class that cannot be instantiated (no objects can be made). An abstract class always has subclasses
- *(Concrete) class*: a class that can be instantiated
- *Interface*: similar to an abstract class, but it provides a template that realizing classes should follow.
- *Enum*: a class containing a set of named constants

Per class the following semantic information is contained in the diagram:

- Name of the class
- Attribute signatures of the class
- Method signatures of the class

An attribute is a property of the class. For example, attributes of the class `human` could be `eyeColor`, `hairColor` and `favoriteFood`. Each attribute also has a signature. The signature consist of four parts: it starts with the visibility which can be `public`, `private` and `protected`, followed by an optional `static`, followed by the attribute type which can be one of the native types (string, int, double, boolean, etc.) or an user defined class, finally the signature ends with the name of the attribute (`eyeColor` for example). A brief description of the three parts follows.

**Visibility** refers to the extent to which other classes can access the attribute. Making an attribute public means that every other class is able to access the attribute directly. A private attribute can only be accessed by the object itself, and is thus not accessible by other classes directly. In a couple of programming languages private attributes can be accessed by subclasses of the class containing the attribute as well. Just as private, protected attributes vary slightly per programming language. In certain languages it means that only subclasses can access it, while in other languages it means that every class in the same namespace and/or package can access the attribute. It is common practise to keep the visibility of each attribute and method as low as possible to reduce the interdependencies between classes.

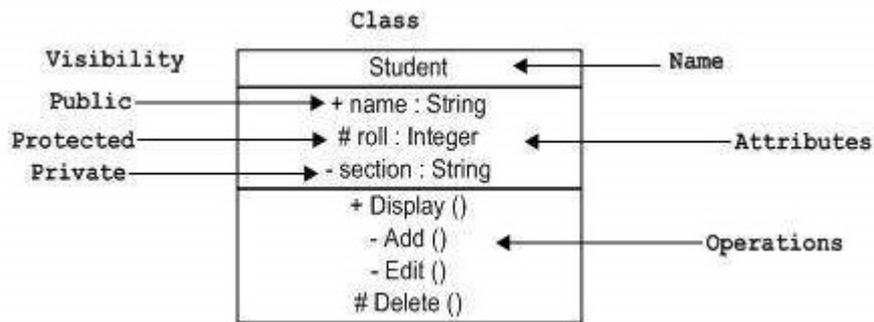


Figure 1: An example of semantic information in a class diagram. Taken from [https://www.tutorialspoint.com/uml/uml\\_basic\\_notations.htm](https://www.tutorialspoint.com/uml/uml_basic_notations.htm)

**Static** is an optional keyword which means that the value of the attribute is the same for each object of the class.

**Type** refers to the type of data the attribute contains. A string contains words or sentences while and int contains numerical data. User defined classes can also be used as type, which means that there is a relationship between the two classes (the class containing the attribute and the class to which the type refers). More on this will follow later in this subsection.

**Attribute name** is the final part of the signature. It refers to the name the developer gives to the attribute.

Method signatures are almost the same as attribute signatures. The main two differences are: methods can also have the abstract keyword which means that the method has no implementation and has to be implemented by subclasses of the class containing the abstract method. The second difference is that a method has brackets at the end of the name and in between the brackets are optional parameters. A parameter has (in most languages) the signature: *type* followed by *parameter name*.

An example of the semantic information in a class diagram can be found in Figure 1.

Next to the semantic information, the class diagram also preserves structural information. The structural information shows how the various classes are related to each other. The types of relationships between classes can be generalized into the following three categories:

- *dependencies*: A dependency means that class  $X$  depends on class  $Y$ . Being dependent means that class  $X$  either has an attribute with class  $Y$  as type, has a function with class  $Y$  as parameter, or uses class  $Y$  in a method.
- *Generalizations*: A generalization means that class  $X$  is a superclass of class  $Y$ . An example would be the following: if class  $X$  is **Animal** and class  $Y$  is **Fish** then  $X$  is a superclass of  $Y$  because a fish is an animal, while not all animals are fish.
- *Realizations*: A realization means that class  $X$  realizes interface  $Y$ . It means that class  $X$  abides by the template defined by interface  $Y$ .

In this thesis we use the definition of dependency given in [19] in which class  $A$  is dependent on class  $B$  if:

- $A$  has an attribute of type  $B$
- $A$  sends a message to  $B$  in which  $B$  is either an attribute, parameter variable, local variable, global variable or class visible (invoking static methods) of  $A$
- $A$  receives a parameter of type  $B$
- $A$  is a subclass of  $B$  or  $A$  implements interface  $B$

As mentioned before, we distinguish dependencies between generalization, implementation and other dependencies.

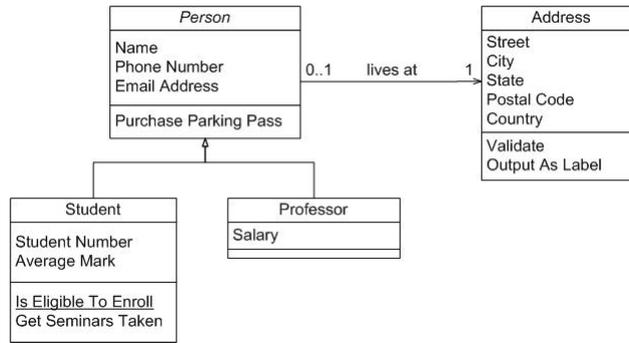


Figure 2: An example of the structural information in a class diagram. Taken from <http://www.agilemodeling.com/artifacts/classDiagram.htm>

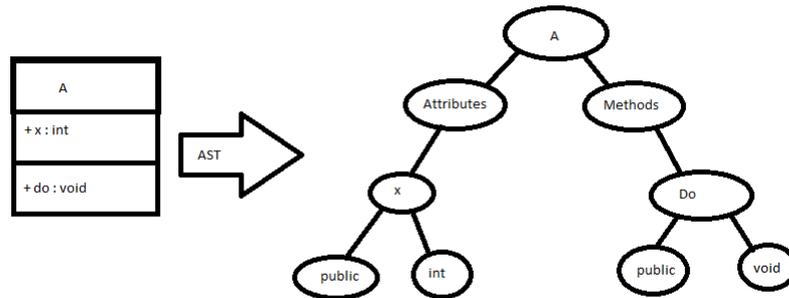


Figure 3: An example of an AST

Figure 2 shows an example of the structural information in a class diagram. The figure shows a generalization, where students and professors are both persons, and a dependency from person to address. Furthermore, person is an abstract class (since it is written in italic) and each person has one address (which can be seen by the number at the relation).

### 2.1.2 Design patterns

A design pattern is a standard solution to a common problem in software development. To be more specific, it is a description of a small part of a class diagram which is designed to provide a general solution to a common problem. An example of a common problem is the undo functionality which can be solved using the memento pattern. During our experiments we do not use design patterns, however to understand the related work discussed in Section 3 this definition was given.

## 2.2 Reverse engineering and the Abstract Syntax Tree

Usually when developing software the process is to first create the abstract model of the system (UML diagrams) and then implement it in code. The act of reverse engineering is to reverse this process, going from the system to an abstract model of it. These abstract models are also called intermediate representations.

Most papers that work with the analysis of software systems (for example identifying design patterns) use an intermediate representation of the system under analysis. These intermediate representations are used because they are easier to analyse than the source code itself. Examples of these intermediate representations are the abstract syntax graph (ASG), relationship matrices and the abstract syntax tree (AST).

In this thesis the Abstract syntax tree (AST) is used in the reverse engineering process. An abstract syntax tree represents a software system by turning it into a tree. For example, an AST of a class `A` with attribute `public int x` and method `public void do()` can be seen in Figure 3. Of course there are multiple ways to construct the AST, as long as it is a tree that represents the system.

One of the main benefits of using an intermediate representation of the system (AST for example) is that it is easier to analyse the system based on such a tree instead of based on the source code in plain text. For example, knowing how many public methods exist in class `A` in the example of Figure 3 only requires to

traverse the branches of "Methods" and count the number of times `Public` occurs at the leaves. To answer the same question using the source code instead of the AST would mean going through every line of the file, use a form of pattern matching to find method signatures and count the number of times `public` occurs. Another benefit of the AST is that there are already libraries/tools available to turn code into an AST.

## 2.3 Network

A network has the underlying structure of a graph, which means that graph theory notations are applicable. A graph  $G$  is a collection of the sets of edges  $E$  and vertices  $V$ . Thus the graph is denoted as  $G = (V, E)$ . Just as a graph, a network can become more complex. For example, the edges can be directed (one-way relationships), both edges and vertices can be labelled, the nodes in the network can be divided in two types of nodes where each edge goes from one type to the other (two-mode networks). In this thesis we define  $n$  to be the number of nodes in the network and  $m$  to be the number of edges.

In the context of this thesis, each vertex in  $V$  represents a class of the class diagram, while an edge  $E$  represents a relationship between the classes. The edges used in this thesis are labelled and directed. In this thesis we define three types of edge labels in the used networks which are  $d$ ,  $in$  and  $im$ , which respectively mean: dependency (node  $u$  depends on node  $v$ ), inherits (node  $u$  is a sub class of node  $v$ ) and implements (node  $u$  implements node  $v$ ).

Some definitions related to networks are:

- *Path*: Given a network with three nodes  $u$ ,  $v$  and  $w$  and three edges  $(u, v)$ ,  $(v, w)$  and  $(u, w)$ , a path from node  $u$  to node  $w$  is a sequence of edges. In our example network there are two paths from  $u$  to  $w$ , which are  $(u, v)$ ,  $(v, w)$  and  $(u, w)$ . A path starts at a certain node (the first node in the first edge of the sequence,  $u$  in our example) and traverses via the edges until the last node of the path (last node in the edge sequence,  $w$  in our example).
- *Shortest path*: A shortest path between nodes  $u$  and  $v$  is a path between  $u$  and  $v$  with the least number of edges in between. In our previous example this would be path  $(u, w)$ . There can be multiple shortest paths from node  $u$  to node  $v$  in a network.
- *Distance*: The distance between nodes  $u$  and  $v$  is the number of edges in a shortest path from  $u$  and  $v$ . The distance from node  $u$  to node  $v$  is represented as  $d(u, v)$ .
- *Node eccentricity*: The length of a longest shortest path starting at a node
- *Node centrality*: Node centrality is an importance measure of a node in the network. There are different forms of node centrality which will be discussed in the next subsection.
- *Degree*: The degree of a node is the number of edges connected to the node. In this thesis we use directed networks, meaning that there is an in-degree (edges going to the node) and an out-degree (edges going out of the node).
- *Component*: A component is a subgraph of nodes for which there exists a path between all the node pairs in the component. The largest component in the network is called the giant component.
- *Diameter*: The diameter of a network is the length of a longest shortest path between any two nodes in the network.
- *Density*: The density of a network gives information about the proportion of potential edges in the network that are actual existing connections.
- *Clustering coefficient*: A measure used to measure to what degree nodes in the network tend to cluster together.
- *Degree assortativity*: A measure used to measure how nodes of similar degrees cluster together. A value of 1 means that nodes with similar degrees tend to cluster together, a value of -1 means that nodes of similar degrees do not cluster together at all.
- *Community*: A subset of nodes connected more strongly to each other than with the rest of the network

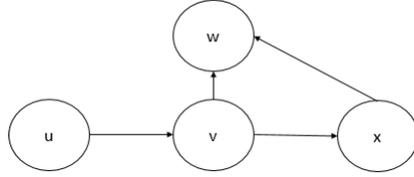


Figure 4: An example of a network

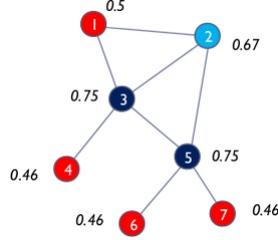


Figure 5: An example of closeness centrality. Node 3 and 5 have the highest value because they both lie in the center of the network, making their average distance to all other nodes low. Image taken from <https://www.slideshare.net/gcheliotis/social-network-analysis-3273045> slide 22.

Figure 4 shows an example of a directed network. In this figure, there are four nodes ( $u, v, w$  and  $x$ ) and four directed edges ( $(u, v), (v, x), (x, w)$  and  $(v, w)$ ). There are two paths going from node  $u$  to node  $w$ , which are  $(u, v), (v, w)$  and  $(u, v), (v, x), (x, w)$ . There is only one shortest path from  $u$  to  $w$  which is  $(u, v), (v, w)$ , with a distance of 2. The eccentricity of node  $u$  is 3. Finally, the in-degree of node  $v$  is 1 while its out-degree is 2.

## 2.4 Centrality measures

Centrality measures in the field of network analysis are often measured based on distance/paths. They are used to determine the importance of a node in the network. The centrality measures used in this thesis are: degree centrality, closeness centrality, betweenness centrality and eccentricity centrality. We will discuss our choice for these centrality measures in more detail in Section 5.

### 2.4.1 Degree centrality

The degree centrality of a node measures the degree of the node (the number of edges linked to/from the node) to the degree of the rest of the nodes. The higher this centrality measure, the higher the degree compared to the degrees of other nodes in the network.

### 2.4.2 Closeness centrality

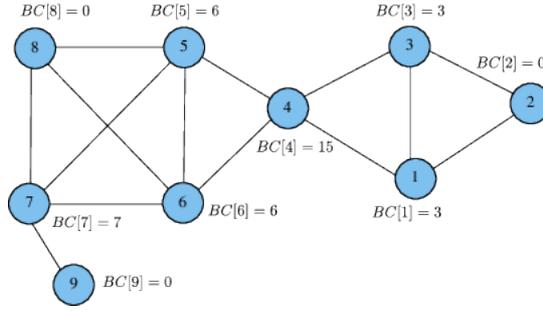
The closeness centrality calculates the average distance from a vertex to each other vertex in the network. It is calculated using the following formula. The idea is that a vertex that is close to every other vertices means that it is less dependent on the other vertices to relay a message [10]. The formula used to compute the closeness centrality of a node is (where  $m$  is the number of edges and  $n$  is the number of nodes in the network):

$$C_c(v) = \left( \frac{1}{n-1} \sum_{w \in V} d(v, w) \right)^{-1} \quad (1)$$

The algorithm runs in  $\mathcal{O}(mn)$  since it has to run one breadth-first search of  $\mathcal{O}(m)$  for each  $n$  vertices of the network. Figure 5 shows an example of closeness centrality values.

### 2.4.3 Betweenness centrality

Betweenness centrality is a centrality measure that measures the relative number of shortest paths that pass through the vertex [9]. Nodes with a high betweenness act as bridges that connect parts of the network.



**Figure 6: An example of betweenness centrality, denoted BC. Node 4 has the highest value because it connects two parts of the network, making all shortest paths from one side to the other going through node 4. Image taken from <https://devblogs.nvidia.com/accelerating-graph-betweenness-centrality-cuda/>**

Figure 6 gives an example of betweenness centrality. Brandes presented an algorithm to calculate this centrality measure in  $\mathcal{O}(2mn)$  time by performing two breadth-first searches for each vertex in [5]. The formula used is:

$$C_b(u) = \sum_{v,w \in V} \frac{\sigma_u(v,w)}{\sigma(v,w)} \quad (2)$$

where  $\sigma(v,w)$  is the number of shortest paths from  $v$  to  $w$  and  $\sigma_u(v,w)$  is the number of such shortest paths going through  $u$ .

#### 2.4.4 Eccentricity centrality

Node eccentricity has already been described in this thesis, it refers to the length of a longest shortest path of a node. It is denoted as:

$$e(v) = \max_{w \in V} (d(v,w)) \quad (3)$$

Using that definition of node eccentricity, eccentricity centrality is defined as follows:

$$C_e(v) = \frac{1}{e(v)} \quad (4)$$

This means that eccentricity centrality is the worst-case variant of closeness centrality. In other words, the further a vertex is to all other vertices, the higher its eccentricity value.

## 2.5 Motifs

A network motif is a frequently recurring significant pattern of interconnections. Motifs are considered building blocks for networks. They can be used for various purposes. For example, Milo *et al.* showed that different motifs occur in different types of networks [22].

Motifs are usually small patterns consisting of two to five nodes. They are often called  $k$ -node motifs, where  $k$  refers to the number of nodes contained in the motif. For example, Figure 7 shows an example of a 3-node motif containing two directed edges. To determine whether the frequency with which a motif occurs is significant, it is compared to the frequency it occurs in random networks (for example by counting its frequency in 1000 random networks of the same size). With this knowledge one is able to identify the type of network being studied by looking at the types of motifs appearing in the network.

There are various ways to generate the random networks. The way it is done has a direct impact on the motifs found. The random networks can be generated in different ways, for example: generate the network with the same number of nodes and edges as the original, generate the network taking the degree distribution into account, etc. In this thesis we generated the random networks taking the number of nodes and number of edges into account.

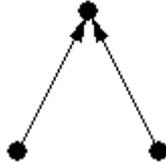


Figure 7: An example of motifs 3-node motif containing two edges

## 2.6 Machine learning

Machine learning is a field of study in which focusses on extracting knowledge from data. It does so by letting the algorithm learn the patterns of the data by itself. The usual steps taken in a machine learning project are: collecting the data, analysing it, extracting features from the data and finally the data is set as input into an algorithm. The algorithm used belongs to a set of algorithms where the machine learns from examples (training instances) to perform better at a certain task. More formally, given a performance measure  $P$ , a task  $T$  and learning experience  $E$ , an algorithm is said to be learning if its performance measured by  $P$  improves for task  $T$  with more learning experience  $E$  [23]. Machine learning algorithms can be divided in three categories: supervised, reinforcement and unsupervised learning. In supervised learning the algorithm is given labelled data as input. The algorithm then learns the optimal function given the labelled input data, and is finally used to predict the labels of future unlabelled data. In reinforcement learning the algorithm learns from a series of reinforcements (rewards or punishments). For example, the lack of a tip at the end of the journey gives the taxi algorithm an indication that he did something wrong, while a large tip indicates that he did something right. Unsupervised learning refers to algorithms that are trained using unlabelled data. These types of algorithms are often used to learn more about the data and find relevant features [25].

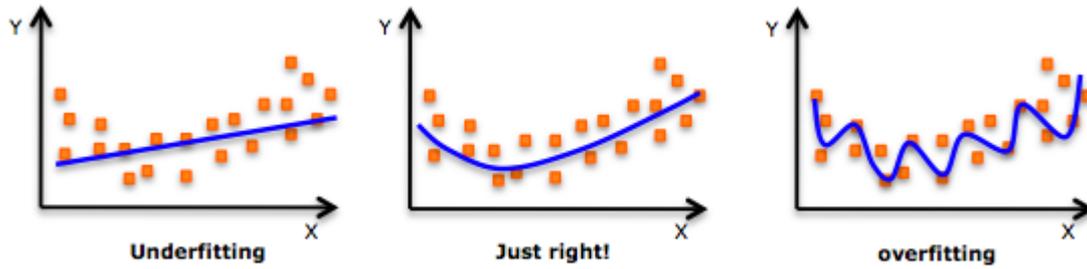
Supervised learning can be used for regression problems as well as classification problems. In this thesis, we are facing a classification problem, thus this subsection shall only focus on supervised classification algorithms.

Supervised learning algorithms are trained using a set of labelled input data  $(X, Y)$  where  $X$  is the set of the input features and  $Y$  is the set of label corresponding to those features. This dataset is often split up into three smaller subsets  $(X_{train}, Y_{train}), (X_{cv}, Y_{cv}), (X_{test}, Y_{test}) \subset (X, Y)$ . These subsets are often split into the following sizes: 60%, 20% and 20% of the total dataset respectively. The idea of splitting the data up in three subsets is that the biggest set  $((X_{train}, Y_{train}))$  is used to train the algorithm, meaning that the algorithm gets  $X_{train}$  as input, tries to predict  $\hat{Y}$  and then gets adjusted according to the difference between  $Y$  and  $\hat{Y}$ . With each new example, the algorithm is improved further and further until all the training data is used.

It is often feasible to train different models at the same time, as this reduces the time that is spend on finding the right model. After various models have been trained, the cross-validation (cv) set is used  $((X_{cv}, Y_{cv}))$ . The cv dataset used to see which of the models performs best. Finally, the chosen model is executed on the test set  $((X_{test}, Y_{test}))$ , which contains data the algorithm has not seen before. The results on the test set show how well the model created by the algorithm generalizes over new data. Therefore, the accuracy on the test set is used to show how good/bad the algorithm performs. Using data the algorithm has not seen before also shows if the algorithm is over- or underfitting. These two terms are explained by Breiman in [6] and will be explained before continuing to the algorithms.

Even though the algorithms learn from the data by itself, they have parameters which have to be tuned manually. These parameters are called hyper-parameters. Hyper-parameters are tuned to optimize the way the algorithm learns from the data. For example, they can influence the learning rate of the algorithm or influence the way the algorithm generalizes to new data.

**Overfitting** means that the model created by the algorithm fits too perfect to the train data. This means that the algorithm got trained in a way that it can predict every value of the train data perfectly. This often means that the model is made too specific for the train data and thus does not generalizes well. Whether a model overfits or not can be seen when the measured error on the training data is very low, but the error on the test data is very high (as the algorithm does not generalize well).



**Figure 8:** The difference between overfitting, underfitting and being just right. Image taken from [https://docs.aws.amazon.com/machine-learning/latest/dg/images/mlconcepts\\_image5.png](https://docs.aws.amazon.com/machine-learning/latest/dg/images/mlconcepts_image5.png)

**Underfitting** is the exact opposite of overfitting. It means that the model the algorithm produced is too simple to fit the data. An underfitting model can be recognized by observing high measured error levels on both the training and the test data. A model that does not overfit, and does not underfit is said to be just right. Figure 8 shows the difference between the three types of fit. As seen in the figure, having a model that is just right is preferred when using the algorithm for predictions.

One way to tackle the overfitting/underfitting problem is to tune the hyper-parameters accordingly. Finding the optimal hyper-parameters for an algorithm can be a complex task. The most straightforward approach is to perform an exhaustive search over (almost) all possible combinations of hyper-parameters in order to find the combination of hyper-parameters which produce the best results. In this process the algorithm is trained and validated using cross-validation for each combination of hyper-parameters. This is the approach we used in our thesis, and will be referred to as *gridsearch*.

The goal of a supervised machine learning algorithm is to minimize a cost (error) function  $J(x)$  such that the error on both the train and the test sets are as low as possible. One example of such a cost function is the mean square error which is defined as follows:

$$MSE = \frac{1}{t} \sum_{i=1}^t (\hat{Y} - Y)^2 \quad (5)$$

Where  $t$  is the number of training examples,  $\hat{Y} = h(x)$  and  $h(x)$  is the result the created model gives for input  $x$ .

### 2.6.1 Classification

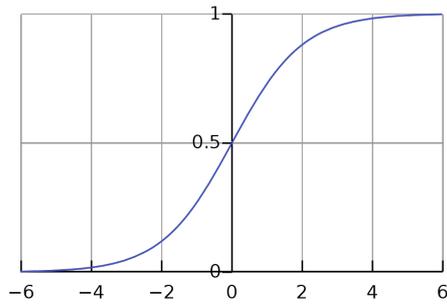
Classification problems refer to problems where input data is mapped to a categorical output. In other words, given a set of input variables  $X$  the algorithm will produce an output variable  $y \in Y$ , where  $Y$  is a (predefined) set of possible outputs. An example of a classification problem is image recognition, where given a set of pixel values  $X$  the algorithm given an output belonging to a set (for example  $Y = car, dog, cat$ ).

There are two main types of classification: binary classification and multi-class classification. Binary classification is classification where the output can only be of two types. A popular example of binary classification is credit card fraud detection. Given a set of input feature values, the output is either "1" (fraud) or "0" (no fraud). Multi-class classification is classification where there are more than two output classes.

### 2.6.2 Multi-class classification

The problem tackled in this thesis is a multi-class classification problem (since there are six labels). However, not every classification algorithm can work with more than two output classes. For example, logistic regression only outputs a 1 or a 0, making it a binary classifier, while decision trees can natively handle multiple output classes [16].

In the previous years, solutions have been found which allow for the use of binary classifiers for multi-class classification tasks. One of these solutions is to implement an one-vs-one strategy. This means that every combination of two classes are treated as a binary classification problem. At the end, the class that is predicted by most of the classifiers is chosen as output. The problem with this approach is that a lot of



**Figure 9: The shape of the sigmoid function**

classifiers have to be trained ( $c!$  where  $c$  is the number of classes) which, especially for complex algorithms, can result in a very long training time.

Another approach, which is often more feasible, to use binary classifiers in multi-class classification problems is to apply an one-vs-all strategy. The one-vs-all strategy means training  $c$  binary classifiers. Each of these classifiers predicts whether a set of input feature values belongs to one specific class or not. For example, in the problem tackled in this thesis six classifiers will be trained. Each of these classifiers only focusses on whether the input is one specific label or not (so one classifier for interfacier, one classifier for information holder etc.). The class with the highest score (usually a probability) output at the end is chosen to be the output class. The main advantage of this approach over the one-vs-one strategy is that only  $c$  classifiers need to be trained instead of  $c!$ . This means less training time.

In 2004, Rifkin and Klautau showed that the one-vs-all strategy can perform almost equally well as native multi-class classification algorithms in [28]. They discovered that what mattered most is to tune the hyper-parameters of the algorithm. When the right hyper-parameters are found, similar results were produced using the one-vs-all strategy and multi-class classification. Knowing that one approach is not necessarily worse than the other, we decided to include a binary classification algorithm to our experiments.

### 2.6.3 Logistic regression

Logistic regression is one of the most basic classification algorithms. It is a binary classification algorithm that is mostly used to predict the probability of an instance to belong to a certain class. In [16] it is described as follows: logistic regression works by computing a weighted sum of the input features and a bias term which is always set to 1. It then outputs the logistic of this weighted sum, using a function that is bounded by 0 and 1 or -1 and 1. One of the most used functions to do so is the sigmoid function. Thus the algorithm can be described by the following two equations:

$$\hat{y} = \delta(\theta^T * x)$$

Where  $x$  is the vector of input features and  $\delta$  equals:

$$\delta(z) = \frac{1}{1 + \exp(-z)}$$

This sigmoid function always returns a value between 0 and 1 (including the boundaries). By rounding the output to either 0 or 1, a binary prediction can then be made. The shape of the sigmoid function can be found in Figure 9.

The objective of the logistic regression algorithm is to set the parameter vector  $\theta$  so that the model estimates high probabilities for positive instances, and low probabilities for negative instances. To do so, an appropriate error measure has to be used. The error made on a single instance should be high when the algorithm outputs zero when the actual value is one (and vice versa) and should be low when the algorithm outputs zero when the actual value is zero.

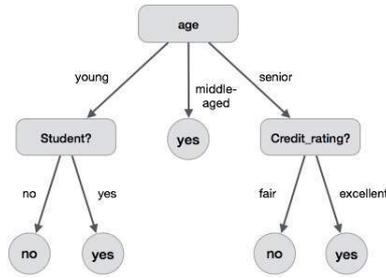


Figure 10: An example of the a decision tree. Image taken from [https://www.tutorialspoint.com/data\\_mining/dm\\_dti.htm](https://www.tutorialspoint.com/data_mining/dm_dti.htm)

One appropriate cost function for this problem is the log loss function. In short, this function works as follows for a single instance:

$$J(\theta) = \begin{cases} -\log(\hat{y}), & \text{if } y = 1 \\ -\log(1 - \hat{y}), & \text{if } y = 0 \end{cases}$$

To compute the cost function over the whole training set, the average cost over all training instances should be taken. This can be written as a single expression as follows:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right]$$

This way  $J(\theta)$  is a function of weights, which can be optimized using the gradient descent algorithm.

#### 2.6.4 Decision trees

Decision trees are a set of machine learning algorithms which belong to logical (symbolic) learning algorithms. They are trees that classify instances by sorting them based on feature values. Each node in a decision tree represents a feature in an instance to be classified, and each branch represents a value that the node can assume. The feature that best divides the training data is the root of the tree. Each non-leaf node within the tree represents a new splitting point based on the features of the instance. Finally, the leaves of the tree represent the prediction.

Once trained, making a prediction means traversing the tree from top to bottom based on the instance its feature values [18]. One of the most useful characteristics of decision trees is their interpretability. Since the results clearly show the effect of each attribute on the results, it is easy to understand and explain why the algorithm produced a certain result. An example of a decision tree can be seen in Figure 10. In this figure *age*, *student* and *credit,ating* are the features, *young*, *middle – aged*, *senior*, *no*, *yes*, *fair*, *excellent* are the possible values of their features respectively and *yes* and *no* are the possible outcomes.

There various different decision tree algorithms. Examples of the decision tree algorithms are J48, Reduces Error Pruning (REP) Tree, C4.5 and Random Forest. Literature shows that random forests have the best performance of them in multiple applications, for example [8]. Another study showed that random forest also work well to predict spatial distribution of the potential yield of ruditapes philippinarum in the Venice lagoon, Italy [33]. For this reason it is decided that random forest will be used in this thesis.

Random forest are a collection of combined decision trees. Each tree is built using a different split of the training data. Each tree has a different train and cross-validation set on which it is built. After going through all the trees a majority voting takes place. The majority voting is a step in which the results of all the instances of decision trees are counted and the output with the most votes gets to be the output of the algorithm. Figure 11 gives an example of a random forest. Taken the three trees given in the figure, Class-B will be the chosen output since two out of the three trees have Class-B as output. The random forest algorithm used in this was proposed by Breiman in [7].

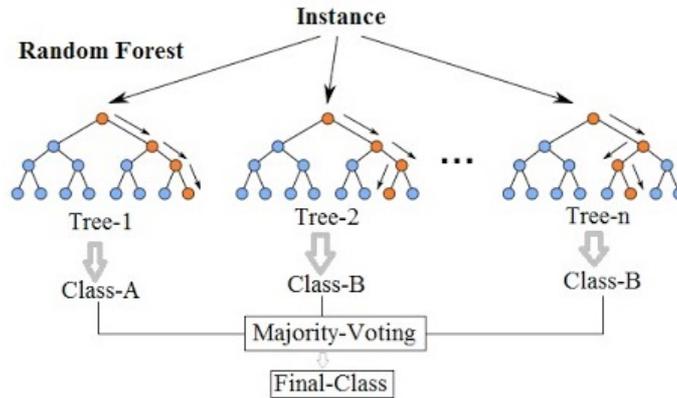


Figure 11: An example of the a random forest. Taken the three trees in the figure, Class-B will be the chosen output since two out of the three trees have Class-B as output. Image taken from <https://medium.com/@williamkoehrsen/random-forest-simple-explanation-377895a60d2d>

## 2.7 Imbalanced data

Imbalanced data refers to a dataset of two or more classes where one or more classes are underrepresented. One often used example is a credit card usage dataset with a certain set of features  $X$  and two classes  $Y \in \{0, 1\}$  where 0 is normal and 1 indicates fraud. Because normal credit card usage occurs much more frequently than fraud, the dataset will contain much more rows belonging to class  $Y = 0$  than class  $Y = 1$ . This means that a classification algorithm that predicts everything as  $Y = 0$  is right most of the time, and thus based on error rate alone it will be a good learning algorithm. Earlier solutions to this problem were to either oversample the minority class ( $Y = 1$ ) or to undersample the majority class ( $Y = 0$ ). In 2002, Chawla *et al.* proposed a method that performs both over- and undersampling at the same time called SMOTE in [11]. In this subsection, both undersampling, oversampling and SMOTE will be explained in more detail.

### 2.7.1 Oversampling

Oversampling refers to a technique where the dataset is made more balanced by increasing the size of the minority class. Early methods did this by picking, at random, certain examples of the minority class multiple times until the ratio minority to majority classes is as desired.

The problem with this way of increasing the minority class is that a classifier gets to see the same example multiple times. In doing so, it will learn very specific rules for classifying rows of the minority class, which increases the chances of overfitting (since the rules learned will be very specific for the few examples it has seen).

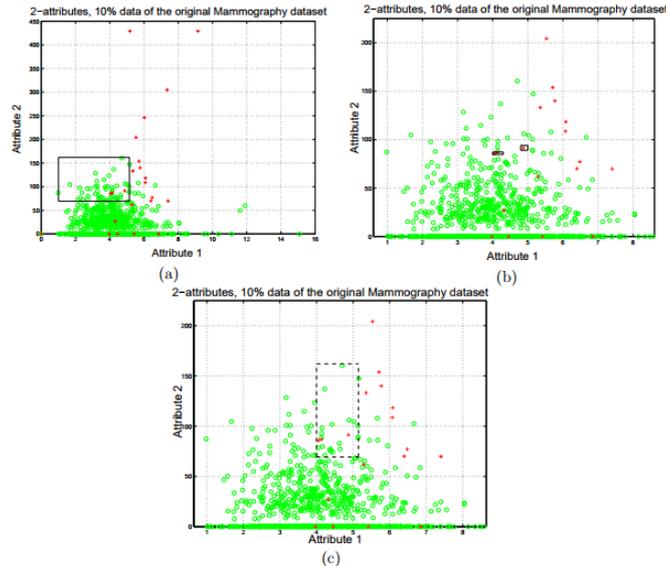
### 2.7.2 Undersampling

Undersampling refers to the opposite of oversampling, namely to a technique where the dataset is made more balanced by decreasing the size of the majority class. Most undersampling methods take the approach of removing examples from the majority class at random until the desired balance is created. In general undersampling usually leads to better classifiers than oversampling. A combination of oversampling together with undersampling does not lead to classifiers that outperform those build using only undersampling.

### 2.7.3 SMOTE Algorithm

The SMOTE algorithm, presented in [11], is an algorithm used to balance out the dataset by taking a different approach to combining over and undersampling. It combines over and undersampling by performing two steps:

1. Undersample the majority classes by randomly removing examples from this class
2. Oversample the minority classes by creating artificial examples representing the minority class



**Figure 12: Various decision regions using different sampling techniques. (a) the decision region without oversampling. (b) the decision region using simple oversampling with replacement. (c) the decision region with oversampling with syntetic generation. Taken from [11]**

The second point is what makes this technique interesting. By creating artificial examples of the minority class, overfitting is prevented. This is because the artificial examples always lie between two minority class examples thus ensuring that the classifier learns more general rules (since the chances of there being two or more examples with the exact same feature set is small). Figure 12 gives an example of the SMOTE algorithm (taken from [11]): (a) shows the decision region in which three minority class samples reside. The decision region is indicated by the solid-line rectangle. (c) is a zoomed in view of the minority class samples for the same dataset, the small solid-line rectangles show the decision regions as a result of oversampling the minority class with over-sampling. (c) is a zoomed-in view of the chosen minority class samples for the same dataset. The dashed lines show the decision region after over-sampling the minority class with syntetic generation. It can be observed that the decision region of (c) is bigger, meaning that the algorithm learned properties of the minority class without overfitting them.

### 3 Related work

A lot of work has been done in the field of UML diagrams, classification and networks. However, most of this work is related to the identification of design patterns instead of labelling classes. In this section we present previous related work we read and explain what our research contributes.

#### 3.1 Previous work

In [12] Dong *et al.* give a review of design pattern mining techniques. They categorized a number of design pattern discovery methods based on the pattern aspect checked. The categories were: structural aspect (the method only looks at the structural aspects of the design pattern, for example the number or types of relationships between classes), behavioural aspect (the method only looks at the behavioural aspects of the design pattern for example function calls), pattern composition (looks at the way a pattern is set-up), semantic aspect (the number of relationships and names of classes for example) and combinations of the four.

Dong *et al.* also list different intermediate representations of the systems in [12], since most methods use algorithms on the system itself, instead of its documentation. They mention that most methods use either the Abstract Syntax Tree, which was explained in Section 2, or the Abstract Syntax Graph as intermediate representation. Interesting aspects of their paper were the features they mentioned, for example number of public, private and protected methods, number of relationships and signatures of methods (incl. parameters). The information of this paper inspired us in acquiring our features, semantic and structural (described in Section 5), and gave us an understanding of intermediate representations of software systems.

In [2], Ballis *et al.* presented a rule-based method to match software patterns against UML models. They defined a language to express software patterns and applied a rule based algorithm which uses the defined patterns and matches them with patterns found in the UML diagram. They used a language to define patterns so that developers/maintainers of the system have the freedom of defining their own patterns, which gives more freedom than static pattern mining algorithms.

Tsantalis *et al.* proposed a design pattern detection algorithm using similarity scoring in [32]. They treated the system as a graph (similar to a part of this thesis) and used a graph similarity scoring algorithm to find matching patterns. They represented the network by using various adjacency matrices. For each of the matrices  $M$ ,  $M_{ij}$  (where  $i$  and  $j$  are vertices) contained an one if the relationship existed between two vertices and a zero otherwise. Each matrix represented a different type of relationship of the system. For example, there is a matrix for dependencies, a matrix for inheritance etc. Their algorithm matches the matrices as they are presented by the user. This means that the user is free to describe both the system and the pattern in whatever way he pleases, as long as it is possible to construct an adjacency matrix of it.

Their algorithm looked for exact graph isomorphism between the described pattern and parts of the system. After the exact matches, it starts looking for inexact matches, since it is possible for developers to make slight modifications to the design pattern. In short, their method can be described in the following steps:

1. *Reverse engineering of system*: Each characteristic of the system under study is represented as a separate  $o \times o$  matrix, where  $o$  is the number of classes
2. *Detection of inheritance hierarchies*: All kinds of generalization relationships are considered for building the inheritance trees. Classes that do not participate in any hierarchy are listed together in a separate group of classes. (feature extraction)
3. *Construction of subsystem matrices*: A subsystem is defined as a portion of the entire system consisting of classes belonging to one or more hierarchies. The system is split up to improve efficiency. (data pre-processing)
4. *Application of similarity algorithm between the subsystem matrices and the pattern matrices*: Normalized similarity scores between each pattern role and each subsystem class are calculated (execution)
5. *Extraction of patterns in each subsystem*: Usually, one instance of each pattern is present in each subsystem, which means that each pattern role is associated with one class. (results)

Although their goal was completely different from ours, we adopted their approach (reverse engineering, feature extraction, data pre-processing, execution and results) in our own approach, which will be described in more detail in Section 4.

Tekin *et al.* took a different approach to applying network analysis in order to find identical design structures in software systems in [30]. Their goal was not to find design patterns, but instead to find identical design structures in a project, and across projects. These patterns could mean that code structures are copied and thus are important to find when maintaining similar parts of the system(s).

Their approach had four steps, which are: (1)The AST of the source code of the system is analysed and the UML-based design level of extraction is created. (2) A software model graph is constructed in which classes and interfaces are the vertices and relationships between them are weighted directed edges. (3) A graph partitioning algorithm is used to divide the weighted and directed software model graph in small pieces. (4) The last step applies a sub-graph mining algorithm called CloseGraph [36] to discover identical structures. These first two steps of their approach are important to our own experiments as we used our own version of these steps to generate networks of the software systems.

The papers that inspired the experiments and research question of this thesis are [1], [14] and [4]. The first uses a XML-based pattern description language to describe the patterns and uses a matching algorithm to match software classes to pattern classes. Their approach works in three steps: first they analyse the C++ source code with the Columbus system [15] which builds an ASG. The second step consist of loading the pattern descriptions. Finally, the algorithm binds classes found in the source code to pattern classes that are part of the pattern description and checks whether they are related in a way that is described in the pattern. The problem they ran into was that the algorithm produced too much false positives.

In the second paper, they enhanced their algorithm by labelling the results of their approach with either true or false. Along with these labels, they also added predictors (public adoptee calls, not public adoptee calls, etc.) for the adapter pattern and inheritance context, is there a base, etc. for the strategy pattern. With these features they performed both a decision tree algorithm (C4.5) and a neural network using the backpropagation algorithm. Applying machine learning to their earlier found results resulted in less false positives. This paper gave us the idea of using the quantified node properties as input to machine learning algorithms.

In the third paper the authors applied supervised machine learning to link prediction. This topic inspired the approach proposed here because it uses the topological attributes of a social network as features for a machine learning algorithm. Some of the topological attributes they used are: the product of node its topological attributes (attributes as the degree and PageRank of a single node), neighbourhood based metrics (attributes as common neighbours of nodes, the jaccard's coefficient etc.), distance based metrics (attributes as shortest path between nodes  $v$  and  $u$ ).

In [31] the fields of software engineering and network science are combined to extract new information out of class diagrams. The goal of this paper was to condense reverse engineered class diagrams so that they become more understandable. In the process of condensing a class diagram, one should be careful not to remove any important classes. To determine whether a class is important or not, the authors used both network metrics as design metrics.

Finally, the last paper related to our work is the research of Dragan *et al.* In [13] they used a deterministic approach to determine the stereotypes of classes. Their approach is based on the frequency and distributions of semantic and behavioural features of classes. Even though their goal is also to label classes, they focus on different labels which stereotype a class. In [24] a tool is proposed which labels the classes and methods of a class automatically using these labels. These papers gave us insights into what semantic software features to use (for example accessors and mutators).

The UML diagrams (and thus the networks) used in this thesis are based on the results of [29] and [17]. In these papers the authors systematically mined over 12.000.000 GitHub projects to find UML files in them.

## 3.2 Contribution

This thesis will contribute to science in the following ways. First, we do not focus on design pattern detection but on labelling classes with roles. We do so by combining the fields of software engineering, network science and machine learning instead of taking a deterministic approach. To the best of our knowledge this combination has not been done yet. Furthermore, there is no research done in labelling software classes automatically using the labels provided in [35].

The results of our research can contribute to both industry and science. Labelling the classes automatically would mean that the complexity of class diagrams can be reduced far more easily (reducing the time it takes to label classes manually), which allows software engineers to spent less time on trying to understand the system and spent more time on improving the system. Also, the labelled classes can be used to develop new approaches in finding design patterns and other low-level software structures.

# 4 Approach

This section describes the approach used to address the research question. It starts describing how the data was acquired/prepared to be analysed in Section 4.1. This is followed the feature extraction in Section 4.2. Finally, in Section 4.3 we describe how we approached the machine learning aspect of the experiments.

## 4.1 Data preparation

For most of the data used in the experiments, the data had to be acquired and labelled manually. Doing so consisted of three steps:

1. Acquiring the source code of the project under study
2. Creating a network representing the UML class diagram based on the source code of the project under study

### 3. Assigning training labels to the classes of the project under study

The following subsections will discuss the used approach for each of these three steps.

#### 4.1.1 Acquiring the source code

All projects studied in this thesis are open-source projects which have their code either on GitHub, SourceForge or a site specifically for that project. We then decided which projects we are going to study. We chose the projects with the following reasons: they are all written in the same programming language (meaning that their structure should be similar and only one way to parse the data is necessary), they are all open-source (the source code can be acquired easily) and their size is not too big (so they can be labelled manually) while also being not too small (between 500 and 2000 classes). Furthermore, all of these projects are worked on actively by the community. The source code was acquired from the site hosting the source code (GitHub or SourceForge).

#### 4.1.2 Creating a network

The next step was to transform the source code to a graph representation of the code, where each node represents a class/interface and each edge represents either a dependency, inheritance or realization. To model the source code as a network, we wrote a program which transforms the source code to an AST and traverses the AST to determine whether the class has any form of dependencies (defined in Section 2) to other user defined classes. If there is a dependency, the system creates the appropriate labelled edges (implements, inherits or depends) between the user defined classes. Finally, the system saves the edgelist for future use.

#### 4.1.3 Assigning training labels

The training labels (labels used to train our algorithms) were assigned manually by following the labelling criteria found in Table 2. The labelling criteria were based on observations that were made when labelling one of the used projects. We checked all classes of each project one-by-one and labelled each class based on the criteria mentioned in the table and on the label descriptions given in Section 1. As mentioned in Section 1, in an ideal design each class should fit one label. However, since we study existing, and open-source, projects the design is not ideal. This means that it is possible for classes to fit multiple labels (as they do more than one thing). Whenever we encountered such a class, we looked at comments and usages of the class to determine its intended role in the overall system.

Label	Criteria
Information holder	<ul style="list-style-type: none"><li>• Name of the class is simple noun</li><li>• Contains 'Java library provided object' type or primitive type variables</li><li>• Contains getters and setter methods for variables</li><li>• May contain persistence features</li><li>• Complete information, not abstract class</li></ul>
Structurer	<ul style="list-style-type: none"><li>• Contains other user defined object types as variables</li><li>• Extends Java its collections framework</li><li>• Has methods that maintain relationships between objects or collections</li></ul>

Service provider	<ul style="list-style-type: none"> <li>• Class name ends with <code>-er</code> or <code>-or</code> but not <code>Controller</code></li> <li>• Has methods and attributes that are easily accessed by other classes</li> <li>• Could be realization of an interface (if the interface is a service provider as well)</li> <li>• Contains or read configurations</li> </ul>
Controller	<ul style="list-style-type: none"> <li>• Class name ends with <code>Controller</code> or <code>Manager</code></li> <li>• Should know information holders, coordinators and/or service providers</li> <li>• Has decision making statement</li> <li>• When work becomes too complicated, a controller could ask a coordinator to coordinate the work</li> </ul>
Coordinator	<ul style="list-style-type: none"> <li>• Holding connection between working objects</li> <li>• Forwards information and requests information</li> <li>• Delegate work to other classes, by creating objects of other class and calling their methods</li> <li>• Called by other classes</li> </ul>
Interfacer	<ul style="list-style-type: none"> <li>• Contains Java Swing, AWT and other UI components (user interface)</li> <li>• Import many user defined other classes</li> <li>• Manage user interface and handle user interaction</li> <li>• Encapsulates functions or objects in the system by providing an interface or an abstract class that can be used outside of the system</li> </ul>

**Table 2: Criteria used for assigning training labels**

Since the criteria in the table can be seen as a check-list, which usually means that it can be automated using a deterministic approach, it could be thought that a simple algorithm with if-statements is enough to label the classes automatically. However, most classes meet criteria of multiple labels meaning that the role of a class within the whole system should be looked at as well.

## 4.2 Feature extraction

As mentioned earlier, the research question of this thesis looks for the improvement of algorithm performance when using a feature set consisting of both network and semantic software features in the of task labelling software classes. To measure this improvement, two different types of features should be extracted from the data, namely semantic software features and network features. Both types of features are extracted from the data/source code in different ways.

Before extracting any features, we looked at the description of each label provided in [35]. For each label, we manually constructed an initial list of possible useful features based on their description. The result of this can be found in Table 3.

Label	Features
Controller	Many functions Out-degree > 0 Contains mostly void methods
Coordinator	Many relations to user defined classes (out-degree)
Structurer	Contains one or more lists Contains list managing methods
Information holder	High number of public attributes More than average accessors and mutators
Service Provider	Number of public methods higher than 0 Low number of public methods Mostly non-void methods High in-degree
Interfacer	Higher betweenness centrality High eccentricity centrality

**Table 3: Translation of description to features**

The reason that this was our initial list was because these features reflect the essence of each role. The controller is responsible for handling complex tasks and making decisions. Complex tasks are often split up in smaller simpler tasks (functions). Also, we expect the controller to have an out-degree of at least one because its smaller tasks can be made reusable by placing the code in either a service provider or a coordinator.

The coordinator is responsible for delegating work to others, which means that he should have a form of dependency to other classes (thus a high out-degree). Structurers are responsible for managing relationships. All enumerable classes (classes that can be iterated through) have this responsibility, which means that structurers should have of list management functions (add to list, remove from list, get by index etc.).

The information holder is responsible for holding information, which would be useless if this information is not accessible. For this reason, we expect it to have either more than average public attributes, or accessors which allow other classes to access the attributes. This brings us to the service provider, whose responsibility is to perform small, simple tasks for other classes (controllers and coordinators). To do this it must have at least one public method, so that other classes can make use of its service. Furthermore, it should not have too much public methods, since he only fulfils one simple task.

This brings us to the final role, which is the interfacer. The interfacer plays the role of a bridge between different software components. This can be interpreted in two ways. The first is that interfacers are placed between software modules, meaning that they have a high betweenness centrality. The second is that they are the bridge between the user (interface) and the back-end of the system, which means that it resides on the edges of the system thus having a high eccentricity centrality value.

These features were extracted from the data, together with features described in Section 4.1 and other features. These features will be described in the following two subsections.

#### 4.2.1 Semantic software features

Similar to the way the edgelist was created, the semantic software features are also extracted from the source code by transforming it into an AST. The following steps are taken in extracting the semantic software features:

1. Go through all classes and transform them to ASTs
2. Extract relevant information from the ASTs
3. Store information in a file for future use

Since the first steps is identical to the steps taken in creating the edgelist (Section 4.1), this section will not go into them again. All the semantic software features we extracted are described in Table 4. The tables also show a category to which a feature belongs, which are: Counting (C), Ratios (R), Statistics (S), Booleans (B) and Textual (T).

Feature	Category	Description
# of methods	C	The total number of methods of a class
# of public methods	C	The number of public methods of a class
# of private methods	C	The number of private methods of a class
# of protected methods	C	The number of protected methods of a class
# of undefined methods	C	The number of methods of which visibility is not defined
# of other methods	C	The number of methods with a visibility other than previously mentioned
# of attributes	C	The total number of attributes of a class
# of public attributes	C	The number of public attributes of a class
# of private attributes	C	The number of private attributes of a class
# of protected attributes	C	The number of protected attributes of a class
# of undefined attributes	C	The number of attributes of which visibility is not defined
# of other attributes	C	The number of attributes with a visibility other than previously mentioned
# of accessors	C	The number of accessors of a class
# of mutators	C	The number of mutators of a class
Method to attribute ratio	R	The ratio of methods to attributes
$\Delta$ attributes from average	S	The difference between number of attributes of a class and the average number of attributes per class in the project
$\Delta$ methods from average	S	The difference between number of methods of a class and the average number of methods per class in the project
Public method ratio	R	Ratio of number of public methods to total number of methods
Public attribute ratio	R	Ratio of number of public attributes to total number of attributes
Private method ratio	R	Ratio of number of private methods to total number of methods
Private attribute ratio	R	Ratio of number of private attributes to total number of attributes
Attribute method ratio	R	Ratio of number of attributes to number of methods
# of lists	C	Number of list attributes the class contains
Name ends with "Controller" or "Manager"	T	Class name ends with one of these strings
Name ends with "er" or "or"	T	Class name ends with one of these strings
# of non-void methods	C	Number of non-void methods
# of list managing methods	C	number of methods related to managing lists
Get set ratio	R	The ratio of mutators + accessors to the total number of methods
List method ratio	R	The ratio of list managing methods to the total number of methods
Get set ratio from average	S	The distances of the get set ratio to the average get set ratio
List ratio from average	S	The distance of the list method ratio to the average list method ratio
Is enum	B	Boolean determining if a class is an enum
Is class	B	Boolean determining if a class is a class
Is interface	B	Boolean determining if a class is an interface
Is abstract	B	Boolean determining if a class is abstract

**Table 4: Extracted semantic software features.**

Some of these features discussed earlier. The rest of them were added because they were extracted to compute further features (number of public methods to compute ratio of public methods for example). Finally, there are features which were extracted because they were similar to other features to extract

(number of public attributes and number of protected attributes), and because they might prove to be more useful than initially expected.

The counting features were chosen to compute the other features. The ratio features were chosen because of our assumption that certain roles have different ratios than other roles. For example, we expect a service provider to have a higher public method ratio than a controller (since a controller handles more complex decisions). The statistical features were chosen because the absolute numbers do not say a lot across projects. For example, in one project having three attributes is a lot, while in another project having three attributes is the minimum. The textual features were chosen because of convention. Class names usually reveal what they are. Things that provide a service often end with "or" or "er" (StringBuilder for example), and controllers are often called controller and manager (based on our own experience). Finally, the boolean features were added because the type of class can be relevant for the determination of certain labels. For example, as we will see in Section 6.1.3, an *Enum* is an Information Holder most of the times.

#### 4.2.2 Network features

To acquire the network features, we used the edgelist created in Section 4.1. The network is treated as directed network (since dependencies, inheritance and realizations are directed). After that, we applied several algorithms to acquire the network its properties (and the properties of its nodes). The list of features extracted from the network can be found in Table 5.

Feature	Description
Out-degree centrality	Centrality measure of node based on out-degree
In-degree centrality	Centrality measure of node based on in-degree
Betweenness centrality	Centrality measure of node based on betweenness
Closeness centrality	Centrality measure of node based on closeness
Out-degree	Number of edges going out of the node
In-degree	Number of edge going in the node
Eccentricity	Maximum graph distance between any other node of the network
Eccentricity centrality	Centrality measure of node based on eccentricity

**Table 5: Extracted network features**

Similar to the semantic software features, these features were are selected based on the expected relevant features and their ease of extraction from the data. The centrality measures were chosen because: we expected coordinators to have a higher out-degree centrality than the other roles, information holders and service providers are more likely to have a higher in-degree centrality and interfacers should either have a higher betweenness centrality (connecting system components) or a higher eccentricity centrality (user interface components). The remaining features were extracted because of the ease of extraction of these features.

It is also important to understand that if we only used a single project, using both the absolute values of in-degree and out-degree are redundant to the relative values of in-degree centrality and out-degree centrality. However, similar to the statistical semantic software features, the relative values might be more relevant over the different projects. The reason for this is that the absolute value of the in-degree and out-degree are dependent on the overall degrees of nodes in each network. For example, in project *A* having an in-degree of two is on the higher end, while in another project having an in-degree of two might be on the lower end.

### 4.3 Machine learning aspects

The next steps were the machine learning aspects of our approach. We first had to split the data in multiple smaller datasets, which we will discuss in Section 4.3.1. We also had to handle the imbalance in our dataset, which will be discussed in Section 4.3.2. The next step was the classification task, which is discussed in Section 4.3.3. Finally, we iteratively used these results to select the optimal features, which will be discussed in Section 4.3.4.

#### 4.3.1 Train, test and validation sets

As explained in Section 2, it is a best practise to split the input data into two subsets: a training set (about 80% of the data) and a test set (about 20% of the data). The training set is then again divided in a

validation set. This validation set can either be a predetermined split of about 20% of the training set, or it can be acquired using cross-validation.

In classification tasks it is important that no labels disappear when making such data splits. This is important because the data needs to see each label in the training process (otherwise it will not learn any patterns related to the missing label), and each label needs to be present in both the validation and the test set to make sure that the predictions made by the algorithm are evaluated in a good way. We used random stratified sampling to achieve such data splits. Stratified sampling is a sampling technique which splits the data in different subgroups (in our case each label is a subgroup) and then samples a number of instances from each subgroup. The number of sampled instances per subgroup is proportional to the size of the subgroup.

Using random stratified sampling to split our dataset into train and test sets makes sure that each label is present in every dataset, and it makes sure that the representation of each label is similar to the representation of the label in the full dataset.

### 4.3.2 Handling data imbalance

In Section 2 we explained the concept of data imbalance. As can be seen in Figure 13 the phenomena of data imbalance also occurred in our dataset (exact numbers are shown in Section 5.1). This imbalance led to poor performance of the algorithm in our earlier experiments.

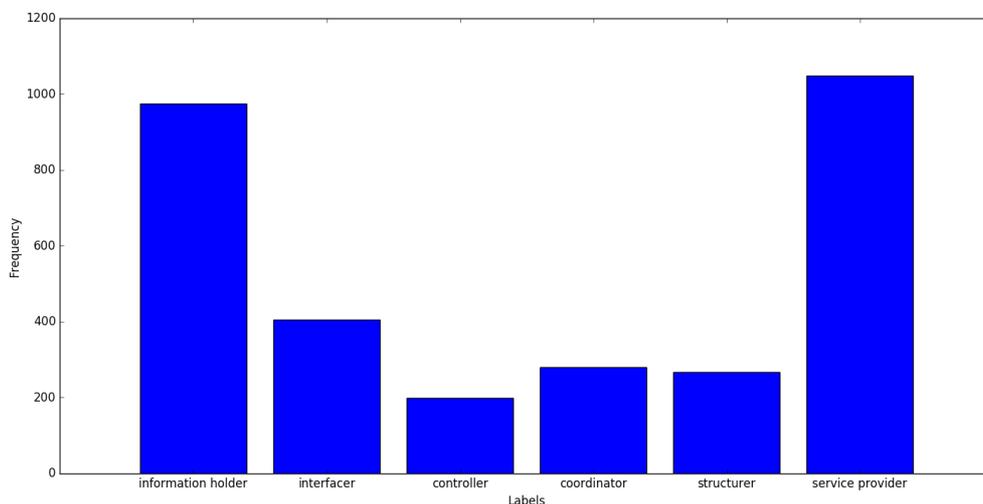


Figure 13: Frequency of each label in the total dataset.

In Section 2 we explained how the SMOTE algorithm can be used to balance out such imbalanced data. However, balancing the full dataset means that the data in the validation and the test sets do not represent to actual unseen data any more. In order to prevent this from happening, we applied the SMOTE algorithm on the train dataset after splitting the data into subsets. This made sure that the evaluation of the algorithms was done on data which the algorithm has not seen before and had a similar distribution to actual data.

### 4.3.3 Classification

The final step in this thesis was to decide what classification algorithm to use, and to implement this. Classification algorithms can become quite complex, to the point that we do not know what features the algorithm used or what decisions the algorithm made to make its final decision. In order to avoid this problem, we decided to start with simpler algorithms and continue to go on with more and more complex algorithms to see how the performance varies. When a simpler model can achieve a similar result as the complex models, the simpler model will be preferred in future applications, hence the choice to use this approach.

One of the most basic classification algorithms is logistic regression, so this was the first algorithm to apply. Another (far more powerful and yet very understandable) algorithm is the decision tree and its follower the random forest. Because of their power and interpretability, these two algorithms were included as well.

It should be noted that the process of splitting the data, handling data imbalance and classifying the instances was performed on three different datasets: the dataset with only network features, the dataset with only semantic software features and the dataset with the combined feature set.

#### 4.3.4 Feature selection

Machine learning algorithms are dependent on the features given as input. Given too few input features, even the most powerful algorithms will give disappointing results. Given too much input features on the other hand can result in overfitting of the data and/or decrease the accuracy of the algorithm.

Our approach to the selection consisted of two parts: first we analysed the individual features and see if we could find clear decision boundaries between the labels. Unfortunately this was not the case. We then applied a more iterative approach. We executed the classification algorithms, looked at the properties it learned from the data, and what features were related to these properties. We then ranked these features based on their importance and re-ran the algorithm with the smaller feature set to measure the differences.

In Section 6.2 we will discuss more implementation details of both the feature selection, extraction and generation of the networks.

## 5 Data

The datasets used in the experiments will be discussed in section. The first subsection will discuss the label distribution of the data. In the second subsection we will discuss the network properties of the datasets, and finally in the third subsection we discuss the semantic software features of the datasets.

The total list of datasets used in this thesis can be found in Table 6. These datasets were used because of their availability, size, the number of users and active development/maintenance of the software (datasets).

Dataset	Version	Downloadlink
K9Mail app	5.304	<a href="https://github.com/k9mail/k-9/releases">https://github.com/k9mail/k-9/releases</a>
SweetHome3D	5.6	<a href="https://sourceforge.net/projects/sweethome3d/files/SweetHome3D-source/SweetHome3D-5.6-src/SweetHome3D-5.6-src.zip/download">https://sourceforge.net/projects/sweethome3d/files/SweetHome3D-source/SweetHome3D-5.6-src/SweetHome3D-5.6-src.zip/download</a>
Mars simulation project	3.1.0	<a href="https://github.com/mars-sim/mars-sim">https://github.com/mars-sim/mars-sim</a>

**Table 6: Datasets used in this thesis**

### 5.1 Label distribution

In Section 4.3.2 it was mentioned that the label distribution was imbalanced. Table 7 presents the frequency of each label in the used datasets.

Label	K9	Home3D	Mars simulation	Total
Controller	26	47	125	198
Coordinator	45	56	179	280
Information holder	221	212	541	974
Interfacer	92	61	252	405
Service provider	359	160	529	1048
Structurer	33	25	208	266
<b>Total</b>	<b>776</b>	<b>561</b>	<b>1834</b>	<b>3171</b>

**Table 7: Label frequency per dataset**

It can be observed that the dataset is mainly dominated by service providers and information holders. Together they make up for about 60% of the dataset. Of the left over 40% the most frequent label is the interfacer which takes up almost the half (about 40%) of the leftover 30% of the data. This shows how skewed the dataset we are working with is. In theory this distribution can be explained by the characteristics of each class. We will discuss this theory in Section 7.

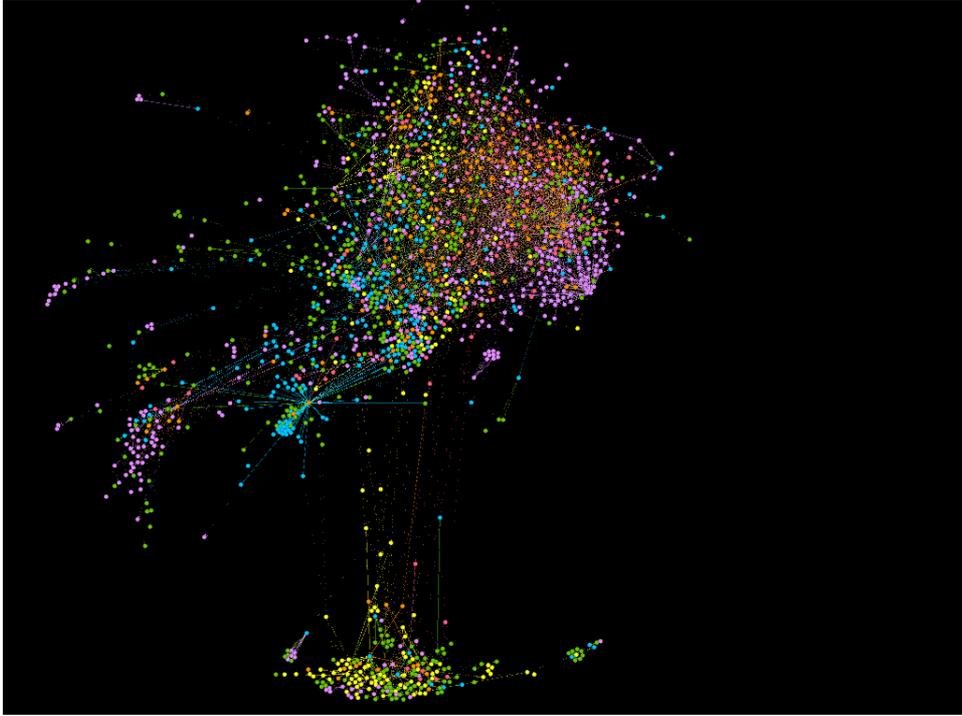


Figure 14: The giant component of the network of the mars simulation project visualized. The color of each node represents its label: Purple-Information Holder, Green-Service Provider, Blue-Interfacer, Organge-Coordinator, Yellow-Structurer and Pink-Controller. The network is visualized using Gephi [3].

As can be seen in Figure 14, the labels tend to mixed together in different parts of the system. This makes sense since the idea of the labels is that they work together to function as a system as whole. However, as on the right side of the network, there are parts where service providers are clustered together. We noticed something similar to this (also with service providers) in the K9 project. This could be explained by the behaviour of service providers. They have the responsibility of doing a small task, meaning that multiple service providers are needed to accomplish slightly bigger tasks.

## 5.2 Network properties

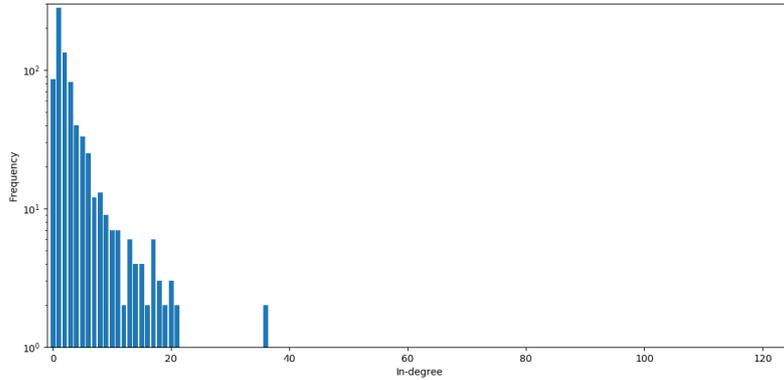
Since we are modelling software systems as networks to acquire the network features, we decided to describe the data in terms of data in two ways. The first is by looking at generic, high level properties of the network and the second by looking at smaller, local properties of the networks (motifs). The following two subsections describe these properties.

### 5.2.1 High level properties

The network properties of the datasets can be found in Table 8 and Table 9.

Dataset	$n$	$m$	Diam Gc	$n$ Gc	# components	Max. in-degree
K9Mail app	776	2834	9	746	26	115
Home3D	561	2932	6	491	71	156
Mars simulation	1834	9775	12	1632	86	316

Table 8: Network properties of the used datasets.  $n$  = number of nodes,  $m$  = number of edges, Diam Gc = diameter of the giant component,  $n$  Gc = number of nodes in the giant component and # components = number of components



**Figure 15: The in degree distribution of the K9Mail app.**

Dataset	Max. out-degree	Density	Avg. distance	Clust. Coef.	Degree assortativity
K9Mail app	63	0.0047	3.5739	0.3811	-0.1019
Home3D	79	0.0093	2.8764	0.3709	-0.1314
Mars simulation	72	0.0028	3.56	0.2852	-0.0702

**Table 9: Network properties of the used datasets continued. Clust. Coef. = the clustering coefficient.**

There are some observations to be made when looking at these properties. First, it appears that the diameter increases when the number of nodes increase even though the difference in average degree is not that big. Second, in each project most nodes are within the giant component. Third, all of the networks are sparse which actually is as expected as one of the quality measures of software systems is to keep dependence among classes as minimal as possible. This property is often assumed to be, and observed, in networks in the real world. Fourth, the clustering coefficient is remarkably high, since we do not expect triangular relationships in software. This can be due to the different types of edges, for example two subclasses having a dependency with each other (which results in a triangle).

Finally, the negative degree assortativity tells us that nodes with a high degree tend to be connected to nodes with low degrees. These observations seem to indicate that the software systems are built in a modular way where certain functionalities reside in a module/package. It seems that within each package there are a few important classes with a high degree which are related to multiple nodes with a low degree, which would explain both the low average distance and the degree assortativity.

The high clustering coefficient could mean that within packages, classes are highly dependent on each other. The increasing diameter could then be explained by the addition of more and more modules/packages to the systems as they get bigger and bigger. Since not every package is dependent on one another, the distances between certain classes gets bigger and bigger.

As seen in Figure 15 and Figure 16 both the in and out degree distributions follow a tailed degree distribution, meaning that the number of nodes with a low in- and out-degree is high, while the number of nodes with a high degrees is low. These is in line with our theory of modular systems with a few important classes with a high number of relationships in each module.

To validate our observation, that the software system is build modular in which each module contains a few classes (nodes) with a high degree which are connected to a multiple nodes with a low degree, we visualized the network of one of the projects (Home3D). We applied a community detection algorithm to distinguish the various communities within the network. The reason we used this technique was that a community in a network is similar to a module/package within a software system, making it an ideal technique to validate our observation. Furthermore, we let the size of each node be determined by its degree. This way we could see if each community consists of a few big nodes and a multiple small nodes.

As can be seen in Figure 17 the system consists of multiple (9) communities. Most of these communities have at least one node which is larger than the other nodes in its community, which is another sign that our theory is correct.

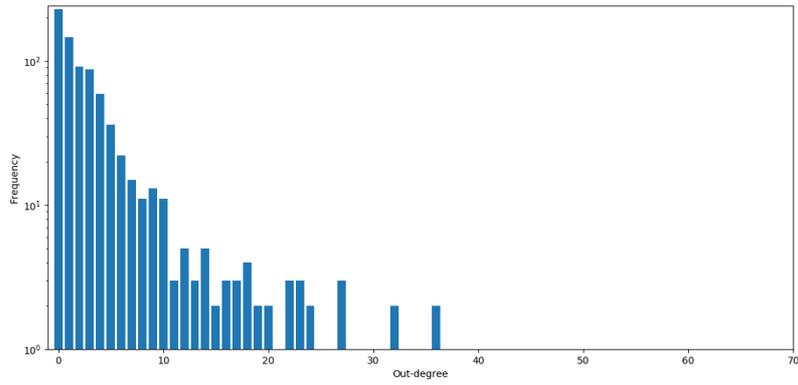


Figure 16: The out degree distribution of the K9Mail app.

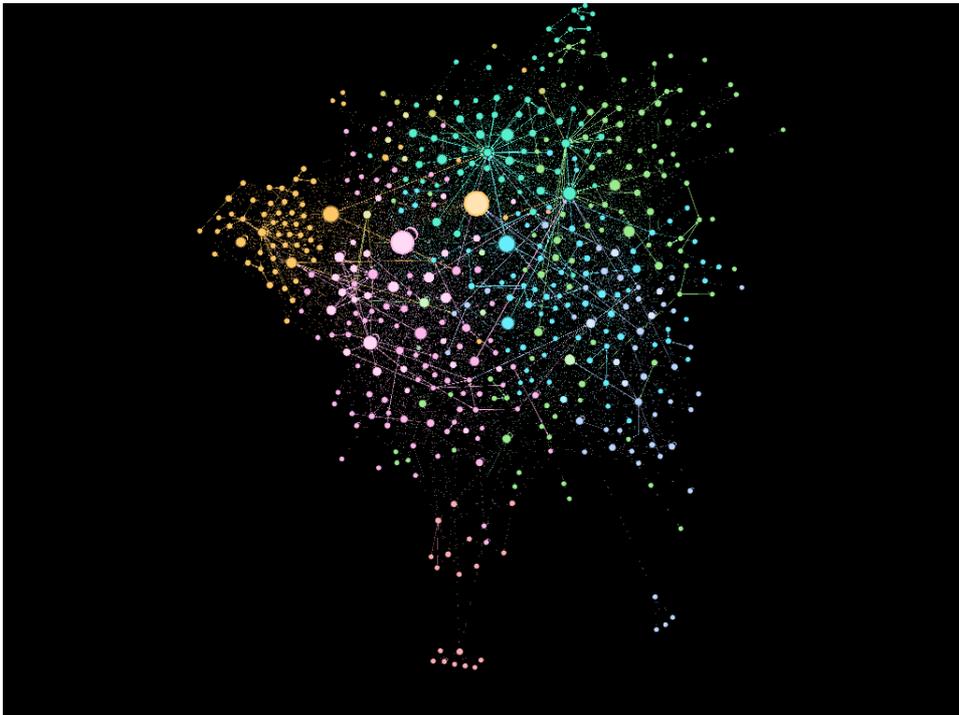
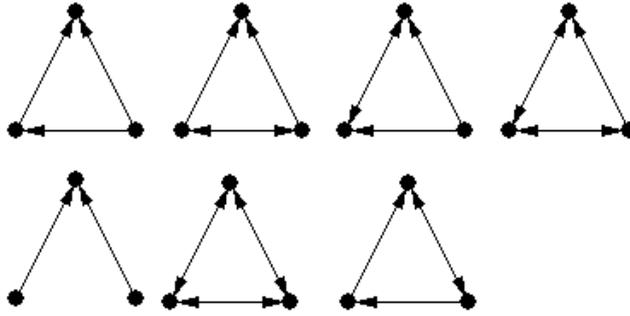


Figure 17: The giant component of the network of the Home3D project visualized. The color of each node displays its community, while the size of each node shows its out degree. The network is visualized using Gephi [3].



**Figure 18: All 3-motifs found in datasets. The motifs in the top row were found in each dataset, the bottom motifs were found in two out of the three datasets.**

### 5.2.2 Motifs

As described in Section 2, a motif is a frequent recurring significant pattern of interconnections. They are small networks that occur more frequent in a given network than in a random network of the same size. Because of our interest in the low level patterns found in the software systems, we decided to acquire both the 3- and 4-node motifs of the networks. We identified these motifs using FANMOD, which is a tool for fast network motif detection [34]. Because this tool does not allow for 2-node networks to be identified, and motifs bigger than four nodes took too long, we did not take these along in our analysis.

In total, seven 3-node motifs were found in the three datasets. Figure 18 shows all of the 3-node motifs found in the datasets. The motifs in the top row were found in each dataset, the bottom motifs were found in two out of the three datasets. What we found interesting, and slightly disturbing, about these motifs is that a most of them have two-way directed edges. In software engineering this is called bi-directional relationships, which means that two classes have knowledge about each other. This means that there exist a high coupling between these classes and is a sign of bad design. If this occurs just a few times it is not that big of a problem, but since motifs are frequent patterns it is a disturbing sign. Related to our earlier theory, these highly interrelated motifs are most likely found within the software modules/packages, and not between the different packages. It should be kept in mind that the motifs did not make use of the labelled edges, and thus each edge was seen as equal. Taking the edge labels into account should give more insights into the meaning of these motifs.

To see which motifs were most dominant in each dataset, we decided to take the top three 3-node motifs of each dataset and visualize them. Table 10 shows these top three along with the Z-score of each motif. The Z-score indicates how many standard deviations the frequency of the motif is away from the average frequency of the motif in 1000 random networks of the same size. As mentioned in Section 2, we generated these random networks using taking only the number of nodes and number of edges of the original network into account. It can be observed that most motifs in each of the top three per dataset is in the top three of at least on other dataset.

These 3-node motifs reveal that there are a lot of circular (triangular) and bi-directional relationships in the analysed projects. This means that there is a high coupling within these systems, where classes are highly dependent on one another. In software engineering, this is a sign of bad design as it decreases the maintainability of the system.

As mentioned before, we performed the same steps for 4-node motifs. In total 143 of these motifs were found in the three datasets. Of these 143 motifs, 52 were found in all three of the datasets, 42 were found in two of them and 49 were only found in one. Because of the size of the found motifs we decided not to visualize them all. Instead, we only took the top three motifs per dataset which can be found in Table 11. As opposed to the 3-node motifs, there is less overlap between the top three motifs in the three datasets. The only motifs that occur in multiple lists are found in the Home3D and the Mars simulation projects. What is interesting about these 4-node motifs is that their Z-score is a lot higher than with the 3-node motifs. This means that these 4-node motifs are very dominant in the software systems.

Nr.	K9Mail	Z-score	Home3D	Z-score	Mars	Z-score
1		202.76		21.87		89.56
2		30.89		16.47		42.83
3		22.8		13.83		33.96

Table 10: Top three 3-node motifs in the three datasets

Nr	K9Mail	Z-score	Home3D	Z-score	Mars	Z-score
1		3848.6		240.2		596.05
2		1851.7		168.35		450.68
3		1424		108.24		352.39

Table 11: Top three 4-node motifs in the three datasets

Just as with the 3-node motifs, we can see that there are circular relationships when looking at the 4-node motifs. Again, this hints to a bad design which can lead to maintainability problems. However, opposed to the fully connected triangles found in the 3-node motifs, we can see that six out of the nine presented motifs are not fully connected. Translating both the 3-node motifs and the 4-node motifs to the field of software engineering, we can see that there are a lot of small parts of the system in which the coupling is high. This is a sign of bad software quality as it makes the system harder to maintain.

### 5.3 Semantic software properties

The number presented in Table 12 describe the systems in terms of semantic software properties. In contrast to the network properties, these semantic software features have a bit more of a difference between the datasets. This tells us that the size of the classes vary between the projects. For example we can observe that, on average, the classes in the Home3D project are the biggest. So even though it has the least classes,

each class itself is larger in terms of number of attributes and number of methods than the classes in other projects. Compared to the other two projects, the K9mail app has the smallest classes. Knowing these facts is not enough to say anything about the design quality. In general having a large number of methods might seem that the class does more than one thing (which is an indication of bad design quality), but it could also mean that large methods are split up in multiple smaller methods (which is one of the principles of clean code [21]). For our experiments the variation in class sizes is a good thing, since it makes our training data more diverse.

Dataset	# classes	Avg. methods	Avg. attributes	Total attributes	Total methods
K9Mail app	776	6.61	4.62	3587	5131
Home3D	561	9.16	7.84	4400	5141
Mars simulation	1834	8	7.23	13251	14681

**Table 12: Average number of attributes and methods in the two datasets**

## 6 Experiments and results

In this section the final results of our experiments will be discussed. The section is split up in a couple of subsections. The first subsection discusses the experimental set-up and implementation details of the feature extracting and selection process. We will explain the hardware used, the software versions used and the implementation details on the process of extracting and selecting features. The second subsection discusses both the training and test results per algorithm (logistic regressions, decision tree and random forest).

In the third section, we give a summary of all the results. After that, we continue with one section describing the top ten most important features of the best performing algorithm. The fifth subsection is dedicated to the mistakes made by the best performing algorithm. While looking at misclassified cases, we noticed that they might be classified correctly when adding behavioural features to the algorithm. To verify this hypothesis, we performed one more experiment in which we added two behavioural features to the feature set.

### 6.1 Experimental set-up, feature extraction and feature selection

In order to measure the results equally, all experiments were performed on a 64-bit Windows 10 laptop with 8GB Ram, an Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz. The experiments were performed using Jupyter notebook version 4.2.1 running on Anaconda 4.1.1 (64-bit) and Python 3.5.2.

The libraries used in these experiments can be found in Table 13.

Library	Version
pandas	0.18.1
numpy	1.13.1
scikit-learn	0.19.0

**Table 13: Python library versions used in experiments**

#### 6.1.1 Scikit-learn

Scikit-learn is a Python module that contains a wide range of state-of-the-art machine learning algorithms. The machine learning algorithms are based on the latest papers in the field, and makes it easier to implement them in new projects [27].

In this thesis we chose to use this module for the following reasons: it saves programming time, it is well documented (including references to papers and external sites), one of the studied books ([16]) walks through a big part of the module and shows how to work with it.

All of the algorithms we use in our experiments are implemented in scikit-learn. This does not mean that the algorithms can be used out of the box. The algorithms have certain hyper-parameters (parameters which are not directly learned by the algorithm, explained in Section 2) which can, and must, be tuned to improve the classification results. Because the number of possible hyper-parameter settings is enormous, it was decided to use gridsearch to find the optimal combination. As mentioned in Section 2, gridsearch is an automatic exhaustive search of (almost) all hyper-parameter value combinations.

Even though using gridsearch saves time (because it is automated), it still takes a long time to compute. This is because the algorithm needs to be trained and evaluated  $h$  times, where  $h$  is the number of possible hyper-parameter combinations.

Finally each algorithm was trained again with the optimal hyper-parameters and evaluated on a test set. The results of the algorithms with the optimal hyper-parameters on the train and test sets can be found in Section 6.2.

### 6.1.2 Feature extraction and network creation

For the generation of the networks from source code and for the extraction of the semantic software features we used the Java library Spoon [26]. We used this library to transform the source code into its intermediate representation, the AST.

Spoon transforms Java code to an AST representation which can then be used to either analyse the system, or modify the system. In the case of this thesis, it is used for analysing purposes. The tool can be used either by the command line or by calling the libraries from a Java project. The latter has been done in this thesis. The code used to parse the projects (and extract the features) can be found in Appendices C and D. The AST was then analysed as described in Section 4.

The reason Java was chosen as programming language for this part was that the projects were written in Java and the Spoon library can be used via Java. Furthermore we were already familiar with the language.

### 6.1.3 Feature selection

As mentioned in Section 4.3.4, we took two approaches to selecting the right features for the algorithms. The first approach was to analyse the individual features one-by-one and see if clear decision boundaries could be found. A couple of features did show a small distinction between certain classes. For example, Figure 19 shows that higher closeness centrality values are more frequent in service providers and interfacers.

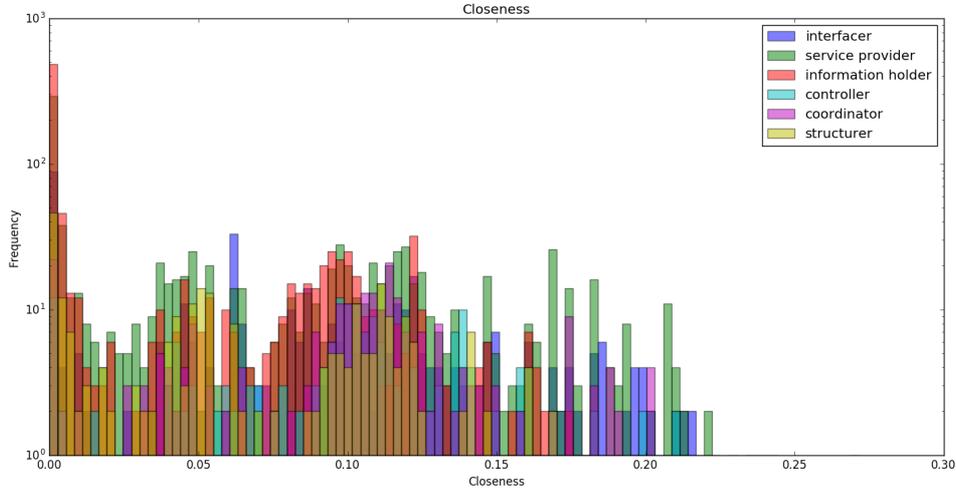
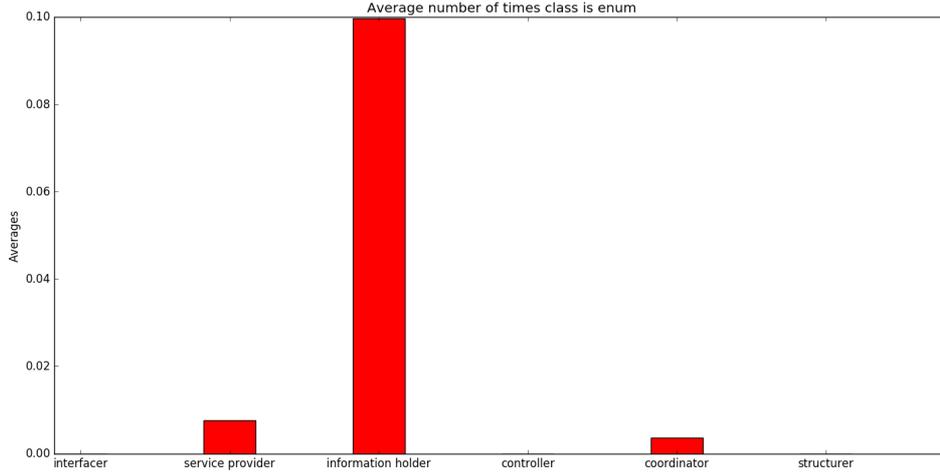


Figure 19: Closeness centrality values per label

Figure 20 plots the average number of times a certain label is an enum. It can be seen, as is expected, that most enums are information holders, while at the same time none of them are controllers. The problem with this approach was that there was not a single feature that showed a clear decision boundary between the six labels. What we did learn from these plots was that there were two features that only had a single value and thus could be dropped from the feature sets. These features were: *other attributes* and *other methods*. All histograms, except for the two removed features, can be found in Appendix B.



**Figure 20: Average number of times the isenum value is set to true per label**

We decided to go for a more practical approach next. The approach was to perform classification algorithms on the full feature sets and see what influence each feature has on the end result based on the best performing algorithm. In case this would be the logistic regression algorithm, it would mean that we would look at the weights used per features (the closer to zero the less influence the feature has). For the decision tree and the random forests the gini importance is used. Before we will explain the gini importance, we will first discuss the gini impurity measure.

Gini impurity measures how "pure" a node (note that a node is either a split on a feature or a leaf node with a prediction) in a decision tree is. A node is set to be pure when all training instances it is applied to represent the same class. A pure node will have gini score of 0. Gini impurity is calculated by [16]:

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

Where  $p_{i,k}^2$  is the ratio of class  $k$  instances among the training instances in the  $i$ th node and  $n$  is the number of classes (labels).

Measuring the importance of a feature is done as follows: "the importance of a variable  $X_m$  for predicting  $Y$  by adding up all the weighted impurity decreases  $p(t) \Delta i(s_t, t)$  for all nodes  $t$  where  $X_m$  is used, averaged over all  $N_T$  trees in the forest:

$$Imp(X_m) = \frac{1}{N_T} \sum_T \sum_{t \in T: v(s_t) = X_m} p(t) \Delta i(s_t, t)$$

where  $p(t)$  is the proportion  $N_t/N$  of samples reaching  $t$  and  $v(s_t)$  is the variable used in split  $s_t$ " [20].

By trial and error the final feature set was finally selected. This feature set will be discussed in more detail in Section 6.2.

## 6.2 Training

In our experiments we used three different algorithms, which were logistic regression, decision tree and random forest. This subsection describes the results of each of these algorithms. Note that one of the things we want to measure are the results acquired using different features sets as input to machine learning algorithms for our problem. To measure this, we present three results per algorithm: the results using only semantic software features, the results using network features, and the results using the combined dataset.

To compare the accuracies presented in the next few subsections, we performed two runs of 10,000 iterations of making random predictions. In the first run we made 10,000 completely random predictions on the test set (so selecting a label at random) which gave an average accuracy of 17.48%. We then did another run of 10,000 random predictions on the test set. The difference this time was that we randomly selected the labels based on the label distribution (presented in Section 5.1 and Section 4.3.2). This approach gave an average accuracy of 23.46%. In both cases the average and maximum accuracy over 10,000 iterations was roughly the same.

We also mentioned that we performed a gridsearch to find the optimal hyper-parameters per algorithm-feature set combination. The optimal hyper-parameters together with their descriptions can be found in Appendix A.

### 6.2.1 Logistic regression

The accuracy scores reached on each set per input feature set can be found in Table 14.

Dataset	Accuracy train set	Accuracy test set
Semantic software features	65.03%	54.64%
Network features	37.15%	28.82%
Full feature set	66.92%	55.59%

**Table 14: Results logistic regression using different input features**

The numbers in the table show that the logistic regression algorithm performs better than making random predictions. More than half of the instances are predicted correctly using both the semantic and the full feature set. Using only the network features results in an accuracy which is only slightly better than random predictions. Furthermore, it can be observed that the algorithm gives the highest accuracy using the full feature set. Finally, the algorithm seems to be overfitting slightly, since the difference between the accuracy on the train set and the accuracy on the test set is about 11%.

Since logistic regression is a linear algorithm (it uses the form  $x_1w_1 + \dots + x_iw_i + b$  where  $x_i$  is the  $i$ th feature of an example and  $b$  is a constant value), and the algorithm only produces an accuracy of 55% we can say that the distribution of labels is not linearly separable. This means that there are no linear hyperplanes which separate the data correctly.

### 6.2.2 Decision tree

The accuracies reached on each set are presented in Table 15. We can see that the decision tree performs better than the logistic regression algorithm. Besides that, we can again observe that the highest accuracy is reached when the full feature set is used. Furthermore, in Section 1 and Section 2 we discussed the concept of overfitting. In the numbers presented in the table, the overfitting phenomena is clearly observable. The algorithm produces almost perfect results on the train set, but has a prediction accuracy of more than 30% less on the test set.

Dataset	Accuracy train set	Accuracy test set
Semantic software features	93.12%	62.68%
Network features	88.15%	45.35%
Full feature set	95.3%	63.89%

**Table 15: Results decision tree using different input features**

### 6.2.3 Random forest

The accuracies reached on each set are presented in Table 16. Just as with the decision tree algorithm this algorithm is also overfitting the training data. However, we can observe that the difference between the train accuracy and the test accuracy is slightly less using this algorithm, meaning that the algorithm generalizes better to unseen data. Furthermore, we can observe that the random forest algorithm reaches the highest test accuracy of all three algorithm. Next to this, we can again observe that the highest accuracy is reached when using the full feature set.

Dataset	Accuracy train set	Accuracy test set
Semantic software features	95.7%	66.14%
Network features	91.67%	49.76%
Full feature set	97.75%	71.81%

**Table 16: Results random forest using different input features**

### 6.3 Results

Looking at the overall results, it shows that the two tree based algorithms are highly overfitting (difference between training predictions and test predictions is around 30%) the data. The logistic regression on the other hand fits the data better with a difference of about 11% between the training data and the test data. Furthermore, it is interesting to see that the performance of each algorithm is better when trained on the full feature set instead of just the network or the semantic software features. Finally, the random forest algorithm outperforms the other two and thus we will continue our analysis with this algorithm.

### 6.4 Most important features

After acquiring the results, we looked into the most important features. Since the random forest performed best, we decided to focus on this algorithm using the optimal hyper-parameters. The feature importances were determined by their gini-importance value, which was explained in Section 6.1.3. The top ten most important features to the algorithm, together with their gini-importance value between brackets (note that if all features are equally important their gini-importance value would be around 0.023), were:

1. Name ends with *-er* or *-or* (0.071392)
2. Delta attributes from average (0.065069)
3. Closeness centrality (0.048196)
4. Name ends with *Controller* or *Manager* (0.048055)
5. Betweenness centrality (0.045990)
6. Number of methods (0.041980)
7. Attribute to method ratio (0.041029)
8. Number of list methods (0.040952)
9. Ratio accessors and mutators to total number of methods (0.040948)
10. Out-degree (0.040611)

As can be seen in the enumeration above, three out of the top ten most important features are network features (closeness centrality, betweenness centrality and out-degree). Two of them are even in the top five most important features. Another noteworthy observation is that the number of methods feature is in the top ten, while the ratio methods to attributes and number of methods from average is not. We expected that these two features would be more important as they scale better over multiple projects than just the number of methods.

It should also be noted that the results acquired were acquired by using all features except the two which we left out (mentioned in Section 6.1.3). Reducing the number of features even more resulted in a lower accuracy.

### 6.5 Errors made

Finally, we decided to zoom in on the misclassified cases. We decided only to look at the predictions of the random forest algorithm, since this algorithm performed best on the test set. We looked at the error at two ways. First, we got an overview of the errors made per class by analysing the confusion matrices using the three confusion matrices. This gave us insights into the labels that were predicted more accurately when using the full feature set instead of just the semantic software features. Afterwards, we looked at the source code of a couple of randomly selected misclassified cases in order to figure out what made it difficult for the algorithm to classify these instances.

### 6.5.1 Confusion matrices

To get more details in the increased accuracy of the algorithm using the full feature set we decided to look at the confusion matrices of the predictions made by the random forests. This way we can analyse what labels were predicted more accurate after adding the network features. Table 17 shows the confusion matrix of the predictions made by the random forest algorithm using only semantic software features. Table 18 shows the confusion matrix using only network features and Table 19 shows the confusion matrix using the full feature set. In each confusion matrix the rows represent actual labels and the columns represent the predictions. Each cell  $C_{ij}$  of matrix  $C$  represents the number of instances of label  $i$  predicted to be  $j$ . The numbers on the diagonal are correct predictions.

	Controller	Coordinator	IH	Interfacer	SP	Structurer	% correct
Controller	16	9	2	2	6	5	<b>40%</b>
Coordinator	7	19	6	10	11	3	<b>33.9%</b>
IH	1	4	162	4	10	14	<b>83.1%</b>
Interfacer	3	11	7	44	14	2	<b>54.3%</b>
SP	7	16	12	13	153	9	<b>72.9%</b>
Structurer	4	2	4	1	6	36	<b>67.9%</b>

**Table 17: Confusion matrix random forest predictions on test set using semantic software features. Rows are actual labels, columns are predictions made. IH stands for Information Holder and SP for Service Provider.**

	Controller	Coordinator	I.H.	Interfacer	S.P.	Structurer	% correct
Controller	8	9	5	11	5	2	<b>20%</b>
Coordinator	8	26	3	4	10	5	<b>46.4%</b>
IH	4	4	129	10	35	13	<b>66.2%</b>
Interfacer	3	12	21	25	17	3	<b>30.9%</b>
SP	13	11	38	27	109	12	<b>51.9%</b>
Structurer	5	0	11	7	11	19	<b>35.8%</b>

**Table 18: Confusion matrix random forest predictions on test set using network features. Rows are actual labels, columns are predictions made. IH stands for Information Holder and SP for Service Provider.**

	Controller	Coordinator	IH	Interfacer	SP	Structurer	% correct
Controller	19	8	0	4	6	3	<b>47.5%</b>
Coordinator	7	23	5	4	14	3	<b>41.1%</b>
IH	2	5	163	4	10	11	<b>83.6%</b>
Interfacer	1	8	8	50	13	1	<b>61.7%</b>
SP	5	7	17	16	162	3	<b>77.1%</b>
Structurer	0	4	4	0	6	39	<b>73.6%</b>

**Table 19: Confusion matrix random forest predictions on test set using full feature set. Rows are actual labels, columns are predictions made. IH stands for Information Holder and SP for Service Provider.**

When looking at these confusion matrices we can make a few statements. The first is that the label coordinator is actually predicted better when using only the network features. This means that the semantic properties of the coordinator are similar to the semantic properties of other labels. To be precise, the coordinator shares semantic software features with the Information Holder and the Service Provider. The reason we state this is that we see an increase in coordinators predicted to be Information Holders and Service Providers when using the full feature set compared to using only network features.

Secondly, we see that every other label performs best using the full feature set. This means that both semantic and network features are necessary to distinguish between these labels. Third, we can see in the confusion matrices, the accuracy of the predictions of each label increases when adding network features. To be more specific, per label the following increases are acquired:

- *Controller*: 7.5%
- *Coordinator*: 7.2%
- *Information Holder*: 0.5%
- *Interfacer*: 7.4%
- *Service Provider*: 4.2%
- *Structurer*: 5.7%

Even though the accuracy of each predicted label increased, the most increase can be found in the *Controller*, *Coordinator* and *Interfacer* labels. When looking at our expectations mentioned in Section 4.2, we expected network features to add value to these classes.

### 6.5.2 Misclassified cases

Table 20 shows the class names of the misclassified cases together with the project it belongs to, the label we assigned to it and the predicted label. The last column briefly explains why we think the class is predicted as it is. Note that in the table, the names of the classes are abbreviated as follows: *IH*: Information Holder, *SP*: Service Provider, *Coord.*: Coordinator, *Contr.*: Controller and *Struc.*: Structurer.

Class name	Project	Expected	Predicted	Explanation
EmergencySupplyMission	Mars	Contr.	Coord.	The class has a lot of relations to other user defined classes. However, it does most of its work itself instead of delegating it.
MasterClock	Mars	IH	Coord.	Has a lot of relations, but its purpose is to contain information and not to delegate
MailItem	K9	IH	SP	Has just two methods which are both focussed on its attributes. Does nothing more than hold information.
ModifiedPolyline	Home3D	Coord.	IH	Has the semantic structure of an information holder, but it actually delegates its tasks instead of altering its own attributes.
EmptyGraph	Mars	Struc.	Contr.	Contains a lot of classes, however all classes are dedicated to relationship management between classes.

**Table 20: Some randomly selected misclassified instances**

Looking at the observations presented in Table 20, most wrongly labelled classes actually do fit the predicted labels based on their features. To classify them correctly behavioural features, described in Section 3, should be taken into account as they contain more information about the content of the methods of a class. However, because behaviour of a class is not described in UML class diagrams, they were not taken into account with our initial feature extraction/selection.

Next to this, we mentioned in Section 4.1 that we deal with real world software systems which are not designed ideally. This means that there are certain classes that do not abide by a single label. In certain cases, this could be what makes it difficult for the algorithms to classify the instances correctly. This could be dealt with in two ways. The first is changing the classification to a multi-class, multi-label problem meaning that multiple classes can be predicted per class. However, since having multiple archetypes is a form of bad design, this would bring too much overhead with it. The second possible solution to the problem is the introduction of an extra label called *bad design* (or something similar) which means that the design of the

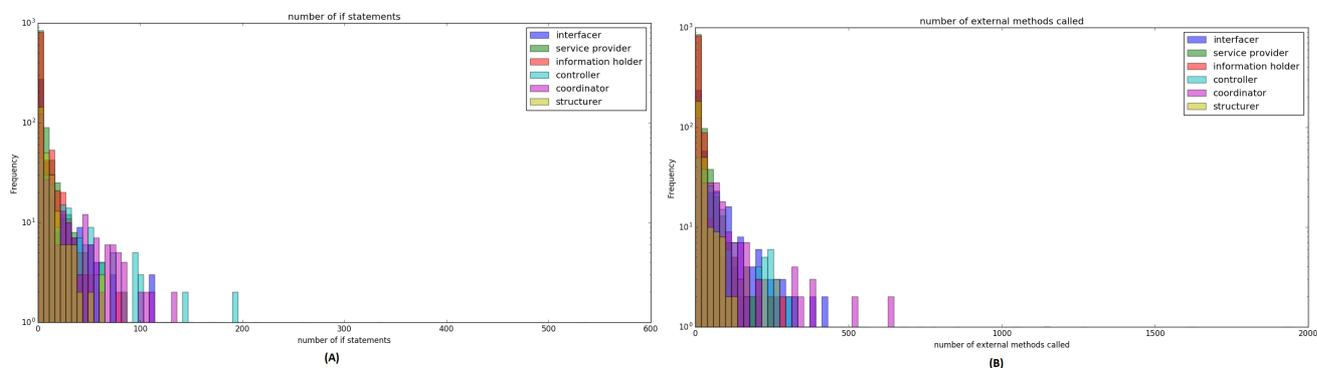
class is flawed and thus should be labelled as such (which in turn gives the developers an indication of classes that need to be improved).

## 6.6 Addition of behavioural features

As mentioned in the previous section, we noticed that behavioural features might be able to make the difference in various misclassified cases. To be more specific, we think that, among others, the following features can improve the algorithms performance:

- Number of *if* and *switch* statements
- Number of work delegated
- Type of communication between classes (for example, does a class only contain small methods in which it calls a method of another class with the same method signature)

We decided to perform one more experiment using the random forest algorithm with the same hyper-parameters as in the previous experiments. The difference is that we added two more features to our total feature set. These feature(s) were: *number of if statements* and *number of external function calls made*. We plotted these feature values in histograms, which can be seen in Figure 21



**Figure 21: Behavioural features per label. (A) shows the number of if statements a class has. (B) shows the number of external methods called per class.**

As can be seen in the histograms the behavioural features do not make clear distinctions between all six of the labels. However, it can be observed that the higher values belong to certain labels (controllers have more if statements, while coordinators call more external methods as expected).

These features led to an additional increase of about 3%, making the new accuracy 74.03% on the test set. This confirms our hypothesis that the addition of behavioural features next to the semantic and network features will improve the prediction results.

It should be stressed that we did not perform a new gridsearch to find the new optimal hyper-parameters or did anything else but extracting the features, re-training the algorithm and making new predictions on the test set. Tuning the hyper-parameters might lead to a small performance boost.

## 7 Conclusion and discussion

In this section we will present our conclusions based on our results and assumptions we had up front. Here we will answer the research question presented in Section 1 (Does extending software class properties with metrics of network science improve the accuracy of machine learning algorithms in the task of labelling classes automatically?), and draw conclusions on the following subjects: the best performing algorithm for this task, the best feature set for this task and the improvement in accuracy when using both network and semantic software features compared to the feature set using only semantic software features. We will end this section with a discussion about the label distribution.

Based on our experiments, we can conclude that the construction a feature set using both network features as well as semantic software features produces the highest prediction accuracy using the machine learning algorithms. As presented in Section 6, The algorithms improved with  $\pm 1$  to  $\pm 5\%$  by the network features, meaning that the extension of software class properties with metrics of network science does improve the accuracy of machine learning algorithms in our task. Also, not all network properties are equally relevant. The features that added the most value were: closeness centrality, betweenness centrality and out-degree. Opposed to our expectations, the in-degree and the eccentricity did not add as much value to our predictions.

Furthermore, the classes that reaped the most benefit from the addition of network features were: controller, coordinator and interfacier. As described in Section 4.2 we expected network features to add value to these classes. This can be explained by their purpose. The controller has a lot of responsibilities which means that it knows multiple other classes. The same holds for the coordinator. The interfacier connects parts of the system which is a network feature instead of a semantic feature.

Next to this, we can conclude that the best algorithm we applied to our problem was the random forest algorithm. It was the only algorithm that returned an accuracy higher than 70%. Also, this algorithm got the most performance gain from the network features. However, this is algorithm is the most difficult to interpret (hard to understand how it made its decisions). As mentioned in Section 1, our selection of the best algorithm was determined by its performance first, and interpretability second only if the performance was similar. Because the difference in performance is bigger than the threshold of one percent, we decided to select the random forest algorithm.

Based on the observed mistakes made, we conclude that the algorithms performance is decent ( better than predicting at random). However, as we mentioned in Section 6.5 and shown in Section 6.6, we believe that adding behavioural features to the algorithm will improve this performance as we already saw a slight increase in performance by adding two easy to extract behavioural features.

Finally, in Section 5 we presented the label distribution in the datasets. We believe that this can be explained by the characteristics of each label. Information holders only hold information and service providers do small tasks which can be called by different other classes. Since these are small classes which are used by various other classes, it makes sense that they are mostly present in the datasets. Controllers, coordinators and structurers on the other hand are classes with bigger responsibilities of which fewer are needed within a system. Finally there are the interfaciers. They are responsible for connecting different components of the system, which makes them less likely to occur. However they are also responsible for the interaction with the user, which increases the number of classes belonging to this label.

## 8 Future work

As we mentioned in Section 6.5, there are classes which are difficult to label because they do not confine to a single archetype. Since this could be seen as bad design, a possible way to improve the results is to add a label *bad design* to indicate such classes. The experiments in this thesis could then be repeated with the extra label to see if this improves the accuracy and at the same time to see whether design flaws can be detected in such way. Also, we saw that the accuracy increases when extending the feature set with behavioural properties of the classes. This can be researched further using more behavioural properties.

Furthermore, we only used a small set of network metrics as input features. These metrics where chosen because we thought, as explained in Section 4.2, they can be used to label the classes. However, there are more network metrics available which might improve the accuracy of the algorithms. Thus our results only serve as a lower bound. Further research could experiment with different network metrics to see whether the results are improved by adding them.

Next to this, we also observed that network properties and methods might be able to determine the structural quality of software systems. However, since we only looked at three software systems we do not have enough information to make general statements. As so, more research can be done into using network measures/properties to determine the quality of software design and/or design choices made. Next to this, the same data can be used to see if common properties can be found in software systems. These two possible future research topics are both about the high level properties as well as the motifs.

On a final note, we want to mention that all results acquired and observations made during this thesis are made because of the combination of different research areas (network science, software engineering and machine learning). Even though this thesis is definitely not the first research to combine different research areas, it can (together with the rest of these researches) be used as a stepping stone for more studies that combine various research areas to gain new insights out of existing data.

## 9 Other contributions

As could be read in this thesis, we developed various scripts and labelled multiple datasets during the process of our experiments. We decided to turn one of these scripts into a tool for future use. Furthermore, we decided to share the labelled datasets so that they can be used in future research. This final section gives a brief overview of the datasets and the tool (how to use them and where to find them).

### 9.1 Datasets

As can be read in Section 5, we manually labelled three datasets during this thesis. These datasets together contain more than 3000 labelled classes which can be used in future research. We shared these datasets on <https://github.com/Xavyr-R/thesis-labelled-files> and <https://git.liacs.nl/s1900110/labelled-classes>.

The labelled datasets are provided as three comma separated .csv files each having the following structure: Class (name of the class), Role (label of the class), Fullname (namespace with class name to locate file in project directory).

### 9.2 Network generator

The first script we turned into a tool is the network generator. As explained in this thesis, this tool is written in Java and makes use of the Spoon library to parse the .java files. The tool consist of two frames. In the first frame you see an overview of given input paths (to be explained in a minute), the given output path and file name and the buttons to add an input path, remove an input path, close the tool and generate the edgelists. The second frame is opened when adding a new input path. This frame consist of a folder selector, a save button and a close button.

To use the tool take the following steps: Open the tool, add all input paths to the input path list, select an output path and an output file name and press on the generate button. The tool will then go through all the input paths, parse all .java files and generate an edgelist. Finally, it will write the edgelist to a csv file in the specified output folder with the given output file name.

The input paths should be paths to folders containing .java files. Since the code crawls through subdirectories as well, giving root directories is sufficient. It is programmed to ignore everything but .java files. For .java files there is one restriction, which is that the file name does not contain **Test** (case insensitive), to prevent it from parsing test files. At the moment it is not possible to give other words to ignore.

This tool can be found on: <https://github.com/Xavyr-R/Java-project-to-edgelist> and <https://git.liacs.nl/s1900110/java-to-network-tool>. Note that this tool does not contain any validation on input folders (for example if the folder even exist) or anything, we expect the user to use it as designed.

## References

- [1] Z. Balanyi and R. Ferenc. Mining design patterns from c++ source code. In *Proceedings of the International Conference on Software Maintenance, ICSM '03*, Washington, DC, USA, 2003. IEEE Computer Society.
- [2] D. Ballis, A. Baruzzo, and M. Comini. A rule-based method to match software patterns against uml models. *Electronic Notes in Theoretical Computer Science*, 219:51–66, 2008.
- [3] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks, 2009.

- [4] N. Benchettara, R. Kanawati, and C. Rouveirol. Supervised machine learning applied to link prediction in bipartite social networks. *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 326–330, 2010.
- [5] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2004.
- [6] L. Breiman. Heuristics of instability and stabilization in model selection. *The Annals of Statistics*, 24(6):2350–2383, 1996.
- [7] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [8] L. D. C. Comparative analysis of random forest, rep tree and j48 classifiers for credit risk prediction. *IJCA Proceedings on International Conference on Communication, Computing and Information Technology*, ICCCMIT 2014(3):30–36, 2015.
- [9] L. C.Freeman. A set of measures of centrality based upon betweenness. *Sociometry*, 40(1):35–41, 1977.
- [10] L. C.Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1978.
- [11] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16(1):321–357, 2002.
- [12] J. Dong, Y. Zhao, and T. Peng. A review of design pattern mining techniques. *International Journal of Software Engineering*, 19(6):823–855, 2009.
- [13] N. Dragan, M. L. Collard, and J. I. Maletic. Automatic identification of class stereotypes. *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.
- [14] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele. Design pattern mining enhanced by machine learning. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, Washington, DC, USA, 2005. IEEE Computer Society.
- [15] R. Ferenc, rpd Beszdes, M. Tarkiainen, and T. Gyimthy. Columbus - reverse engineering tool and schema for c++. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, ICSM '02, Washington, DC, USA, 2002. IEEE Computer Society.
- [16] A. Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., St Georges, Farnham, UK, 1 edition, 2017.
- [17] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez. The quest for open source projects that use uml: Mining github. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, New York, NY, USA, 2016. ACM.
- [18] S. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica*, 31(3):249–268, 2007.
- [19] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3 edition, 2004.
- [20] G. Louppe, L. Wehenkel, A. Sutera, and PierreGeurts. Understanding variable importances in forests of randomized trees. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, USA, 2013. Curran Associates Inc.
- [21] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.
- [22] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.
- [23] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.
- [24] L. Moreno and A. Marcus. Jstereocode: Automatically identifying method and class stereotypes in java code. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, New York, NY, USA, 2012. ACM.

- [25] P. Norvig and S. J. Russell. *Artificial Intelligence: A Modern Approach*. Pearson Education Limited, London, 3 edition, 2016.
- [26] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2015.
- [27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and douard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [28] R. Rifkin and A. Klautau. In defense of one-vs-all classification. *Journal of Machine Learning Research*, 5:101–141, 2004.
- [29] G. Robles, T. Ho-Quang, R. Hebig, M. R. V. Chaudron, and M. A. Fernandez. An extensive dataset of uml models in github. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR ’17, Piscataway, NJ, USA, 2017. IEEE Press.
- [30] U. Tekin and F. Buzluca. A graph mining approach for detecting identical design structures in object-oriented design models. *Science of Computer Programming*, 95(P4):406–425, 2014.
- [31] F. Thung, D. Lo, M. H. Osman, and M. R. V. Chaudron. Condensing class diagrams by analyzing design and network metrics using optimistic classification. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, New York, NY, USA, 2014. ACM.
- [32] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006.
- [33] S. Vincenzi, M. Zucchetta, P. Franzoi, M. Pellizzato, F. Pranovi, G. A. D. Leo, and P. Torricelli. Application of a random forest algorithm to predict spatial distribution of the potential yield of ruditapes philippinarum in the venice lagoon, italy. *Ecological Modelling*, 222(8):1471–1478, 2011.
- [34] S. Wernicke and F. Rasche. Fanmod: a tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 2006.
- [35] R. J. Wirfs-Brock. Characterizing classes. *IEEE Software*, 23(2):9–11, 2006.
- [36] X. Yan and J. Han. Closegraph: Mining closed frequent graph patterns. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’03, New York, NY, USA, 2003. ACM.

## A Optimal hyper-parameters

This appendix contains the optimal hyper-parameters for each combination of algorithm and feature set. The appendix has three subsections in which the optimal hyper-parameters are presented together with their meaning. Each subsection presents the hyper-parameters per algorithm

### A.1 Logistic regression

The hyper-parameters which resulted in the highest accuracy for the logistic regression algorithm can be found in Table 21.

Parameter	Value full set	Value network set	Value semantic software set
penalty	l1	l1	l2
fit_intercept	True	True	True
C	1	1	1
dual	False	False	True
solver	liblinear	liblinear	newton-cg
class_weight	None	balanced	None

**Table 21: Optimal hyper-parameters for logistic regression on each dataset**

Table 22 presented the meaning of each of the hyper-parameters of the logistic regression algorithm and the possible values they can have.

Parameter	Meaning	Possible values
penalty	Specifies the norm used in the penalization	l1, l2
fit_intercept	Specifies if a constant should be added to the decision function	True, False
C	Regularization strength	Positive floats, smaller values mean stronger regularization
dual	Dual or primal formulation	True, False
solver	Algorithm used for optimization	newton-cg, lbfgs, liblinear, sag, saga
class_weight	Weights associated with classes	dictionary (class.label: weight), None or Balanced

**Table 22: Hyper-parameters of logistic regression explained**

## A.2 Decision tree

The hyper-parameters which resulted in the highest accuracy for the decision tree algorithm can be found in Table 23.

Parameter	Value full set	Value network set	Value semantic software set
min_samples_leaf	2	2	2
max_leaf_nodes	None	None	None
splitter	best	best	best
max_depth	50	50	50
criterion	entropy	gini	gini
class_weight	None	balanced	balanced
max_features	None	None	None

**Table 23: Optimal hyper-parameters for the decision tree on each dataset**

Table 24 presented the meaning of each of the hyper-parameters of the logistic regression algorithm and the possible values they can have.

Parameter	Meaning	Possible values
min_samples_leaf	The minimum number of samples required to be at a leaf node	int and float above 0
max_leaf_nodes	The maximum number of leave nodes allowed	integer, None
splitter	Strategy used to split at each node	best, random
max_depth	Maximum depth of the tree	integer, None
criterion	Function used to measure quality of split	gini, entropy
class_weight	Weights associated with classes	dictionary, list of dictionaries, None
max_features	Number of features to consider when looking for best split	integer, float, None

**Table 24: Hyper-parameters of decision tree explained**

## A.3 Random forest

The hyper-parameters which resulted in the highest accuracy for the random forest algorithm can be found in Table 25.

Parameter	Value full set	Value network set	Value semantic software set
n_estimators	15	15	15
max_depth	75	100	25
max_leaf_nodes	None	None	None
criterion	gini	entropy	gini
oob_score	False	True	False
min_samples_leaf	2	2	2
class_weight	None	balanced	None
max_features	5	5	5
min_samples_split	2	2	2

**Table 25: Optimal hyper-parameters for the random forest on each dataset**

Table 26 presented the meaning of each of the hyper-parameters of the logistic regression algorithm and the possible values they can have.

Parameter	Meaning	Values
n_estimators	Number of trees in the forest	integer
max_depth	Maximum depth of the tree	integer, None
max_leaf_nodes	Maximum number of leave nodes allowed	integer, float, None
criterion	Function used to determine quality of a split	gini, entropy
oob_score	Whether to use out-of-bag (not trained on) samples to estimate generalization accuracy	True, False
min_samples_leaf	Minimum number of samples required to be at a leaf node	integer, float
class_weight	Weights associated with the classes	dictionary, list of dictionaries, None
max_features	Number of features to consider when looking for the best split	integer, float, None
min_samples_split	Minimum number of samples required to split an internal node	integer, float

**Table 26: Optimal hyper-parameters for the random forest on each dataset**

## B Histograms individual features

This appendix presents all histograms of the individual features that are not presented in the thesis itself. The appendix is split up into two subsections, the first containing the network features and the second containing the semantic software features.

### B.1 Network features

The histograms of all network features, except for the closeness centrality, used this thesis can be found in this subsection.

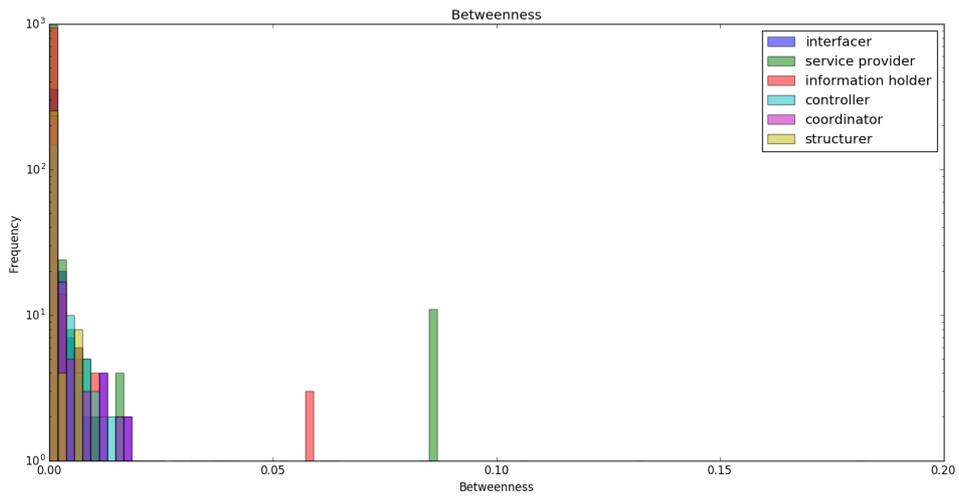


Figure 22: Betweenness centrality values per label

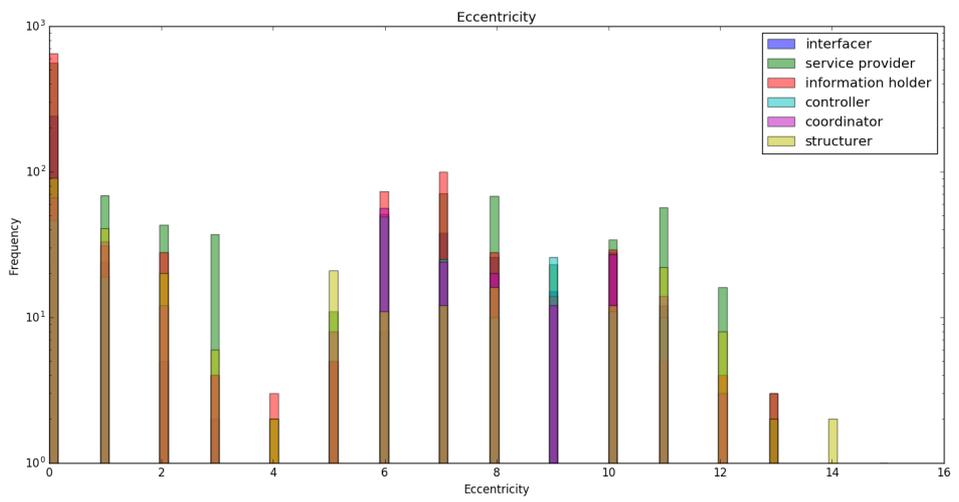


Figure 23: Eccentricity values per label

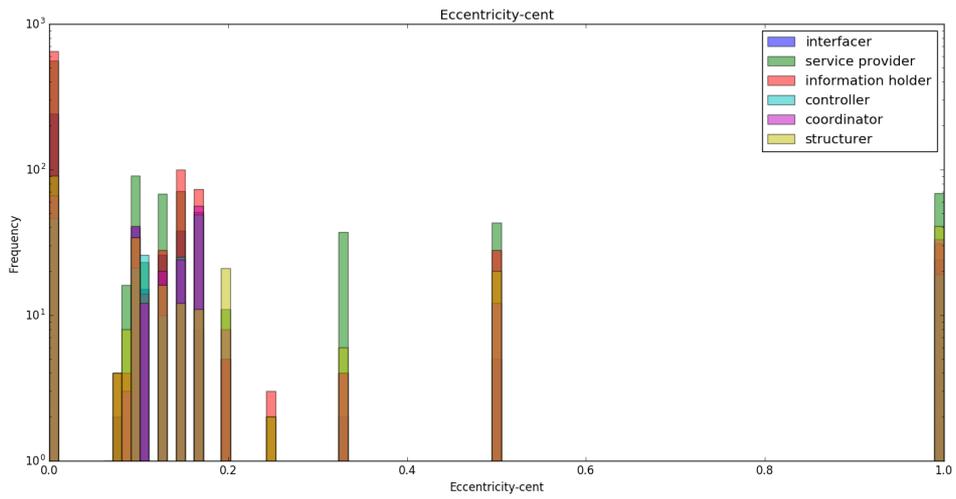


Figure 24: Eccentricity centrality values per label

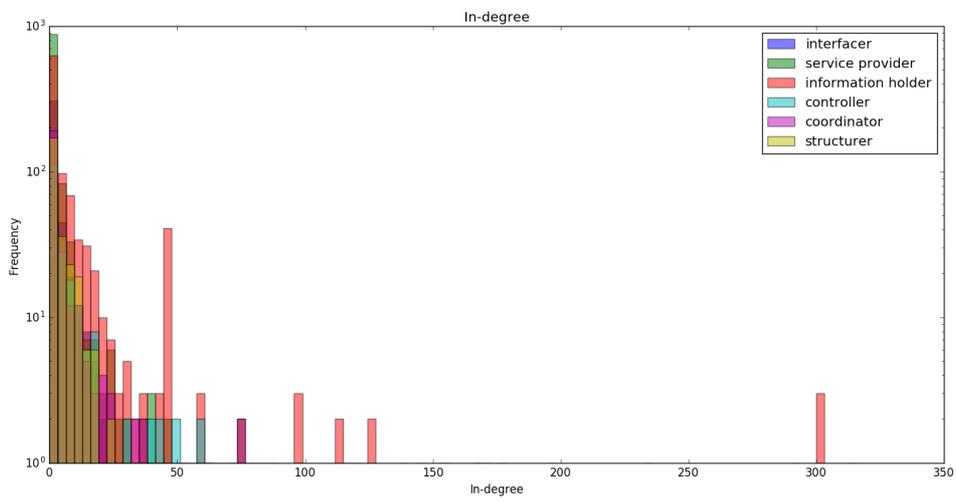


Figure 25: In-degree values per label

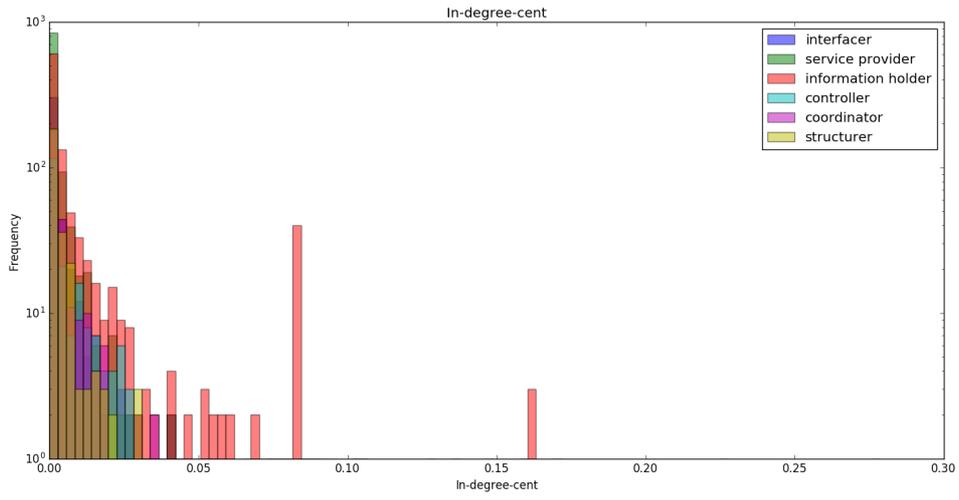


Figure 26: In-degree centrality values per label

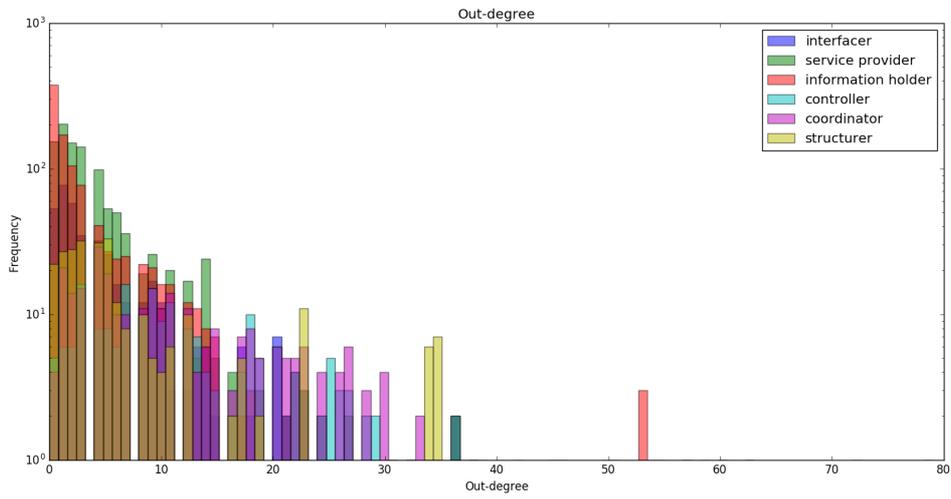


Figure 27: Out-degree values per label

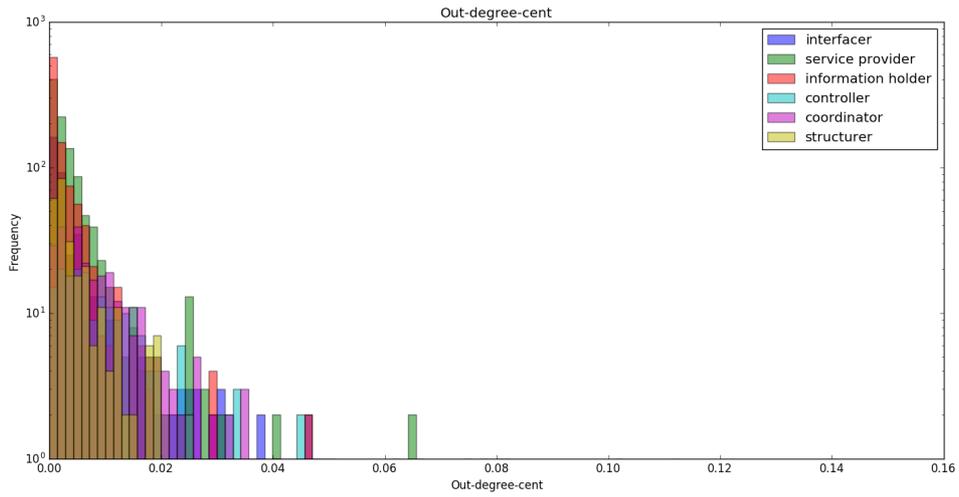


Figure 28: Out-degree centrality values per label

## B.2 Semantic software features

The histograms of all semantic software features, except for the behavioural features and the isenum feature, can be found in this subsection.

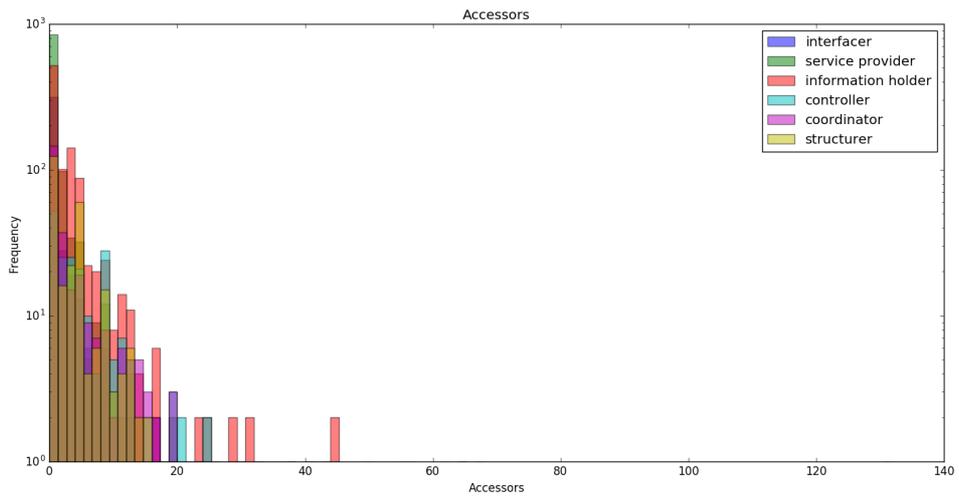


Figure 29: Number of accessors values per label

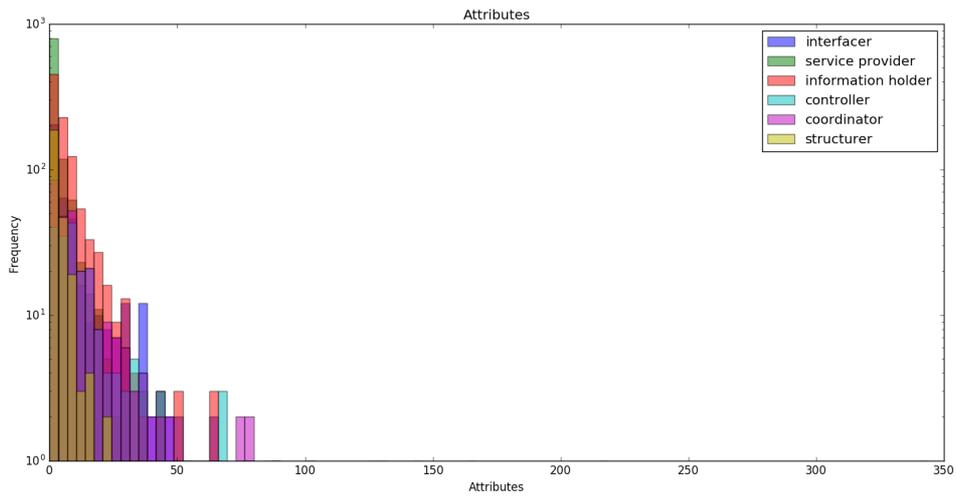


Figure 30: Number of attributes values per label

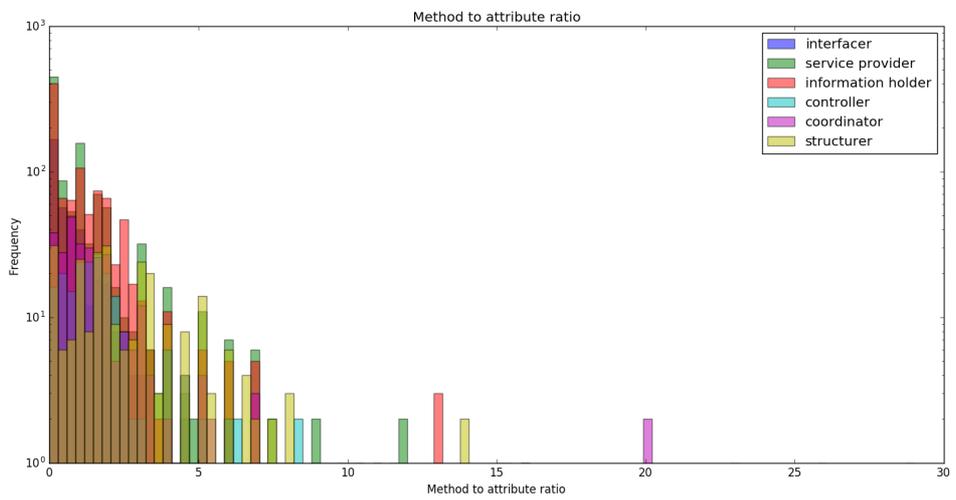


Figure 31: Attribute to method ratio values per label

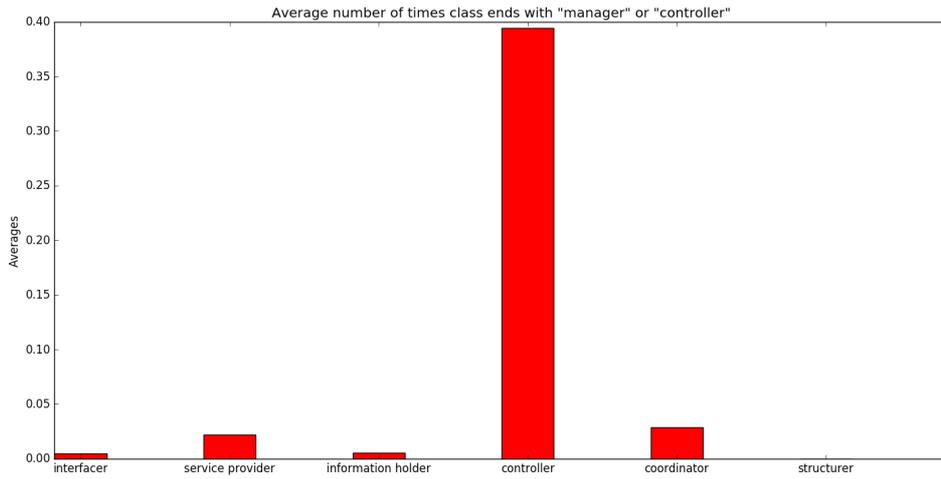


Figure 32: Average number of times a classname ends with "controller" or "manager" values per label

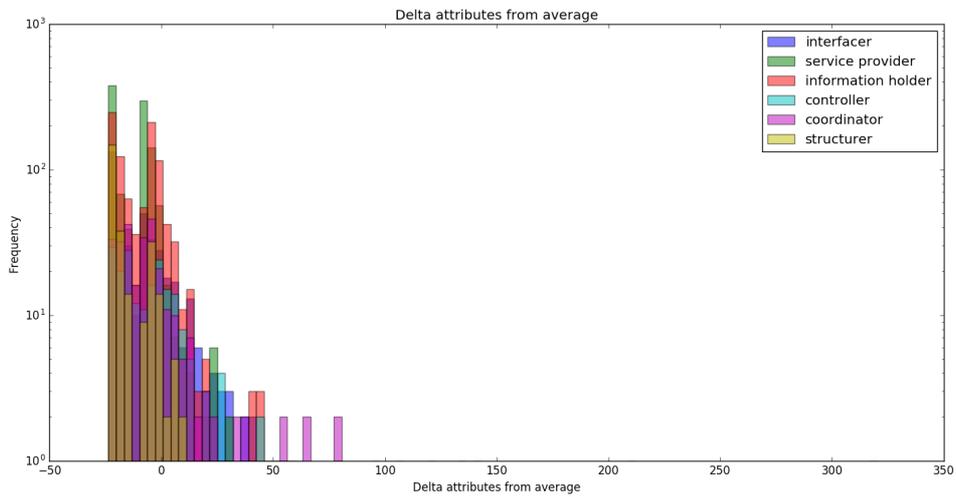


Figure 33: Delta number of attributes from average values per label

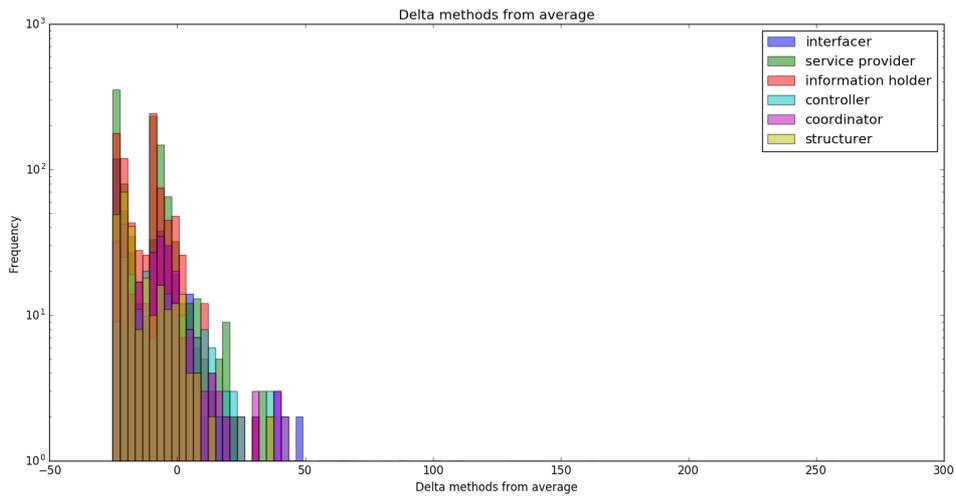


Figure 34: Delta number of methods from average values per label

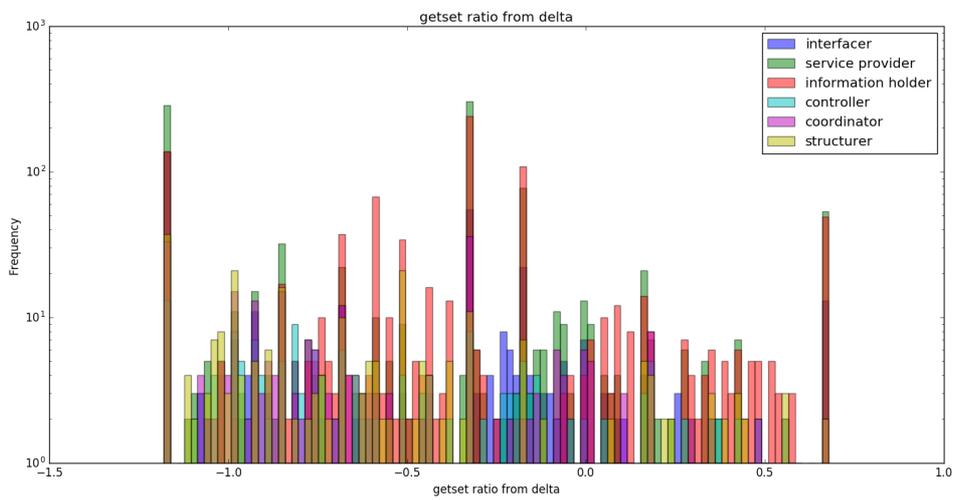


Figure 35: Delta number of accessors+mutators from average values per label

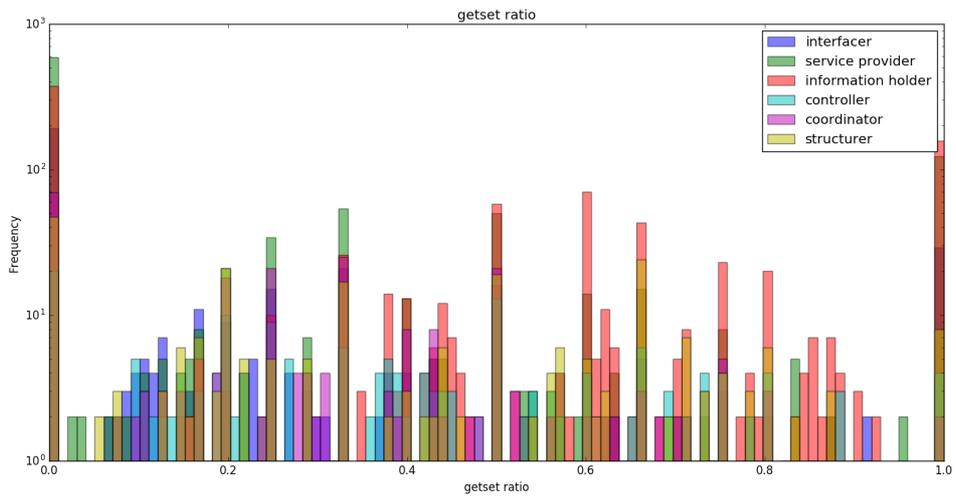


Figure 36: Ratio number of accessors+mutators to total number of methods values per label

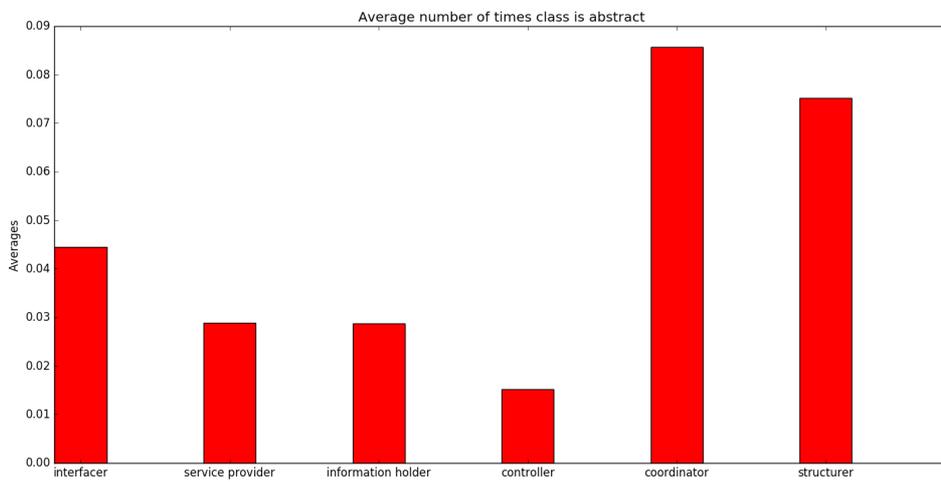
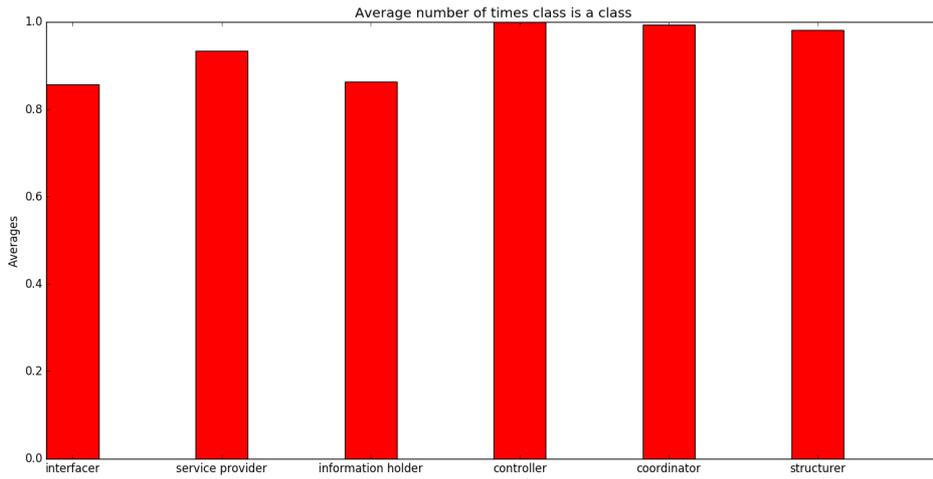
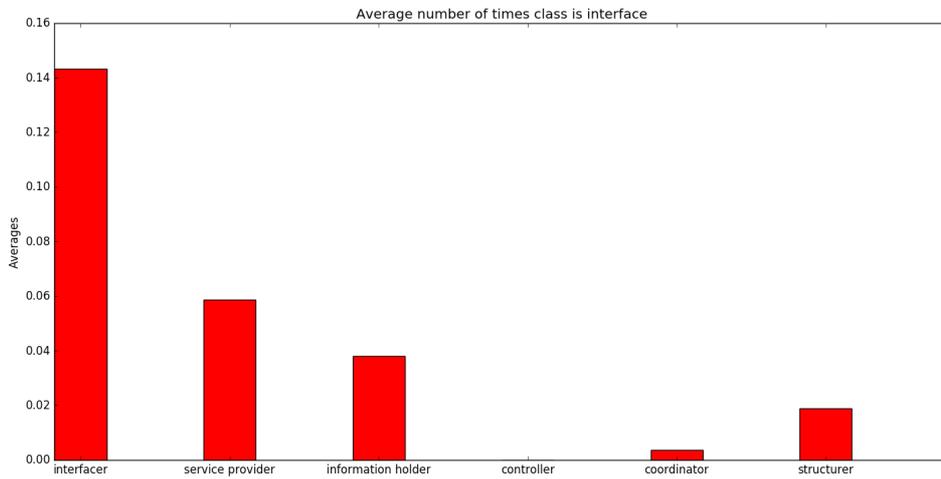


Figure 37: Average number of times a class is abstract per label



**Figure 38:** Average number of times a class is a class (not an interface or enum) per label



**Figure 39:** Average number of times a class is an interface per label

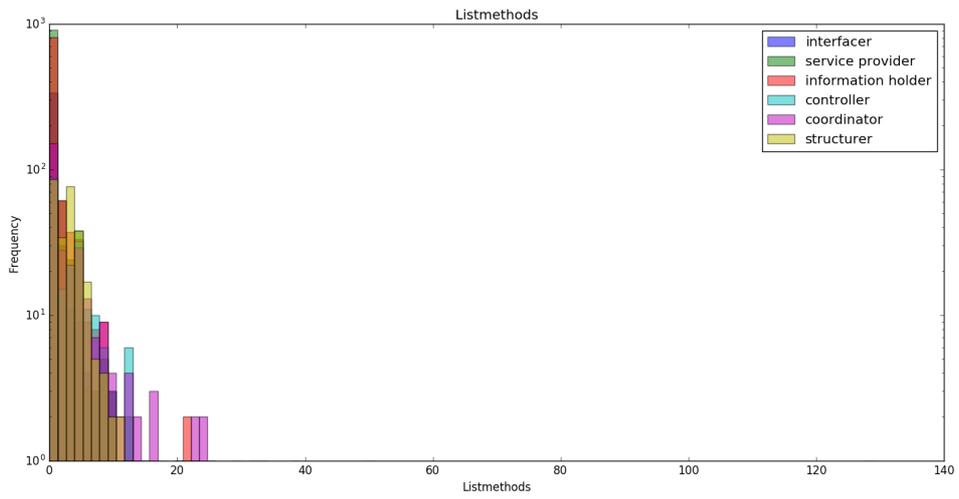


Figure 40: Number of listmethods values per label

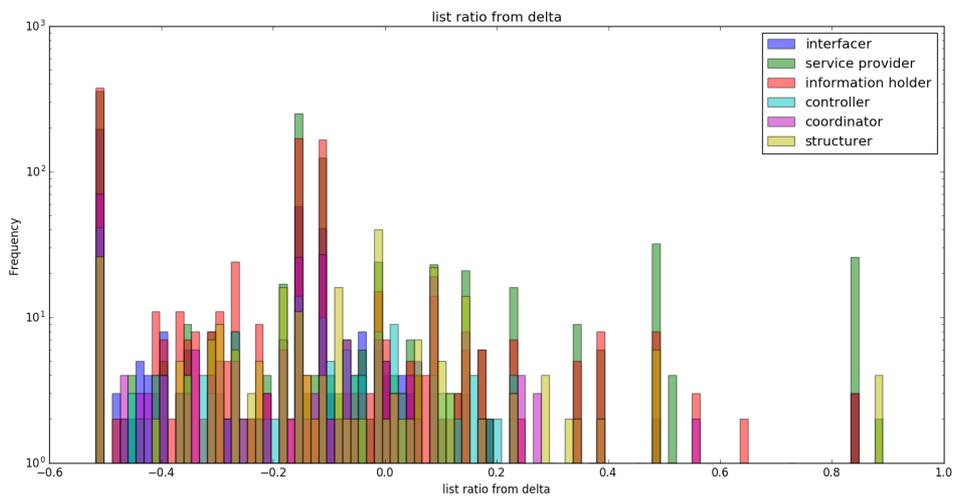


Figure 41: Delta number of list methods from average per label

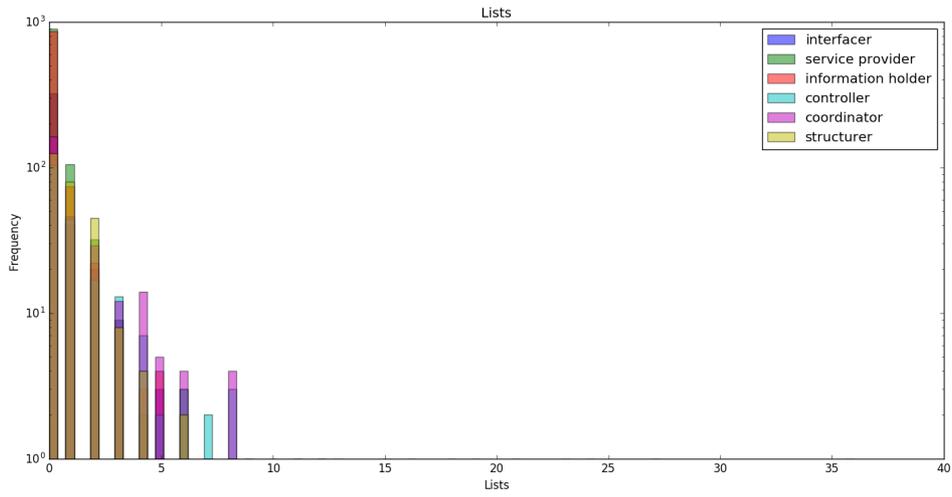


Figure 42: Number of list values per label

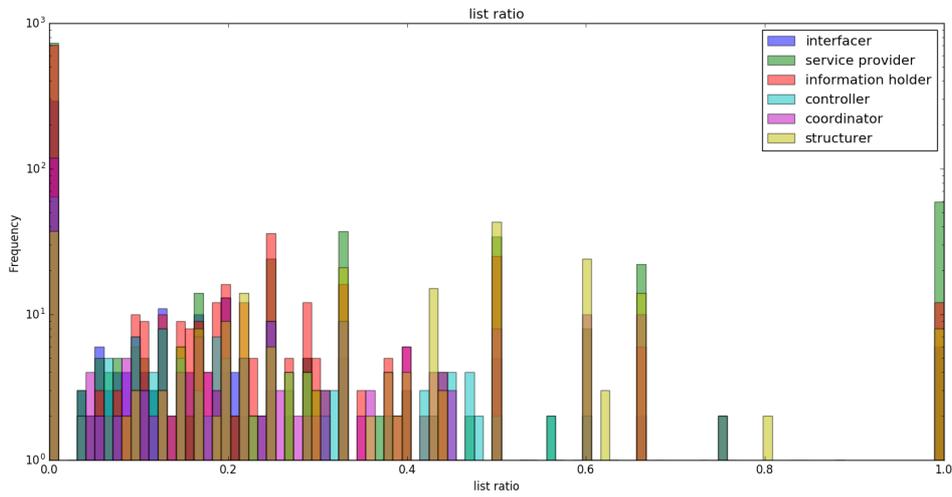


Figure 43: Ratio of list methods from average values per label

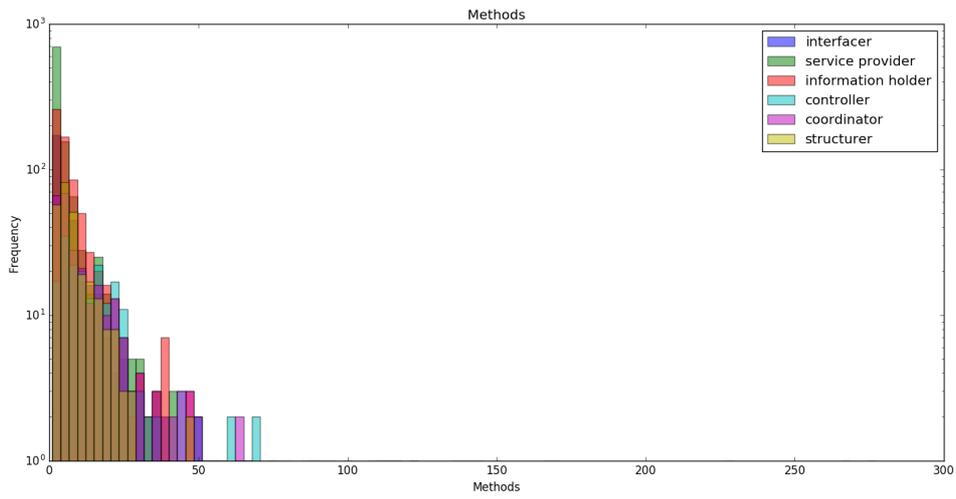


Figure 44: Number of methods values per label

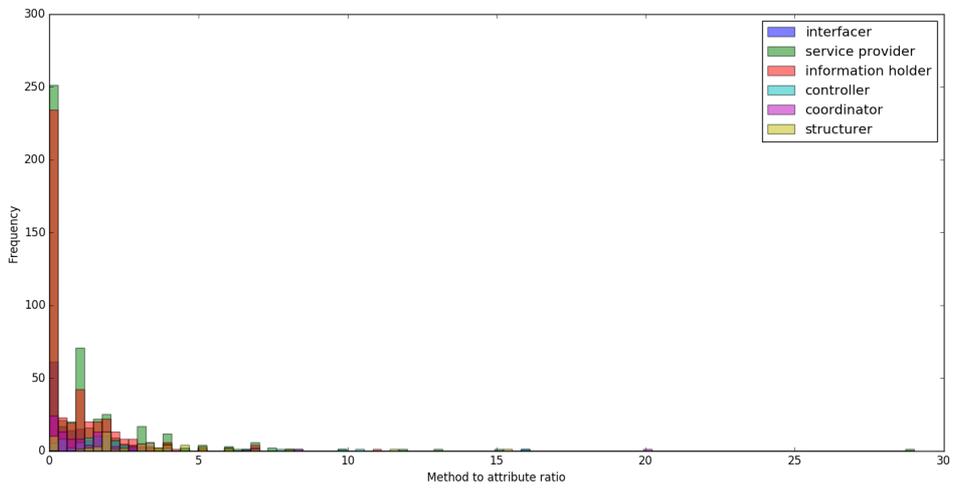


Figure 45: Number methods to attributes ratio values per label

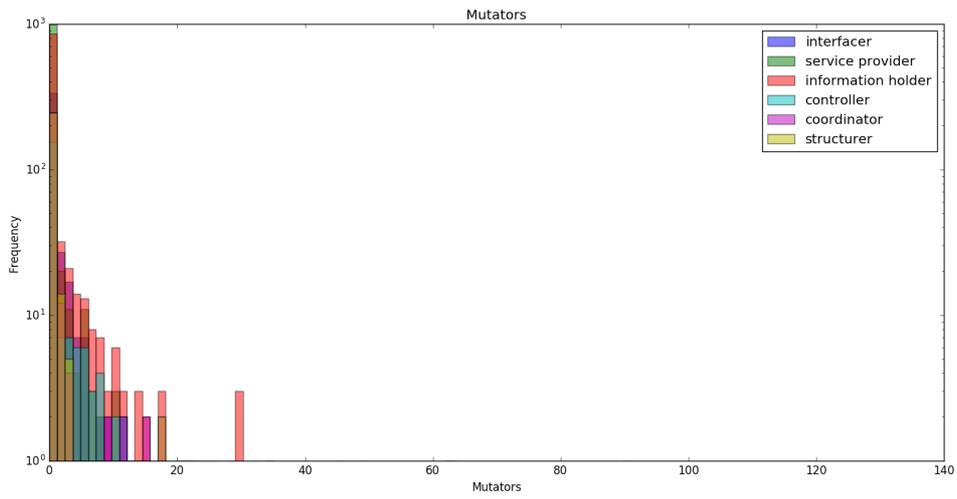


Figure 46: Number mutators values per label

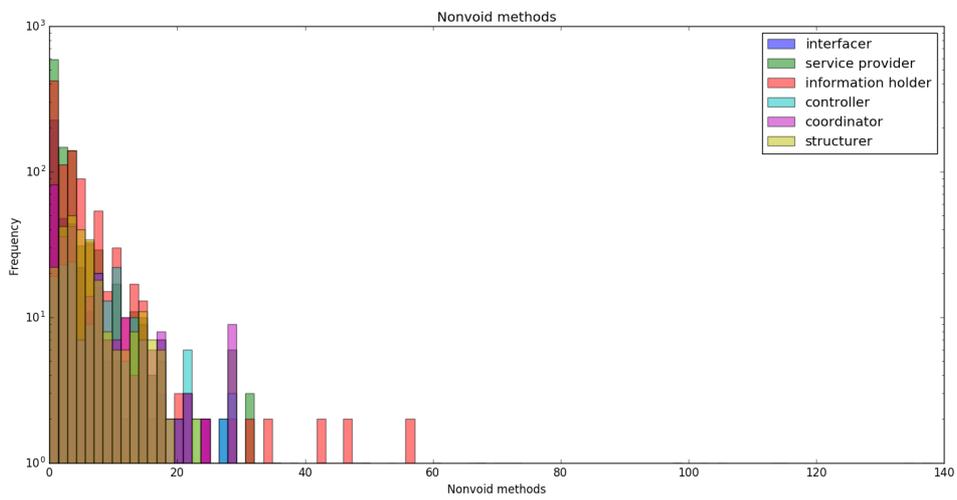


Figure 47: Number non-void methods values per label

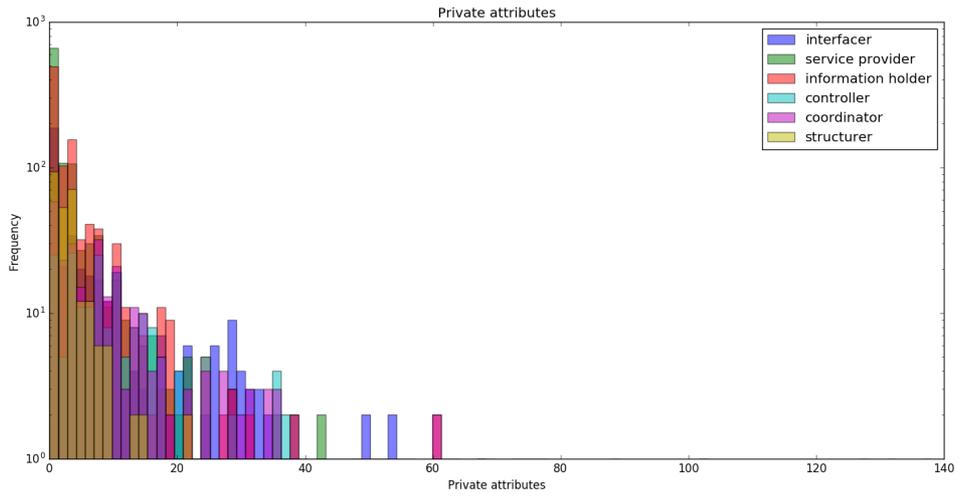


Figure 48: Number private attributes values per label

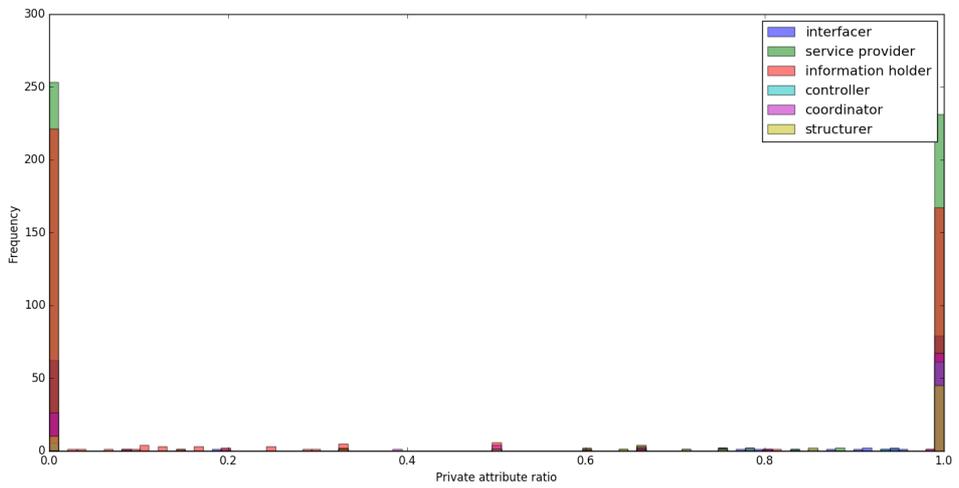


Figure 49: Private attribute ratio values per label

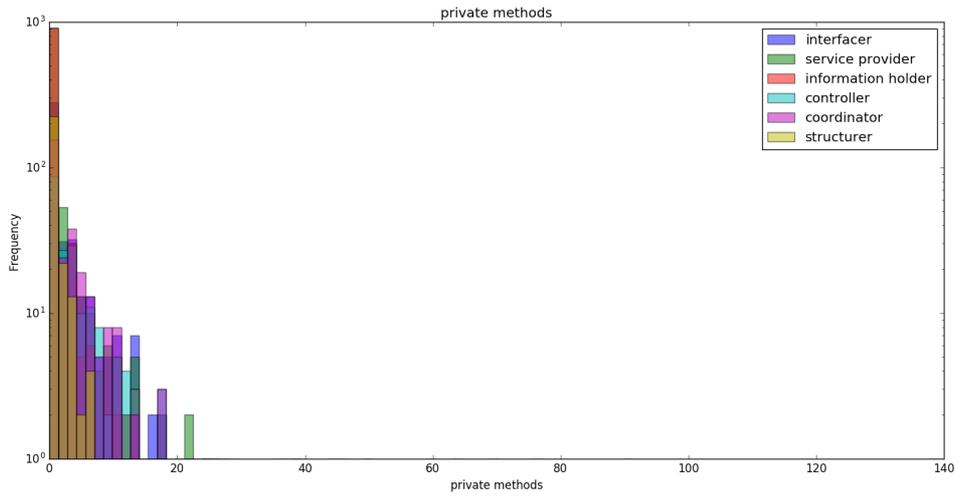


Figure 50: Number of private methods values per label

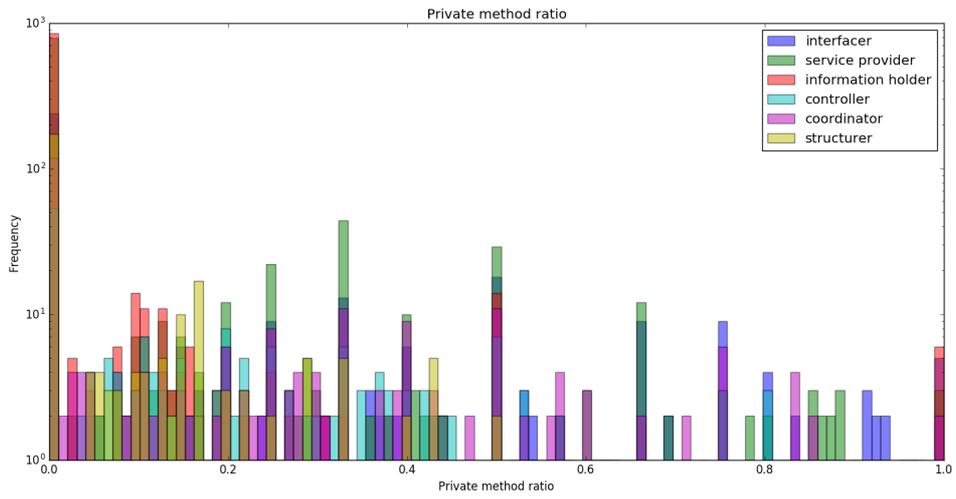


Figure 51: Private method ratio values per label

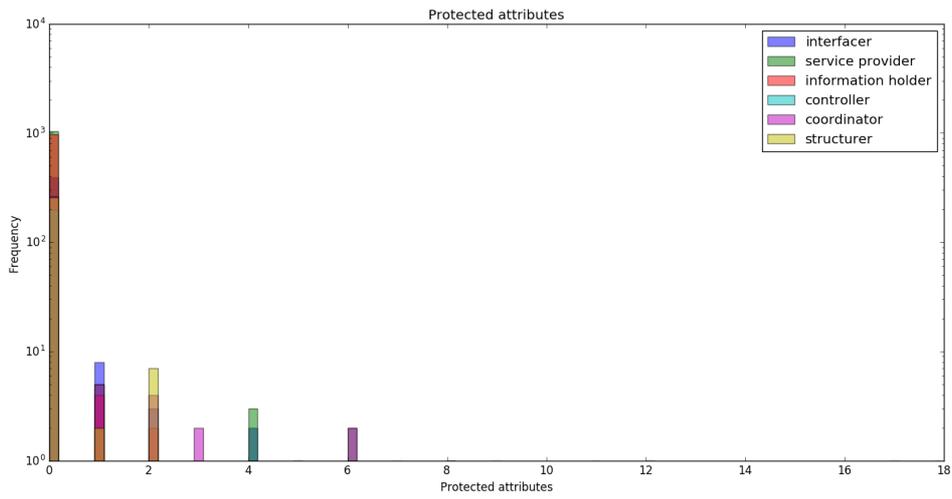


Figure 52: Number of protected attributes values per label

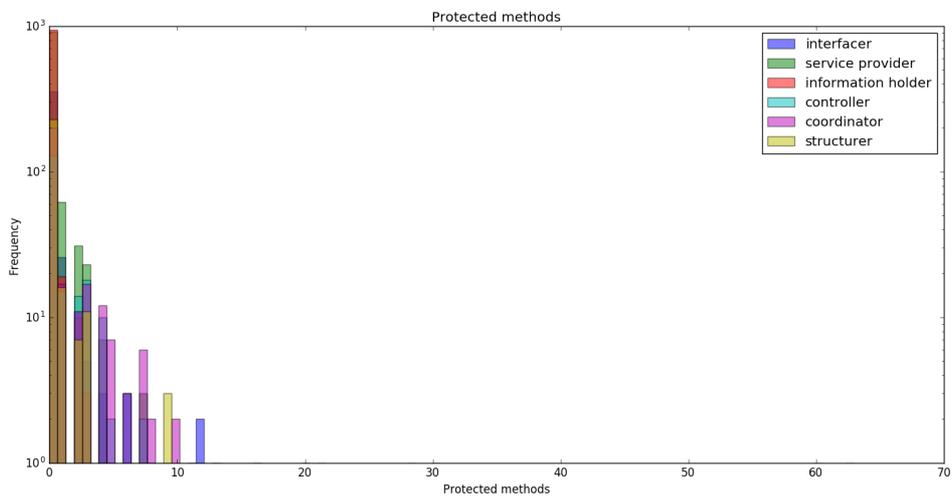


Figure 53: Number of protected methods values per label

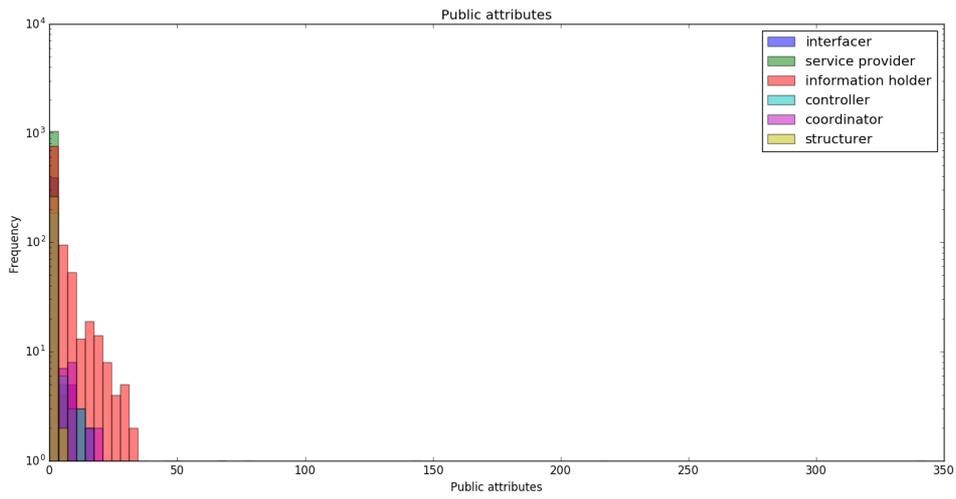


Figure 54: Number of public attributes values per label

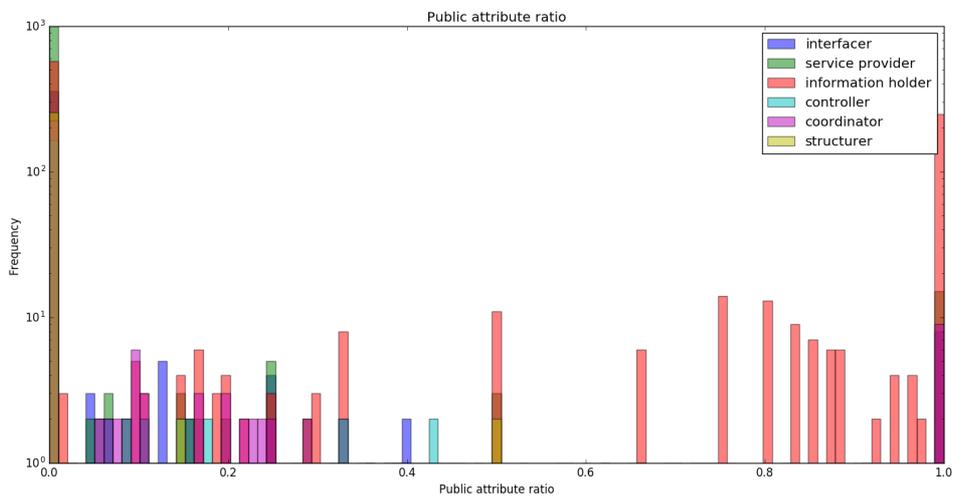


Figure 55: Public attribute ratio values per label

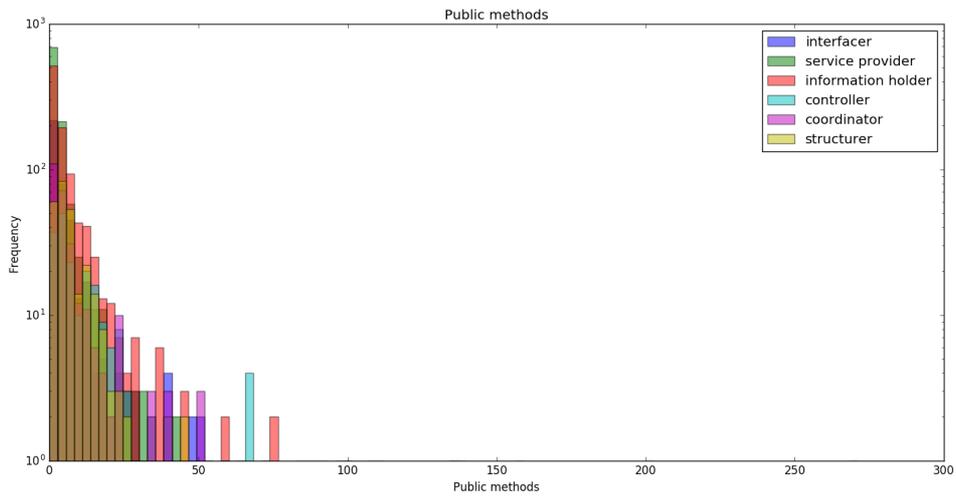


Figure 56: Number of public method values per label

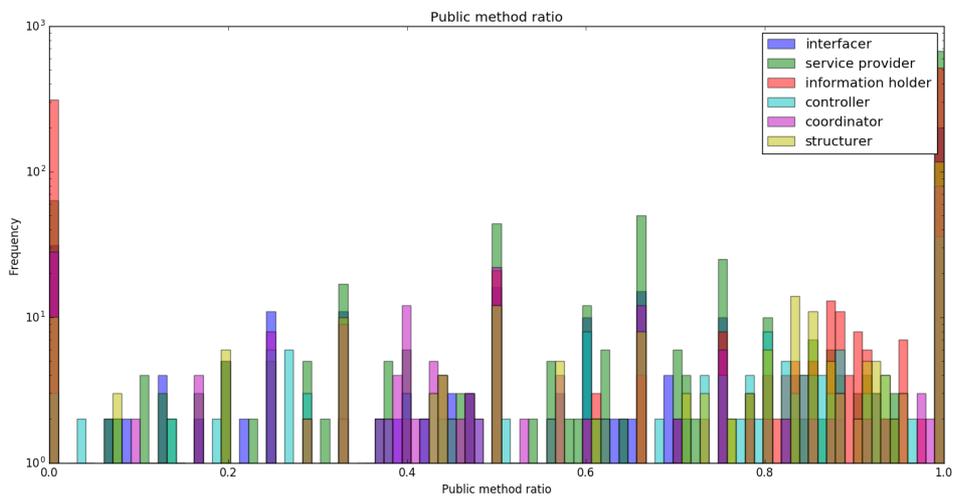


Figure 57: Public method ratio values per label

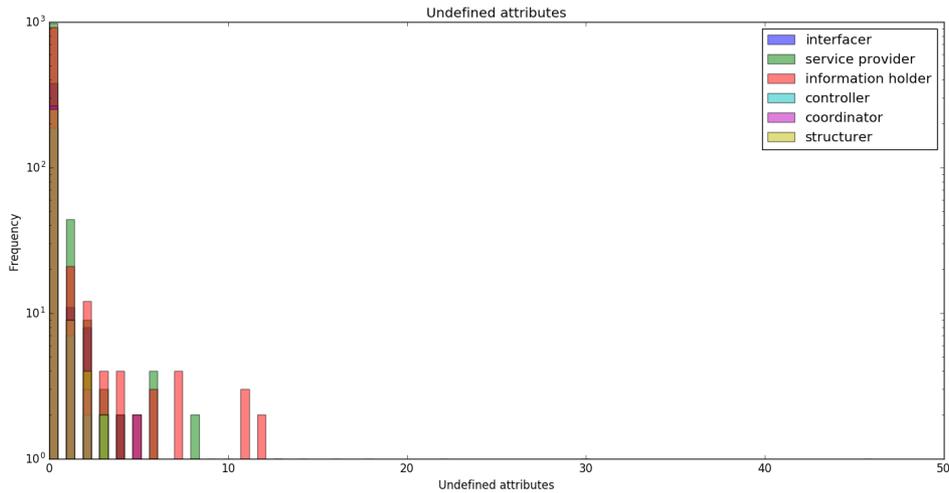


Figure 58: Number of undefined attributes values per label

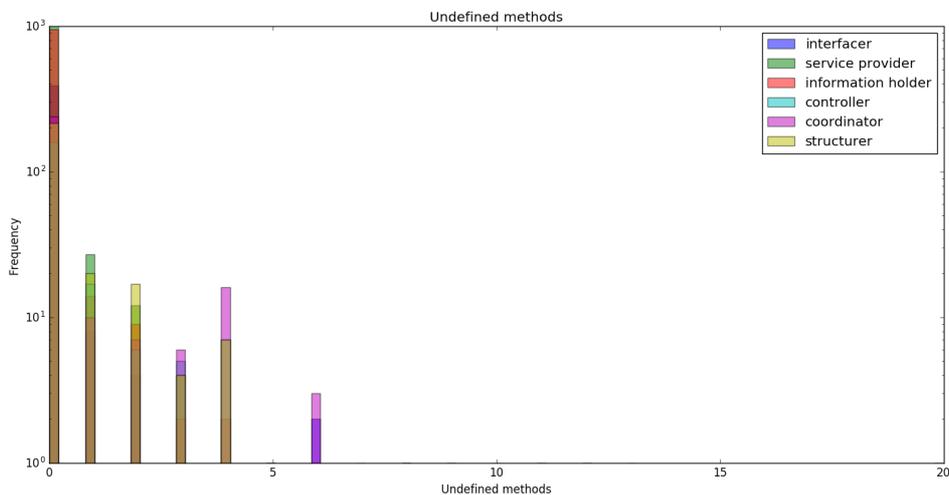


Figure 59: Number of undefined methods values per label

## C Source code network creator

This appendix contains the source code used transform the source code of a project of interest to a labelled adjacency list. The source code uses the Spoon library to parse the classes/interfaces from the code and turn it into an AST. In order to run the code in this appendix, both the Spoon .jar file (spoon-core-6.0.0-jar-with-dependencies.jar via <http://spoon.gforge.inria.fr/>) and the jackson-all .jar file (jackson-all-1.9.0.jar via <http://www.java2s.com/Code/Jar/j/Downloadjacksonall190jar.htm>) should be loaded into the project. Note that the versions mentioned here are the versions used in the experiment, later versions might give the same results.

### C.1 Information holder

The source code uses one information holder to store intermediate data. This information holder is called "IdNamespaceContainer" and only stores a combination of assigned id and the namespace of the parsed class/interface.

---

```
public class IdNamespaceContainer {
    public int id;
    public String fullName;
```

```

public IdNamespaceContainer(){
}

public IdNamespaceContainer(int id, String fullname){
    this.id = id;
    this.fullName = fullname;
}
}

```

---

## C.2 Main script

The main script has four steps to transform the source code. These steps are:

1. Read all .java files in the specified root folder and subfolders and convert them to File objects
2. Fill a Map (dictionary) with the name of the class as key and the IdNamespaceContainer as value
3. Parse all classes and create the edges by doing the following:
  - Look if the class has a superclass that exists in the Map, if so create an edge and store it in the edgelist with the ext. label
  - Look if the class implements an interface that exists in the Map, if so create an edge and store it in the edgelist with the impl. label
  - Look if any of the types of the class its attributes exists in the Map, if so create an edge and store it in the edgelist with the dep. label
  - Look if any of the parameter types of the class its methods exists in the Map, if so create an edge and store it in the edgelist with the dep. label
  - Look if any of the methods of the class declares a type that exists in the Map, if so create an edge and store it in the edgelist with the dep. label

The source code of this algorithm is as follows:

---

```

public class ProjectToGraphProgram {

    //stackoverflowerror -> dependency parameter & usage in body
    /**
     * @param args the command line arguments
     */
    private static Map<String, IdNamespaceContainer> classes = new Hashtable<>();
    private static Set<String> edgelist = new HashSet<>();

    public static void main(String[] args) {
        // TODO code application logic here
        List<String> paths = new ArrayList<>();

        //argo paths.add("C:\\Users\\xavyr\\Desktop\\CS and SBB\\Thesis\\Data\\ArgoUML\\src");
        //k9 paths.add("C:\\Users\\xavyr\\Desktop\\CS and
            SBB\\Thesis\\Data\\k-9-5.304\\k9mail\\src\\main\\java\\com\\fsc\\k9");
        //k9 paths.add("C:\\Users\\xavyr\\Desktop\\CS and
            SBB\\Thesis\\Data\\k-9-5.304\\k9mail-library\\src\\main\\java\\com\\fsc\\k9");

        //mars paths.add("C:\\Users\\xavyr\\Desktop\\CS and SBB\\Thesis\\Data\\mars-sim-master");

        paths.add("C:\\Users\\xavyr\\Desktop\\CS and
            SBB\\Thesis\\Data\\SweetHome3D-5.6-src\\src\\com\\eteks\\sweethome3d");

        List<File> files = getFiles(paths);
        System.out.println("Files loaded");
        fillDict(files);
    }
}

```

```

System.out.println("Dict created");
createEdges(files);
System.out.println("Edges created");
// addOtherDependancies(files);
writeToCSV();
System.out.println("CSV created");
writeIdDictToCSV();
System.out.println("Mapping CSV created");
}

private static void writeIdDictToCSV(){
    try
    {
        //argo BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
        FileOutputStream("C:\\Users\\xavyr\\Desktop\\CS and
        SBB\\Thesis\\Data\\Edgelists\\Argo\\mapping-argo-auto.csv"), "UTF-8"));

        // k9BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
        FileOutputStream("C:\\Users\\xavyr\\Desktop\\CS and
        SBB\\Thesis\\Data\\Edgelists\\K9\\mapping-k9-auto.csv"), "UTF-8"));

        //MARS BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
        FileOutputStream("C:\\Users\\xavyr\\Desktop\\CS and
        SBB\\Thesis\\Data\\Edgelists\\Mars\\mapping-mars-auto.csv"), "UTF-8"));

        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
        FileOutputStream("C:\\Users\\xavyr\\Desktop\\CS and
        SBB\\Thesis\\Data\\Edgelists\\Home3D\\mapping-home3d-auto.csv"), "UTF-8"));

        String header = "Id,Class,Fullname";
        bw.write(header);
        bw.newLine();
        for (String key : classes.keySet())
        {
            StringBuilder oneLine = new StringBuilder();
            oneLine.append(classes.get(key).id).append(",").append(key).append(",").append(classes.get(key).ful
            bw.write(oneLine.toString());
            bw.newLine();
        }
        bw.flush();
        bw.close();
    }
    catch (UnsupportedEncodingException e) {}
    catch (FileNotFoundException e){}
    catch (IOException e){}
}

private static void writeToCSV()
{
    try
    {
        //argo BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
        FileOutputStream("C:\\Users\\xavyr\\Desktop\\CS and
        SBB\\Thesis\\Data\\Edgelists\\Argo\\edgelist.csv"), "UTF-8"));
        //k9 BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
        FileOutputStream("C:\\Users\\xavyr\\Desktop\\CS and
        SBB\\Thesis\\Data\\Edgelists\\K9\\edgelist.csv"), "UTF-8"));
        //mars BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
        FileOutputStream("C:\\Users\\xavyr\\Desktop\\CS and
        SBB\\Thesis\\Data\\Edgelists\\Mars\\edgelist.csv"), "UTF-8"));
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
        FileOutputStream("C:\\Users\\xavyr\\Desktop\\CS and
        SBB\\Thesis\\Data\\Edgelists\\Home3D\\edgelist.csv"), "UTF-8"));
    }
}

```

```

String header = "From,To,Weight";
bw.write(header);
bw.newLine();
for (String line : edgelist)
{
    StringBuilder oneLine = new StringBuilder();
    oneLine.append(line);
    bw.write(oneLine.toString());
    bw.newLine();
}
bw.flush();
bw.close();
}
catch (UnsupportedEncodingException e) {}
catch (FileNotFoundException e){}
catch (IOException e){}
}

private static void createEdges(List<File> files){
    for (File file : files) {
        String path = file.getAbsolutePath();

        Launcher launcher = new Launcher();

        launcher.addInputResource(path);

        launcher.getEnvironment().setAutoImports(true);

        launcher.getEnvironment().setNoClasspath(true); // optional

        launcher.buildModel();
        CtModel model = launcher.getModel();

        for (Object e : model.getAllTypes()) {
            addSuperclass((CtType)e);
            addInterfaces((CtType) e);
            addDependancies((CtType)e);

            for (Object nested : ((CtType) e).getNestedTypes()) {
                addSuperclass((CtType)nested);
                addInterfaces((CtType)nested);
                addDependancies((CtType)nested);
            }
        }
    }
}

private static void addDependancies(CtType currentClass){
    addAttributeDependancies(currentClass);
    addOtherDependancies(currentClass);
}

private static void addUsageDependancies(CtMethod method, CtType currentClass){
    if(method.getBody() != null){
        for (Object t : method.getBody().getReferencedTypes()){
            String typeName = ((CtTypeReference)t).getSimpleName();

            if(classes.containsKey(typeName)){
                int currentId = classes.get(currentClass.getSimpleName()).id;
                int typeId = classes.get(typeName).id;
                if(currentId != typeId){
                    edgelist.add(currentId + "," + typeId + "," + 1);
                }
            }
        }
    }
}
}

```

```

    }
}

private static void addOtherDependencies(CtType currentClass){
    for(Object method : currentClass.getMethods()){
        addParameterDependencies((CtMethod)method, currentClass);
        addUsageDependencies((CtMethod)method, currentClass);
    }
}

private static void addParameterDependencies(CtMethod method, CtType currentClass){
    for(Object param : method.getParameters()){
        String paramType = ((CtParameterImpl)param).getType().getSimpleName();

        if(classes.containsKey(paramType)){
            int currentId = classes.get(currentClass.getSimpleName()).id;
            int typeId = classes.get(paramType).id;
            edgelist.add(currentId + "," + typeId + "," + 1);
        }
    }
}

private static void addAttributeDependencies(CtType currentClass){
    for(Object attribute : currentClass.getAllFields()){
        String type = ((CtFieldReferenceImpl)attribute).getType().getSimpleName();
        if(classes.containsKey(type)){
            int currentId = classes.get(currentClass.getSimpleName()).id;
            int typeId = classes.get(type).id;
            edgelist.add(currentId + "," + typeId + "," + 1);
        }
    }
}

private static void addInterfaces(CtType currentClass){
    for(Object interf : currentClass.getSuperInterfaces()){
        String interfaceName = ((CtTypeReferenceImpl)interf).getSimpleName();
        if(classes.containsKey(interfaceName)){
            int currentId = classes.get(currentClass.getSimpleName()).id;
            int interfId = classes.get(interfaceName).id;
            edgelist.add(currentId + "," + interfId + "," + 3);
        }
    }
}

private static void addSuperclass(CtType currentClass){
    CtTypeReference superclass = (CtTypeReference)((CtType)currentClass).getSuperclass();
    if(superclass != null){
        String superName = superclass.getSimpleName();

        if(classes.containsKey(superName)){
            int currentId = classes.get(currentClass.getSimpleName()).id;
            int superId = classes.get(superName).id;
            edgelist.add(currentId + "," + superId + "," + 2);
        }
    }
}

private static List<File> getFiles(List<String> paths) {
    List<File> files = new ArrayList<>();

    for(String path : paths){
        files.addAll(listFiles(path));
    }
}

```

```

    return files;
}

private static String getFileExtension(File file) {
    String name = file.getName();
    try {
        return name.substring(name.lastIndexOf(".") + 1);
    } catch (Exception e) {
        return "";
    }
}

private static String getFileName(File file) {
    String s = file.getName();
    return s;
}

public static List<File> listFiles(String directoryName) {
    File directory = new File(directoryName);

    List<File> resultList = new ArrayList<File>();

    // get all the files from a directory
    File[] fList = directory.listFiles();
    //resultList.addAll(Arrays.asList(fList));
    for (File file : fList) {
        if (file.isFile() && "java".equals(getFileExtension(file)) &&
            getFileName(file).indexOf("Test") == -1) {
            resultList.add(file);
            //System.out.println(file.getAbsolutePath());
        } else if (file.isDirectory()) {
            resultList.addAll(listFiles(file.getAbsolutePath()));
        }
    }
    //System.out.println(fList);
    return resultList;
}

private static void fillDict(List<File> files){
    int idCounter = 0;
    for (File file : files) {
        String path = file.getAbsolutePath();

        Launcher launcher = new Launcher();

        launcher.addInputResource(path);

        launcher.getEnvironment().setAutoImports(true);

        launcher.getEnvironment().setNoClasspath(true); // optional

        launcher.buildModel();
        CtModel model = launcher.getModel();

        for (Object e : model.getAllTypes()) {
            String className = ((CtType) e).getSimpleName();
            String namespace = ((CtType)e).getQualifiedName();
            classes.put(className, new IdNamespaceContainer(idCounter, namespace));
            idCounter++;

            for (Object nested : ((CtType) e).getNestedTypes()) {
                String nestedClassName = ((CtType) nested).getSimpleName();
                String nestedNamespace = ((CtType)nested).getQualifiedName();

                classes.put(nestedClassName, new IdNamespaceContainer(idCounter,

```



```

public int numberOfPublicMethods;
public int numberOfPrivateMethods;
public int numberOfProtectedMethods;
public int numberOfUndefinedMethods;
public int numberOfOtherMethods;

public int numberOfAttributes;
public int numberOfPublicAttributes;
public int numberOfPrivateAttributes;
public int numberOfProtectedAttributes;
public int numberOfUndefinedAttributes;
public int numberOfOtherAttributes;

public int numberOfAccessors;
public int numberOfMutators;

public List<MethodSemantic> methods;
public List<AttributeSemantic> attributes;

public double methodAttributeRatio;
public double deltaAttributesFromAverage;
public double deltaMethodsFromAverage;
public double publicMethodRatio;
public double publicAttributeRatio;
public double privateMethodRatio;
public double privateAttributeRatio;
public double attributeMethodRatio;

public double getsetRatio;
public double listRatio;
public double getsetFromAverageRatio;
public double listFromAverageRatio;
public boolean isEnum;
public boolean isClass;
public boolean isInterface;
public boolean isAbstractClass;

public boolean er_or;
public boolean controller_manager;
public int numberOfLists;
public int listMethods;
public int numberNonVoidMethods;
public String typeOfClass;

public int numberOfExternalMethodCalls;
public double numberOfExternalMethodCallsToLinesRatio;

public int numberOfIfStatements;
public double numberOfIfStatementsToLinesRatio;

public ClassSemantics(){
    this.methods = new ArrayList<>();
    this.attributes = new ArrayList<>();

    this.numberOfAttributes = 0;
    this.numberOfMethods = 0;
    this.numberOfPrivateAttributes = 0;
    this.numberOfPrivateMethods = 0;
    this.numberOfProtectedAttributes = 0;
    this.numberOfProtectedMethods = 0;
    this.numberOfPublicAttributes = 0;
    this.numberOfPublicMethods = 0;
    this.numberOfUndefinedMethods = 0;
    this.numberOfUndefinedAttributes = 0;
    this.numberOfOtherAttributes = 0;
}

```

```
        this.numberOfOtherMethods = 0;
    }
}
```

---

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
```

```
package filecrawler;
```

```
/**
```

```
 *
 * @author xavyr
 */
```

```
public class MethodSemantic {
```

```
    public String signature;
    public String methodName;
    public String methodVisibility;
    public String methodReturnType;
```

```
    public boolean isGetter(){
        boolean returnVal = false;
```

```
        if(this.methodName.toLowerCase().startsWith("is") ||
            (this.methodName.toLowerCase().startsWith("get") &&
             !(this.methodName.toLowerCase().contains("at") ||
              this.methodName.toLowerCase().contains("by")))){
            return true;
        }
    }
```

```
        return returnVal;
```

```
    }
```

```
    public boolean isSetter(){
        boolean returnVal = false;
```

```
        if(this.methodName.toLowerCase().startsWith("set")){
            return true;
        }
    }
```

```
        return returnVal;
```

```
    }
```

```
}
```

---

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
```

```
package filecrawler;
```

```
/**
```

```
 *
 * @author xavyr
 */
```

```
public class AttributeSemantic {
```

```
    public String attributeName;
    public String attributeVisibility;
    public String attributeReturnType;
```

```
}
```

---

### D.1.2 Feature extraction

This subsection presents the source code used to extract the semantic software features. Note that the value of *totalNumberOfClasses* should be changed to the number of classes in the project (together with the paths to the project source code).

---

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package filecrawler;

import java.io.BufferedWriter;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import org.apache.commons.io.filefilter.NameFileFilter;
import org.codehaus.jackson.map.ObjectMapper;
import spoon.Launcher;
import spoon.reflect.CtModel;
import spoon.reflect.code.CtStatement;
import spoon.reflect.declaration.CtClass;
import spoon.reflect.declaration.CtField;
import spoon.reflect.declaration.CtMethod;
import spoon.reflect.declaration.CtType;

/**
 *
 * @author xavyr
 */
public class FileCrawler {

    /**
     * @param args the command line arguments
     */
    public static int totalNumberOfMethods;
    public static int totalNumberOfAttributes;
    public static int totalNumberOfClasses = 561;

    public static double averageNumberOfMethods;
    public static double averageNumberOfAttributes;

    public static double averageListRatio;
    public static double averageGetsetRatio;

    public static void main(String[] args) {
        // TODO code application logic here
        List<String> paths = new ArrayList<>();

        //k9 paths.add("C:\\Users\\xavyr\\Desktop\\CS and
        SBB\\Thesis\\Data\\k-9-5.304\\k9mail\\src\\main\\java\\com\\fscck\\k9");
        //k9 paths.add("C:\\Users\\xavyr\\Desktop\\CS and
        SBB\\Thesis\\Data\\k-9-5.304\\k9mail-library\\src\\main\\java\\com\\fscck\\k9");
        //argo paths.add("C:\\Users\\xavyr\\Desktop\\CS and SBB\\Thesis\\Data\\ArgoUML\\src");

        paths.add("C:\\Users\\xavyr\\Desktop\\CS and SBB\\Thesis\\Data\\mars-sim-master");
```

```

//home paths.add("C:\\Users\\xavyr\\Desktop\\CS and
    SBB\\Thesis\\Data\\SweetHome3D-5.6-src\\src\\com\\eteks\\sweethome3d");

List<File> files = getFiles(paths);
System.out.println("Files loaded");
List<ClassSemantics> semantics = getSemantics(files);
System.out.println(semantics.size() + " classes created");

addTruongFeatures(semantics);

addRatioFeatures(semantics);

String jsonArray = "";
try {
    jsonArray = writeListToJsonArray(semantics);
} catch (IOException e) {
    System.out.println(e);
}
System.out.println("Jsonarray created");

// writeJsonToFile(jsonArray, "C:\\Users\\xavyr\\Desktop\\CS and
    SBB\\Thesis\\Data\\semantics.json");
// System.out.println("JSON file created");

writeToCSV(semantics);
System.out.println("CSV file created");
}

private static void writeToCSV(List<ClassSemantics> semantics)
{
    String CSV_SEPARATOR = ",";
    try
    {
        //k9 BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
            FileOutputStream("C:\\Users\\xavyr\\Desktop\\CS and
                SBB\\Thesis\\Data\\Features\\K9\\semantics.csv"), "UTF-8"));
        //argo BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
            FileOutputStream("C:\\Users\\xavyr\\Desktop\\CS and
                SBB\\Thesis\\Data\\Features\\Argo\\semantics.csv"), "UTF-8"));
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
            FileOutputStream("C:\\Users\\xavyr\\Desktop\\CS and
                SBB\\Thesis\\Data\\Features\\Mars\\semantics.csv"), "UTF-8"));
        //home BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new
            FileOutputStream("C:\\Users\\xavyr\\Desktop\\CS and
                SBB\\Thesis\\Data\\Features\\Home3D\\semantics.csv"), "UTF-8"));

        String header = "Class name,Accessors,Attributes,Methods,Mutators,Other
            attributes,Other methods,Private attributes,private methods,Protected
            attributes,Protected methods,Public attributes,Public methods,Undefined
            attributes,Undefined methods,Method to attribute ratio,Delta attributes from
            average,Delta methods from average,Public method ratio,Public attribute
            ratio,Private method ratio,Private attribute ratio,Attribute to method
            ratio,er-or,controller-manager,Lists,Listmethods,Nonvoid methods,getset
            ratio,list ratio,getset ratio from delta,list ratio from
            delta,isEnum,isInterface,isClass,isAbstractClass,number of if statements,
            number of external methods called";
        bw.write(header);
        bw.newLine();
        for (ClassSemantics semantic : semantics)

```

```

{
    StringBuilder oneLine = new StringBuilder();
    oneLine.append(semantic.className);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfAccessors);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfAttributes);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfMethods);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfMutators);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfOtherAttributes);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfOtherMethods);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfPrivateAttributes);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfPrivateMethods);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfProtectedAttributes);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfProtectedMethods);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfPublicAttributes);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfPublicMethods);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfUndefinedAttributes);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfUndefinedMethods);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.methodAttributeRatio);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.deltaAttributesFromAverage);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.deltaMethodsFromAverage);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.publicMethodRatio);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.publicAttributeRatio);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.privateMethodRatio);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.privateAttributeRatio);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.attributeMethodRatio);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.er_or);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.controller_manager);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberOfLists);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.listMethods);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.numberNonVoidMethods);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.getsetRatio);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.listRatio);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.getsetFromAverageRatio);
    oneLine.append(CSV_SEPARATOR);
    oneLine.append(semantic.listFromAverageRatio);
}

```

```

        oneLine.append(CSV_SEPARATOR);
        oneLine.append(semantic.isEnum);
        oneLine.append(CSV_SEPARATOR);
        oneLine.append(semantic.isInterface);
        oneLine.append(CSV_SEPARATOR);
        oneLine.append(semantic.isClass);
        oneLine.append(CSV_SEPARATOR);
        oneLine.append(semantic.isAbstractClass);
        oneLine.append(CSV_SEPARATOR);
        oneLine.append(semantic.numberOfIfStatements);
        oneLine.append(CSV_SEPARATOR);
        oneLine.append(semantic.numberOfExternalMethodCalls);
        bw.write(oneLine.toString());
        bw.newLine();
    }
    bw.flush();
    bw.close();
}
}
catch (UnsupportedEncodingException e) {}
catch (FileNotFoundException e){}
catch (IOException e){}
}

private static void addRatioFeatures(List<ClassSemantics> semantics){
    double averageAttributes = (double)totalNumberOfAttributes / (double)totalNumberOfClasses;
    double averageMethods = (double)totalNumberOfMethods / (double)totalNumberOfClasses;
    double totalListRatio = 0;
    double totalGetsetRatio = 0;

    for(ClassSemantics semantic : semantics){
        semantic.deltaAttributesFromAverage = (double)semantic.numberOfAttributes -
            averageAttributes;
        semantic.deltaMethodsFromAverage = (double)semantic.numberOfMethods - averageMethods;
        if(semantic.numberOfAttributes > 0){
            semantic.methodAttributeRatio =
                (double)semantic.numberOfMethods/(double)semantic.numberOfAttributes;
            semantic.publicAttributeRatio =
                (double)semantic.numberOfPublicAttributes/(double)semantic.numberOfAttributes;
        }
        if(semantic.numberOfMethods > 0){
            semantic.attributeMethodRatio =
                (double)semantic.numberOfAttributes/(double)semantic.numberOfMethods;
            semantic.publicMethodRatio =
                (double)semantic.numberOfPublicMethods/(double)semantic.numberOfMethods;
            semantic.privateMethodRatio =
                (double)semantic.numberOfPrivateMethods/(double)semantic.numberOfMethods;
            semantic.getsetRatio = ((double)semantic.numberOfAccessors +
                (double)semantic.numberOfMutators) / (double)semantic.numberOfMethods;
            semantic.listRatio = (double)semantic.listMethods / (double)semantic.numberOfMethods;
            totalListRatio += semantic.listRatio;
            totalGetsetRatio += semantic.getsetRatio;
        }
    }
}

averageGetsetRatio = totalGetsetRatio / totalNumberOfClasses;
averageListRatio = totalListRatio / totalNumberOfClasses;

for(ClassSemantics semantic : semantics){
    semantic.getsetFromAverageRatio = semantic.getsetRatio - averageGetsetRatio;
    semantic.listFromAverageRatio = semantic.listRatio - averageListRatio;
}
}

```

```

private static void writeJsonToFile(String jsonArray, String pathToNewFile) {
    try {
        // Writing to a file
        File file = new File(pathToNewFile);
        file.createNewFile();
        FileWriter fileWriter = new FileWriter(file);
        System.out.println("Writing JSON object to file");
        System.out.println("-----");

        fileWriter.write(jsonArray);
        fileWriter.flush();
        fileWriter.close();

    } catch (IOException e) {
        e.printStackTrace();
    }
}

private static String writeListToJsonArray(List<ClassSemantics> list) throws IOException {
    final ByteArrayOutputStream out = new ByteArrayOutputStream();
    final ObjectMapper mapper = new ObjectMapper();

    mapper.writeValue(out, list);

    final byte[] data = out.toByteArray();
    return new String(data);
}

private static List<ClassSemantics> getSemantics(List<File> files) {
    List<ClassSemantics> semantics = new ArrayList<>();

    for (File file : files) {
        String path = file.getAbsolutePath();

        Launcher launcher = new Launcher();

        launcher.addInputResource(path);

        launcher.getEnvironment().setNoClasspath(true); // optional

        launcher.buildModel();
        CtModel model = launcher.getModel();

        for (Object e : model.getAllTypes()) {
            ClassSemantics cs = new ClassSemantics();
            cs.className = ((CtType) e).getSimpleName();

            Set methods = ((CtType) e).getMethods();
            List attributes = ((CtType) e).getFields();

            cs.isEnum = ((CtType)e).isEnum();
            cs.isInterface = ((CtType)e).isInterface();
            cs.isClass = ((CtType)e).isClass();
            cs.isAbstractClass = ((CtType)e).isClass() && ((CtType)e).isAbstract();

            addMethods(cs, methods);

            addAttributes(cs, attributes);

            semantics.add(cs);

            for (Object nested : ((CtType) e).getNestedTypes()) {
                ClassSemantics csNested = new ClassSemantics();
                csNested.className = ((CtType) nested).getSimpleName();
            }
        }
    }
}

```

```

        Set methodsNested = ((CtType) nested).getMethods();
        List attributesNested = ((CtType) nested).getFields();

        csNested.isEnum = ((CtType)e).isEnum();
        csNested.isInterface = ((CtType)e).isInterface();
        csNested.isClass = ((CtType)e).isClass();
        csNested.isAbstractClass = ((CtType)e).isClass() && ((CtType)e).isAbstract();

        addAttributes(csNested, attributesNested);

        addMethods(csNested, methodsNested);

        semantics.add(csNested);
    }
}

return semantics;
}

private static void addAttributes(ClassSemantics cs, List attributes) {
    for (Object a : attributes) {
        AttributeSemantic attribute = new AttributeSemantic();
        cs.numberOfAttributes++;
        totalNumberOfAttributes++;
        attribute.attributeName = ((CtField) a).getSimpleName();
        attribute.attributeReturnType = ((CtField) a).getType().getSimpleName();

        if (((CtField) a).getVisibility() != null) {
            attribute.attributeVisibility = ((CtField) a).getVisibility().name();

            switch (attribute.attributeVisibility.toLowerCase()) {
                case "public":
                    cs.numberOfPublicAttributes++;
                    break;
                case "private":
                    cs.numberOfPrivateAttributes++;
                    break;
                case "protected":
                    cs.numberOfProtectedAttributes++;
                    break;
                default:
                    cs.numberOfOtherAttributes++;
                    break;
            }
        } else {
            cs.numberOfUndefinedAttributes++;
            attribute.attributeVisibility = "Undefined";
        }
        cs.attributes.add(attribute);
    }
}

private static void addBehaviour(ClassSemantics cs, CtMethod m){
    if(m.getBody() != null){
        if(m.getBody().getStatements().size() > 0){
            for(CtStatement statement : m.getBody().getStatements()){
                try{
                    String state = statement.toString().toLowerCase();
                    if(state != null && !state.contains("throw")){
                        String[] words = state.split(" ");
                    }
                }
            }
        }
    }
}

```



```

    } else if(method.isSetter()){
        cs.numberOfMutators++;
    }
}

private static List<File> getFiles(List<String> paths) {
    List<File> files = new ArrayList<>();

    for(String path : paths){
        files.addAll(listFiles(path));
    }

    return files;
}

private static String getFileExtension(File file) {
    String name = file.getName();
    try {
        return name.substring(name.lastIndexOf(".") + 1);
    } catch (Exception e) {
        return "";
    }
}

private static String getFileName(File file) {
    String s = file.getName();
    return s;
}

public static List<File> listFiles(String directoryName) {
    File directory = new File(directoryName);

    List<File> resultList = new ArrayList<File>();

    // get all the files from a directory
    File[] fList = directory.listFiles();
    //resultList.addAll(Arrays.asList(fList));
    for (File file : fList) {
        if (file.isFile() && "java".equals(getFileExtension(file)) &&
            getFileName(file).indexOf("Test") == -1) {
            resultList.add(file);
            //System.out.println(file.getAbsolutePath());
        } else if (file.isDirectory()) {
            resultList.addAll(listFiles(file.getAbsolutePath()));
        }
    }
    //System.out.println(fList);
    return resultList;
}

private static void addTruongFeatures(List<ClassSemantics> semantics) {
    for(ClassSemantics semantic : semantics){
        addNameFeatures(semantic);
        for(MethodSemantic method : semantic.methods){
            addListMethodFeatures(method, semantic);
            addNonVoidMethodFeatures(method, semantic);
        }
        for(AttributeSemantic attribute : semantic.attributes){
            addListsFeatures(attribute, semantic);
        }
    }
}

private static void addNameFeatures(ClassSemantics semantic) {
    if(semantic.className.contains("Controller") || semantic.className.contains("Manager")){

```

```

        semantic.controller_manager = true;
    }else{
        if(semantic.className.endsWith("er") || semantic.className.endsWith("or")){
            semantic.er_or = true;
        }
    }
}

private static void addListMethodFeatures(MethodSemantic method, ClassSemantics semantic) {
    String param = method.signature.substring(method.signature.indexOf("("));
    String[] params = param.split(",");
    boolean index = false;
    for(String s : params){
        if(s.equals("int")){
            index = true;
            break;
        }
    }
    if(method.methodName.toLowerCase().contains("add") ||
        method.methodName.toLowerCase().contains("delete") ||
        method.methodName.toLowerCase().contains("remove") ||
        method.methodName.toLowerCase().contains("size") ||
        method.methodName.toLowerCase().contains("count") ||
        method.methodName.toLowerCase().contains("child") ||
        method.methodName.toLowerCase().contains("parent") ||
        method.methodName.toLowerCase().contains("has") ||
        (method.methodName.toLowerCase().contains("get") &&
         (method.methodName.toLowerCase().contains("at") ||
          method.methodName.toLowerCase().contains("by") || index))){
        semantic.listMethods++;
    }
}

private static void addListsFeatures(AttributeSemantic attribute, ClassSemantics semantic) {
    if(attribute.attributeReturnType.toLowerCase().contains("list") ||
        attribute.attributeReturnType.toLowerCase().contains("set") ||
        attribute.attributeReturnType.toLowerCase().contains("map") ||
        attribute.attributeReturnType.toLowerCase().contains("collection"))
    {
        semantic.numberOfLists++;
    }
}

private static void addNonVoidMethodFeatures(MethodSemantic method, ClassSemantics semantic) {
    if(method.methodReturnType != "void"){
        semantic.numberNonVoidMethods++;
    }
}
}

```

---

## D.2 Network features (Python)

All of the network features were extracted using Python with the networkX library. The code used to extract these features is as follows:

```

import networkx as nx
import numpy as np
import csv

G_undirected = nx.Graph()
G_directed = nx.DiGraph()

```

```

def createNodes(amount):
    global G_undirected
    global G_directed

    nodelist = list(map(str,range(0,amount)))
    output_dicts = list()

    G_undirected = nx.Graph()
    G_undirected.add_nodes_from(nodelist)

    G_directed = nx.DiGraph()
    G_directed.add_nodes_from(nodelist)

def readEdgelist(path):
    edgelist = open(path, 'r')
    datareader = csv.reader(edgelist, delimiter=',')
    data = []
    i = 0
    for row in datareader:
        if(i > 0):
            data.append(tuple(row))
            i += 1

    return data

def addEdgelist(data):
    global G_undirected
    global G_directed

    G_undirected.add_weighted_edges_from(data)
    G_directed.add_weighted_edges_from(data)

def calculateEccCent(g):
    ecc = dict()
    ecc_cent = dict()
    graphs = list(nx.strongly_connected_component_subgraphs(g))

    for i in range(0, len(graphs)):
        graph = graphs[i]
        if nx.number_of_edges(graph) > 0:
            ecc_sub = nx.eccentricity(graph)
            for k,v in ecc_sub.items():
                ecc[k] = v
                if(v > 0):
                    ecc_cent[k] = 1/v
                else:
                    ecc_cent[k] = 0
        else:
            for node in graph.nodes():
                ecc[node] = 0
                ecc_cent[node] = 0
    return ecc_cent

def calculateEcc(g):
    ecc = dict()
    ecc_cent = dict()
    graphs = list(nx.strongly_connected_component_subgraphs(g))

    for i in range(0, len(graphs)):
        graph = graphs[i]
        if nx.number_of_edges(graph) > 0:
            ecc_sub = nx.eccentricity(graph)
            for k,v in ecc_sub.items():
                ecc[k] = v
        else:

```

```

        for node in graph.nodes():
            ecc[node] = 0
    return ecc

def createOutputDict(closeness_cent, bc_directed, in_deg, out_deg, indeg_cent, outdeg_cent, ecc,
                    ecc_cent):
    output_dicts = list()
    for i in range(0, len(closeness_cent)):
        features = dict()
        valId = str(i)
        features["Id"] = valId
        features["Closeness"] = closeness_cent[valId]
        features["Betweenness"] = bc_directed[valId]
        features["In-degree"] = in_deg[valId]
        features["Out-degree"] = out_deg[valId]
        features["In-degree-cent"] = indeg_cent[valId]
        features["Out-degree-cent"] = outdeg_cent[valId]
        features["Eccentricity"] = ecc[valId]
        features["Eccentricity-cent"] = ecc_cent[valId]
        output_dicts.append(features)
    return output_dicts

def writeCSV(output_dicts, outputPath):
    with open(outputPath + 'structural_features-auto.csv', 'w') as f: # Just use 'w' mode in 3.x
        w = csv.DictWriter(f, output_dicts[0].keys(), delimiter=',', lineterminator='\n')
        w.writeheader()
        w.writerows(output_dicts)

def extractFeatures(outputPath):
    global G_undirected
    global G_directed

    bc_directed = nx.betweenness_centrality(G_directed, normalized = True)
    cc_directed = nx.closeness_centrality(G_directed)
    in_deg = G_directed.in_degree()
    out_deg = G_directed.out_degree()
    indeg_cent = nx.in_degree_centrality(G_directed)
    outdeg_cent = nx.out_degree_centrality(G_directed)
    closeness_cent = nx.closeness_centrality(G_directed)
    ecc = calculateEcc(G_directed)
    ecc_cent = calculateEccCent(G_directed)

    output = createOutputDict(closeness_cent, bc_directed, in_deg, out_deg, indeg_cent,
                              outdeg_cent, ecc, ecc_cent)
    writeCSV(output, outputPath)

def numberOfNodes(data):
    highest = 0
    for row in data:
        highest = max(highest, int(row[0]), int(row[1]))
    return highest

def execute(inputPath, outputPath):
    data = readEdgelist(inputPath)
    n = numberOfNodes(data)
    createNodes(n)
    addEdgelist(data)
    extractFeatures(outputPath)

inp = list()

inp.append('C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Edgelist/K9/edgelist.csv')
inp.append('C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Edgelist/Mars/edgelist.csv')
inp.append('C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Edgelist/Home3D/edgelist.csv')
inp.append('C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Edgelist/Argo/edgelist.csv')

```

```

outp = list()
outp.append('C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/K9/')
outp.append('C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/Mars/')
outp.append('C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/Home3D/')
outp.append('C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/Argo/')

for i in range(0, 4):
    execute(inp[i], outp[i])
    print(G_directed.number_of_nodes())

```

---

## E Source code combining features

This appendix contains the source code used to combine the network and semantic software features of the different projects together to three files: full-feature set, network-feature set and semantic software feature set. The code is one python scripts which uses the outputs of Appendix D as input.

---

```

import csv as csv
def getSemanticFeatureList(path):
    reader = csv.DictReader(open(path, 'r'))
    feature_list = []
    for line in reader:
        feature_list.append(line)
    return feature_list

def getLabelList(path):
    reader2 = csv.DictReader(open(path, 'r'), delimiter=",")
    label_list = []
    for line in reader2:
        label_list.append(line)
    return label_list

def getStructuralFeatureList(path):
    reader3 = csv.DictReader(open(path, 'r'), delimiter=",")
    structural_feature_list = []
    for line in reader3:
        structural_feature_list.append(line)
    return structural_feature_list

def writeToCsv(featurelist, path):
    with open(path, 'w') as f: # Just use 'w' mode in 3.x
        w = csv.DictWriter(f, featurelist[0].keys(), delimiter=',', lineterminator='\n')
        w.writeheader()
        w.writerows(featurelist)

def mergeFeatures(struct, sem):
    merged = []
    for i in range(0, len(struct)):
        for j in range(0, len(sem)):
            stid = struct[i]["Id"]
            semid = sem[j]["Id"]
            if struct[i]["Id"] == sem[j]["Id"]:
                merge = {**struct[i], **sem[j]}
                merged.append(merge)
    return merged

def addLabels(merged, labels):
    for i in range(0, len(merged)):
        for j in range(0, len(labels)):
            if merged[i]["Id"] == labels[j]["Id"]:
                merged[i]["Label"] = labels[j]["Role"]
    return merged

```

```

def addIds(sem, labels):
    for i in range(0, len(sem)):
        for j in range(0, len(labels)):
            if sem[i]["Class name"].strip().lower() == labels[j]["Class"].strip().lower():
                sem[i]["Id"] = labels[j]["Id"]
    return sem

def execute(inp, outp):
    struct = getStructuralFeatureList(inp["Structural"])
    sem = getSemanticFeatureList(inp["Semantic"])
    labels = getLabelList(inp["Labels"])
    sem = addIds(sem, labels)
    merged = mergeFeatures(struct, sem)
    final = addLabels(merged, labels)
    #print(final[0])
    writeToCsv(final, outp)

paths = dict()

paths["Structural"] = "C:/Users/xavyr/Desktop/CS and
    SBB/Thesis/Data/Features/Mars/structural_features-auto.csv"
paths["Semantic"] = "C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/Mars/semantics.csv"
paths["Labels"] = "C:/Users/xavyr/Desktop/CS and
    SBB/Thesis/Data/Edgelists/Mars/mapping-mars-auto-labelled.csv"

output = "C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/Mars/full-labelled.csv"

execute(paths, output)

paths = dict()

paths["Structural"] = "C:/Users/xavyr/Desktop/CS and
    SBB/Thesis/Data/Features/K9/structural_features-auto.csv"
paths["Semantic"] = "C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/K9/semantics.csv"
paths["Labels"] = "C:/Users/xavyr/Desktop/CS and
    SBB/Thesis/Data/Edgelists/K9/mapping-k9-auto-labelled.csv"

output = "C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/K9/full-labelled.csv"

execute(paths, output)

paths = dict()

paths["Structural"] = "C:/Users/xavyr/Desktop/CS and
    SBB/Thesis/Data/Features/Home3D/structural_features-auto.csv"
paths["Semantic"] = "C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/Home3D/semantics.csv"
paths["Labels"] = "C:/Users/xavyr/Desktop/CS and
    SBB/Thesis/Data/Edgelists/Home3D/mapping-home3d-auto-labelled.csv"

output = "C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/Home3D/full-labelled.csv"

execute(paths, output)

reader = csv.DictReader(open("C:/Users/xavyr/Desktop/CS and
    SBB/Thesis/Data/Features/Home3D/full-labelled.csv", 'r'))
feature_list = []
for line in reader:
    feature_list.append(line)

reader = csv.DictReader(open("C:/Users/xavyr/Desktop/CS and
    SBB/Thesis/Data/Features/K9/full-labelled.csv", 'r'))
feature_list2 = []
for line in reader:
    feature_list2.append(line)

```

```

reader = csv.DictReader(open("C:/Users/xavyr/Desktop/CS and
    SBB/Thesis/Data/Features/Mars/full-labelled.csv", 'r'))
feature_list3 = []
for line in reader:
    feature_list3.append(line)

full_features = feature_list + feature_list2 + feature_list3
final_features = []
for dic in full_features:
    row = dict()
    for k, v in dic.items():
        if k != "Id": #and k != "Class name":
            row[k] = v
    final_features.append(row)

with open("C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/full/full-labelled-classes.csv",
    'w') as f: # Just use 'w' mode in 3.x
    w = csv.DictWriter(f, final_features[0].keys(), delimiter=',', lineterminator='\n')
    w.writeheader()
    w.writerows(final_features)

full_network = []
full_semantic = []
network_features =
    ["In-degree", "Eccentricity-cent", "Eccentricity", "Out-degree", "Closeness", "In-degree-cent", "Betweenness", "Out-
for dic in full_features:
    row_n = dict()
    row_s = dict()
    for k,v in dic.items():
        if k in network_features:
            row_n[k] = v
        elif k == "Label":
            row_n[k] = v
            row_s[k] = v
        elif k != "Id" and k != "Class name":
            row_s[k] = v
    full_network.append(row_n)
    full_semantic.append(row_s)

with open("C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/full/network-labelled.csv", 'w')
    as f: # Just use 'w' mode in 3.x
    w = csv.DictWriter(f, full_network[0].keys(), delimiter=',', lineterminator='\n')
    w.writeheader()
    w.writerows(full_network)

with open("C:/Users/xavyr/Desktop/CS and SBB/Thesis/Data/Features/full/semantic-labelled.csv",
    'w') as f: # Just use 'w' mode in 3.x
    w = csv.DictWriter(f, full_semantic[0].keys(), delimiter=',', lineterminator='\n')
    w.writeheader()
    w.writerows(full_semantic)

```

---

## F Source code classifiers

This appendix contains the source code of the classification scripts. This includes:

- Imports and loading the data
- Smote algorithm
- Gridsearches and results
- Optimal hyper-parameters per algorithm-dataset combination
- Printing class names of misclassified instances

- Feature importances

Since it is one long script we added comments to make it clear where each part starts.

---

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.linear_model import SGDClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_predict
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler

import numpy as np
from sklearn.metrics import f1_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix
from sklearn.svm import LinearSVC
from sklearn.svm import SVC
from imblearn.over_sampling import SMOTE
import collections
from sklearn.tree import DecisionTreeClassifier
from sklearn.base import clone
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import GridSearchCV

import random

def load_data(csv_path):
    return pd.read_csv(csv_path)

def accuracy_percentage(actual, predictions):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predictions[i]:
            correct += 1
    return correct / len(actual)

full_data = load_data("C:/Users/xavyr/Desktop/CS and
    SBB/Thesis/Data/Features/full/full-labelled-classes.csv")
network = load_data("C:/Users/xavyr/Desktop/CS and
    SBB/Thesis/Data/Features/full/network-labelled.csv")
other = load_data("C:/Users/xavyr/Desktop/CS and
    SBB/Thesis/Data/Features/full/semantic-labelled.csv")

#split data in sets
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(full_data, full_data["Label"].get_values()):
    train_set = full_data.loc[train_index]
    test_set = full_data.loc[test_index]
    train_network = network.loc[train_index]
    test_network = network.loc[test_index]
    train_sem = other.loc[train_index]
    test_sem = other.loc[test_index]

train_X = train_set.drop(["Label", "Other attributes", "Other methods", "Class name"], axis=1)
train_y = train_set["Label"].copy()
train_classes = train_set["Class name"].copy()
test_X = test_set.drop(["Label", "Other attributes", "Other methods", "Class name"], axis=1)
test_y = test_set["Label"].copy()
test_classes = test_set["Class name"].copy()

train_net_X = train_network.drop("Label", axis=1)
train_net_y = train_network["Label"].copy()
test_net_X = test_network.drop("Label", axis=1)
```

```

test_net_y = test_network["Label"].copy()
train_sem_X = train_sem.drop(["Label", "Other attributes", "Other methods"], axis=1)
train_sem_y = train_sem["Label"].copy()
test_sem_X = test_sem.drop(["Label", "Other attributes", "Other methods"], axis=1)
test_sem_y = test_sem["Label"].copy()

#SMOTE
sm = SMOTE(random_state=42, ratio = 1.0, k_neighbors=4)
x_train_sm, y_train_sm = sm.fit_sample(train_X, train_y)
x_train_net_sm, y_train_net_sm = sm.fit_sample(train_net_X, train_net_y)
x_train_sem_sm, y_train_sem_sm = sm.fit_sample(train_sem_X, train_sem_y)

#gridsearch&results logistic regression
logit_clf = LogisticRegression()
logit_params = [{'penalty': ["l2"], 'dual': [False, True], 'C':[0.01, 0.1, 0.5, 0.75, 1],
                'fit_intercept':[False, True],
                'class_weight':['balanced', None], 'solver':['liblinear']}],
                {'penalty': ["l2"], 'dual': [False], 'C':[0.01, 0.1, 0.5, 0.75, 1],
                'fit_intercept':[False, True],
                'class_weight':['balanced', None], 'solver':['newton-cg', 'sag', 'lbfgs', 'saga']}],
                {'penalty': ["l1"], 'dual': [False], 'C':[0.01, 0.1, 0.5, 0.75, 1],
                'fit_intercept':[False, True],
                'class_weight':['balanced', None], 'solver':['saga', 'liblinear']}]
grid_search_full = GridSearchCV(logit_clf, logit_params, cv=5, verbose=3, n_jobs=-1)
grid_search_full.fit(x_train_sm, y_train_sm)

grid_search_net = GridSearchCV(logit_clf, logit_params, cv=5, verbose=3, n_jobs=-1)
grid_search_net.fit(x_train_net_sm, y_train_net_sm)

grid_search_sem = GridSearchCV(logit_clf, logit_params, cv=5, verbose=3, n_jobs=-1)
grid_search_sem.fit(x_train_sem_sm, y_train_sem_sm)

print("train sm logit: ",f1_score(y_train_sm, grid_search_full.predict(x_train_sm),
                                average="macro"))
print("test sm logit: ",f1_score(test_y, grid_search_full.predict(test_X), average="macro"))
print("train sm logit acc: ",accuracy_percentage(y_train_sm, grid_search_full.predict(x_train_sm)))
print("test sm logit acc: ",accuracy_percentage(test_y.values, grid_search_full.predict(test_X)))
print(confusion_matrix(y_train_sm, grid_search_full.predict(x_train_sm)))
print(confusion_matrix(test_y.values, grid_search_full.predict(test_X)))

print("-----Network-----")
print("train sm logit net: ",f1_score(y_train_net_sm, grid_search_net.predict(x_train_net_sm),
                                average="macro"))
print("test sm logit net: ",f1_score(test_y, grid_search_net.predict(test_net_X), average="macro"))
print("train sm logit net acc: ",accuracy_percentage(y_train_net_sm,
                                grid_search_net.predict(x_train_net_sm)))
print("test sm logit net acc: ",accuracy_percentage(test_y.values,
                                grid_search_net.predict(test_net_X)))
print(confusion_matrix(y_train_net_sm, grid_search_net.predict(x_train_net_sm)))
print(confusion_matrix(test_y.values, grid_search_net.predict(test_net_X)))

print("-----Non-Network-----")
print("train sm logit sem: ",f1_score(y_train_sem_sm, grid_search_sem.predict(x_train_sem_sm),
                                average="macro"))
print("test sm logit sem: ",f1_score(test_y, grid_search_sem.predict(test_sem_X), average="macro"))
print("train sm logit sem acc: ",accuracy_percentage(y_train_sem_sm,
                                grid_search_sem.predict(x_train_sem_sm)))
print("test sm logit sem acc: ",accuracy_percentage(test_y.values,
                                grid_search_sem.predict(test_sem_X)))
print(confusion_matrix(y_train_sem_sm, grid_search_sem.predict(x_train_sem_sm)))
print(confusion_matrix(test_y.values, grid_search_sem.predict(test_sem_X)))

#gridsearch & results random forest
dec_tree = DecisionTreeClassifier()

```

```

tree_params = [{'criterion':['gini', 'entropy'], 'splitter': ['best', 'random'], 'max_depth':[5,
10, 25, 50, 75, 100, None],
'min_samples_leaf': [2, 5, 10, 25, 50, 75, 100], 'max_features':[1, 3, 5, None],
'max_leaf_nodes':[2, 5, 10, 25, None], 'class_weight':['balanced', None]]}

tree_search_full = GridSearchCV(dec_tree, tree_params, cv=5, verbose=3, n_jobs=-1)
tree_search_full.fit(x_train_sm, y_train_sm)

tree_search_net = GridSearchCV(dec_tree, tree_params, cv=5, verbose=3, n_jobs=-1)
tree_search_net.fit(x_train_net_sm, y_train_net_sm)

tree_search_sem = GridSearchCV(dec_tree, tree_params, cv=5, verbose=3, n_jobs=-1)
tree_search_sem.fit(x_train_sem_sm, y_train_sem_sm)

print("train sm tree: ",f1_score(y_train_sm, tree_search_full.predict(x_train_sm),
average="macro"))
print("test sm tree: ",f1_score(test_y, tree_search_full.predict(test_X), average="macro"))
print("train sm tree acc: ",accuracy_percentage(y_train_sm, tree_search_full.predict(x_train_sm)))
print("test sm tree acc: ",accuracy_percentage(test_y.values, tree_search_full.predict(test_X)))
print(confusion_matrix(y_train_sm, tree_search_full.predict(x_train_sm)))
print(confusion_matrix(test_y.values, tree_search_full.predict(test_X)))

print("-----Network-----")
print("train sm tree net: ",f1_score(y_train_net_sm, tree_search_net.predict(x_train_net_sm),
average="macro"))
print("test sm tree net: ",f1_score(test_y, tree_search_net.predict(test_net_X), average="macro"))
print("train sm tree net acc: ",accuracy_percentage(y_train_net_sm,
tree_search_net.predict(x_train_net_sm)))
print("test sm tree net acc: ",accuracy_percentage(test_y.values,
tree_search_net.predict(test_net_X)))
print(confusion_matrix(y_train_net_sm, tree_search_net.predict(x_train_net_sm)))
print(confusion_matrix(test_y.values, tree_search_net.predict(test_net_X)))

print("-----Non-Network-----")
print("train sm tree sem: ",f1_score(y_train_sem_sm, tree_search_sem.predict(x_train_sem_sm),
average="macro"))
print("test sm tree sem: ",f1_score(test_y, tree_search_sem.predict(test_sem_X), average="macro"))
print("train sm tree sem acc: ",accuracy_percentage(y_train_sem_sm,
tree_search_sem.predict(x_train_sem_sm)))
print("test sm tree sem acc: ",accuracy_percentage(test_y.values,
tree_search_sem.predict(test_sem_X)))
print(confusion_matrix(y_train_sem_sm, tree_search_sem.predict(x_train_sem_sm)))
print(confusion_matrix(test_y.values, tree_search_sem.predict(test_sem_X)))

#gridsearch & results random forest
ran_for = RandomForestClassifier()
for_params = [{'n_estimators':[2, 5, 10, 15], 'criterion':['gini', 'entropy'], 'max_features':[1,
3, 5, None],
'max_depth':[5, 10, 25, 50, 75, 100, None], 'min_samples_leaf': [2, 5, 10, 25, 50,
75, 100],
'min_samples_split': [2, 5, 10, 25], 'max_leaf_nodes':[2, 5, 10, 25, None],
'class_weight':['balanced', None],
'oob_score': [False, True]]}

for_search_full = GridSearchCV(ran_for, for_params, cv=5, verbose=3, n_jobs=-1)
for_search_full.fit(x_train_sm, y_train_sm)

for_search_net = GridSearchCV(ran_for, for_params, cv=5, verbose=3, n_jobs=-1)
for_search_net.fit(x_train_net_sm, y_train_net_sm)

for_search_sem = GridSearchCV(ran_for, for_params, cv=5, verbose=3, n_jobs=-1)
for_search_sem.fit(x_train_sem_sm, y_train_sem_sm)

print("train sm forest: ",f1_score(y_train_sm, for_search_full.predict(x_train_sm),

```

```

        average="macro"))
print("test sm forest: ",f1_score(test_y, for_search_full.predict(test_X), average="macro"))
print("train sm forest acc: ",accuracy_percentage(y_train_sm, for_search_full.predict(x_train_sm)))
print("test sm forest acc: ",accuracy_percentage(test_y.values, for_search_full.predict(test_X)))
print(confusion_matrix(y_train_sm, for_search_full.predict(x_train_sm)))
print(confusion_matrix(test_y.values, for_search_full.predict(test_X)))

print("-----Network-----")
print("train sm forest net: ",f1_score(y_train_net_sm, for_search_net.predict(x_train_net_sm),
        average="macro"))
print("test sm forest net: ",f1_score(test_y, for_search_net.predict(test_net_X), average="macro"))
print("train sm forest net acc: ",accuracy_percentage(y_train_net_sm,
        for_search_net.predict(x_train_net_sm)))
print("test sm forest net acc: ",accuracy_percentage(test_y.values,
        for_search_net.predict(test_net_X)))
print(confusion_matrix(y_train_net_sm, for_search_net.predict(x_train_net_sm)))
print(confusion_matrix(test_y.values, for_search_net.predict(test_net_X)))

print("-----Non-Network-----")
print("train sm forest sem: ",f1_score(y_train_sem_sm, for_search_sem.predict(x_train_sem_sm),
        average="macro"))
print("test sm forest sem: ",f1_score(test_y, for_search_sem.predict(test_sem_X), average="macro"))
print("train sm forest sem acc: ",accuracy_percentage(y_train_sem_sm,
        for_search_sem.predict(x_train_sem_sm)))
print("test sm forest sem acc: ",accuracy_percentage(test_y.values,
        for_search_sem.predict(test_sem_X)))
print(confusion_matrix(y_train_sem_sm, for_search_sem.predict(x_train_sem_sm)))
print(confusion_matrix(test_y.values, for_search_sem.predict(test_sem_X)))

#best hyper-parameters per algorithm-dataset combination
print("Forest:")
print("Full ", for_search_full.best_params_)
print("Net ", for_search_net.best_params_)
print("Sem ", for_search_sem.best_params_)

print("Logit:")
print("Full ", grid_search_full.best_params_)
print("Net ", grid_search_net.best_params_)
print("Sem ", grid_search_sem.best_params_)

print("Tree:")
print("Full ", tree_search_full.best_params_)
print("Net ", tree_search_net.best_params_)
print("Sem ", tree_search_sem.best_params_)

#feature importances random forest
forest_clf = RandomForestClassifier(n_estimators=15, max_depth=75, max_leaf_nodes=None,
        criterion="gini", oob_score=False, min_samples_leaf=2, class_weight=None, max_features=5,
        min_samples_split=2)
forest_clf.fit(x_train_sm, y_train_sm)
test_predict_sem_sm = forest_clf.predict(test_X)
print(accuracy_percentage(test_y.values, test_predict_sem_sm))

importances = forest_clf.feature_importances_
std = np.std([tree.feature_importances_ for tree in forest_clf.estimators_],
        axis=0)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(x_train_sem_sm.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

#print misclassified class names

```

```
forest_clf = RandomForestClassifier(n_estimators=15, max_depth=75, max_leaf_nodes=None,
    criterion="gini", oob_score=False, min_samples_leaf=2, class_weight=None, max_features=5,
    min_samples_split=2)
forest_clf.fit(x_train_sm, y_train_sm)
pred = forest_clf.predict(test_X)
print(accuracy_percentage(test_y.get_values(), pred))
count = 0
for i in range(len(pred)):
    predict = pred[i].strip().lower()
    tar = test_y.get_values()[i].strip().lower()
    if predict != tar:
        count = count + 1
        print(test_classes.get_values()[i], " ", predict, " - ", tar)
print(count)
print(len(test_y))
```

---