# DETECTING CODE SMELLS WITH SPNS

by

**Mustafa el Bouazzaoui**

in partial fulfillment of the requirements for the degree of

**Master of Science**

in Software Engineering

at the Open University, faculty of Science
Master Software Engineering

to be defended publicly on Friday November 18, 2022 at 10:00 AM.

| | | |
|---|---|---|
| Course code: | IM9906 | |
| Supervisors: | dr. Arjen Hommersom, | Open University |
| | dr. ir. Harrie Passier, | Open University |

Open Universiteit
de beste!
www.ou.nl

# CONTENTS

# Abstract

Software systems are getting more complex, leading to the risk of introducing technical debt [27], i.e., sub-optimal implementation decisions that provide short-term benefits but cause a decrease of software quality. The presence of technical debt usually also indicates the presence of code smells within that same system [15]. Code smells, introduced in [15], indicate that important software design and implementation principles were violated in the source code of a software application during its life cycle. These code smells lead to an increase in the complexity of the software, hence also leading to difficulties regarding the comprehensibility and maintenance of the software application.

Checking the source code to identify code smells manual is a time-consuming and complex process. This is mainly due to a lack of knowledge and the fact that the detection of potential code smells is prone to subjective interpretation by developers. Hence research has been performed in detecting code smells automatic to support developers. This led to a variety of tools, i.e., static analysis tools, implementing heuristic-based approaches that are simple and easy. However, this again leads to a lot of uncertainty during the process of identifying code smells. This was mainly because the list of potential code smells, produced by the static analysis tools, was also prone to subjective interpretation by developers. Therefore, manual inspection is still necessary when using these types of tools, thus making the entire process of detection still time-consuming. This in turn limits the adoption of code smell detection in practice.

To overcome the limitations of these types of tools researchers proposed various code smell detection mechanisms using machine learning techniques. For this study, a deep learning approach using sum-product networks (SPNs) is proposed to detect code smells. This is done by learning from both code metrics and word embeddings extracted from the source code. Several experiments have been carried out to detect the following three code smells Long Method, Feature Envy, and Large Class. The results were compared to a deep learning approach, using neural networks, to detect code smells as covered in [26]. This deep learning approach outperformed the state-of-the-art static analysis tools currently available. The results of the SPN models showed that detecting code smell Long Method performed at least as well as the deep learning approach defined in [26]. With regards to code smells Feature Envy and Large class the SPN models under-performed the deep learning approach. Therefore, more research is needed regarding the potential of SPNs to detect code smells. This further research should also include a more extensive dataset than was used for this study.

Furthermore, as of this writing no code smell detection tools are available that use SPNs. With several experiments, part of a case study, it is shown that SPNs do have potential regarding practical applicability. Therefore, to evaluate the practical applicability in a development session, a tool should be developed that employs SPNs to detect code smells.

# 1

# INTRODUCTION

Software systems are getting more complex by the day. Furthermore, companies are required to continuously update their software applications to offer new features. These continuous changes frequently occur under time pressure and lead developers to set aside good design/programming practices and principles to deliver a workable, but still immature software application. A side effect of this process is the risk of introducing technical debt [27], i.e., sub-optimal implementation decisions that provide short-term benefits but cause a decrease in software quality.

The presence of technical debt is usually also an indication of the presence of code smells within a software system [15]. Code smells are defined as bad patterns in source code that violate important principles of software design and implementation, e.g., the introduction of complex and/or long classes, excessive coupling between objects, etc. These bad patterns usually decrease the software quality of an application due to an increase in complexity, and thus maintenance efforts. In [15], the authors defined a list of 22 code smells, but stop short of outlining how to detect a code smell. In addition, developers may not be aware of the various code smells and the reasoning behind them due to a lack of experience and knowledge. Therefore, checking the source code manually to identify code smells can be a tiresome, tedious, and time-consuming process. Furthermore, this process is also prone to subjective interpretation.

Therefore, several techniques have been proposed to detect code smells automatically, intending to support developers in their software development process [34][1]. This led to a variety of static analysis tools, used by developers, based on various software quality metrics and thresholds. Though these heuristic-based approaches are simple and easy to implement, there is still a lot of uncertainty in identifying code smells with these types of tools. A major cause for this is the following, namely that the list of potential code smells generated by these tools is also prone to subjective interpretation by developers. There is no uniform use of metrics and thresholds to determine whether a piece of code, i.e., code entity, is a code smell or not [13]. This in turn can lead to a long list "smelly" code entities that developers might not perceive as code smells, thus false positives. Therefore, manual checking would still be necessary, making the entire process of code smell detection still time-consuming, thus limiting the adoption of code smell detectors in practice [35].

To overcome the limitations of these tools based on static analysis, researchers have proposed new detection strategies by employing various machine learning (ML) techniques [14][9]. By employing ML techniques to detect code smells, the aim is to decrease the false positives that are common when using static analyzers. Though the use of ML techniques to detect code smells looks promising, thresholds are still needed to be configured. This can lead to the same limitations as most static analyzers, and hence more research is needed [26][22]. Therefore, these types of code smell detectors (i.e., discriminative models [11]) tend to misclassify code smells due to uncertainty when a code entity is a code smell or not, and its dependence on feature selection, i.e., which code metrics to use.

To decrease the uncertainty in the process of code smell detection, probabilistic graphical models (PGMs) seem like a promising alternative. PGMs are a type of deep learning (DL) technique aiming to model the relationships that exist between a set of random variables (RVs), represented by a list of values of a classification feature, in a graphical way [38]. The key insight in these probabilistic modeling techniques is that they could capture relationships between different (classification) features that are not obvious when using the discriminative models [9]. Hence, making it powerful to draw inferences on some unobserved variables, given the evidence on observed variables [18]. However, most probabilistic models take a very long time to perform various inference tasks, as these inference tasks are considered intractable. Therefore, we chose a deep learning approach using sum-product networks (SPNs) [33] to detect code smells for this study. SPNs are a considerable improvement when compared to many PGMs, as the time of inference with these type of models are much more efficient and thus faster [39]. With SPNs, the underlying models will eventually learn how to identify whether a code entity is a code smell or not based on the training data. The results of this SPN-based approach to detect code smells are compared to the deep learning approach, based on neural networks, defined in [26].

The remainder of this thesis is structured as follows. Chapter 2 outlines the necessary preliminaries needed for this study. In Chapter 3, the related work relevant to this study is covered. In Chapter 4, the research design is outlined, by defining the problem statement and research questions. In Chapter 5, a proposal is outlined for a more detailed SPN-based approach to detect code smells. In Chapter 6, the setup and results of the different experiments are covered. In Chapter 7, a case study regarding the applicability of SPNs is discussed. Finally, this chapter is followed by discussion, conclusion, and future work.

# 2

# PRELIMINARIES

This chapter provides the background on several concepts relevant to this study. Therefore, in Section 2.1 a general overview of several code smells is given. In Section 2.2, several machine learning techniques are covered. This leads to a general overview of sum-product networks in Section 2.3. Finally, in Section 2.4 the widely used word embedding technique *word2vec* will be covered.

## 2.1 CODE SMELLS

In this section, a background on code smells is provided. Additionally, three of the most common code smells are also covered. These three code smells are also the focus of this study.

### 2.1.1 DEFINITIONS

Countless hours and significant resources are lost on maintenance because of poor design and bad choices about the implementation of software. Studies have shown that this results in source code that is difficult to comprehend and hard to maintain [45]. Furthermore, this also increases the risk of introducing technical debt [27], i.e., sub-optimal implementation decisions that provide short-term benefits. This causes the quality of the source code to be impacted negatively, leading to a range of issues, and this in turn leads to a lot of time spent on maintenance [15]. Therefore, the presence of technical debt is usually also an indication of the presence of code smells. Code smells are defined as bad patterns in source code that violate important principles of software design and implementation, e.g., the introduction of complex and/or long classes, excessive coupling between objects, etc. These bad patterns usually decrease software quality. In [15], the authors defined a list of 22 code smells that indicate when the source code of a software system needs refactoring to improve the software quality [15]. Furthermore, with each code smell several refactor strategies were introduced to resolve the specific code smells. Although the 22 code smells imply poor design/implementation choices, their frequencies in source code repositories differ greatly. The three code smells that affect the software quality the most are the following: Long Method, Feature Envy, and Large Class [32]. These code smells are also the main focus of this study and will be discussed in more detail in the next sections. Different code metrics also have been defined to serve as indicators to help identify the different code smells [23]. These code metrics will be covered in more detail in Chapter 5. However, the focus will be on the code metrics that help identify the three aforementioned code smells.

### 2.1.2 Long Method

The Long Method code smell refers to methods that have become too long and do too many things, leading to increased complexity. Therefore, long methods are generally hard for developers to comprehend and maintain. The Long Method code smell is usually refactored by employing one of the following refactor strategies: Extract Method, Replace Temp with Query, Introduce Parameter Object, and Preserve Whole Object [15]. The essence of these refactor strategies is to decompose the method into smaller methods, or data objects, which can then be invoked/called from the original method. For example, a given heuristic is that whenever a block of code in a method could use commenting, it should be extracted to a method instead [15]. Fowler and Beck in [15] did not specify metric(s) nor which threshold values to consider when certain methods are too long. Therefore, it is not always clear when a certain method is too long.

### 2.1.3 Feature Envy

The Feature Envy code smell refers to a method that seems to be more "interested" in a class other than its parent class. This usually means that the specific method accesses lots of features, i.e., fields and methods, of another class than its parent class [15]. Thus the method "envies" another class more than its parent class, as it is strongly coupled with that other class. This impacts the cohesion of the parent class negatively, making it riskier to change something without causing potential bugs in other parts of the software.

The Feature Envy code smell is refactored by employing one of the following refactor strategies: Move Method, and Extract Method [15]. The essence of the refactor strategies is to simply move the method to the class that the method is "interested" in. A basic example of a method that suffers from the code smell Feature Envy is shown in Listing 2.1[1]. The method `getMobilePhoneNumber()` is "smelly" and its logic should be moved to the Phone class. The result of this refactor strategy is shown in Listing 2.2.

Listing 2.1: A basic example of a method that suffers from the code smell Feature Envy.

```java
public class Phone {
   private final String unformattedNumber;
   public Phone(String unformattedNumber) {
      this.unformattedNumber = unformattedNumber;
   }
   public String getAreaCode() {
      return unformattedNumber.substring(0,3);
   }
   public String getPrefix() {
      return unformattedNumber.substring(3,6);
   }
   public String getNumber() {
      return unformattedNumber.substring(6,10);
   }
```

[1]https://sourcecodeera.com/blogs/Samath/Phone-Class-using-Java.aspx

```java
}

public class Customer{
   private Phone mobilePhone;
   public String getMobilePhoneNumber() {
      return "(" +
         mobilePhone.getAreaCode() + ") " +
         mobilePhone.getPrefix() + "-" +
         mobilePhone.getNumber();
   }
}
```

Listing 2.2: A basic example of a method refactored with the Move Method refactor strategy.

```java
public class Phone {
   private final String unformattedNumber;
   public Phone(String unformattedNumber) {
      this.unformattedNumber = unformattedNumber;
   }
   private String getAreaCode() {
      return unformattedNumber.substring(0,3);
   }
   private String getPrefix() {
      return unformattedNumber.substring(3,6);
   }
   private String getNumber() {
      return unformattedNumber.substring(6,10);
   }
   public String toFormattedString() {
      return "(" + getAreaCode() + ") " + getPrefix() + "-" + getNumber();
   }
}

public class Customer
   private Phone mobilePhone;
   public String getMobilePhoneNumber() {
      return mobilePhone.toFormattedString();
   }
```

Therefore, the heuristic used to detect this code smell is to determine whether the method under investigation envies another class more than its parent class. Following this, a measure should be given for envy between the method under investigation and different potential target classes, including its parent class. This is not always straightforward, as detecting this code smell also depends on context, knowledge, and experience. It is also possible that a method envies several classes, to varying degrees, thus increasing the complexity. There also might be valid reasons to not view this pattern as a code smell Feature Envy, as there are sophisticated patterns that break this rule (e.g., Visiting Pattern).

### 2.1.4 Large Class

The Large Class code smell is a class (e.g., in Java or C#) that has become too large over time. That is because these large classes have evolved over time leading to too many responsibilities, thus having too many fields and methods. Such large classes are usually a breeding ground for duplicated code, complexity, and chaos, thus decreasing the ability to comprehend the code by a developer [15]. Finally, large classes often break the single responsibility principle. This design principle states that any given class should only have one reason for change, and thus a class should only have one responsibility [27]. The reasoning is that responsibilities should be decoupled so changes to one responsibility within a class cannot unexpectedly "break" other classes. This reasoning is closely related to the degree to which the fields/methods inside a class belong together, i.e., cohesion, and should not depend too much on other classes.

The Large Class code smell is usually refactored by employing one of the following refactor strategies: Extract Class, Extract Subclass, and Duplicate Observed Data [15]. The essence of these refactor strategies entails the following. Create a new class, or subclass, and move the relevant fields and methods from the old class into the new class. In addition to making the old class more maintainable, it also makes it more comprehensible. Fowler and Beck in [15] did not specify specific metric(s) nor threshold values to consider when classes are too large. Hence, uncertainty can arise in some cases whether a class is too large or not.

## 2.2 Machine Learning

The main goal of this section is to cover the different machine learning techniques relevant to this study. In Section 2.2.1, the basic concepts of machine learning are covered. In Section 2.2.2, the concepts of deep learning are covered. Finally, Section 2.2.3 covers another sub-field of machine learning, namely probabilistic graphical models that will serve as a prelude to sum-product networks, covered in the next section.

### 2.2.1 Basic Concepts

When machine learning (ML) is mentioned, usually a lot of excitement and confusion arises. People picture an intelligent robot, or a computer system, akin to Skynet[2]. However, it is important to realize that ML is a sub-field of Artificial Intelligence (AI). With ML, models or systems are created by learning from data without being explicitly programmed [11]. This learning process also includes improving the model. The algorithms used to create these ML models use statistical techniques to recognize patterns in data and then make predictions. ML has been around for many years and powers many of the services used today, such as Netflix, Google, Facebook, etc.

The underlying models of ML systems are based on the following idea. Let $X$ denote data we know about, i.e., instance values or samples, and let $Y$ denote the results we are interested in, i.e., the labeled instance classes. The key question in ML is how to model the relationship between $X$ and $Y$. The type of model to be used is based on the problem at hand. However, many types of models can be employed with regard to different ML tech-

[2]In the movie the Terminator, Skynet is an advanced Artificial Intelligence system threatening humankind. Skynet is therefore often used as an analogy when Artificial Intelligence systems are sufficiently advanced in such a way that they can threaten the status quo in a negative way.

niques, so we will classify them into broad categories first. This is done using the following criteria:

- Whether or not the models are training, i.e., learning, with supervision (**supervised** vs **unsupervised** vs **reinforcement**).
- The way input data is mapped on labeled data (**discriminative** vs **generative**).

Note that the types of ML techniques grouped by the criteria above are not mutually exclusive. With **supervised learning**, the data is labeled to tell the ML model exactly what patterns it should look for. For example, Netflix suggests a show that a user would like based on his or her profile. The model used by Netflix is trained using supervised learning, as the model learns from preferences/choices of other users that have a similar profile. With **unsupervised learning**, the data have no labels. The ML model identifies data clusters that have a similar pattern. For instance grouping images based on certain specifics, such as images of cars or animals. Finally, with **reinforcement learning** the ML model learns by means of trial-and-error and converges to a specified objective. This means that it tries out lots of different routes and is rewarded (or penalized) depending on whether its behavior helps (or hinders) it from reaching the specified objective. AlphaGo, created by Google, is "powered" by a reinforcement learning model. This AI system beat some of the best human players in the game of Go, known for being highly complex to play [42].

**Discriminative** classifiers train by learning the decision boundaries between the different classes [40]. The result is a discriminative model that creates a direct mapping from $X$ to $Y$, where $X$ is the input data, and $Y$ is the class label [31][49]. Thus the trained model aims to predict the labels from the input data. Examples of such classifiers are the following [40]: Logistic regression, support vector machines, decision trees, and deep neural network models, discussed in Section 2.2.2. Alternatively, **generative** classifiers train a model by modeling the probability distributions of each class in $Y$ [40]. By using Bayes' rule the most likely class label is selected [31][49]. Thus the trained model aims to explain how the input data was generated. Examples of such models are the following [40]: Hidden Markov Models, Naïve Bayes, and Bayesian Belief Networks, discussed in Section 2.2.3.

Finally, for an ML model to learn, the following components also need to be taken into consideration:

**Datasets** ML models are created by training on a collection of labeled instance data/samples, i.e., datasets.

**Features** Features, or classification features, are relevant characteristics within the dataset that the ML model employs for training purposes.

**Algorithm** Different algorithms can be used to solve the problem at hand [17][11]. It is very important to select the right algorithm, or an ensemble of algorithms, as different algorithms can lead to different accuracy scores and speed. These algorithms are then incorporated into the ML models.

### 2.2.2 Deep Learning

A sub-field of ML is deep learning (DL). However, before elaborating on DL, it is important to start with neural networks. Neural networks (NNs) is a technique that tries to "emulate" the working of the human brain. The brain can be seen as an interconnected network of neurons, hence the name neural network (NN). In a NN, a neuron (also called a node) is a unit that computes an activation in the form of a real value. Each node receives input, also a real value, from other nodes or external sources and computes an output. Each input has a weight ($w$), indicating its relative importance to other input values. The node computes an output by applying a function $f$ to the weighted sum of its input values, as shown in Figure 2.1.

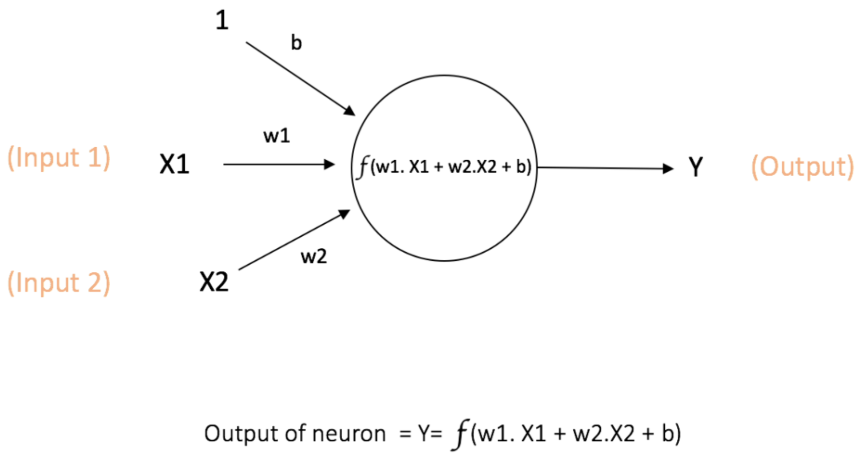


Figure 2.1: An illustration of a single neuron.

Depending on the problem and how the nodes are connected, a large number of layers of neurons may be needed between the input layer and output layer. This is called a deep neural network, as shown in Figure 2.2, because it has at least two hidden layers of neurons that work together to process a collection of data. Therefore, this type of learning is also called deep learning, a term coined back in 2006 [41].

### 2.2.3 Probabilistic Graphical Models

In the previous section, neural networks and deep learning were discussed. Nowadays these types of models often have to deal with lots of complex interconnected data when training a model for prediction. Therefore, large and complex neural network architectures may have to be created manually, which is also time-consuming. An alternative technique, that captures such complex data relationships more robustly are probabilistic graphical models (PGMs). PGMs aim to model relationships that exist between the data in a dataset in a graphical way, for example, relationships between different code metrics of a code entity on various "levels".

A PGM that can be used for this purpose is a Bayesian Belief Network (BBN). A BBN is a directed acyclic graph (DAG) describing which random variables (RVs) in a joint probability distribution interact with each other directly via a conditional dependency [38]. Let us define a graph as a tuple $G = (V, E)$ of sets, with $V$ being a set of nodes (or vertices) and the elements of E, with $E \subseteq V \times V$, being the edges in $G$. The graph $G$ is said to be a DAG, meaning that $G$ is directed and does not contain any directed cycles. Formally, if an edge $(X_1, X_2)$ exists in the graph connecting RVs $X_1$ and $X_2$, it means that $P(X_2|X_1)$ is a factor in the joint probability distribution, with $X_1$ serving as the "parent" of $X_2$. Because of this, we

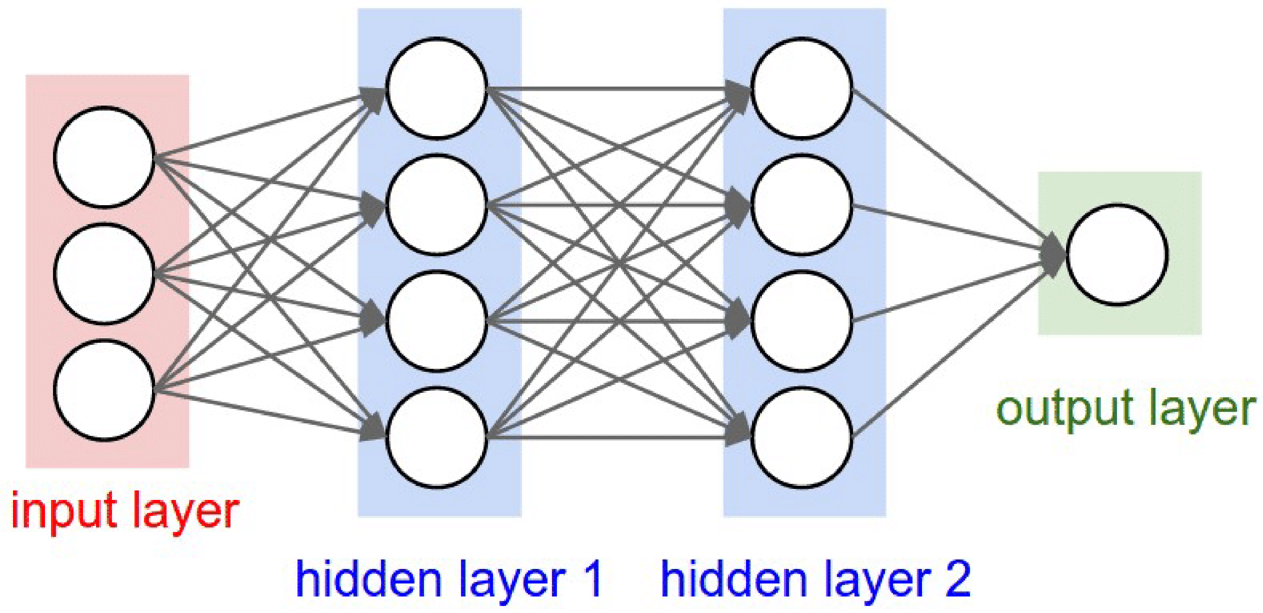


Figure 2.2: An example of a deep neural network.

must know $P(X_2|X_1)$ for all values of $X_1$ and $X_2$ to conduct an inference task. Each node $X_i$, for $i = 0, \ldots, N$, have associated probability values that quantify the effects of different parent nodes, i.e., $P(X_i|parents(X_i))$, using a finite number of parameters of the probability distribution [40]. These values gathered for each node are presented in Conditional Probability Tables (CPTs). The probability values themselves are called beliefs, hence the name Bayesian Belief Network. A well-known example given in Figure 2.3 [40], shows a connected network defining a daily lawn routine. This network is also called a multiply connected network, as multiple paths between two nodes exist [43]. Based on the probability that it is going to be cloudy or not, i.e., the CPT in node *Cloudy*, the sprinkler is turned on or off. This is given by the CPT in node *Sprinkler* and is based on the probabilities in node *Cloudy*. The probability of rain also depends on whether it is cloudy or not, which in turn is given by the CPT in node *Rain*. Based on both the probabilities in nodes *Sprinkler* and *Rain*, the grass will be wet given by the probabilities of CPT of node *WetGrass*. Thus two paths affect the event *WetGrass*. Focusing for example on the edge from node *Rain* to node *WetGrass*, means that $P(\mathit{WetGrass}|\mathit{Rain})$ will be a factor, whose probability values are specified next to the *WetGrass* node in a CPT. However, such tables can become quite big as they store one probability value for every combination of states, or influences.

Since CPTs can store multiple levels of influence, a BBN can become quite big, thus causing inference tasks to be intractable. This makes computing the exact probability intractable in #P in this case [33]. Thus the process of reasoning with these type of models become challenging. Therefore, in [39] sum-product networks were proposed and will be covered in the next section.

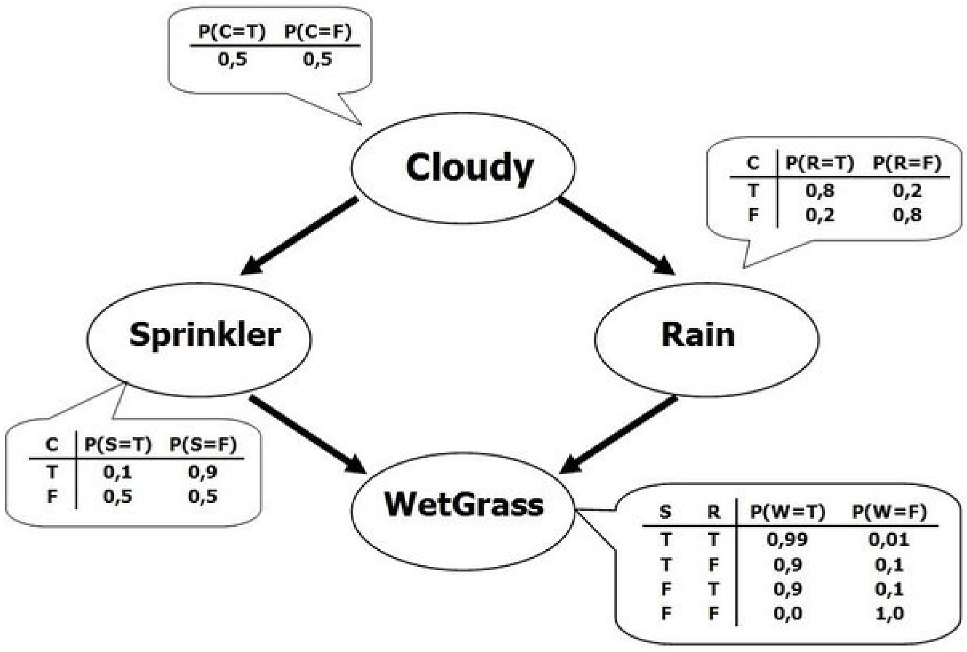


Figure 2.3: A simple example of a Bayesian Belief Network describing a daily lawn routine [40].

## 2.3 Sum-Product Networks

This section starts with a general overview of sum-product networks (SPNs). Following the basics of SPNs, the learning process of SPNs is covered and how it is related to NNs.

### 2.3.1 Basic Definitions Of Sum-Product Networks

SPNs are also a class of probabilistic graphical models that allow exact and efficient computation of a large number of inference tasks, e.g. marginalization [39][33]. Let us define an SPN as a connected and rooted DAG $S$, over random variables (RVs) $\mathbf{X}$. Let $V(S)$ be the set of all nodes in graph $S$, and let $\mathsf{ch}(n)$ denote the set of child nodes of node $n \in V(S)$. An inner node $n$ within graph $S$ is either a sum or product node and its scope is defined as $\mathsf{sc}(n) = \bigcup_{c \in \mathsf{ch}(n)} \mathsf{sc}(c)$ in recursion [46]. To be more precise, every node in an SPN graph is associated with a univariate distribution function[3], as shown in Figure 2.4[4]. This means that the RVs associated with the different leaf nodes are reachable from the internal nodes. A sum node $S_n$ computes a weighted sum over its child nodes as follows: $S_n = \sum_{j \in \mathsf{ch}(n)} w_{nj} S_j$, with $w_{nj} \geq 0$, and also $w_{nj}$ denoting the weight value between node $n$ and node $j$. Subsequently, a product node $S_n$ computes a product over its child nodes as follows: $S_n = \prod_{j \in \mathsf{ch}(j)} S_j$. Finally, a leaf node $n$ is usually associated with an arbitrary probability distribution [46]. The value of a leaf node $n$ is calculated by evaluating its respective probability distribution, i.e., $P(n|\theta)$. With $\theta$ denoting the parameters of the probability distribution of a leaf node $n$. This also makes the difference between PGMs and SPNs clear. Namely, every node in a PGM represents an RV and the different edges between the nodes represent probabilistic dependencies presented as a row in a CPT, while in an SPN every node represents a probability distribution [46].

[3]The term probability distribution is used most of the time in this thesis.
[4]https://blog.pollithy.com/spn/sum-product-network

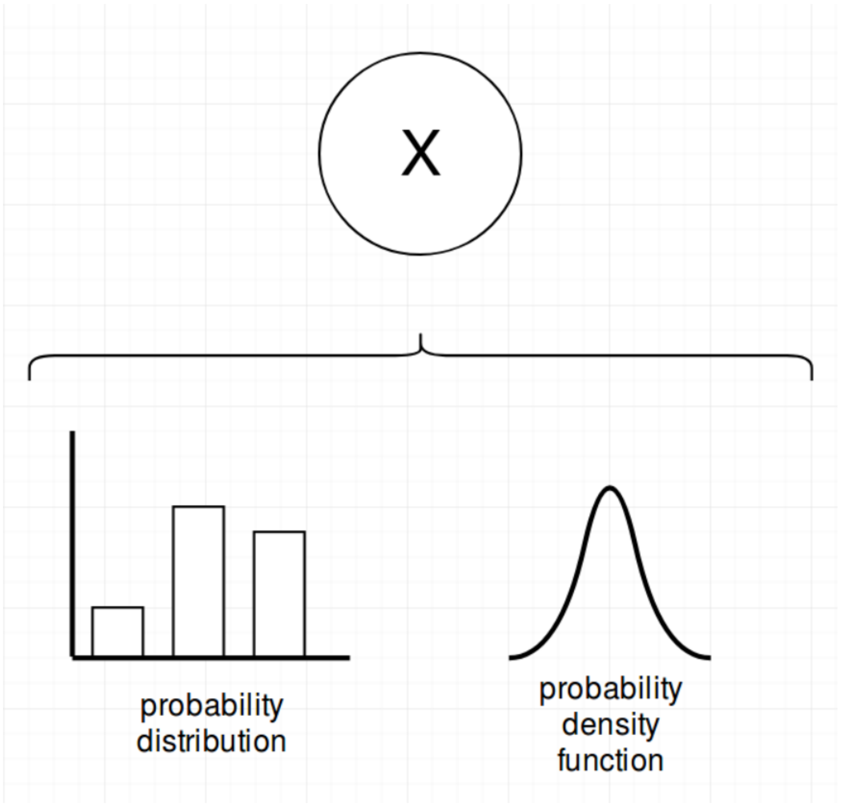


Figure 2.4: A univariate distribution of an SPN starting with one node.

To ensure that an SPN $S$ is a sound probability distribution that guarantees efficient inference, it is required that an SPN is complete, and consistent or decomposable [33]. Completeness and decomposability are essential to render many inference scenarios tractable in an SPN. An SPN is complete if all its sum nodes are complete, and a sum node is complete if all its child nodes have the same scope [33]. An SPN is consistent if its product nodes have disjoint scopes, i.e., the probability distributions of the child nodes that compute the product values are independent of each other [33]. Furthermore, all SPNs are assumed locally normalized, thus the sum of all weights is equal to 1, i.e., $\sum_{j \in \mathrm{ch}(n)} w_{nj} = 1$, with $nj$ denoting an edge from node $n$ to node $j$ [37]. In Figure 2.5, we show a simple illustration of an SPN graph visualizing the concepts discussed above.

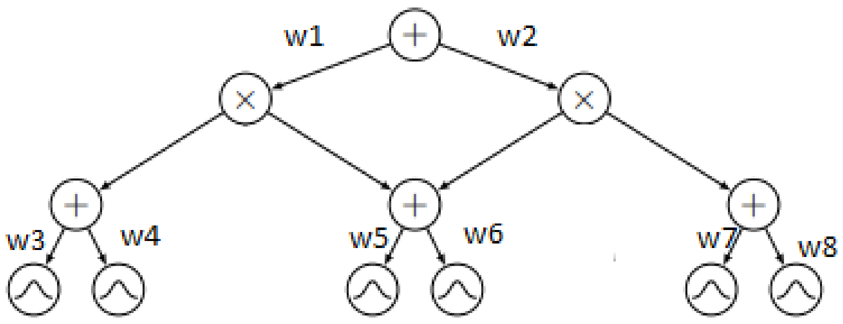


Figure 2.5: An illustration of a simple SPN graph containing sum, product and leaf nodes.

Also, an SPN $S$ outputs a value that is the normalized output of its root node depending on the structure of the DAG, the set of weights ($\mathbf{W}$), and the set of parameters ($\Theta$) of the distributions of the leaf nodes. Computing the probability of observations $\mathbf{X} = \mathbf{x}$, consists of a bottom-up evaluation from leaf nodes to the root node of graph $S$. First, each leaf node evaluates the probability function $P(\mathbf{X} = \mathbf{x})$, then each inner node evaluates the probability function $S_n(\mathbf{x})$ before cascading the output to its parent node, until the root node $S_0$. For the computation to be tractable, the following requirement has to be met. The size of the SPN graph has to be polynomial in the size of the total number of RVs [37]. Therefore, performing a marginalization inference only has to be performed at the leaf nodes, followed by a bottom-up evaluation of the graph.

### 2.3.2 Learning Process

The learning process of SPNs can be organized into parameter and structural learning. Firstly, parameter learning entails the process of finding the optimal parameters, i.e., weights and distribution parameters, for a fixed SPN network structure from training data [33]. To learn the parameters of an SPN model for generative learning, the likelihood of the respective parameters is maximized, given the training data. Alternatively, regarding discriminative learning, the conditional likelihood of the parameters for each output value of class variables in $\mathbf{Y}$ are maximized. A wide range of parameter learning techniques exist, including various maximum likelihood approaches using either gradient-based optimisation or expectation-maximisation [33][37]. Furthermore, several Bayesian techniques have also been developed over the years [33]. For example, for an SPN model regarding generative learning, the maximum likelihood estimation (MLE) can be employed to find the optimal parameters in the sets: $\mathbf{w} \in \mathbf{W}$ and $\theta \in \Theta$. The MLE method is applied on the training data $\mathbf{D} = \mathbf{X}^1, \mathbf{X}^2, \ldots, \mathbf{X}^N$ of $N$ features with $N$ different distributions. With $\mathbf{W}$ as the set of weights between the nodes (only the edges from sum nodes) of the SPN graph, and $\Theta$ denoting the set of parameter values of the distributions associated with the leaf nodes. This optimal configuration of both weights and parameters maximizes the logarithmic likelihood of $L_{\mathbf{D}(\mathbf{w},\theta)}$ [33]:

$$L_{\mathbf{D}(\mathbf{w},\theta)} = \log P(\mathbf{D}|\mathbf{w},\theta) = \sum_{i=1}^{N} \log S(x^i|\mathbf{w},\theta) \tag{2.1}$$

Secondly, with structural learning the situation is different. The issue with parameter learning is that the network structure is fixed, thus domain dependent [16]. This would mean that when applying an SPN model to a new problem, a new network structure has to be created manually. However, several algorithms have been devised that learn both the parameters and network structure from the training data [16]. After the training process of finding an optimal model, that same model can be applied to different problems in different domains [33][37]. However, the approaches of most algorithms are best described as heuristic, as mostly intuitive schemes are presented for structure learning [16]. A commonly used algorithm is LearnSPN [33], it creates an SPN recursively in a top-down fashion by splitting ("chopping") the features, and then clustering ("slicing") the instances of a dataset for training. The training data in this case is presented as a matrix. The chopping in this process creates a product node, and this can be visualized by partitioning the dataset along the columns or features. When the chopping outputs one column, i.e., one feature or variable, a leaf node is created. The leaf nodes have a univariate distribution that is found by employing the MLE procedure. The slicing creates a sum node and can be visualized by partitioning the dataset into similar clusters of instances over multiple rows and columns. This process is also done recursively using the expectation-maximization procedure to find the probability models for the different sum nodes [33]. This algorithm will also be the basis for the training of the SPN models used in this study when conducting the different experiments.

### 2.3.3 Sum-Product Networks As Neural Networks

The configuration of how nodes are connected within an SPN graph is similar to a multi-layered NN. Therefore, SPNs can be seen as a type of feedforward NNs because of the flow of signals (values) from the input nodes (leaves) to the output node (root), see Figure 2.5. The product nodes can be seen as activation nodes, similar to a neuron in a neural network. However, in [33] the term "neural network" is mainly reserved for models that are structured in layers and connected by the following operators: sigmoid, ReLU, softmax, etc. The main difference between these types of deep learning techniques is that SPNs have a probabilistic interpretation while multi-layered NNs do not. Therefore, every node in a multi-layered NN needs to have a threshold value assigned regarding inference, and with SPNs it is not necessary. Furthermore, multi-layered NNs have to be configured manually and it is necessary to examine different architectures of different sizes with different parameter settings (hyper-parameters) until an optimal model is found. This process of finding an optimal model can be time-consuming, and can also lead to a sub-optimal model [11]. On the other hand, with SPNs the underlying model for classification is learned from the training data directly, thus generative. This makes a classifier employing SPNs more robust as fewer parameter settings and configurations are needed to be set. Hence, using SPNs for code smell detection is a worthwhile endeavor as will be made clear in the next chapters.

## 2.4 Word Embeddings

To recover the semantic relationships embedded in method names, field names, and class names within source code is quite challenging. Lexical similarity is insufficient in measuring the semantical relationship between different methods within a class for example. That is because lexically dissimilar code entities can still be closely related semantically. To fully exploit the semantics between different code entities in source code, NNs are employed to extract useful characteristics from the texts that make up a code entity in source code [28]. One of the first techniques for this purpose was related to natural language processing (NLP) research and called *word2vec* [28]. The model computes the likelihood that certain word relationships occur. To predict the semantic meaning of a word, *word2vec* makes a suggestion based on the words used in previous appearances. These predictions can then be used to find associations between different words.

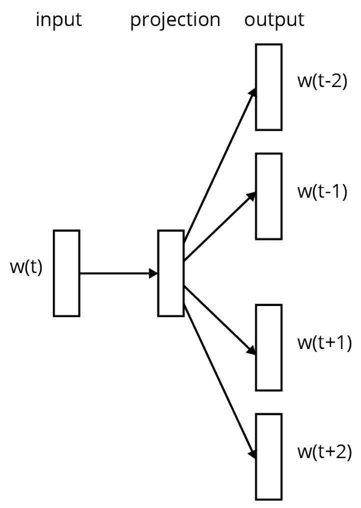


Figure 2.6: The Skip-gram model architecture employed by *word2vec*.

In Figure 2.6, it is shown that the input is a word $W_t$ that passes through the encoding layer in the center. The objective is to recover the nearest neighbor of $W_t$ across all instances of a word and its usages in the instances of the training data. This simple architecture, trained on billions of word instances, is sufficient to create deeply informative word vectors. This strategy of using encoder-decoder NNs to embed words is also a way to generate vector embeddings or word embeddings, for code entities in source code [28]. Identifiers described in a natural language are fed as input into the NNs. Then the words in the identifiers are converted into numerical vectors. This approach, shown in Figure 2.6, has been proven efficient for producing word embeddings to capture syntactic and semantic word relationships [28].

# 3 Related Work

In [10], a comprehensive literature review of 84 code smell detection tools was performed. Only four tools, available for download, were selected for a comparative study. The focus was on the following code smells: Large Class and Long Method [15]. The study relied on the following metrics for comparison: precision, recall, and F1 [11]. Whereas precision measures the accuracy of positive predictions, recall measures the ratio of positive code smell detections that are correctly detected, and F1 is the harmonic mean of precision and recall. These metrics will be covered in more detail in the next chapter. Furthermore, every code smell detection tool was configured with a threshold value to identify a code smell. They found that the detection tools can have different results regarding the detection of code smells. The researchers concluded that the differences were due to the fact that the tools use different detection methods. The list of code smells detected by the different tools, based on static analysis, does not mean that every item listed should have been identified as a code smell. Meaning, a developer had to manually review the list of potential code smells, and this process itself is prone to subjective interpretation. The different tools had to be configured with different metrics and thresholds, which in turn led to more uncertainty regarding the detection of code smells.

To solve the issue of uncertainty regarding code smell detection, researchers have proposed the use of various ML techniques. In [9], the experiments of [13] were replicated by using ML techniques to detect code smells. A large study was conducted where 32 different ML models were applied to detect the following four code smells: Data Class, Large Class, Feature Envy, and Long Method [15]. The researchers reported that most of the classifiers in the code smell detection tools exceeded 95% in terms of accuracy. However, when the datasets were extended by multiple code smells and non-code smells, the accuracy dropped significantly. Their conclusion was that this was mainly due to more uncertainty in the reporting of the various code smells. Therefore, the results were non-conclusive and more research is needed towards structuring datasets appropriately when training the models to detect the different code smells.

As discussed above, there is always some confusion when a potential code smell is detected. That is because the assessment of whether a code entity is a code smell or not is still a process that needs manual checking afterward as there is too much uncertainty.

Therefore, in [20] and [21], an approach is proposed to extend the DECOR (DEtection & CORrection) approach in order to decrease uncertainty in the detection of code smells. In DECOR, so-called rule cards were used to specify code smells by using a domain-specific language (DSL). Due to the limited expressiveness of the rule cards, BDTEX (Bayesian Detection Expert) was suggested. BDTEX is based on the GQM (Goal Question Metric) technique to extract relevant information from a code smell definition. This made it possible for Bayesian Belief Networks (BBNs) to be built systematically without relying on the rule cards. Code smells are then reported with a probability corresponding to the degree of certainty regarding the detection of the respective code smell. Furthermore, the Bayesian theory that underlies BBNs can also be used to improve the detection mechanism in such a way that it can "learn" from past experiences regarding the detection of code smells. This is an important step to reduce uncertainty, hence more research is needed.

An alternative to capturing uncertainty in the process of code smell detection is to perceive software code as a form of communication similar to human language. In [1], the aforementioned idea is proposed and called the Naturalness hypothesis. This means that source code "text" has the same statistical properties as natural languages. Based on this, researchers have built probabilistic models to learn intermediate encodings (e.g., word embeddings) of the source code. These models predict the probability distribution of code entities, called representational code models in this study, meaning that these models can "learn" from the source code. These properties have the potential that is can be used with static analysis tools.

An approach akin to code smell detection, using deep learning, is proposed in [25]. In this research, a deep learning technique is proposed to automate the detection of security vulnerabilities in source code. This system is called Vulnerability Deep Pecker (VulDeePecker), and at its basis is a deep learning model to detect security vulnerabilities. To detect the vulnerabilities, code gadgets were introduced. Code gadgets represent a number of lines of code that are semantically related to each other. These lines are then transformed into numerical vectors that are fed into the underlying model of VulDeePecker. VulDeePecker was applied to three different software projects: Xen[1], Seamonkey[2], and Libav[3]. The experiments showed promising results, as VulDeePecker is more effective in detecting code vulnerabilities compared to static analysis tools such as VUDDY[4], and VulPecker[5].

Another deep learning approach focused on code smell detection is proposed in [26]. The code smells that were the focus of this research are the following: Long Method, Feature Envy, Large Class, and Misplaced Class [15]. To identify the different code smells, different complex neural network architectures were created manually. After this step, the different NN models were trained to detect the different code smells. However, to detect the different code smells different features were needed for each code smell. To identify the code smell Long Method only code metrics were used to train the models. To identify the code smells Feature Envy and Large Class, code metrics and textual features were used. The textual features (e.g., class names, field names, method names, etc.) were extracted from source

[1]https://github.com/xen-project
[2]https://www.seamonkey-project.org
[3]https://libav.org
[4]https://github.com/squizz617/vuddy
[5]https://github.com/vulpecker/Vulpecker

code, then converted to numerical vectors by employing *word2vec* [28]. These numerical features, or code metrics and textual features, are fed into the different models that classify a code entity as "smelly" or "not smelly". Whether a code entity is a code smell or not depends on the thresholds set for prediction. The results were compared with the results of the study in [44]. It was shown that the deep learning approach in detecting code smells outperformed the state-of-the-art detection techniques. As the results are only based on the four aforementioned code smells the deep learning approach has lots of potential.

Another important issue to consider regarding code smell detection is employing these ML-based approaches during the software development process. There are a number of tools that support automatic code smell detection during the development process. DECOR was already discussed above. The following tools are static analysis tools. The first tool is PMD[6] and detects the following two code smells in Java: Large Class and Long Method. To detect the code smell Large class, detection strategies are employed as defined in [23]. On the other hand, to detect the code smell Long Method only one metric is used: LOC (Lines Of Code). Secondly, JDeodorant[7] is a tool that detects the following code smells: Long Method, Feature Envy, Large Class [44]. The detection strategies are based on different code metrics to identify the different code smells [44]. Finally, JSpIRIT[8] detects and prioritizes ten code smells, including the three smells that are the focus of this study: Long Method, Feature Envy, and Large Class [47]. The detection strategies of these code smells are based on the strategies as defined in [23].

The proposed approach (outlined in the following chapters) in this study employs an alternative deep learning technique to detect code smells, i.e., sum-product networks (SPNs). As of this writing, there is no research and tooling available regarding the use of SPNs to detect code smells during the software development process. This is confirmed by searching through different digital libraries and resources given in Table 3.1. The search involved 4 digital libraries, aiming to identify relevant studies regarding SPNs and code smell detection. Additionally, Mendeley[9] was also used in searching and organizing the papers related to SPNs and tooling. Furthermore, the snowballing technique was applied by checking the references of each selected study, to avoid missing any relevant research [48].

Table 3.1: Digital libraries with criteria regarding date used for search.

| **Database** | **URL** | **Date Criteria** |
|---|---|---|
| ACM Digital Library | http://dl.acm.org/ | 2000-01-01 to 2021-01-01 |
| IEEE Xplore | http://ieeexplore.ieee.org/Xplore/ | 2000-01-01 to 2021-01-01 |
| Science Direct | http://www.sciencedirect.com/ | 2000-01-01 to 2021-01-01 |
| Google Scholar | https://scholar.google.com/ | 2000-01-01 to 2021-01-01 |
| Springer | http://link.springer.com/ | 2000-01-01 to 2021-01-01 |

[6]https://pmd.github.io/
[7]https://github.com/tsantalis/JDeodorant
[8]https://github.com/hcvazquez/JSpIRIT
[9]http://www.mendeley.com/

# 4 Research Design

In this chapter, the overall approach of the research design for this study will be covered. Section 4.1 covers the problems that are associated with code smell detection. In Section 4.2, the research questions relevant to this study are discussed. Following the research questions, an outline of the general approach to answering the research questions is discussed in Section 4.3.

## 4.1 Problem statement

As described in previous chapters, detecting code smells manually or aided by state-of-the-art static analysis tools is a complex and time-consuming process. In either case, the process of detecting code smells is prone to subjective interpretation. That is especially the case when examining source code manually by a developer, as part of a review process for example. When using tools to detect code smells automatically, a set of code metrics combined with their respective thresholds are set to determine when a code entity under investigation is a code smell or not. Tuning these thresholds is usually done for specific projects and is often complex and also time-consuming. Furthermore, these tools also have different techniques to detect code smells automatically. These static analyzers used in many integrated development environments (IDEs) produce a long list of potential code smells. A developer usually needs to evaluate this long list of potential code smells, as most of them are not code smells. Therefore, a lot of uncertainty exists regarding code smell detection when static analyzers are used. Due to the subjective interpretation by different developers and their level of knowledge, there is also a possibility that more code smells are introduced.

To overcome these issues regarding uncertainty, research into the use of various ML techniques to detect code smells has been conducted [9]. The goal of using ML techniques was to make code metrics and their respective thresholds, which needed to be set manually, redundant. Though using classification algorithms, or classifiers, employed by ML systems is a promising way of detecting code smells, there still is a lot of room for improvement [14]. That is because there is still a lot of uncertainty when potential code smells are reported. However, in [26] research has shown that code smell detection using a deep learning approach performs at least as well as most state-of-the-art detection techniques employing static analysis. The issue with a classifier employing deep learning, such as multi-layered NN architectures, is that it can take a long time to create. That is because it is necessary to examine different architectures of varying size and parameter (hyper-parameter) values, such as the different thresholds that need to be set in the different NN nodes. This process of finding a satisfying model can also be time-consuming, thus leading to a sub-optimal model [11].

However, the major issue (problem) with most (ML) techniques is that these types of models have no inherent way to deal with "vagueness", i.e., whether a code entity might be labeled as a code smell up to a certain degree. That is because the ML techniques mentioned above create discriminative models. Hence, these models lack the possibility to deal with uncertainty in the form of "vagueness". This leads us to focus on probabilistic models that are able to deal with this kind of uncertainty, and thus "vagueness". By using SPNs we want to deal with this kind of uncertainty by creating the appropriate models from "the data", i.e., the training data. This is done by creating probabilistic classifiers, estimating the probability distributions over the different feature values, such as code metrics, which can then be used to classify code entities into classes of code smell or non-code smell [39]. Thus making the models used for code smell detection more robust.

To assess the feasibility of SPN models detecting code smells, three experiments will be replicated from the study in [26]. In [26], a deep learning approach is used by employing multi-layered NNs to detect code smells. To evaluate the performance of the approach using SPNs, the results of the experiments are compared to that of the DL approach in [26]. The code smells selected for the different experiments are repeated in Table 4.1, as they were discussed in more detail in Chapter 2. Furthermore, it is also important to assess the practical applicability of SPNs by means of a case study regarding code entities that are not part of training or test data. Also, as of this moment of writing, there is no tooling available to detect code smells by employing SPNs. Therefore, the focus of this thesis will be twofold, the feasibility of employing SPNs models detecting code smells, and the assessment of the practical applicability of SPNs.

Table 4.1: The code smells within the scope of our study

| **Code Smell** | **Definition** |
|---|---|
| Long Method | A method that is too long or bloated in size, thus doing too much. |
| Feature Envy | A method more interested in features of other classes then the ones of the class the method it is currently located. |
| Large Class | A class that is too large or bloated in size, thus having too much responsibility. |

## 4.2 Research Questions

The main objective of this study is to investigate the feasibility of employing SPNs to detect code smells with high confidence and asses its practical applicability. To accomplish this, an approach was devised that will be discussed in the next section.

Therefore, the following main research question is defined:

**RQ** To what extent do SPNs improve the detection of code smells compared to state-of-the-art detection techniques, and can this be applied in practice?

This main question will be investigated by answering the following sub-questions:

**RQ-1** To what extent do SPNs improve the detection of code smells when compared to state-of-the-art detection techniques?

**RQ-2** To what extent does the performance of code smell detection improve when a word embedding technique, *word2vec*, is used?

**RQ-3** Do some individual features carry more weight when detecting code smells?

**RQ-4** To what extent are SPNs applied in a development environment?

The main objective of **RQ-1** is to investigate the performance (e.g., precision and recall) of the proposed approach using SPNs to detect code smells: Long Method, Feature Envy, and Large Class. To answer this research question, three different SPN models were created to detect the different code smells. The different models were used to replicate the experiments in [26] using the same (publicly available) datasets. The main reason to replicate the experiments is to compare the results of both approaches. Furthermore, the DL approach is a major improvement when compared to state-of-the-art techniques such as JDeoderant [12] and DECOR [29].

The main objective of **RQ-2** is to improve the prediction of code smells with higher confidence by introducing word embeddings. Therefore, the word embedding technique *word2vec* was incorporated within the different SPN models. By including *word2vec* to detect code smells, the aim is to increase accuracy by exploiting the semantic relationships of a code entity under investigation when detecting a code smell. The reason to include *word2vec* embeddings as classification features to train the different SPN models for the different code smells, is to investigate if an improvement of performance occurs when compared with the results of **RQ1**, and the results of the DL approach in [26].

The main objective of **RQ-3** is to investigate to what extent individual code metrics influence the detection of code smells. That is because SPNs are able to allow exact and efficient computation of a probability value to detect a code smell based on relevant code metrics. With this marginal inference task, the code metric(s) that influence code smell detection the most can be validated by means of a case analysis. Hence, this research question also serves the goal to assess the practical applicability of SPNs. That is because the data instances used in this case study are not part of the training or test data. For this research question, the focus was only on code metrics and not on word embeddings. That was due to resource constraints with regard to memory and computational power.

The main objective of **RQ-4** is to investigate to what extent tooling exists regarding code smell detection using SPNs. This research question also answers if the tool should be integrated into an IDE to assist developers or whether it should run as a stand-alone application.

The results of the aforementioned research questions will be covered in the next two chapters. Thus, the results of research questions **RQ-1** and **RQ-2** will be covered in Chapter 6. In Chapter 7 the results of research questions **RQ-3** and **RQ-4** will be covered.

## 4.3 Code Smell Detection Approach

To answer the research questions of the previous section, an approach was devised to detect code smells using SPNs. To detect the code smells defined in Table 4.1, the following steps need to be performed: extracting and analyzing the dataset, data preparation, training of SPN models, validation of the SPN models, and finally evaluating the trained SPN models. The steps are shown in Figure 4.1 and are covered in more detail below.

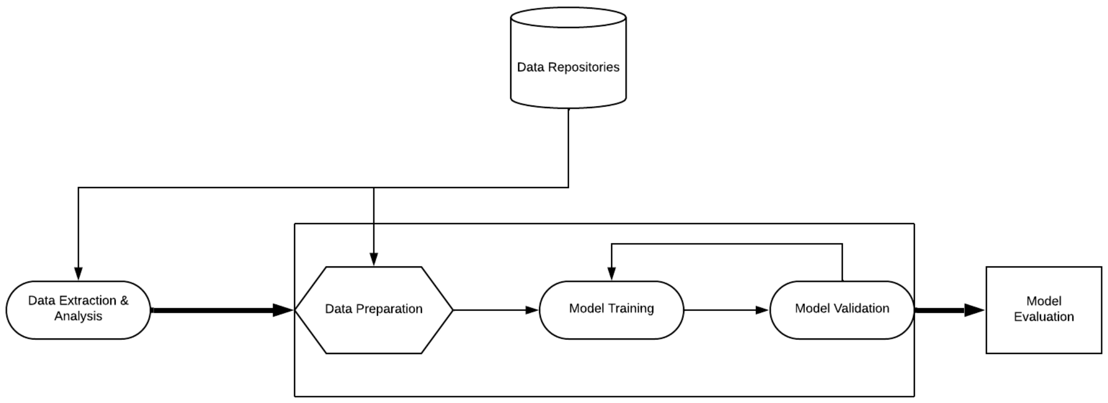


Figure 4.1: Overview of the general SPN approach.

### Data Extraction and Analysis

The datasets were extracted from the same data repository[1] used in [26]. To be able to use the datasets to create the SPN models, code metrics and word embeddings have to be extracted from the source code. Based on the code smells, the same classification features as in [26] were selected, i.e., code metrics and *word2vec* vectors. Furthermore, analysis of the different code smells generated is also part of this step.

### Data Preparation

The aim of this step is to create training, validation, and test datasets. This is done for the different SPN models that detect their respective code smells: Long Method, Feature Envy, and Large Class. The data will be gathered and processed in such a way that it will be presented in the format *csv*[2], as it is supported by libraries in Python, Java, C#, etc. Furthermore, every instance in the training, validation, and test set is extended with the source file location, the code entity under investigation, begin and end line number of the code entity itself. This is done for the purpose of further analysis if necessary.

### Model Training

The aim of this step is to create and then train the different SPN models to detect the code smells: Long Method, Feature Envy, and Large Class. In this step, the following two parameters are estimated: the weights that connect the two nodes within an SPN graph, and the parameters of the probability distributions at the leaf nodes. Fine-tuning the SPN models is also part of this overall step. Because the different models will be validated after a training session. This process is repeated until an optimal SPN model is created.

[1]https://github.com/liuhuigmail/DeepSmellDetection
[2]https://docs.fileformat.com/spreadsheet/csv/

## Evaluation

A necessary step after creating the different SPN models to detect the different code smells is to find out how effective the different models are. Different performance metrics are available to evaluate the models. As the objective of the different SPN models is to classify whether a code entity is a code smell or not, the focus is on performance metrics used that evaluates the accuracy of models with regard to binary classification problems.

A metric widely used to evaluate ML models and compare them to each other is the Area Under Curve or AUC [11]. This metric computes the area underneath the entire receiver operating characteristic (ROC) curve, hence it is also called ROC AUC. The ROC curve is a graph showing the performance of a classification model at all classification thresholds. But for probabilistic classifiers, it gives a probability or score, reflecting the degree to which class an instance (or sample) belongs. Although the ROC Curve is a helpful diagnostic tool, it can be challenging to compare two or more classifiers based on their curves. Therefore, the ROC AUC provides the ability for a classifier to distinguish between classes and is used as a summary of the ROC curve. The higher the ROC AUC score, the better the performance of the model is at distinguishing between positive and negative classes. For imbalanced classification with a severe skew and few samples of the "minority" class, the ROC AUC can be misleading [5]. That is because a small number of correct or incorrect classifications can result in a large change in the ROC AUC score. The reason is that the datasets used for training/testing are skewed, as non-code smells are more prevalent than code smells, hence code smells are called the minority class. Therefore, when using ROC AUC a high rate can still be achieved by simply guessing.

Thus, using the performance metric ROC AUC alone is not enough to fully evaluate the effectiveness of the SPN models [5]. Therefore, the following metrics are also included[11]:

$$Precision = \frac{\#\, of\ correct\ code\ smells}{\#\, of\ code\ smells} = \frac{TP}{TP + FP} \tag{4.1}$$

$$Recall = \frac{\#\, of\ correct\ code\ smells}{\#\, of\ true\ code\ samples} = \frac{TP}{TP + FN} \tag{4.2}$$

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4.3}$$

Where TP = True Positives, TN = True Negatives, FP = False Positives, and FN = False Negatives. A true positive is an output where the SPN model in this case correctly predicts that the code entity is a code smell. Similarly, a true negative is an output where the SPN model correctly predicts that the code entity is not a code smell. A false positive is an output where the SPN model incorrectly predicts that the code entity is a code smell, and similarly a false negative is where the SPN model incorrectly predicts that the code entity is not a code smell. With the performance metric precision, an attempt is made to answer the question of what proportion of predicted code smells are actual code smells. Because it is possible to get a 100% precision with a small test set, it is therefore important to also use recall as part of the evaluation of the SPN models. Therefore, with the performance metric recall, an attempt is made to answer the question of what proportion of predicted code smells were predicted correctly. Unfortunately, precision and recall are often in tension. That is, improving precision typically reduces recall and vice versa. Another performance metric that has been developed that relies on both precision and recall, is the F1 score. The F1 score is calculated from the precision and recall as it is the harmonic mean between the two ratios. This performance metric will also be used as a measure to evaluate the accuracy of the different SPN models. Furthermore, these performance metrics listed above were also used to evaluate the DL approach in [26].

# 5

# CODE SMELL DETECTION USING SPNS

In this chapter, the different aspects of our approach to detecting code smells using SPNs are covered. In Section 5.1, the datasets that will be used to train the different SPNs to detect the code smells: Long Method, Feature Envy, and Large Class, are discussed. In Section 5.2, the steps to generate the code smells in the datasets, and the format in which it will be used to train, validate, and test the models are discussed. In Section 5.3, the features that will be used to classify whether a code entity is code smell or not are discussed. In Section 5.4, the process to create the different SPN models is discussed. Finally, in Section 5.5 the different datasets used for the different experiments are analyzed.

## 5.1 DATA SELECTION

To answer the first two research questions, the experiments covered in [26] were replicated. Therefore, datasets were generated from the same corpus of open-source projects, and are listed in Table 5.1. These selected open-source projects are all written in Java. The table shows the following information: project name, version number, NOC (Number of Classes), NOM (Number of Methods), and LOC (Lines of Code) respectively. The datasets were generated by a refactoring process, discussed in the next section, which introduces code smells to reduce the quality of the software. The datasets are publicly available and provided by the authors of [26]. We opted for this strategy of procuring the datasets for the following two reasons. Firstly, by using the same datasets it will be much easier to compare the results. Secondly, it is a cumbersome undertaking to initiate the task of generating a large dataset to train, validate, and test the different SPN models for code smell detection. Furthermore, with regard to the entire dataset, the assumption is that the selected open-source projects contain high-quality code.

Table 5.1: The open-source projects used for our study.

| **Project** | **Version** | **NOC** | **NOM** | **LOC** |
|---|---|---|---|---|
| Areca | 7.4.7 | 473 | 5055 | 88126 |
| Freeplane | 1.3.12 | 787 | 6938 | 124937 |
| jEdit | 4.5.0 | 513 | 5964 | 185571 |
| JUnit | 4.10 | 123 | 866 | 11734 |
| PMD | 5.2.0 | 250 | 2.097 | 23783 |
| Weka | 3.9.0 | 1348 | 20182 | 444493 |
| AbdExtractor (Android) | 20140630 | 1695 | 12608 | 304458 |
| Grinder | 3.6 | 502 | 3037 | 101293 |
| Art of Illustration (AoI30) | 3.0 | 492 | 6188 | 152207 |
| JExcelAPI | 2.6.12 | 424 | 3118 | 90555 |

## 5.2 Code Smell Generation

The datasets for training, validation, and testing were generated by using the same process as specified in [26]. To generate a data sample (or instance) having a code smell or not, the following steps were followed:

**Step 1**, a validation process is performed on whether a code entity is a code smell or not. This means the following. In the case of the Long Method code smell, a check is made whether a number of lines of code could be replaced within a method under investigation with another method executing those same replaced lines of code. In the case of a Feature Envy, a test is performed on whether a method under investigation could be moved to another potential target class. In the case of a Large Class code smell, an extraction from a potentially large class is attempted. With the Eclipse[1] refactor functionality a test is made if the different code smells could be created, and with the requirement that the different applications still work after these refactor procedures.

**Step 2**, a random selection is made to create a sample with a code smell (positive sample) or a sample without a code smell (negative sample). The positive or negative samples are generated randomly with a 50% chance.

**Step 3**, a negative sample is generated by keeping the code entity under investigation unmodified, thus not performing any refactoring to introduce one of the code smells given in Table 4.1. The features, which will be used to train a specific SPN model to detect a code smell, are extracted to identify this code entity as a non-code smell and appended with an $output$ of value 0. This sample is then added to the dataset in the following format:

$$\begin{aligned} \textit{negative sample} &= (\textit{input, output}) \\ \textit{input} &= (\textit{metrics, embedding}) \\ \textit{output} &= 0 \end{aligned}$$

**Step 4**, to generate a positive sample, a refactoring from step 2 is randomly selected and applied. Then the features are extracted to classify the code entity as a code smell and appended with $output$ having value 1. This sample is then added to the dataset in the

[1] https://www.eclipse.org/

following format:

$$\begin{aligned} \textit{positive sample} &= \textit{(input, output)} \\ \textit{input} &= \textit{(metrics, embedding)} \\ \textit{output} &= \textit{1} \end{aligned}$$

As given in steps 3 and 4, the input to an SPN model consists of code metrics and word embeddings using *word2vec*, which will be discussed in more detail in the next section. The word embeddings (vectors) for every method or class in the code smells Feature Envy and Large Class respectively, are transposed and added to the list of code metrics. Combined with the output it will serve as a training sample (or training instance). However, the samples regarding the code smell Long Method only need code metrics to train its SPN model. In [26], a similar choice is made. Furthermore, the samples will also be extended with the code entity under investigation and other metadata, regarding source file location, begin and end of line of the respective code entity. However, the metadata will not be used during the learning process.

## 5.3 Feature Selection

The classification features consist of both structural information (code metrics) and textual information (semantic relationships) regarding the code entities under investigation. The main reason for using the same features as in [26], is to compare the accuracy between the DL approach and the proposed approach using SPN models. The structural information refers to code metrics extracted from the source code. The code metrics to detect the different code smells will be covered below in more detail. The textual information refers to the semantic information described by identifiers(i.e., field names, method names, and class names) within a class or method under investigation. The names of identifiers are converted into numerical vectors by *word2vec* as introduced in the previous chapter. A given identifier is partitioned into a sequence of words according to capital letters and underscores, and each word is converted into a fixed-length numerical vector with a dimension of 200, as the corpus used for this study is relatively small[2]. This value for the dimension is also used in [26]. Furthermore, each identifier should contain no more than 5 words, as this covers more than 98% of all identifiers of the open-source projects in Table 5.1. An identifier with fewer than 5 words will be padded with zeros. The word embeddings consisting of fixed-length numerical vectors are processed as features by the SPN models:

$$id(e) = (w_1, w_2, \ldots, w_5) = (V(w_1), V(w_2), \ldots, V(w_5))$$

where $id(e)$ is the identifier of a code entity $e$, and $(w_1, w_2, \ldots, w_5)$ is a sequence of five words. Finally, $V(w_i)$ converts $w_i$ into a vector (dimension of 200) with *word2vec*. Thus, the number of features will increase by a 1000 per $id(e)$ when using word embeddings.

### Code Smell: Long Method

The features that are extracted to detect the code smell Long Method consists of only structural information, namely the code metrics from the method under investigation. The

[2] https://moj-analytical-services.github.io/NLP-guidance/

same code metrics were used in [26], as it is shown that these code metrics have been promising in detecting the code smell Long Method. Therefore, the following code metrics will be used at method-level: Size or Lines Of Code (LOC), Lack of Cohesion Methods (LCOM1, LCOM2, and LCOM4), Cohesion (COH), Class Cohesion (CC) [6]. Besides these code metrics, the DL approach (proposed in [26]) also used the following code metrics: Number Of Accessed Variables (NOAV), McCabe's Cyclomatic Number (MCN), and Coupling Dispersion (CD). NOAV and MCN measure the complexity of the method whereas CD measures how much the method is "coupled" with external classes. In Appendix A, these code metrics are covered in more detail.

As a result, for a given method, $m$, to detect the code smell Long Method the following classification features are defined:

$$\begin{aligned}&\textit{feature(m)} = \textit{(metrics(m))}\\&\textit{metrics(m)} = \textit{(Size, LCOM1, LCOM2, LCOM4, COH, CC, NOAV, MCN, CD)}\end{aligned}$$

## Code Smell: Feature Envy

To detect code smell Feature Envy, the model should determine if a method should be moved from its enclosing class to a target class. This SPN model exploits both structural and textual information. Thus the features that are extracted to detect the code smell Feature Envy consists of two parts: extracting code metrics from the class and method under investigation and extracting textual information using *word2vec*. Regarding structural information, the distance metrics proposed in [26] are used. These metrics have been proven promising in detecting the code smell Feature Envy [26], and are defined in more detail in Appendix A. For textual information, the method under investigation, the enclosing class, and the potential target class are processed with *word2vec*.

This resulted in the following specification of the features:

$$\begin{aligned}\textit{feature(m)} &= \textit{(embedding(m), metrics(m))}\\\textit{embedding(m,ec,tc)} &= \textit{(id(m), id(ec), id(tc))}\\\textit{metrics(m)} &= \textit{(dist(m,ec), dist(m,tc))}\end{aligned}$$

where $id(m)$ is the name of the method $m$ under investigation, $id(ec)$ is the name of the enclosing class of method $m$, and $id(tc)$ is the name of the potential target class. The value computed by $dist(m, ec)$ is the distance between method $m$ and its enclosing class $ec$, and the $dist(m, tc)$ computes the distance between method $m$ and the potential target class $tc$. Every identifier has a max length of 5 words.

## Code Smell: Large Class

The features that are extracted to detect the code smell Large Class are also twofold: textual information using *word2vec* and code metrics from the class under investigation. The textual information refers to the identifiers (i.e., field names and method names) declared within the class under investigation. The names of fields and methods are converted into word embeddings using *word2vec*. With regards to structural information, the 10 code metrics from [26] are reused: Access To Foreign Data (ATFD), Direct Class Coupling (DCC), Depth of Inheritance Tree (DIT), Lack of Cohesion in Methods (LCOM), Weighted Method

Count (WMC), Size or Lines Of Code (LOC), Number Of Public Attributes (NOPA), Number Of Accessor Methods (NOAM), Number Of Attributes (NOA), and Number Of Methods (NOM). These code metrics are defined in more detail in Appendix A.

This resulted in the following specification of the features:

$$\begin{aligned} \textit{feature(c)} &= \textit{(embedding(c), metrics(c))} \\ \textit{embedding(c)} &= \textit{fields(c), methods(c)} \\ \textit{metrics(c)} &= \textit{(ATFD, DCC, DIT, LCOM, WMC, Size, NOPA, NOAM, NOA, NOM)} \end{aligned}$$

where $embedding(c)$ is the semantic information of the source code of the class, $c$, that is under investigation. This entails the field names and method names of class $c$, defined by $fields(c)$, and $methods(c)$. All identifiers have a max length of 5 words. In $metrics(c)$ the code metrics of class $c$ are specified.

## 5.4 Creating SPN Models

Code smell detection can be defined as a classification problem. For this study, SPNs were used to model different classifiers to detect the different code smells: Long Method, Feature Envy, and Large Class. To create the different code smell detection models using SPNs, the library SPFlow [3] was used [30]. SPFlow is a library written in Python to develop comprehensive, simple, and extensible libraries for SPNs. It implements methods for inference (marginal, conditional probabilities, etc.), parameter learning with gradient descent, and several structural learning algorithms. An implementation of the learning algorithm Learn-SPN was used to train the different SPN models [33]. Furthermore, the SPFlow library can also be extended and customized to implement new algorithms. SPNs created with this library are also able to employ TensorFlow[4] as part of their back-end process. This library can be used for a wide range of applications regarding ML. However, for this study, TensorFlow was mainly used to speed up computational-intensive operations.

There are two options when creating an SPN model. Firstly, an SPN graph can be created manually using SPFlow, and then it can be compiled into an SPN model. This option is a tedious process and can take a long time to reach an optimal SPN model. Secondly, SPFlow can also be used to create an SPN model based on the training data. The only thing the two options got in common is the number of leaf nodes needed to detect the different code smells, as they are the same for both options. In the first option, a (probability) model has to be configured for every leaf node manually. However, the second option lets SPFlow model the distribution for every leaf node based on the training data, making it more flexible than the first option. Furthermore, this option is more realistic as the model created adopts different distributions of the different features from the training data. Therefore, the second option was preferred when creating the different SPN models.

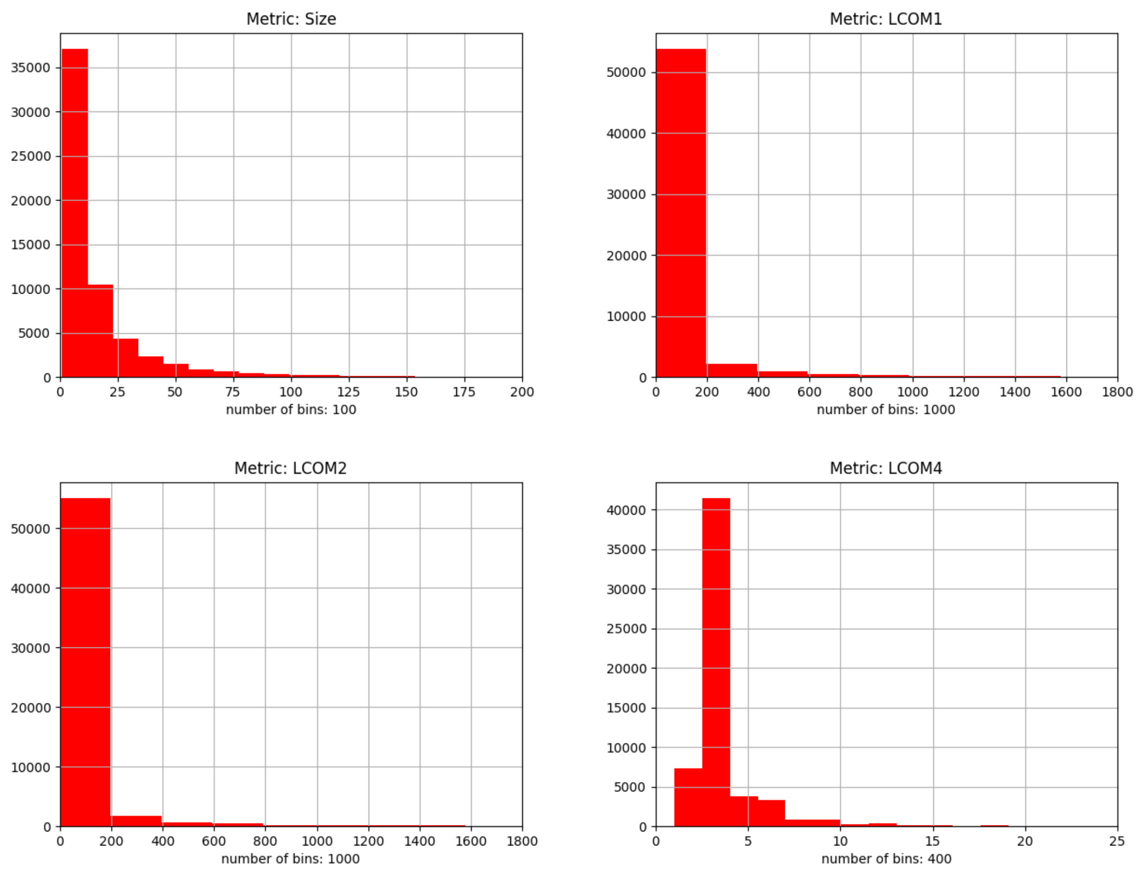
Finally, when creating the different SPN models, different hyper-parameters have to be set. To validate if an SPN model for a specific code smell is optimal, a validation set has to be created. Thus to perform this validation process, the datasets from Table 5.1 were split up in a training set, a validation set, and a test set. In the next chapter, this process is discussed in

[3] `https://githumb.com/SPFlow/SPFlow`
[4] `https://www.tensorflow.org/`

more detail. This type of validation of the SPN models by extracting a validation set, hold-out validation method, from the dataset is much less time-consuming than other (more cumbersome) available validation methods [11].

## 5.5 Analyzing Data sets

Before carrying out the different experiments, the distributions of the different code metric values regarding the code smells: Long Method, Feature Envy, and Large Class were analyzed. In Appendix B, a complete overview is shown of the distributions of the code metric values grouped by the different aforementioned code smells. Most code metrics in the training data are skewed to their lower minimum for both the code smells Long Method and Large Class. This is shown in Figure 5.1 regarding code smell Long Method. Regarding the code metric CC there is a skewness to both the minimum and maximum values, as can be seen in Figure 5.1 also. This code metric measures the number of client calls, in different classes, of the method under investigation. The reason that the distribution of this code metric is also skewed to the maximum value, i.e. 1, is because when computing this metric a threshold was set [26].



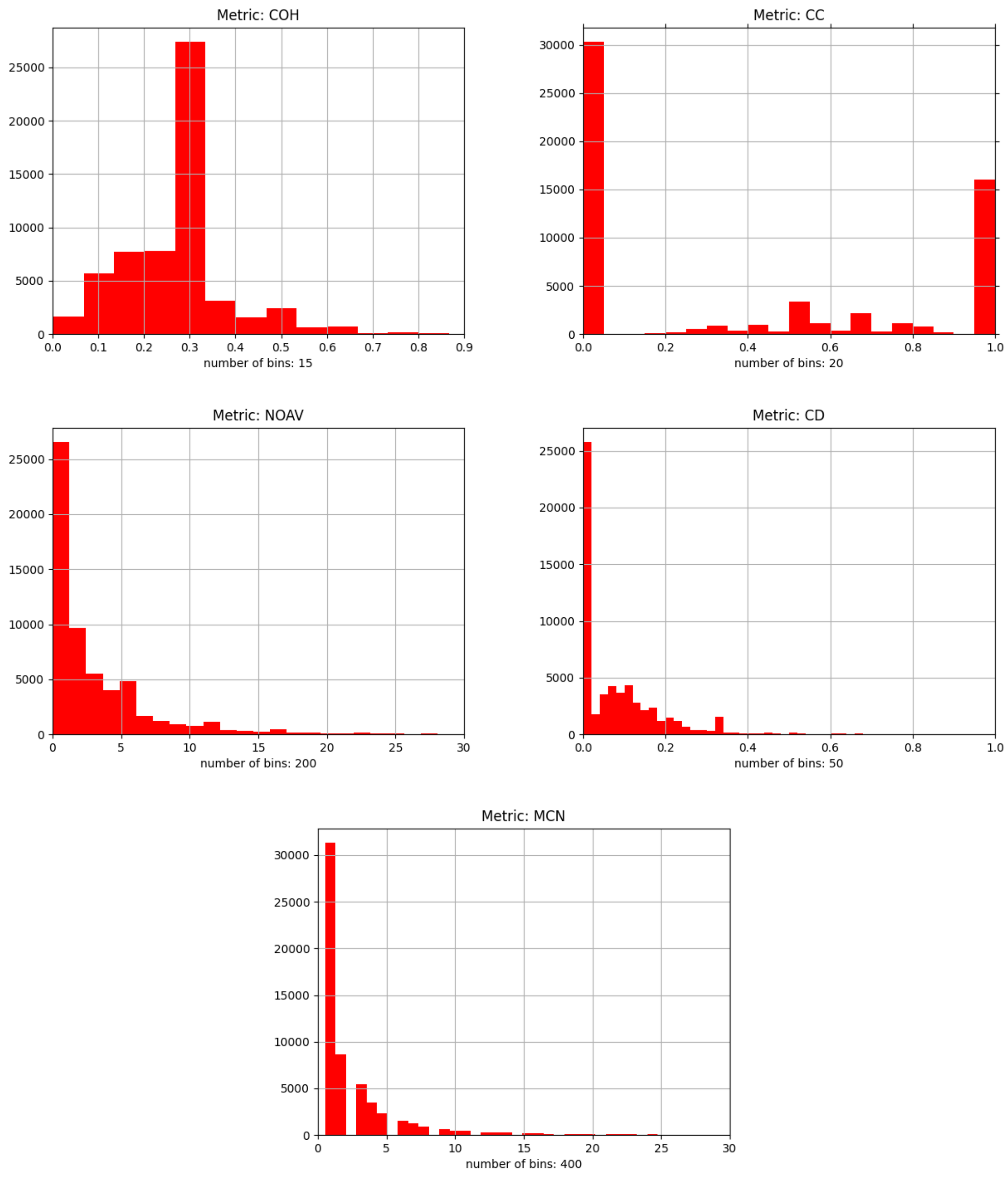


Figure 5.1: Long Method: Histograms showing the distributions of the various method level code metrics.

On the other hand regarding the distance metrics relevant to code smell Feature Envy, a skewness is also present at the maximum value, namely 1. This is also shown in Figure 5.2:

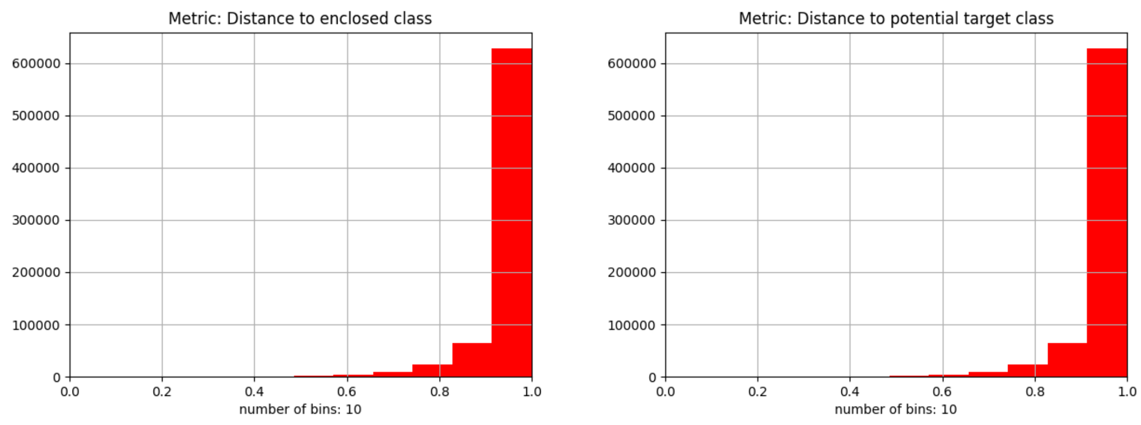


Figure 5.2: Feature Envy: Histograms showing the distributions of the two distance metrics.

# 6
# Experimental Results

In this chapter, the aim is to answer the research questions, mentioned in Chapter 4, regarding the following. To what extent do SPNs outperform state-of-the-art techniques (**RQ-1**)? Does including word embedding in SPNs improve the performance of code smell detection (**RQ-2**)? The results of these different experiments will be compared to the DL approach presented in [26].

Before evaluating the results, the setup of the different experiments is outlined in Section 6.1. In Section 6.2, the results of the different experiments are covered. Finally, in Section 6.3 the different results of the different experiments are discussed.

## 6.1 Experimental Setup

Several experiments were performed to determine the scale of effectiveness regarding the detection of the following three code smells: Long Method, Feature Envy, and Large Class. To use the datasets as input for the three different SPN models, the datasets had to be cleaned and pre-processed. That resulted in training, validation, and test data for the different SPN models. As mentioned in the previous chapter, to create the different SPN models the library SPFlow was used and the library Gensim[1] was used to incorporate the word embeddings of the code entities. Gensim is a Python implementation of the *word2vec* algorithm. Subsequently, the following hyper-parameters were set to configure the different SPN models:

- The first hyper-parameter is the **type** of the SPN model to employ. With SPFlow it is possible to create two types of SPN models: parametric and mixed (non-parametric) SPN models. Figure 2.5, from Chapter 2, illustrates a simple probabilistic graph of an SPN model. In this graph, the leaves represent the distributions of the different classification features (i.e., code metrics and word embeddings) to train the SPN models. Every feature is a vector of numbers with a dimension equal to the number of instances. Within SPFlow a feature (leaf node) is considered a random variable having a probability distribution. In SPFlow, when creating an SPN model it is possible to configure the distributions to model the different features with the following types:

[1] `https://radimrehurek.com/gensim/models/word2vec.html`

Gaussian, Bernoulli, Gamma, etc. Hence these types of SPN models created by the SPFlow library are called parametric SPNs. On the other hand, in the case of mixed SPN models the probability distributions of the features, i.e., the leaf nodes in Figure 2.5, are inferred from the training data by the SPFlow library.

- The second hyper-parameter is the **threshold**. The prediction made by the SPN model whether a code entity is classified as a code smell (or not) depends on the fact that the output, i.e., probability, is greater than the threshold value.
- The third hyper-parameter is the **min instances slice** within SPFlow. SPFlow clusters instances (slicing) of a dataset to create sum nodes after its rows are recursively split into independent subsets. The number of instances in a cluster must not go below the threshold set by this hyper-parameter. It is important to set an appropriate value for this parameter to prevent over-fitting.
- The fourth hyper-parameter is **min features slice** within SPFlow. With this parameter, features are split up (chopping) until a threshold, set by this hyper-parameter, is reached to create a product node. It is also important to set an appropriate value for this parameter to prevent over-fitting.
- To utilize the semantic information of the source code, described by identifiers (i.e., field names, method names, and class names) within a class or method under investigation, the *word2vec* implementation of the library Gensim is used. This leads to the fifth hyper-parameter, the **word sequence** (or the maximum number of words) of an identifier. A given identifier is partitioned into a sequence of words according to capital letters and underscores. Each identifier should contain no more than a max number of words.
- The sixth hyper-parameter is also set by the Gensim library. That is the **dimension** of the numerical vector of a word embedding in the hyper-parameter word sequence. Each word in the word sequence is converted to a numerical vector having a fixed-length set by the given dimension.

As part of the hyper-parameter optimization process, there were two options. The first option was to fiddle with the hyper-parameters manually, until a combination of hyper-parameters values was found in which the specific SPN model performed best. This would have been a very time-consuming process to explore the many combinations. Therefore, we opted for the second option, namely **grid search** [3][17]. With grid search, different SPN models are trained with different combinations of hyper-parameters values. As described in the previous chapter, a validation set is extracted from the training set, i.e., the hold-out validation set. On the validation set, a respective SPN model for its specific code smell was evaluated. Though the number of combinations of hyper-parameter values can become quite large, in this case the number of combinations of hyper-parameter values were few. Furthermore, the grid search is also embarrassingly parallel, therefore Scikit-Learn's **GridSearchCV**[2] was employed to automate the search for the optimal hyper-parameter values. However, not all hyper-parameters have been set by means of grid search. Only the hyper-parameters **min instances slice** and **min features slice** were set by means of grid search. The hyper-parameters list in Table 6.1 has been adopted from the study in [26].

[2]https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Table 6.1: Hyper-parameters set for the different classifiers.

| Hyper-Parameter | Value |
|---|---|
| Threshold | 0.5 |
| Word sequence | 5 |
| Dimension | 200 |

## 6.2 Results

This section covers the results regarding the different experiments to detect the code smells: Long Method, Feature Envy, and Large Class.

### 6.2.1 Detecting Code Smell Long Method

The results of the experiments covered in this section aim to answer part of the research question **RQ-1**: finding out if SPNs outperform state-of-the-art techniques to detect code smells, as covered in [26]. The classification features to detect the code smell Long Method consists of code metrics only. The distributions of the data sets as shown in Appendix B presume that Gamma distribution to model the different features would suffice for most features. Therefore, 4 different parametric SPN models were created, and one mixed SPN model. The reason to include a mixed SPN model in this experiment was also that the data distributions of the datasets showed some skewness. For the different parametric SPN models, different probability distributions for both classification features (i.e., input) and output were used. These different SPN models are also labeled in Table 6.2.

Table 6.2: The different SPN models to detect code smell Long Method.

| Label | Distribution Input | Distribution Output |
|---|---|---|
| SPN1 | Gaussian | Gaussian |
| SPN2 | Gaussian | Bernoulli |
| SPN3 | Gamma | Gaussian |
| SPN4 | Gamma | Bernoulli |
| SPN5 | Mixed | Mixed |

The results produced by the different SPN models listed in Table 6.2 are reported in more detail in Appendix C. In Table 6.3, the average results of the performance of the different SPN models and that of the DL approach, specified in [26], are shown. Based on the results of Table 6.3, it is clear that SPN5, i.e., the SPN model configured with non-parametric distributions (mixed in SPFlow), outperforms the other models for every performance metric. Therefore, the following observations are made:

- First, the mixed SPN model (SPN5) significantly outperforms the DL approach regarding the F1 metric and to a lesser extent regarding the ROC AUC metric. With SPN5 the F1 metric improved by 34.38%(=89.91%-55.53%), and the ROC AUC improved by 4.41%(=83.65%-79.24%) respectively.

- Second, SPN5 can identify most of the Long Method smells. Its average recall is up to 86.43%. When compared to the DL approach, the improvement is significant, 7.44%(=86.43%-78.99%).

- Third, SPN5 also results in a higher precision (78.96%) than that of DL approach (42.81%).

Table 6.3: Results modeled with different SPN models configured with different distributions for input and output to detect code smell Long Method.

| Model | Precision | Recall | F1 | ROC AUC |
|---|---|---|---|---|
| SPN1 | 66.03% | 71.20% | 67.63% | 73.77% |
| SPN2 | 52.04% | 66.42% | 61.39% | 48.21% |
| SPN3 | 68.04% | 74.77% | 70.69% | 73.67% |
| SPN4 | 62.88% | 84.01% | 71.08% | 64.21% |
| SPN5 | **78.96%** | **86.43%** | **81.91%** | **79.65%** |
| DL Approach | 42.81% | 78.99% | 55.53% | 79.24% |

### 6.2.2 Employing Word Embedding

Partly due to the results of the mixed SPN model in the previous section, we opted for this type of SPN model to evaluate the performance of the SPN models to detect code smells Feature Envy and Large Class. The results of both experiments are reported in more detail in Appendix D and Appendix E. With these experiments, the aim is to answer the following research questions. **RQ-1** as discussed in the previous section, and **RQ-2**. With **RQ-2** the aim is to assess whether using word embeddings improves the performance of code smell detection.

In Appendix D, the results are reported of the SPN model to detect the code smell Feature Envy with and without the use of word embeddings. The average results are shown in Table 6.4. The first row in Table 6.4 shows the average results of the SPN model in which the word embeddings were used. However, these results are skewed when studying the results in Appendix D. The scores for all projects regarding the recall metric are quite extreme, namely 100%. Furthermore, every project gives a ROC AUC score of 50% unlike the ROC AUC scores of the DL approach. Thus the specific SPN model classified the code smell Feature Envy in the different projects at random.

Appendix D also reported the results of the SPN model trained with only code metrics as classification features. The average results are shown in the second row of Table 6.4. The performance scores for precision and recall hover around the average values seen in Table 6.4. Therefore, the following observations from Table 6.4 are made:

- First, the SPN model (trained with code metrics only) significantly outperforms the DL approach regarding the F1 metric. With SPNs the F1 metric improved by 12.08%(=63.99%-51.91%).
- Second, the DL approach performs significantly better with regards to the ROC AUC metric, 35.81%(=84.9%-49.09%). We do have to note that the low average of the accuracy score was because some projects gave a ROC AUC score of 0.00%. However, the performance is somewhat similar to that of JDeoderant [26].
- Third, the SPN model (trained with code metrics only) identifies most Feature Envy smells. Its recall is up to 91.68%. Compared to the DL approach, the recall is improved, though not by a large margin as the rates for both approaches were already

high, 3.57%(=91.68%-88.11%).

- Fourth, the SPN model (trained with code metrics only) also has a higher precision (49.23%) compared to the DL approach (36.79%).

Table 6.4: Average results regarding the detection of code smell Feature Envy.

| **Model** | **Precision** | **Recall** | **F1** | **ROC AUC** |
|---|---|---|---|---|
| SPN (metrics + word embeddings) | *13.43%* | *100%* | *17.78%* | *50.00%* |
| SPN (metrics only) | **49.23%** | **91.68%** | **63.99%** | 49.09% |
| DL Approach | 36.79% | 88.11% | 51.91% | **84.90%** |

In Appendix E, the results for the SPN model to detect code smell Large Class are reported. The average results of the SPN model, incorporating word embeddings, are shown in the first row of Table 6.5. The results for most of the projects in the dataset also seem skewed based on the results in Appendix E. The SPN model gives a ROC AUC score of 0.00% for these 5 projects, thus being unable to detect code smell Large Class. The only projects that do seem to show valid results are the following: Freeplane, Android, Grinder, and JexcelAPI. For these projects, the SPN model outperforms the DL approach with regard to the F1 metric by a margin of more than 30%. The ROC AUC scores for these projects are also larger than 50%.

Appendix E also reports the detailed results of the SPN model with only code metrics as classification features. The average scores are shown in the second row of Table 6.5, and the following observations stand out:

- First, the SPN model significantly outperforms the DL approach regarding the F1 metric. The performance improved by 47.64%(=69.37%-22.33%).
- Second, the DL approach shows a much higher performance with regard to the ROC AUC metric. The DL approach outperforms the SPN model significantly, by 34.74%(=75.77%-41.03%). We also note that the low ROC AUC metric for the SPN model is caused because the model showed an extremely low performance for several projects.
- Third, the SPN model identifies the code smell Large Class with a high rate, just like the DL approach. However, when the approaches are compared to each other the recall of the SPN model still shows an improvement of 3.12%(=84.07%-80.95%).
- Fourth, the high recall (84.07%) of the SPN model did not result in a much lower precision when compared to the DL approach. The SPN model shows a significant improvement of precision by 46.60%(=59.55%-12.95%).

Table 6.5: Average results regarding the detection of code smell Large Class.

| **Model** | **Precision** | **Recall** | **F1** | **AUC** |
|---|---|---|---|---|
| SPN (metrics + word embedding) | *87.39%* | *41.68%* | *79.02%* | *39.29%* |
| SPN (metrics only) | **59.55%** | **84.07%** | **69.37%** | 41.03% |
| DL Approach | 12.95% | 80.95% | 22.33% | **75.77%** |

## 6.3 Discussion

The non-parametric SPN models outperformed the DL approach, defined in [26], regarding the detection of the code smell Long Method only. The SPN models to detect code smells Feature Envy and Large Class, trained with only code metrics, performed worse than the DL approach based on the ROC AUC scores. The SPN models in these cases reported a ROC AUC score of 50% and less, thus not better than a random classifier. However, based on precision and recall the SPN models did perform better, which was not expected. The SPN models that incorporated word embeddings performed even worse on average when taking all performance metrics into account. In this study word embeddings were only used to detect the code smells Feature Envy and Large Class. The SPN models to detect both Feature Envy and Large Class also did not show better results than a random classifier, as the ROC AUC scores for both models hover around 50% on average (or lower). In the case of detecting code smell Large Class the ROC AUC score was 0.00% for more than half of the projects. The following three factors seem to contribute to this under-performance are the following. Firstly, an imbalance between positive and negative samples, i.e. code smells vs no code smells. Secondly, the wrong features might have been selected when testing the SPN models to detect code smells Feature Envy and Large Class. Finally, the size of the dataset might also not have been sufficient regarding the training of SPN models.

In [34] and [35] it is shown that data balancing and an extensive dataset could be a key factor in improving the reliability of SPN models. Figure 6.1 does not show any issues regarding imbalance between positive versus negative samples regarding the code smell Feature Envy. The ratio of features versus training instances is also far below 1. However, when word embeddings were used the number of features increased to 3002 per instance. That is because for every instance the word embeddings account for 3000 of the 3002 features, thus increasing the imbalance regarding the type of features. Additionally, the combination of distance metrics and output did seem random, as the difference between the two distance metrics was very small, see Figure 5.2.

In Figure 6.2, we see that there is a major imbalance between positive versus negative samples regarding the dataset to detect code smell Large Class. When using word embeddings, the ratio of features versus training instances is well above 1. That is because the number of training instances is not that big for the different projects and the number of features is at least 2010. As most classes have at least one method and one field. Both these factors have major consequences regarding the SPN model incorporating word embeddings during the training phase, as the word embeddings account for more than 2000 of the 2010 features. This might lead to unreliable results.

Extending the SPN models with *word2vec* to incorporate word embeddings as features with code metrics might therefore have to be reconsidered, as these features did not lead to an improvement in performance.

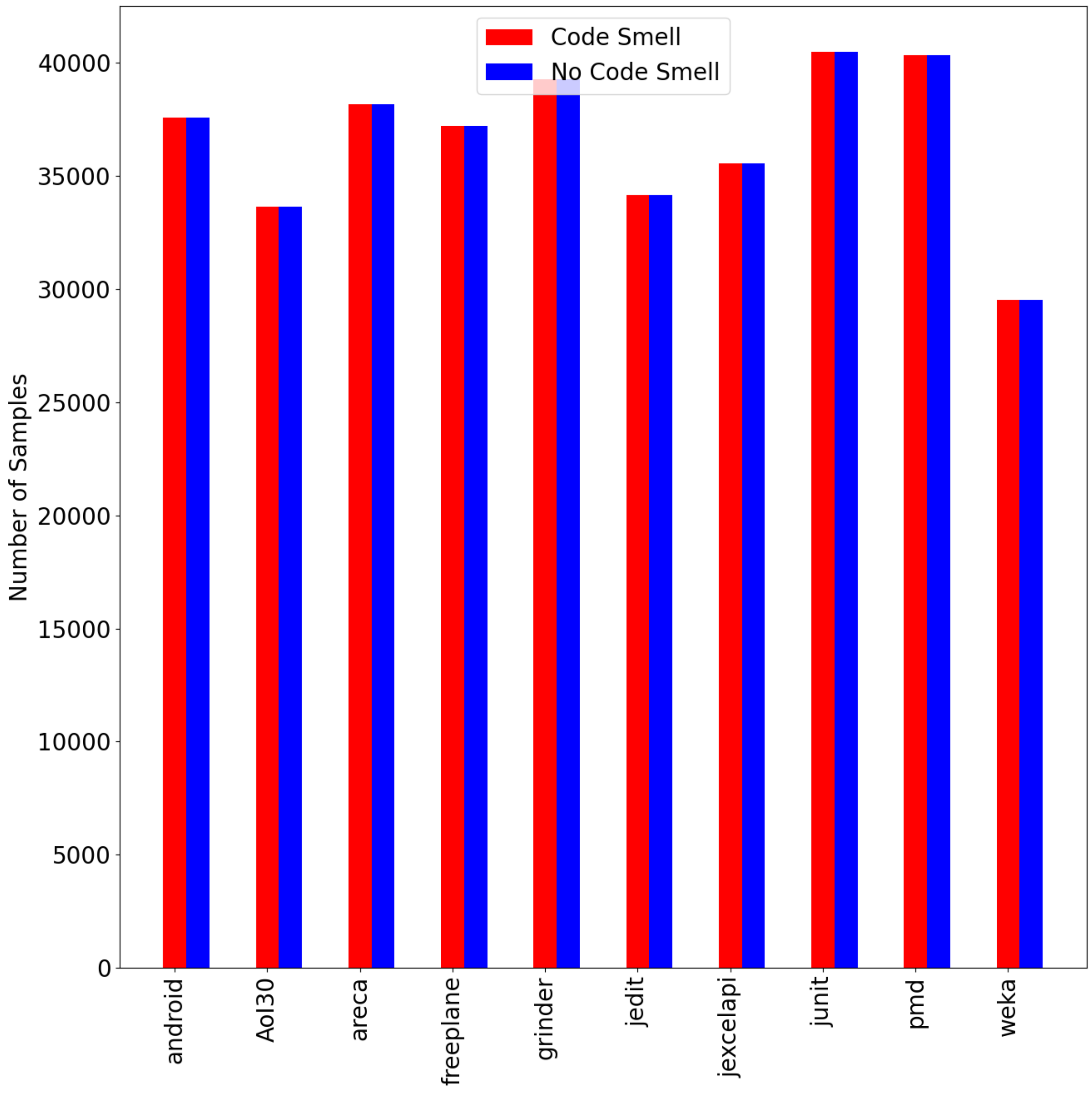


Figure 6.1: A distribution of instances, or samples, per project regarding code smell Feature Envy.

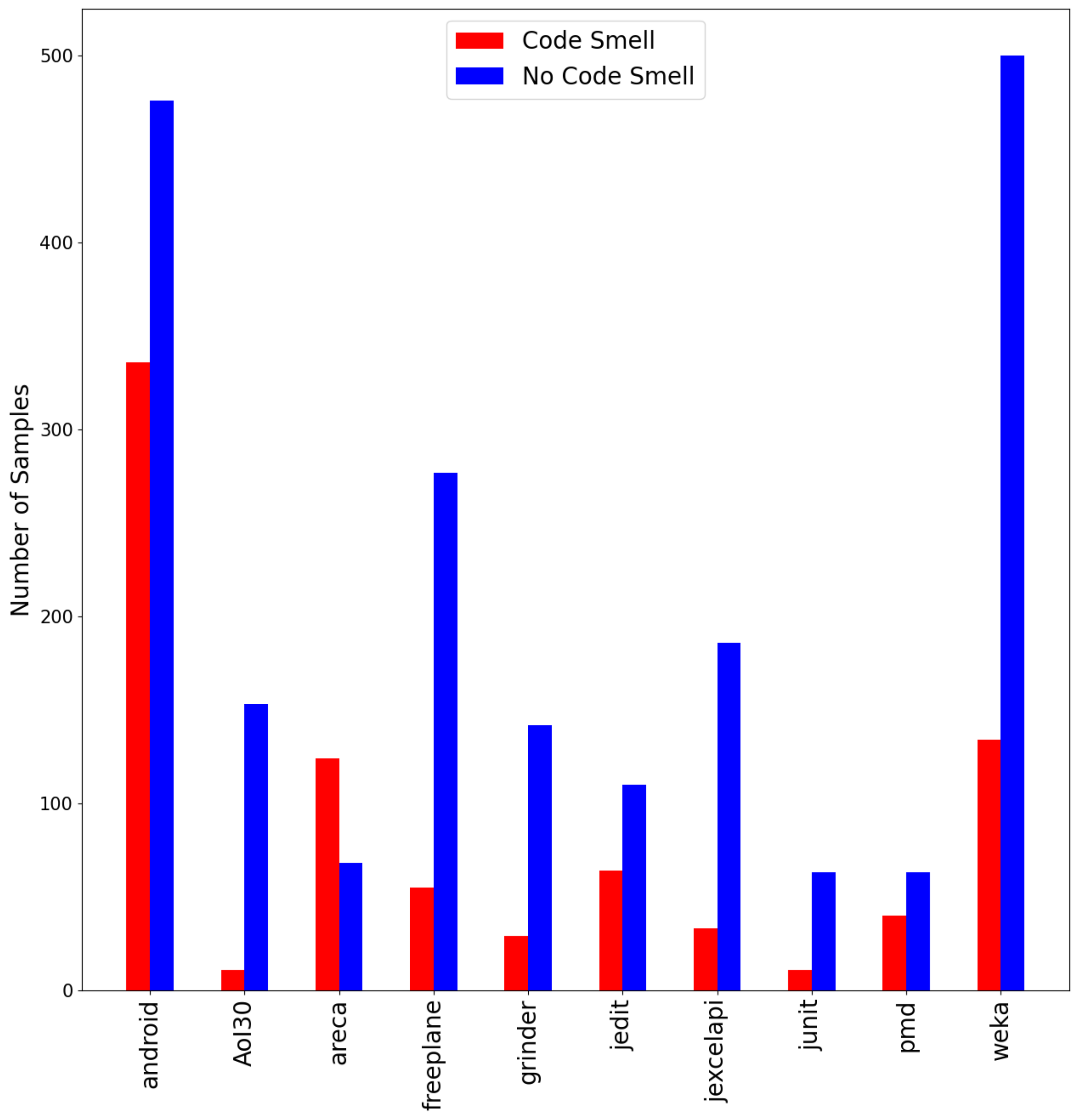


Figure 6.2: A distribution of instances, or samples, per project regarding code smell Large Class.

# 7 Applicability of SPNs in Code Smell Detection

In this chapter, the focus will mainly be on the assessment of the practical applicability of SPNs regarding code smell detection. This is done by answering research questions: **RQ-3** and **RQ-4** as defined in Chapter 4. With research question **RQ-3**, the aim is to find out which features, i.e., code metrics, are of significance when detecting the different code smells using SPNs. Following this, two cases (not part of the datasets used to train the SPNs) are analyzed to assess the validity of the results of research question **RQ-3**. Finally the last research question, **RQ-4**, essentially investigates the practical applicability of SPNs with regard to tooling. Therefore, in Section 7.1 the results of **RQ-3** are discussed. This is followed by an analysis of two cases using the results of Section 7.1 in Section 7.2. Finally, in Section 7.3 the results of **RQ-4** are discussed.

## 7.1 Relevant Features

In this section, the results of **RQ-3** are discussed, namely to what extent are the individual features relevant in detecting the different code smells. Thus, is it possible to exclude some low-quality features in detecting code smells, and is it applicable in practice? As outlined in previous chapters, an SPN can be thought of as a set of sums/products nodes and leaf nodes, i.e., classes and features, in a DAG. The probability distributions of the internal nodes and leaf nodes are estimated over class and feature values [39]. After performing the marginal inference task on the feature of interest, it is conditioned on the likelihood of a code smell. With this procedure the relevancy of the feature of interest is computed regarding the event of a code smell or not.

Different experiments were performed to assess the relevancy of the different code metrics to detect the different code smells: Long Method, Feature Envy, and Large Class by using the dataset defined in Table 5.1. In Appendix F, the results are reported for every code metric of the three different code smells, and below the results are discussed for every code smell separately.

## Long Method

In Figure 7.1, the results are shown of the code metrics that affect detecting code smell Long Method the most. These code metrics mentioned below are covered in more detail in Appendix A. It is clear that the probability of a code smell is significantly higher than the probability of a non-code smell when the values for the cohesion metrics CC, LCOM4, and COH are low. Whereas, regarding code metrics that measure the complexity of the method under investigation, the following are of importance: CD, Size, and NOAV. As expected, a high value for the metric CD gives a higher probability of a code smell than the probability of a non-code smell. However, with regard to outliers it seems that the SPN model does have some issues. This was not expected, as we expected an increase in the probability of a code smell as complexity increases for example.

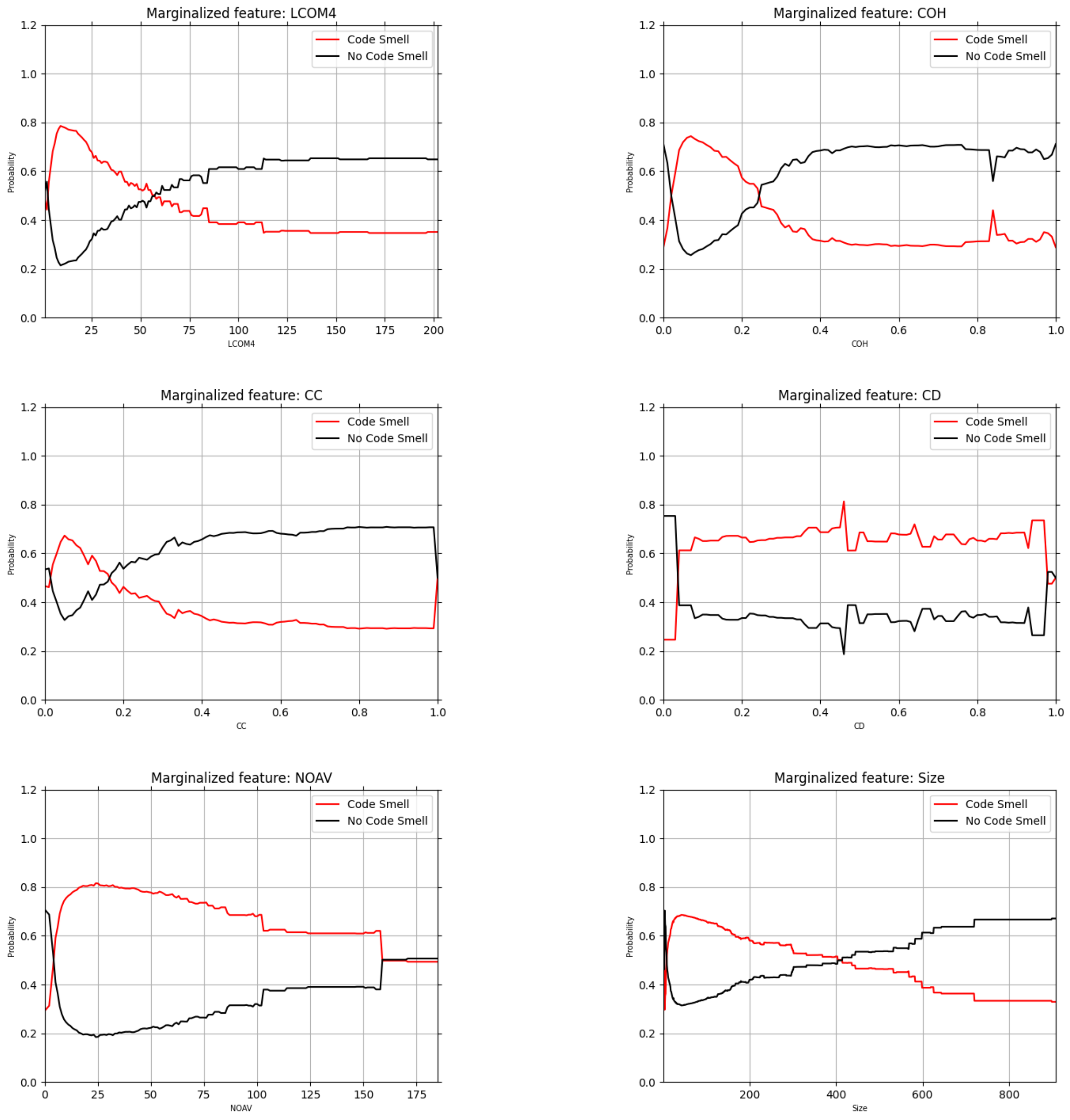
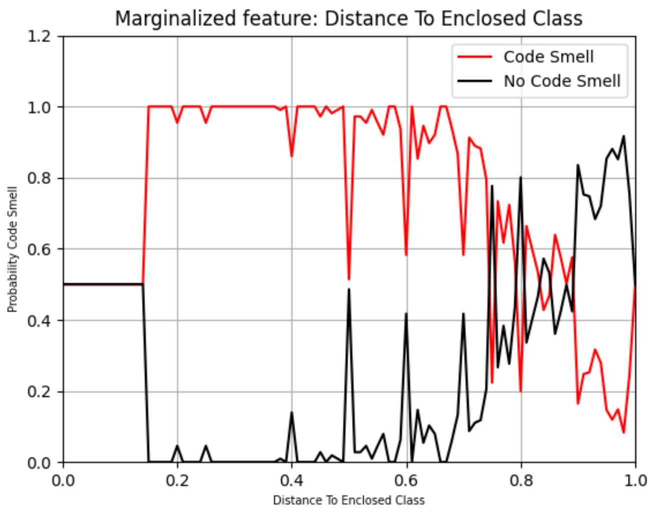


Figure 7.1: The selected features that are most relevant in detecting code smell Long Method.



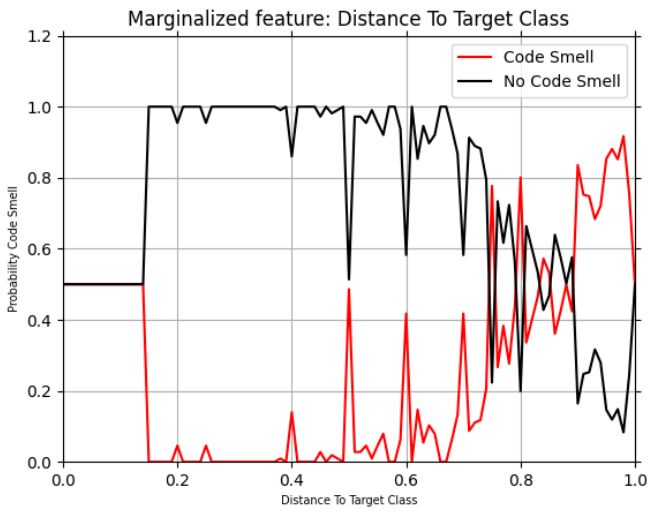


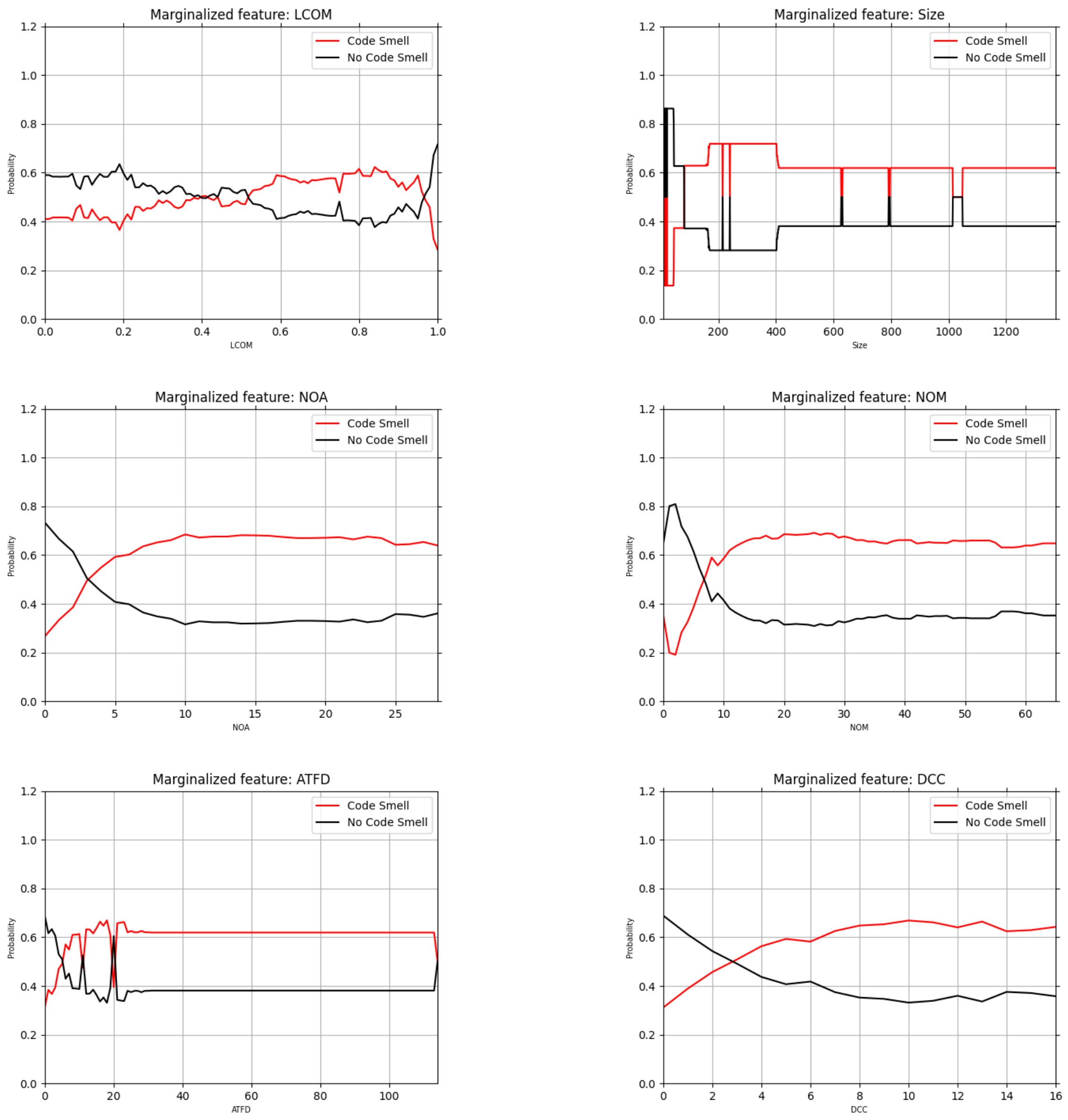
Figure 7.2: Feature Envy: The selected features that have the most influence to detect the code smell.

## Feature Envy

In Figure 7.2, both distance metrics used to detect the code smell Feature Envy is shown in such a way, that it is clear that both views are the inverse of each other. This is logical as the probability of a code smell increases when the distance to the target class increases. The probability of a non-code code smell increases when the distance to the enclosed class decreases. In this case, it is possible to use only one of the distance metrics. However, it is clear from both views that uncertainty increases as both distance values go to 1. Both these code metrics are covered in more detail in Appendix A.

## Large Class

In Figure 7.3, the code metrics that affect the detection of code smell Large Class the most are shown. The cohesion metric LCOM shows an increase in the probability of a code smell when the value increases to 1, as an increasing value (from 0 to 1) means a decrease in cohesion in this case. The code metrics measuring complexity that are the most relevant are the following: NOA, NOM, ATFD, Size, DCC, and WMC. It shows, intuitively, that the probability of a code smell increases as complexity increases, and the probability of a non-code smell decreases when complexity decreases. Also in this case there seem to be some issues when dealing with outliers. The code metrics mentioned above are covered in more detail in Appendix A.



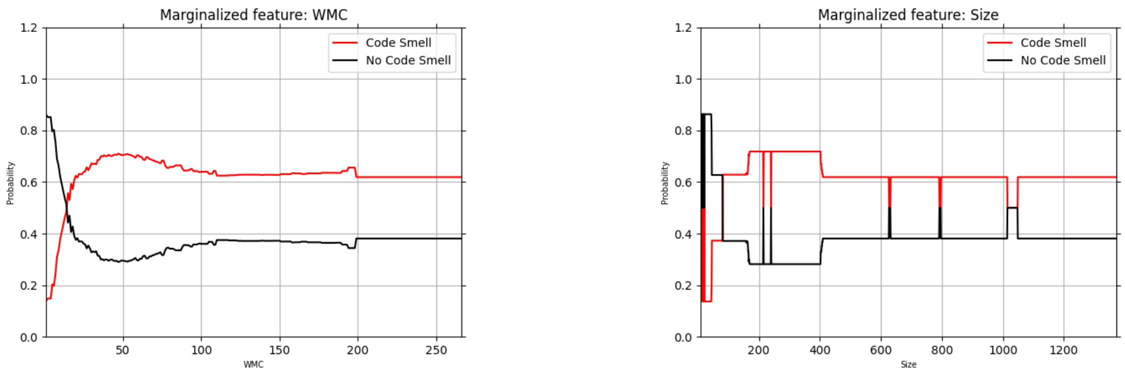


Figure 7.3: Large Class: The selected features that have the most influence to detect the code smell.

## 7.2 Case: Code Smell Detection with SPNs

In this section, the potential practical applicability will be assessed as a follow regarding research question **RQ-3**, with the results of the previous section taken into consideration. To perform this case study, the Java application *Neuroph*[1] version 2.9 was selected. This open-source software application was also used in [26] as part of their case study. *Neuroph* is a lightweight neural network framework providing a library and GUI tool to facilitate in creating training, and saving of different neural networks.

Furthermore, the focus will be on code smell Long Method when performing the analysis regarding research question **RQ-3**. With the analysis of two code listings, shown in Listing 7.1 and Listing 7.2, we want to assess the validity to the notion that certain features carry more weight in detecting code smells. However, before starting with the analysis the following steps were performed. The first step is a manual inspection of the source code regarding the code smell Long Method. The second step is to validate the code smell if present. The third step is to train the SPN model with the same training data used in the previous chapter. Finally, the SPN model is applied to detect the Long Method code smell.

In Table 7.1, the results are shown with code metrics that carry the most weight regarding code smell detection. Based on the code metrics, shown in Figure 7.1, the code in listing Listing 7.1 is indeed a code smell according to the SPN model, and also according to the DL approach in [26]. It is also very obvious that a number of methods can be extracted from the code in Listing 7.1. SPFlow creates an SPN model based on the training data with sum, product, and leaf nodes akin to the SPN graph of Figure 7.4. This figure shows the "marginalized" SPN graph that SPFlow created by "removing" the non-relevant code metrics. Furthermore, the internal nodes are not shown, and thus not all connections of the leaf nodes with the internal nodes are shown. Additionally, every leaf node consists of its probability distribution that models the code metric of relevance.

With the SPN model, an inference task is performed by marginalizing the code metrics that are not relevant. Every leaf node computes a probability given a value of a code metric associated with Listing 7.1. All leaf nodes cascade the probabilities via internal sum/product nodes to the root node, with the leaf nodes of not relevant code metrics having a probability of 100%. The root node computes in this case a marginal likelihood value, that is eventu-

[1]http://neuroph.sourceforge.net

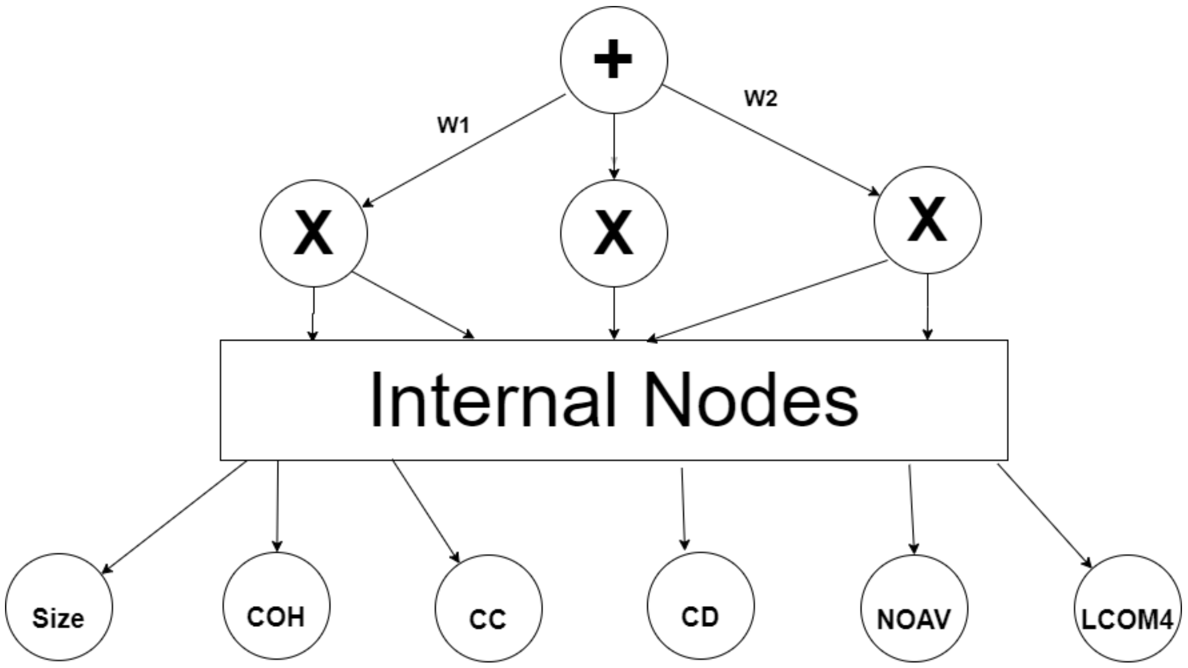


Figure 7.4: The SPN graph generated by SPFlow for the case study shows one sum node at the root and three product nodes.

ally converted to a probability of a code smell Long Method. Thus, the model created by SPFlow outputs a probability higher than 76% that Listing 7.1 is a code smell Long Method, given a threshold of 50%.

| **Code Smell** | **Metric** | | | | | | **Output** | |
|---|---|---|---|---|---|---|---|---|
| | **Size** | **COH** | **CC** | **CD** | **NOAV** | **LCOM4** | **DL Approach NNs** | **SPN Code Smells Detector** |
| Code Smell 1 | 69 | 0.175 | 0.1701 | 0 | 5 | 2 | 1 | 1 |
| Code Smell 2 | 15 | 0.21 | 0.2222 | 0.5 | 4 | 4 | 0 | 1 |

Table 7.1: Metrics for the two code samples to detect code smell Long Method.

Listing 7.1: Code smell 1

```
    /*
    * initialize the form.
    * WARNING: Do NOT modify this code. The content of this method is
    * always regenerated by the Form Editor.
    */
   private void initComponents()//GEN-BEGIN:initComponents
   {
      m_popupMenu = new javax.swing.JPopupMenu();
      m_menuAddItem = new javax.swing.JMenuItem();
      m_menuRemoveItem = new javax.swing.JMenuItem();
      jScrollPane1 = new javax.swing.JScrollPane();
      m_table = new javax.swing.JTable();

      m_menuAddItem.setText(""Add item"");
      m_menuAddItem.addActionListener(new java.awt.event.ActionListener()
      {
         public void actionPerformed(java.awt.event.ActionEvent evt)
         {
            m_menuAddItemActionPerformed(evt);
         }
      });

      m_popupMenu.add(m_menuAddItem);

      m_menuRemoveItem.setText(""RemoveItems"");
      m_menuRemoveItem.addActionListener(new java.awt.event.ActionListener()
      {
         public void actionPerformed(java.awt.event.ActionEvent evt)
         {
            m_menuRemoveItemActionPerformed(evt);
         }
      });

      m_popupMenu.add(m_menuRemoveItem);

      setLayout(new java.awt.BorderLayout());

      jScrollPane1.setBorder(null);
      m_table.setModel(new javax.swing.table.DefaultTableModel(
      new Object [][]
      {
         {null, null, null, null},
         {null, null, null, null},
         {null, null, null, null},
         {null, null, null, null}
      },
      new String []
      {
```

```
49          ""Title 1"", ""Title 2"", ""Title 3"", ""Title 4""
50       }
51       ));
52       m_table.setTableHeader(null);
53       m_table.addMouseListener(new java.awt.event.MouseAdapter()
54       {
55          public void mouseClicked(java.awt.event.MouseEvent evt)
56          {
57             tableMousePressed(evt);
58          }
59          public void mousePressed(java.awt.event.MouseEvent evt)
60          {
61             tableMousePressed(evt);
62          }
63          public void mouseReleased(java.awt.event.MouseEvent evt)
64          {
65             tableMousePressed(evt);
66          }
67       });
68
69       jScrollPane1.setViewportView(m_table);
70
71       add(jScrollPane1, java.awt.BorderLayout.CENTER);
72
73    }
```

Listing 7.2 is not labeled as a code smell by the DL approach, whilst the SPN model did label it as a code smell, as shown in Table 7.1. The procedure to compute the probability of code smell Long Method is similar to that of Listing 7.1. As expected the cohesion features CC, COH, CD, and LCOM4 mainly affected the prediction of whether Listing 7.2 is a code smell or not. These metrics for Listing 7.2 show a significantly higher probability of a code smell, as shown in Figure 7.1. The trained SPN model outputs Listing 7.2 as a code smell Long Method with a probability of more than 57%, thus classifying it as a code smell given a threshold of 50%. Though the probability of a code smell is much lower than the case regarding Listing 7.1, as expected. When analyzing the code it is easy to spot which lines of code can be extracted to a separate method, though the size (or lines of code) of the method is only 15. Lines 12-18 in Listing 7.2 can be extracted to a separate method, which can then be called from the original method. This is because the static method invocations in those lines have a low cohesion regarding the overall method. Hence, it can easily be extracted to a separate method. The refactored method is shown in Listing 7.3.

Listing 7.2: Code smell 2

```
    /**
     * Prints Neuroph data set
     *
     * @param neurophDataset Dataset Neuroph data set
     */
    public static void printDataset(DataSet neurophDataset) {
        System.out.println(""Neuroph dataset"");
        Iterator iterator = neurophDataset.iterator();

        while (iterator.hasNext()) {
          DataSetRow row = (DataSetRow) iterator.next();
            System.out.println(""inputs"");
            System.out.println(Arrays.toString(row.getInput()));
            if (row.getDesiredOutput().length > 0) {
                System.out.println(""outputs"");
                System.out.println(Arrays.toString(row.getDesiredOutput()));
                System.out.println(row.getLabel());
            }
        }
}
```

Listing 7.3: Code smell 2 refactored

```
    /**
     * Prints Neuroph data set
     *
     * @param neurophDataset Dataset Neuroph data set
     */
    public static void printDataset(DataSet neurophDataset) {
        System.out.println(""Neuroph dataset"");
        Iterator iterator = neurophDataset.iterator();

        while (iterator.hasNext()) {
          DataSetRow row = (DataSetRow) iterator.next();
          printDataSetRow(row);
        }

    public static void printDataSetRow(DataSetRow row) {
        System.out.println(""inputs"");
        System.out.println(Arrays.toString(row.getInput()));
        if (row.getDesiredOutput().length > 0) {
            System.out.println(""outputs"");
            System.out.println(Arrays.toString(row.getDesiredOutput()));
            System.out.println(row.getLabel());
        }
}
```

## 7.3 Practical Applicability in an IDE

In Chapter 3, it was already concluded that practically no tooling exists employing SPNs to detect code smells in an existing integrated development environment (IDE). However, with research question **RQ-4** the aim was to find out if it is possible to incorporate SPNs, to detect code smells, within an existing IDE. To get relatively quick results, a code smell detection tool using SPNs as a plug-in within Eclipse[2] is the preferred option. Eclipse enjoys our preference as it is one of the most popular IDEs for Java, as the focus of this study is detecting code smells in Java source files. This is an alternative to IntelliJ IDEA[3], which is an expensive commercial IDE that can also be used to develop Java software. Eclipse is also easily extendable via its well-known plug-in architecture[4], hence another reason to prefer an Eclipse plug-in. This eliminates the need to develop a whole new user interface from scratch, which is something that has to be done when incorporating the code smell detection mechanism in a stand-alone tool. Furthermore, a stand-alone tool separated from the IDE would be inconvenient, as developers have to switch back and forth between the source code in the IDE and the stand-alone tool regarding code smell detection.

Before outlining the approach to detect code smells within Eclipse, we evaluated three Eclipse plug-ins that are available for download. We selected these three tools because they analyze Java source files and are open-source projects. The results were covered in Chapter 3. However, the source code (on a high level) of the different tools was also evaluated. Based on the evaluations of these tools the following approach with regards to the plug-in to detect code smells using SPNs is defined. There are two main components to integrate code smell detection in an IDE. The first component is the IDE Eclipse, and the second component is the code smell detection mechanism using SPNs that runs in the back-end. Eclipse serves as the front-end that provides the user interface, interacts with the user, and handles all user actions. The code smell detection plug-in is responsible for detecting the code smells within a Java program and outputting a list of potential code smells. Thus another important requirement is that the plug-in should provide a list of potential code smells, with the following information for each code smell:

- type of code smell: Long Method, Feature Envy, or Large Class for example
- Java file name,
- location (line number) of code smell in the Java source file,
- likelihood of a code smell.

In Figure 7.5, a high-level overview is shown of the aforementioned tool to detect code smells. It shows that after Java source files are pre-processed it produces a dataset suitable for code smell detection. The code smell detection plug-in processes the dataset and detects potential code smells, and outputs a list of code smells and where to locate the "smelly" code entities. This list of code smells can be a combination of the three different types of code smells Long Method, Feature Envy, and Large Class. After this step, it is up to the developer to take the necessary actions regarding refactoring, if necessary.

[2]https://www.eclipse.org/

[3]https://www.jetbrains.com/idea/

[4]https://www.vogella.com/tutorials/EclipsePlugin/article.html

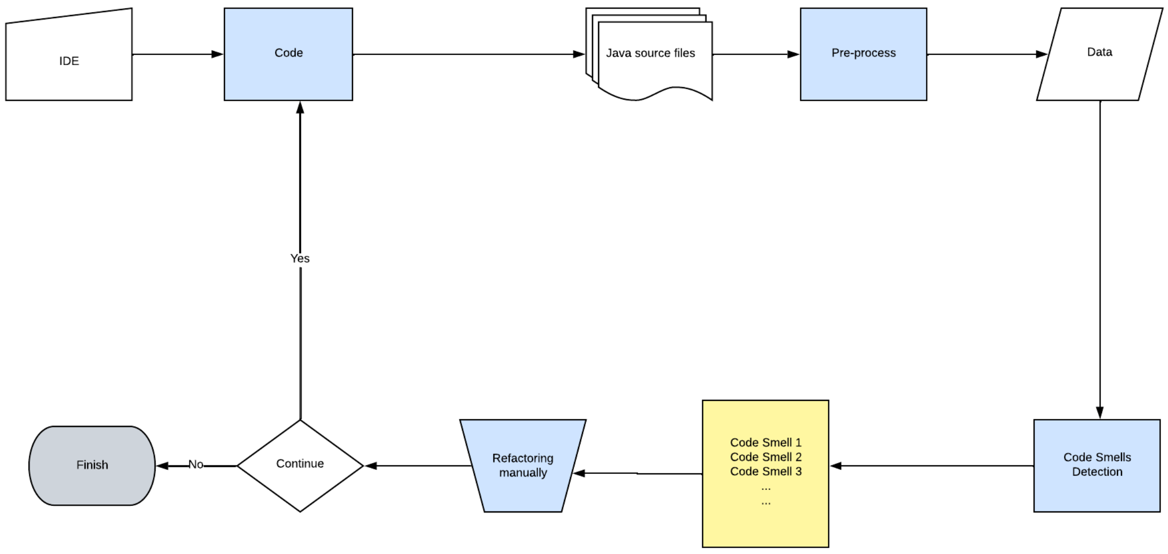


Figure 7.5: An overall view integrating code smell detection in an IDE.

The biggest issue when developing an Eclipse plug-in to detect code smell using SPNs is that there are no libraries available in Java[5] regarding SPNs. The SPN models to detect code smells that were used for the different experiments in this study were developed in Python using the library SPFlow, also written in Python. However, this is a software development question and the SPFlow library could for example be ported to Java or Python modules could be called within a Java program. Therefore, an extended up-front design is not needed as the workflows regarding input and output are simple. Furthermore, it is also important that the first version of the plug-in does nothing more than what was specified by the requirements. This to avoid the "what if" trap by doing too much[6]. Therefore, based on the evaluation of the aforementioned static code analyzers, mentioned in Chapter 3, it is possible to create a plug-in that can be used during development to evaluate the process of code smell detection. In [36], it is shown that not all code smells are "equal" and that some code smells have a higher priority than others.

[5]https://github.com/
[6]https://codeopinion.com/stop-over-engineering-software/

# 8 DISCUSSION

For this study, we have compared the SPN approach to the DL approach in [26]. The results in Chapter 6 show that SPNs do improve code smell detection in Java code when compared to state-of-the-art techniques, to some extent. That is especially the case with regards to the code smell Long Method. The SPN model outperformed the DL approach for every performance metric: precision, recall, and ROC AUC.

Although the experiments showed promising results, especially in the case of code smell Long Method, there are some critical notes. With the follow-up experiments regarding code smells Large Class and Feature Envy also code metrics were used as classification features as part of answering research question **RQ-1**. The SPN models showed an average ROC AUC score of around 41% and 49% respectively. This suggests that the SPN models performed no better than a random classifier with the datasets used that were procured for this study. However, the datasets that were used regarding code smell Feature Envy and Large Class were not balanced. The imbalance regarding the datasets to detect the code smell Feature Envy had more to do with the distribution of the distance metrics. The distributions regarding the distance of a method to its parent class and a potential target class were both skewed to the maximum value of 1. The datasets used to detect code smell Large Class was relatively small, and as SPNs typically learn from data, inferences produced with the models could be unreliable [8][34][35]. Furthermore, though the datasets used are available publicly, still a lot of pre-processing was needed to make the datasets suitable for training, validation, and testing.

Another issue concerning the features to detect the code smell Feature Envy is the following. These distance metrics might not have been appropriate for the respective experiments. This leads us to the discussion regarding the other code metrics selected for this study. The focus was mainly on the code metrics used in [26]. However, many code metrics have been developed over the years. It is not an easy task to decide which code metrics are suitable for the task of code smell detection. Although, as discussed in Chapter 7, by employing SPNs it is much easier to evaluate which code metrics are "more suitable" for code smell detection and which are not. Though code metrics are mainly developed to measure and improve software quality [23]. Hence, the question arises if code metrics in general are appropriate for code smell detection.

Additional experiments employing word embeddings, in this case using *word2vec*, did not lead to an improvement in the performance of the SPNs regarding code smell detection of Feature Envy and Large Class. The models showed results that seemed skewed regarding the metrics precision, recall, and ROC AUC. Furthermore, it also took multiple days to train the SPN models to detect a code smell Feature Envy. The combination of more than 10K training instances and more than 3000 features lead to a data explosion. *Word2vec* was originally developed with NLP in mind [4]. Hence, leading to the question of whether a more code-minded technique would have led to a better performance regarding code smell detection of Feature Envy and Large Class. However, this does not mean we should write off techniques akin to *word2vec*. In this study, every numerical vector of a word embedding was transposed in such a way that every vector element served as a classification feature. Therefore, another massive imbalance was created as the number of features extracted from word embeddings was at least 2000, and the number of features extracted from code metrics was not more than 10. This might have also led to several distortions regarding the different performance scores. Thus, it is also possible that *word2vec* was not applied correctly.

Two cases were analyzed to assess the practical applicability of SPNs. Though the results of these experiments showed promising results, the number of cases was limited. The next step would have been incorporating SPNs in a tool, or an Eclipse plug-in in this case. This was not possible for this study due to time constraints.

# 9 Conclusions

This study basically assesses the potential of using SPN models to detect code smells. Three common code smells were selected, i.e., Long Method, Feature Envy, and Large Class, and an SPN model was created for each code smell using SPFlow. To assess the effectiveness of SPNs regarding the detection of code smells several experiments have been carried out to compare the performance to that of the DL approach in [26]. In [26], the DL approach outperformed the state-of-the-art static analysis tools, hence the same dataset was used. In this chapter, the answers to the research questions are discussed based on the results of the experiments and assessments. First, the sub-questions will be answered, followed by an answer to the main research question.

**RQ-1: To what extent do SPNs improve the detection of code smells when compared with state-of-the-art approaches?**
Results showed that SPNs have potential regarding the detection of code smells, based on the dataset used in this study. The different non-parametric SPN models showed the best performance when compared to parametric SPN models. There were also no trial-and-error approaches with different network configurations and sizes needed, as was the case with the DL approach in [26]. Thus, the SPN models were relatively easy to create. The SPN approach to detect the code smell Long Method outperformed the DL approach of [26], thus also the state-of-the-art static analysis tools. However, the SPN models to detect code smells Feature Envy and Large Class did not perform better than a random classifier based on the ROC AUC scores. Though, the SPN models did not perform any worse than JDeodorant regarding the ROC AUC scores [26]. The SPN models did perform better than the DL approach based on the metrics precision and recall. This all means that no definitive claims can be made about whether SPNs are an alternative to current state-of-the-art analysis tools. Therefore, more research is needed regarding the employment of SPNs in code smell detection. The procurement of datasets needed for training, validation, and testing is also very important in this regard.

**RQ-2: To what extent does the performance of code smell detection improve when a word embedding technique, *word2vec*, is used?**
The second part of the experimental phase consisted of incorporating *word2vec* to detect code smells Feature Envy and Large Class. With *word2vec* the aim was to capture the syn-

tactic and semantic word relationship of a code entity under investigation. However, employing word embeddings resulted in a data explosion as the number of features increased by more than 3000 regarding the detection of code smell Feature Envy. In the case of detecting code smell Large Class it was even worse as the number of features was larger than the number of instances. Hence, with the current datasets the SPN models showed no overall improvements when *word2vec* was used. However, this does not mean that no embedding technology would have the potential to increase the performance of code smell detection when incorporated into an SPN model. Therefore, more research is needed with regard to the employment of word embeddings and also with regards to alternative embedding techniques. Additionally, the procurement of more extensive datasets is also very important in this regard.

**RQ-3: Do some individual features carry more weight when detecting code smells?**
Different SPN models were created to perform marginal inference tasks based on the selected code metrics. The first experiment showed that some code metric values correlate strongly with whether a code entity is a code smell or not. This means that some code metrics showed a significantly higher marginal probability, higher than the threshold when a code entity was labeled a code smell. Based on these results, the code metrics that had the highest impact in detecting the code smell Long Method were selected for the analysis of two cases, that were not part of the training data. These cases showed that by high-level analysis of the process itself that SPNs can perform marginal inference without the loss of efficacy. Therefore increasing the potential to apply SPNs in a development environment aiding developers in detecting code smells. However, there might be some challenges with regard to robustness, because the SPN models do seem to have issues when dealing with outliers. Hence, more research is needed with a more extensive dataset.

**RQ-4: To what extent are SPNs applied in a development environment?**
In Chapter 3, we already concluded that there are no code smell detection tools incorporating SPNs available to assist developers in their software development process. However, creating a stand-alone or a plug-in for an existing IDE such as Eclipse is a software development issue. Based on analysis of the different existing Eclipse plug-ins to detect code smells, the same template can be used to create the plug-in to incorporate SPN models. The tools all follow the extensive and well-documented template[1] to create Eclipse plug-ins. The next step could be porting the python code from the SPFlow library to Java.

Based on the answers to the research questions above, we answer the main research question as follows:
**RQ: To what extent do SPNs improve the detection of code smells compared with state-of-the-art detection methods, and can this be applied in practice?**
The overall results of this study show that SPNs do have potential regarding code smell detection, however, more research is needed. The SPN models were "created" from the training data, hence no trial-and-error approaches were needed to create the different models. However, the experiments lead to mixed results when compared to the DL approach in [26]. Using the word embeddings, *word2vec*, also did not lead to observable improvements. Hence, more research is also needed with a more extensive and balanced dataset for the different code smells. Though, with the SPN approach it was possible to discern which fea-

[1]https://www.eclipse.org/pde/

tures, in this case code metrics, carry the most weight when detecting a code smell. Thus having the potential in applying SPNs in a development environment.

# 10 Future Work

This study leads us to several directions for future work, which will be outlined below.

The first, and one of the most important topics, is the procurement of an extensive and balanced dataset for training, validation, and testing. This is imperative for getting valid results about the accuracy, and robustness, of the SPN models in question. The datasets used for this study were the same datasets used in [26]. However, the datasets to train and test the models to detect code smells Feature Envy and Large Class were heavily imbalanced. In the case of code smell Large Class the datasets were too small. Therefore, not enough data is available to garner support to make statements about the efficacy of the SPN models. Hence, as part of future work we recommend setting up a repository for ML experiments with regard to code smell detection. This could be a dedicated repository on Github, or a dedicated repository hosted by the Open University[1]. An important aspect in this regard is that a format should be specified in such a way that a minimal amount of pre-processing is needed. This should be a continuous process of addition and evaluation so that the repository can grow over time. A mechanism should also be developed to create code smells in the datasets, in which the code smells are balanced over subsets resembling "real data" as much as possible. Besides procuring data, gathering classification features (e.g., code metrics) as much as possible is also part of this process. How this should be organized, and more importantly, maintained is an interesting follow-up challenge.

Another topic for future work is more research regarding the use of word embeddings or alternative embedding techniques. *Word2vec* was originally developed with NLP in mind, however, a more code-minded technique was developed, namely *code2vec* [2]. This approach parses a code entity first into an Abstract Syntax Tree (AST). Then it extracts syntactic paths between all the leaf nodes traversing through their lowest common ancestor [2]. Each path is represented as a sequence of intermediate AST nodes between two leaf nodes. With *code2vec*, the code entity is then also converted into a numerical presentation. Recent research has shown that using *code2vec* regarding code smell detection has a lot of potential [19][22]. Using *code2vec* to extend SPN models in detecting code smells is therefore also an interesting follow-up endeavor. However, in [26] *word2vec* showed promising results detecting code smells. Hence, it might also be worthwhile to follow up on *word2vec*

[1]https://research.ou.nl/en/datasets/

using a different setup or strategy.

The third topic for future work is to build an eclipse plug-in to detect code smells during the software development process in Java. As there is no such tooling available, it is important to build such a tool to evaluate the performance and usage in practice. Another issue is whether to use different SPNs for every code smell, or one SPN model to detect different code smells[2]. Therefore, evaluating code smell detection using SPNs in a development environment is a necessary follow-up project.

A final topic for future work is extending code smell detection by predicting the correct refactor strategy. This is because code smells detection and refactoring are connected. In [15], a list of 22 code smells was defined, and for each code smell several refactor strategies were also defined. While code smells represent design flaws in the software, refactoring is the process that restructures and transforms the software in such a way that it adheres to certain quality standards to solve those flaws. In other words, code smells tell what the problems are, and refactoring can then be used to correct such problems. Integrating these two processes would provide the complete process of locating the design flaws and improving software design. In [22], an ML approach to predict refactoring strategies to solve the code smell Feature Envy showed promising results. Hence it is important to investigate methods to predict refactor strategies based on code smells detected.

[2]https://blog.google/technology/ai/introducing-pathways-next-generation-ai-architecture/

# Bibliography

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4), 2018. 1, 16

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2Vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages*, 3:1–29, 2019. 56

[3] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, 2012. 33

[4] Piotr Bojanowski, Onur Celebi, Tomas Mikolov, Edouard Grave, and Armand Joulin. Updating pre-trained word vectors and text classifiers using monolingual alignment. *arXiv*, 2019. 52

[5] Paula Branco, Luis Torgo, and Rita Ribeiro. A survey of predictive modelling under imbalanced distributions, 2015. 22

[6] Sofia Charalampidou, Apostolos Ampatzoglou, and Paris Avgeriou. Size and cohesion metrics as indicators of the long method bad smell: An empirical study. *ACM International Conference Proceeding Series*, 2015-October, 2015. 27, v

[7] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. v

[8] Denis Deratani Mauá, Diarmaid Conaty, Fabio Gagliardi Cozman, Katja Poppenhaeger, and Cassio Polpo de Campos. Robustifying sum-product networks. *International Journal of Approximate Reasoning*, 101:163–180, 2018. 51

[9] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: Are we there yet? *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*, 2018-March:612–621, 2018. 2, 15, 18

[10] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. pages 1–12, 06 2016. 15

[11] Peter Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, USA, 2012. 2, 6, 7, 13, 15, 19, 22, 29

[12] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1037–1039, 2011. 20

[13] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2), 2012. 1, 15

[14] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. Anti-pattern and code smell false positives: Preliminary conceptualisation and classification. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016*, 2016-Januari(c):609–613, 2016. 2, 18

[15] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley object technology series. Addison-Wesley, 1999. iii, 1, 3, 4, 6, 15, 16, 57

[16] Robert Gens and Domingos Pedro. Learning the structure of sum-product networks. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 873–880, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. 12

[17] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019. 7, 33

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. 2

[19] Mouna Hadj-Kacem and Nadia Bouassida. Deep representation learning for code smells detection using variational auto-encoder. *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2019. 56

[20] Foutse Khomh, Stéephane Vaucher, Yann Gaël Guéehéeneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. *Proceedings - International Conference on Quality Software*, pages 305–314, 2009. 16

[21] Foutse Khomh, Stephane Vaucher, Yann Gaël Guéhéneuc, and Houari Sahraoui. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559–572, 2011. 16

[22] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. Recommendation of move method refactoring using path-based representation of code. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 315–322, 2020. 2, 56, 57

[23] M. Lanza, S. Ducasse, and R. Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Berlin Heidelberg, 2007. 4, 17, 51, v, vi

[24] W. Li and S. Henry. Maintenance metrics for the object oriented paradigm. In *[1993] Proceedings First International Software Metrics Symposium*, pages 52–60, 1993. v

[25] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. (February), 2018. 16

[26] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang. Deep learning based code smell detection. *IEEE Transactions on Software Engineering*, pages 1–1, 2019. iii, 2, 16, 18, 19, 20, 21, 23, 24, 25, 26, 27, 29, 32, 33, 34, 35, 37, 44, 51, 53, 54, 56, xi

[27] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, 1 edition, 2008. iii, 1, 3, 6

[28] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, pages 1–12, 2013. 13, 14, 17

[29] N. Moha, Y. Gueheneuc, L. Duchien, and A. Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010. 20

[30] Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks, 2019. 28

[31] Andrew Y. Ng and Michael I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In Thomas G. Dietterich, Suzanna Becker, and Zoubin Ghahramani, editors, *NIPS*, pages 841–848. MIT Press, 2001. 7

[32] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188–1221, 2018. 3

[33] Iago París, Raquel Sánchez-Cauce, and Francisco Javier Díez. Sum-product networks: A survey. 2020. 2, 9, 10, 11, 12, 13, 28

[34] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. Comparing heuristic and machine learning approaches for metric-based code smell detection. *IEEE International Conference on Program Comprehension*, 2019-May:93–104, 2019. 1, 37, 51

[35] Fabiano Pecorelli, Fabio Palomba, Foutse Khomh, and Andrea De Lucia. Developer-driven code smell prioritization. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 220–231, New York, NY, USA, 2020. Association for Computing Machinery. 1, 37, 51

[36] Fabiano Pecorelli, Fabio Palomba, Foutse Khomh, and Andrea De Lucia. Developer-driven code smell prioritization. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 220–231, New York, NY, USA, 2020. Association for Computing Machinery. 50

[37] Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf, and Pedro Domingos. On Theoretical Properties of Sum-Product Networks. In Guy Lebanon and S. V. N. Vishwanathan, editors, *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38 of *Proceedings of Machine Learning Research*, pages 744–752, San Diego, California, USA, 09–12 May 2015. PMLR. 11, 12

[38] David L. Poole and Alan K. Mackworth. *Artificial Intelligence: Foundations of Computational Agents.* Cambridge University Press, USA, 2nd edition, 2017. 2, 8

[39] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. *Proceedings of the IEEE International Conference on Computer Vision*, pages 689–690, 2011. 2, 9, 10, 19, 40

[40] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach.* Pearson, 4 edition, 2022. 7, 9, 10

[41] Jürgen Schmidhuber. Deep Learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. 8

[42] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017. 7

[43] H.Jacques Suermondt and Gregory F. Cooper. Probabilistic inference in multiply connected belief networks using loop cutsets. *International Journal of Approximate Reasoning*, 4(4):283–306, 1990. 9

[44] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009. 17, v

[45] M. Tufano, D. Poshyvanyk, F. Palomba, A. DeLucia, G. Bavota, R. Oliveto, and M. Penta. When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Transactions on Software Engineering*, 43(11), 2017. 3

[46] Antonio Vergari, Nicola Di Mauro, and Floriana Esposito. Visualizing and understanding sum-product networks. *Machine Learning*, 108(4):551–573, aug 2018. 10

[47] Santiago Vidal, Hernan Vazquez, J. Andres Diaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. Jspirit: a flexible tool for the analysis of code smells. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6, 2015. 17

[48] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, New York, NY, USA, 2014. Association for Computing Machinery. 17

[49] Jing Hao Xue and D. Michael Titterington. Comment on "on discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes". *Neural Processing Letters*, 28(3):169–187, 2008. 7

# APPENDIX A: CODE METRICS DEFINITIONS

Below a brief overview is given of the software code metrics, or code metrics, relevant for this study.

**Distance** This metric is two-fold. The first distance metric computes the relative distance of a method to its enclosed class. The second distance metric computes the relative distance of a method to a target class [44].

**Size or Lines of Code (LOC)** This metric defines the number of lines of code of of a code entity, including blank lines [23].

**LCOM - Lack of Cohesion Of Methods** This metric defines a measure to define the difference between the number of method pairs not having fields in common and the number of method pairs having fields in common [24]. In [6], the formulas for LCOM and LCOM1, ..., LCOM4 are defined.

**COH - Cohesion** This metric measures the relationship of the different lines of code in a method. Thus a metric for cohesion is computed as defined in [6].

**CC - Class Cohesion** This metric measures the relationship of methods and fields within a class. In this case a cohesion metric at class-level is computed as defined in [6].

**NOAV - Number of Accessed Variables** This metric computes the total number of instance fields, of a class, that are accessed [23].

**MCN - McCabe's Cyclomatic Complexity** This metric computes "the number of linearly-independent paths through a method" [23]. This means that by adding a condition in an **if** statement for example, the complexity increases as the number of paths also increases.

**CD - Coupling Dispersion** This metric measures the number of classes/instances accessed from the method under investigation [23].

**ATFD - Access To Foreign Data** This metric computes the number of fields from unrelated class that are accessed from the class under invetigation [23].

**DCC - Direct Class Coupling** This metric computes the number of classes that are related to a class under investigation [7].

**DIT - Depth Inheritance Tree** This metric computes the maximum steps from the class under investigation to its parent class at the root [7].

**TCC - Tight Class Cohesion** This metric computes the relative number of method pairs of a class that access the same class-level fields of the class under investigation [23].

**WMC - Weighted Method Count** This metric computes the sum of the statistical complexity of all methods of a class under investigation [23].

**NOPA - Number of Public Attributes** This metric computes the number of public fields of a class under investigation [23].

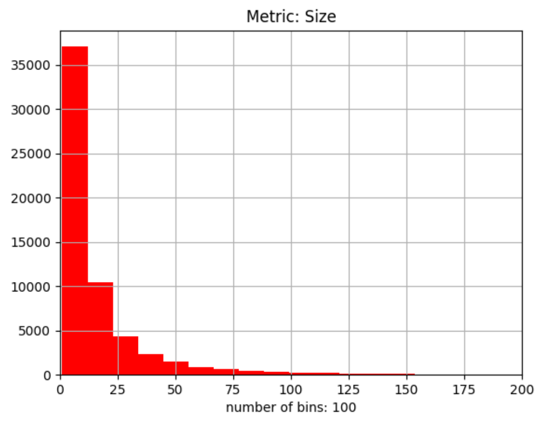
**NOAM - Number of Accessor Methods** This metric computes the number of accessor methods, i.e., **get** and **set** methods, of a class under investigation [23].

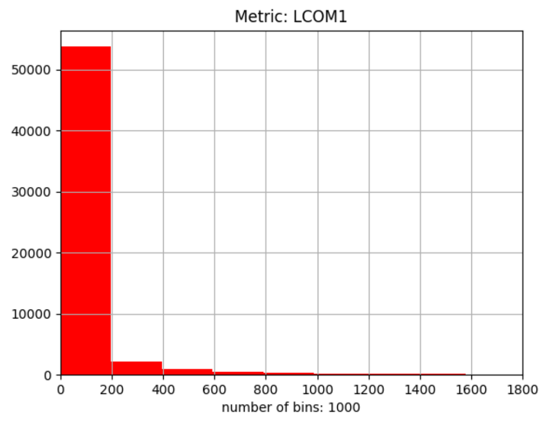
**NOA - Number of Attributes** This metric computes the number of all fields of a class under investigation [23].

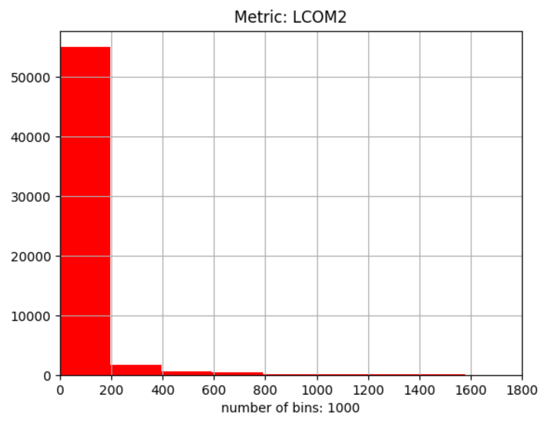
**NOM - Number of Methods** This metric computes the number of all methods of a class under investigation [23].

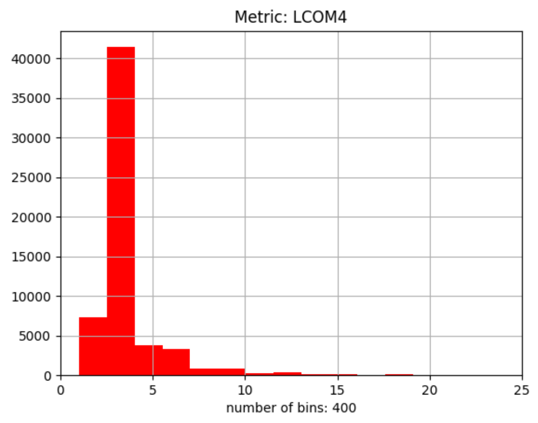
# Appendix B: Data distributions

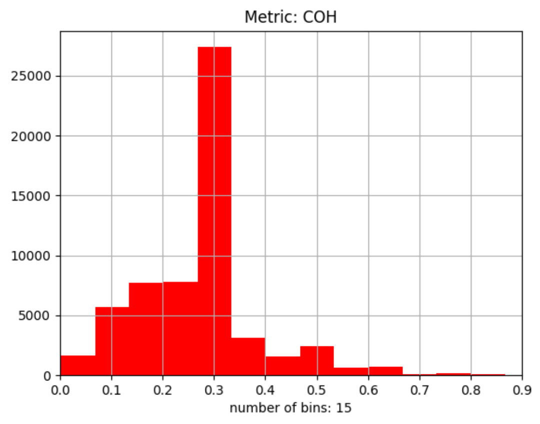
Below the distributions of the training data over various bin sizes are shown. This is visualized by histograms for the code metrics of the three different code smells: Long Method, Feature Envy, and Large Class. Firstly, the histograms for the code metrics regarding the code smell Long Method are shown:

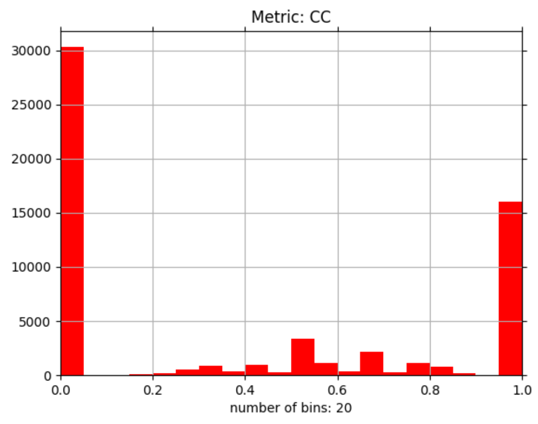












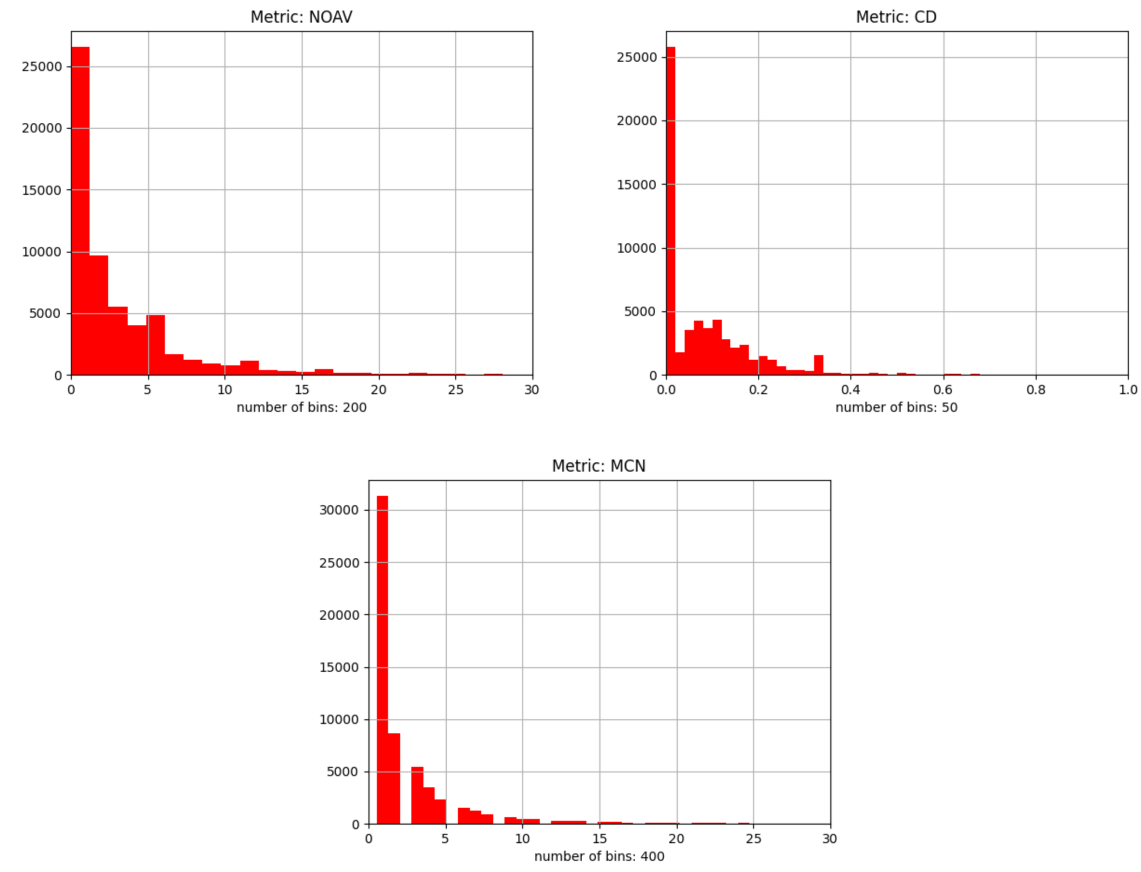


Figure 10.1: Long Method: Histograms showing the distributions of the various method level code metrics.

Secondly, the histograms for the code metrics regarding the code smell Feature Envy are shown below:

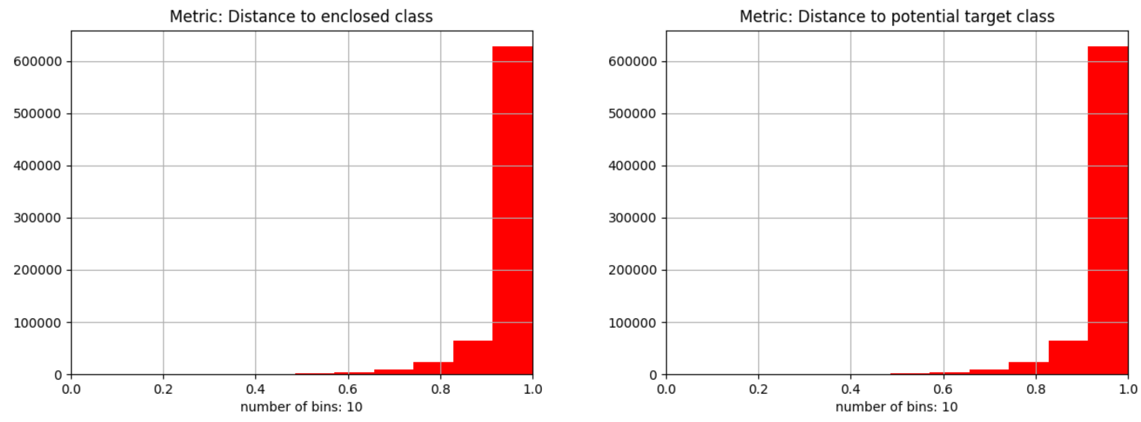
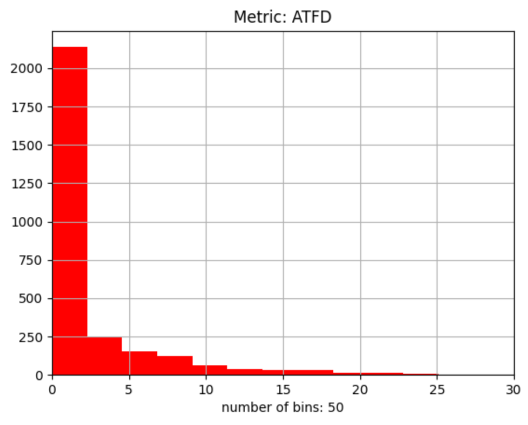
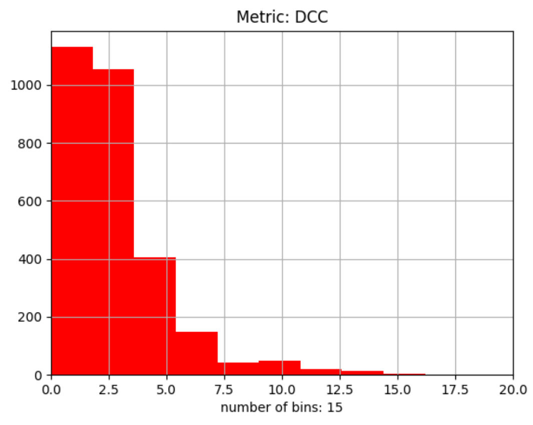
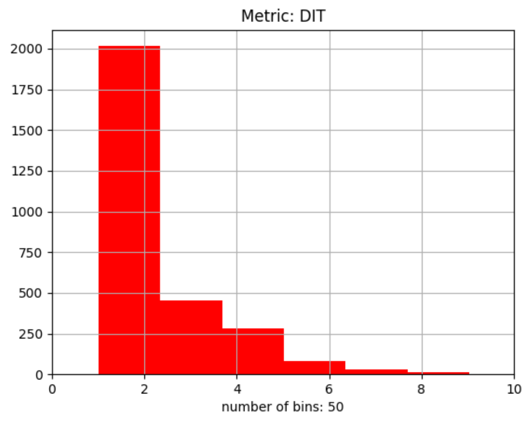


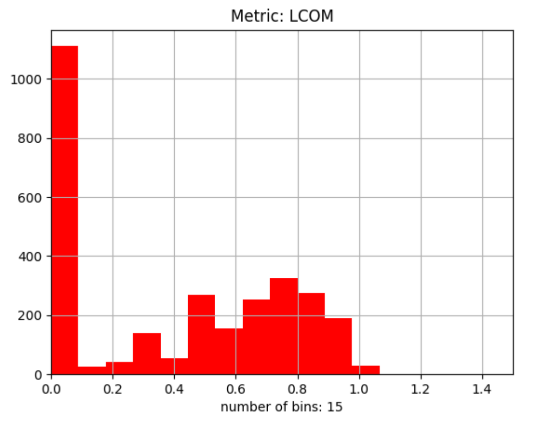
Figure 10.2: Feature Envy: Histograms showing the distributions of the two distance metrics.

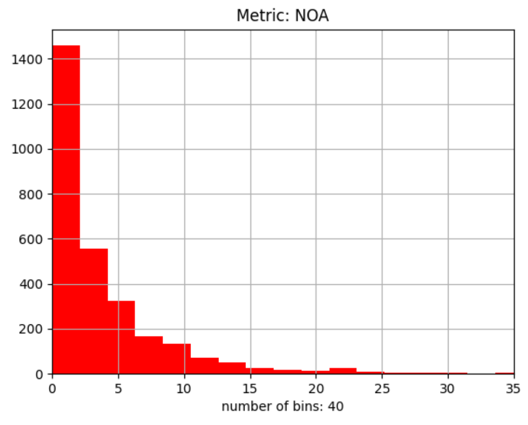
Finally, the histograms for the code metrics regarding the code smell Large Class are shown below:

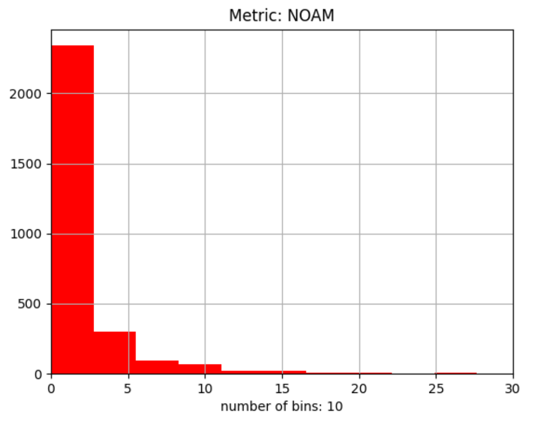

Metric: ATFD
number of bins: 50

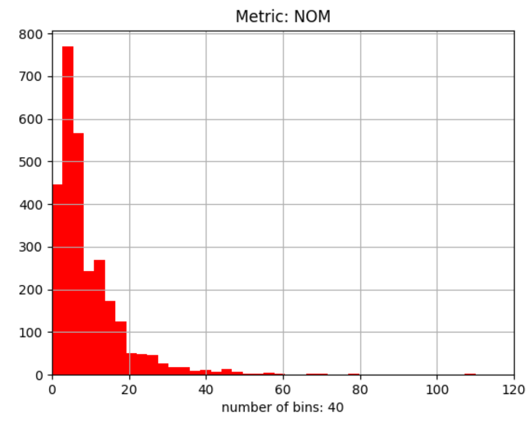

Metric: DCC
number of bins: 15

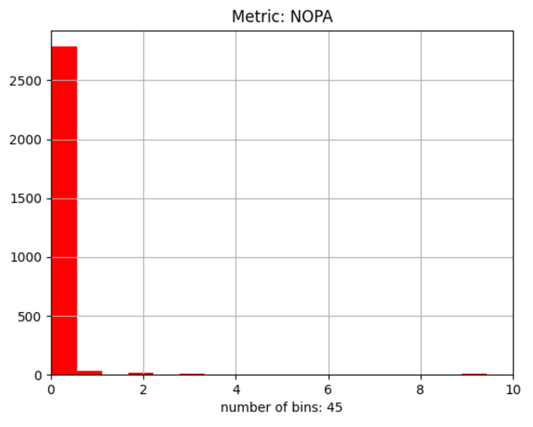

Metric: DIT
number of bins: 50

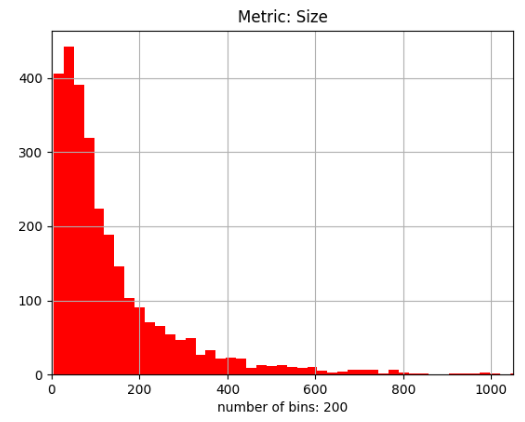

Metric: LCOM
number of bins: 15


Metric: NOA
number of bins: 40


Metric: NOAM
number of bins: 10


Metric: NOM
number of bins: 40


Metric: NOPA
number of bins: 45



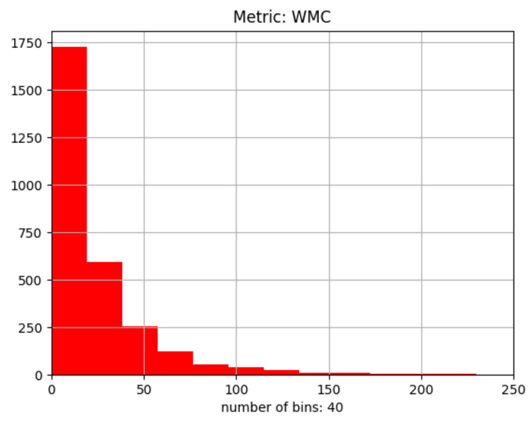


Figure 10.3: Large Class: Histograms showing the distributions of the various class level code metrics.

# APPENDIX C: RESULTS LONG METHOD EXPERIMENTS

In this Appendix, the results are shown of the experiments detecting the code smell Long Method using parametric and non-parametric SPN models. In Table 10.1, the results of the experiments to detect the code smell Long Method are shown. The SPN model is configured with a Gaussian distribution for both input and output. Figure 10.4 shows the precision vs recall curves. It is clear from Table 10.1 that the SPN model outperforms the DL approach specified in [26] based on the F1 metric. However, regarding the metric ROC AUC, the DL approach outperforms the SPN model for all projects.

Table 10.1: Results of the SPN model configured with a Gaussian distribution for both input and output.

| Applications | SPN models | | | | DL Approach NNs | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | F1 | AUC | Precision | Recall | F1 | AUC |
| Areca | 69.35% | 76.40% | 72.71% | 75.24% | 42.72% | 73.83% | 54.13% | 78.53% |
| Freeplane | 67.64% | 75.72% | 71.45% | 73.07% | 46.42% | 75.61% | 57.52% | 78.79% |
| jEdit | 64.18% | 84.84% | 73.08% | 73.24% | 52.17% | 83.45% | 64.20% | 77.15 |
| jUnit | 69.81% | 45.83% | 55.35% | 63.42% | 58.53% | 52.91% | 55.58% | 72.63% |
| PMD | 71.71% | 66.17% | 68.83% | 71.99% | 37.09% | 70.59% | 48.63% | 77.37% |
| Weka | 72.09% | 82.24% | 77.69% | 77.69% | 80.37% | 79.25% | 79.99% | 81.75% |
| Android | 73.37% | 63.19% | 67.90% | 78.88% | 32.34% | 80.63% | 46.16% | 78.81% |
| Grinder | 63.39% | 69.61% | 66.35% | 67.45% | 37.20% | 71.64% | 48.15% | 74.07% |
| Art of Illusion | 71.50% | 77.39% | 74.33% | 79.34% | 37.68% | 87.67% | 52.22% | 80.27% |
| jExcelAPI | 37.09% | 70.59% | 48.63% | 77.37% | 32.04% | 83.89% | 49.93% | 88.53% |
| **Average** | **66.03%** | **71.20%** | **67.63%** | **73.77%** | **42.81%** | **78.99%** | **55.53%** | **79.24%** |

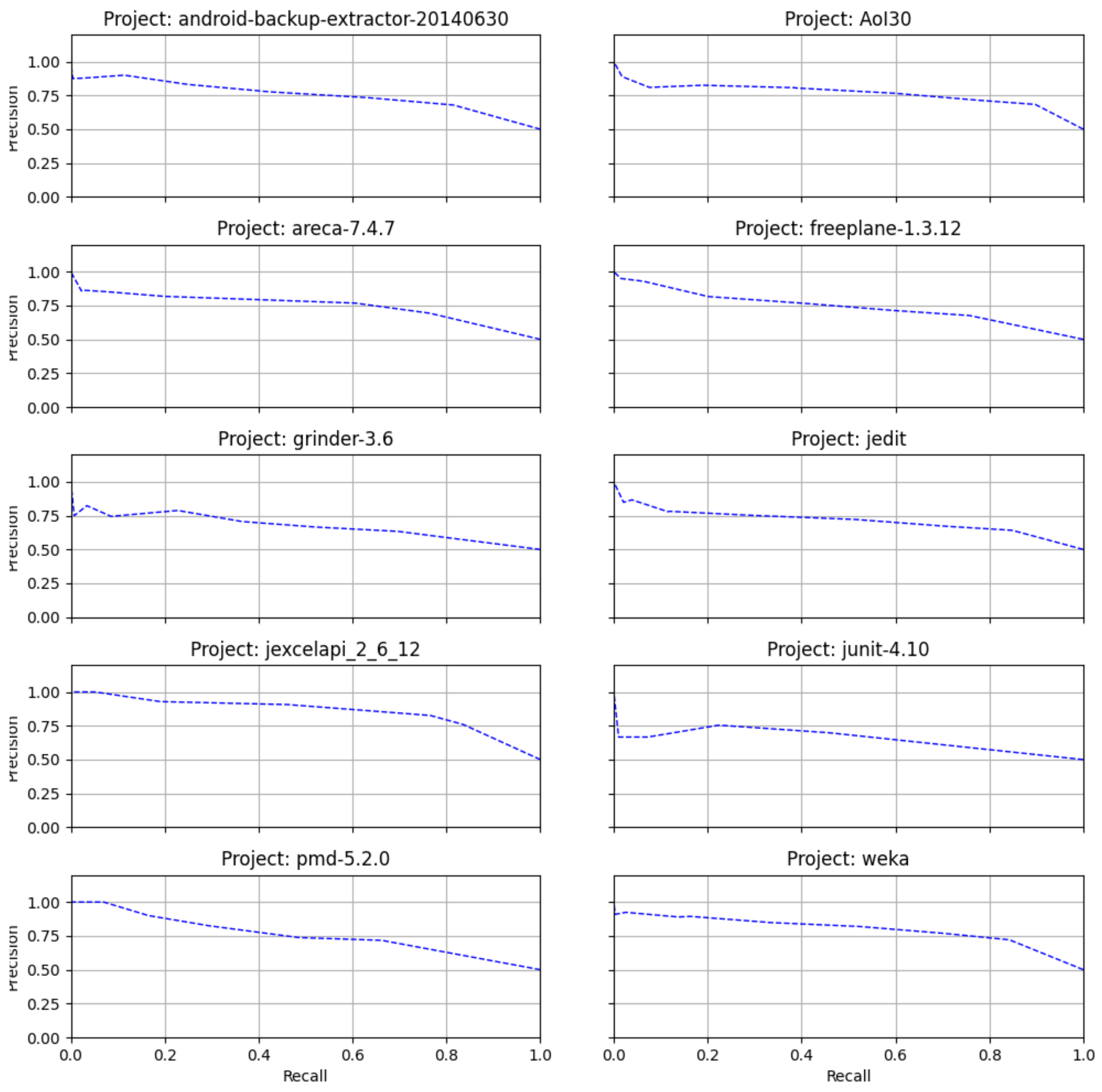


Figure 10.4: Performance scores of precision vs recall of the SPN model configured with a Gaussian distribution for both input and output.

In Table 10.2, the results are shown of the experiments to detect the code smell Long Method configured with a Gaussian distribution for input and a Bernoulli distribution for output. Figure 10.5 shows the precision vs recall curves. Except for project jUnit, the SPN model outperforms the DL approach regarding the F1 metric. However, for the metric ROC AUC, the DL approach outperforms the SPN model for all projects with a factor of 9%-800%.

Table 10.2: Results of the SPN model configured with a Gaussian distribution for input and a Bernoulli distribution for output.

| Applications | SPN models | | | | DL Approach NNs | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | F1 | AUC | Precision | Recall | F1 | AUC |
| Areca | 52.88% | 75.63% | 62.24% | 40.29% | 42.72% | 73.83% | 54.13% | 78.53% |
| Freeplane | 52.95% | 75.39% | 62.20% | 42.89% | 46.42% | 75.61% | 57.52% | 78.79% |
| jEdit | 64.18% | 84.84% | 73.08% | 73.24% | 52.17% | 83.45% | 64.20% | 77.15 |
| jUnit | 32.69% | 44.79% | 37.80% | 8.31% | 58.53% | 52.91% | 55.58% | 72.63% |
| PMD | 44.05% | 68.01% | 53.47% | 30.81% | 37.09% | 70.59% | 48.63% | 77.37% |
| Weka | 52.58% | 84.27% | 64.76% | 51.37% | 80.37% | 79.25% | 79.99% | 81.75% |
| Android | 55.92% | 83.17% | 68.98% | 53.99% | 32.34% | 80.63% | 46.16% | 78.81% |
| Grinder | 52.82% | 68.88% | 59.78% | 57.23% | 37.20% | 71.64% | 48.15% | 74.07% |
| Art of Illusion | 59.65% | 79.18% | 68.04% | 63.58% | 37.68% | 87.67% | 52.22% | 80.27% |
| jExcelAPI | 52.64% | 80.22% | 63.57% | 60.44% | 32.04% | 83.89% | 49.93% | 88.53% |
| **Average** | **52.04%** | **66.42%** | **61.39%** | **48.21%** | **42.81%** | **78.99%** | **55.53%** | **79.24%** |

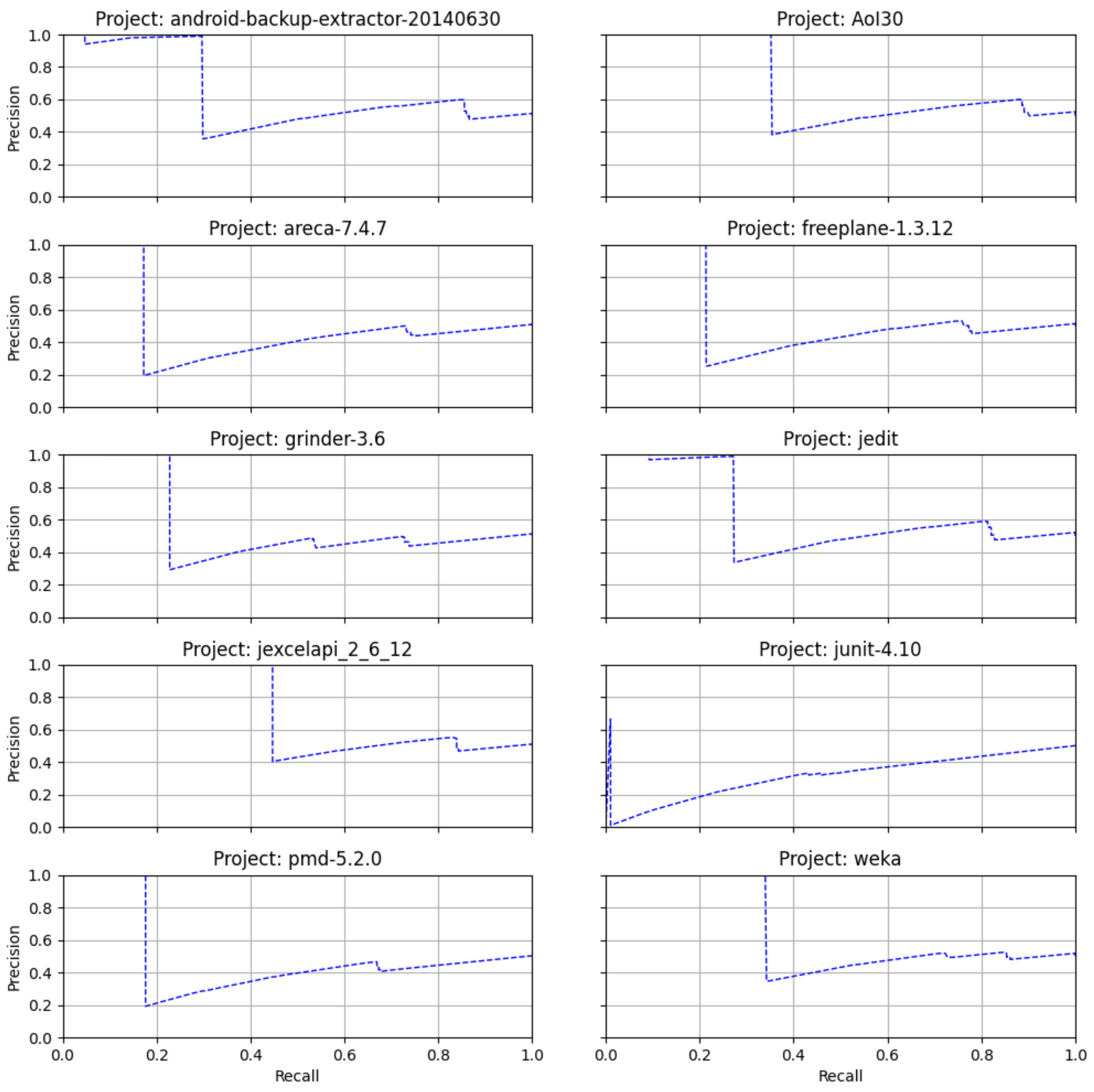


Figure 10.5: Performance scores of precision vs recall of the SPN model configured with a Gaussian distribution for input and Bernoulli distribution for output.

In Table 10.6, the results are shown of the experiments to detect the code smell Long Method configured with a Gamma distribution for input and Gaussian distribution for output. Figure 10.6 shows the precision vs recall curves. What stands out is that the SPN models outperform the DL approach at a much larger margin than vice-versa concerning the ROC AUC metric. Regarding the ROC AUC metric, the DL approach outperforms the SPN models only for the project jUnit with a factor of more than 10%.

Table 10.3: Results of the SPN model configured with a Gamma distribution for input and a Gaussian distribution for output.

| Applications | SPN models | | | | DL Approach NNs | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | F1 | AUC | Precision | Recall | F1 | AUC |
| Areca | 70.37% | 78.05% | 74.01% | 76.37% | 42.72% | 73.83% | 54.13% | 78.53% |
| Freeplane | 67.06% | 77.15% | 71.75% | 73.56% | 46.42% | 75.61% | 57.52% | 78.79% |
| jEdit | 63.47% | 82.67% | 71.81% | 71.74% | 52.17% | 83.45% | 64.20% | 77.15 |
| jUnit | 68.25% | 43.29% | 54.09% | 62.42% | 58.53% | 52.91% | 55.58% | 72.63% |
| PMD | 72.51% | 66.91% | 69.59% | 73.79% | 37.09% | 70.59% | 48.63% | 77.37% |
| Weka | 71.52% | 85.57% | 76.27% | 77.92% | 80.75% | 79.25% | 79.99% | 81.75% |
| Android | 65.02% | 85.17% | 73.74% | 76.74% | 32.34% | 80.63% | 46.16% | 78.81% |
| Grinder | 62.75% | 68.14% | 65.33% | 66.42% | 37.20% | 71.64% | 48.15% | 74.07% |
| Art of Illusion | 68.10% | 77.03% | 73.22% | 76.33% | 37.68% | 87.67% | 52.22% | 80.27% |
| jExcelAPI | 71.34% | 83.88% | 77.10% | 81.45% | 32.04% | 83.89% | 49.93% | 88.53% |
| **Average** | **68.04%** | **74.77%** | **70.69%** | **73.67%** | **42.81%** | **78.99%** | **55.53%** | **79.24%** |

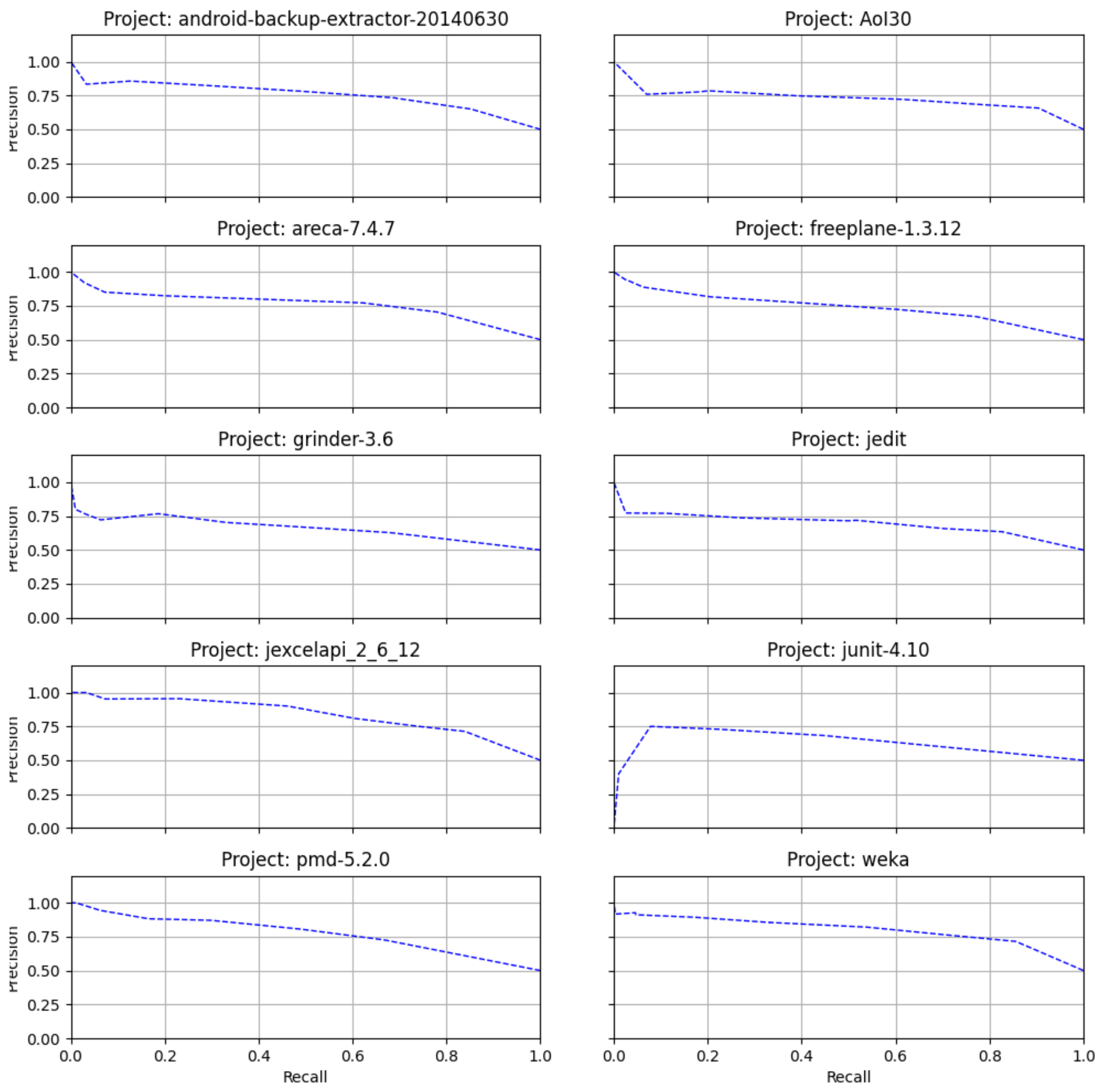


Figure 10.6: Performance results of precision vs recall of the SPN model configured with a Gamma distribution for input and Gaussian distribution for output.

In Table 10.4, the results are shown of the experiments to detect the code smell Long Method configured with a Gamma distribution for input and Bernoulli distribution for output. Figure 10.7 shows the precision vs recall curves. From Table 10.4 it is clear that the SPN model outperforms the DL approach by a large margin, up to 90%, regarding the F1 metric. Regarding the ROC AUC metric, the DL approach outperforms the SPN model at a much smaller margin, except for project PMD. For the project PMD the DL approach outperforms the SPN model with a factor of more 40%.

Table 10.4: Results of the SPN model configured with a Gamma distribution for input and a Bernoulli distribution for output.

| Applications | SPN models | | | | DL Approach NNs | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | F1 | AUC | Precision | Recall | F1 | AUC |
| Areca | 46.93% | 78.68% | 60.55% | 41.43% | 42.72% | 73.83% | 54.13% | 78.53% |
| Freeplane | 67.06% | 82.32% | 73.91% | 70.94% | 46.42% | 75.61% | 57.52% | 78.79% |
| jEdit | 65.58% | 82.97% | 73.61% | 69.72% | 52.17% | 83.45% | 64.20% | 77.15 |
| jUnit | 62.26% | 83.33% | 71.27% | 66.41% | 58.53% | 52.91% | 55.58% | 72.63% |
| PMD | 51.82% | 91.54% | 65.35% | 51.47% | 37.09% | 70.59% | 48.63% | 77.37% |
| Weka | 76.32% | 71.30% | 73.72% | 74.59% | 80.75% | 79.25% | 79.99% | 81.75% |
| Android | 68.70% | 76.94% | 72.59% | 70.94% | 32.34% | 80.63% | 46.16% | 78.81% |
| Grinder | 52.00% | 94.85% | 65.48% | 50.00% | 37.20% | 71.64% | 48.15% | 74.07% |
| Art of Illusion | 61.62% | 95.66% | 74.96% | 68.04% | 37.68% | 87.67% | 52.22% | 80.27% |
| jExcelAPI | 76.53% | 82.42% | 79.37% | 78.57% | 32.04% | 83.89% | 49.93% | 88.53% |
| **Average** | **62.88%** | **84.01%** | **71.08%** | **64.21%** | **42.81%** | **78.99%** | **55.53%** | **79.24%** |

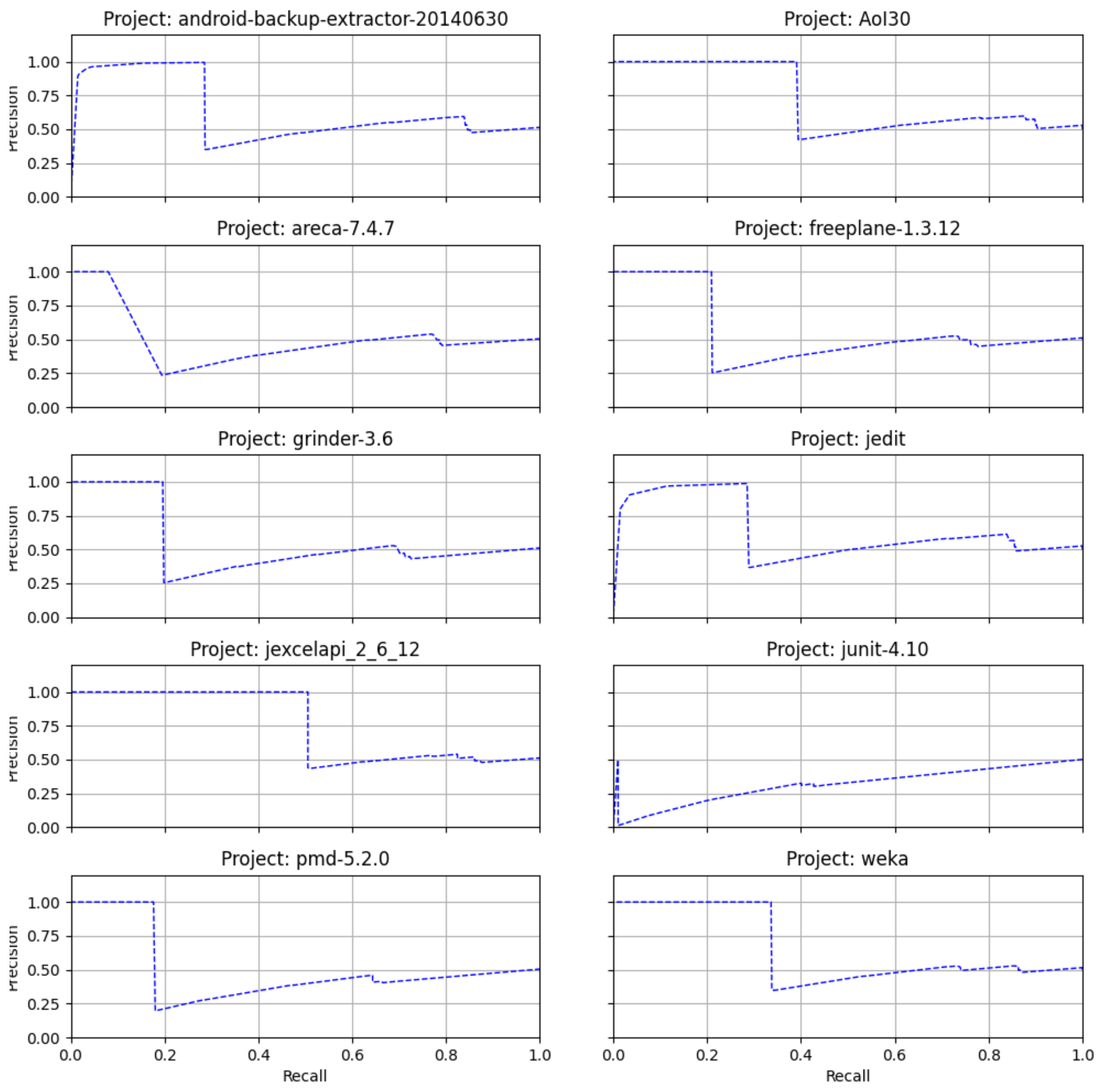


Figure 10.7: Performance scores of precision vs recall of the SPN model configured with a Gamma distribution for input and Bernoulli distribution for output.

In Table 10.5 the results are shown of the experiments to detect the code Long Method configured with a distribution for input and output learned from the data. This is a non-parametric model or mixed model in SPFlow. Figure 10.8 shows the precision vs recall curves. From Table 10.5 it is clear that the SPN model outperforms the DL approach by a large margin regarding the F1 metric, except for the projects jUnit and Weka. In the case of project jUnit the SPN model under-performs the DL approach by more than 35%. Regarding the ROC AUC metric, the SPN model also outperforms the DL approach for almost all projects, except for the projects jUnit and Weka.

Table 10.5: Results of a non-parametric SPN model.

| Applications | SPN models | | | | DL Approach NNs | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | F1 | AUC | Precision | Recall | F1 | AUC |
| Areca | 97.08% | 97.34% | 97.21% | 97.02% | 42.72% | 73.83% | 54.13% | 78.53% |
| Freeplane | 93.97% | 90.52% | 92.21% | 97.40% | 46.42% | 75.61% | 57.52% | 78.79% |
| jEdit | 90.49% | 91.71% | 91.09% | 97.29% | 52.17% | 83.45% | 64.20% | 77.15 |
| jUnit | 36.44% | 46.88% | 41.00% | 9.70% | 58.53% | 52.91% | 55.58% | 72.63% |
| PMD | 92.19% | 91.18% | 91.68% | 88.80% | 37.09% | 70.59% | 48.63% | 77.37% |
| Weka | 51.51% | 83.33% | 63.68% | 48.16% | 80.75% | 79.25% | 79.99% | 81.75% |
| Android | 90.09% | 95.76% | 92.87% | 95.61% | 32.34% | 80.63% | 46.16% | 78.81% |
| Grinder | 94.51% | 92.89% | 93.69% | 97.41% | 37.20% | 71.64% | 48.15% | 74.07% |
| Art of Illusion | 91.37% | 93.21% | 92.28% | 97.43% | 37.68% | 87.67% | 52.22% | 80.27% |
| jExcelAPI | 51.99% | 81.31% | 63.43% | 67.66% | 32.04% | 83.89% | 49.93% | 88.53% |
| **Average** | **78.96%** | **86.43%** | **81.91%** | **79.65%** | **42.81%** | **78.99%** | **55.53%** | **79.24%** |

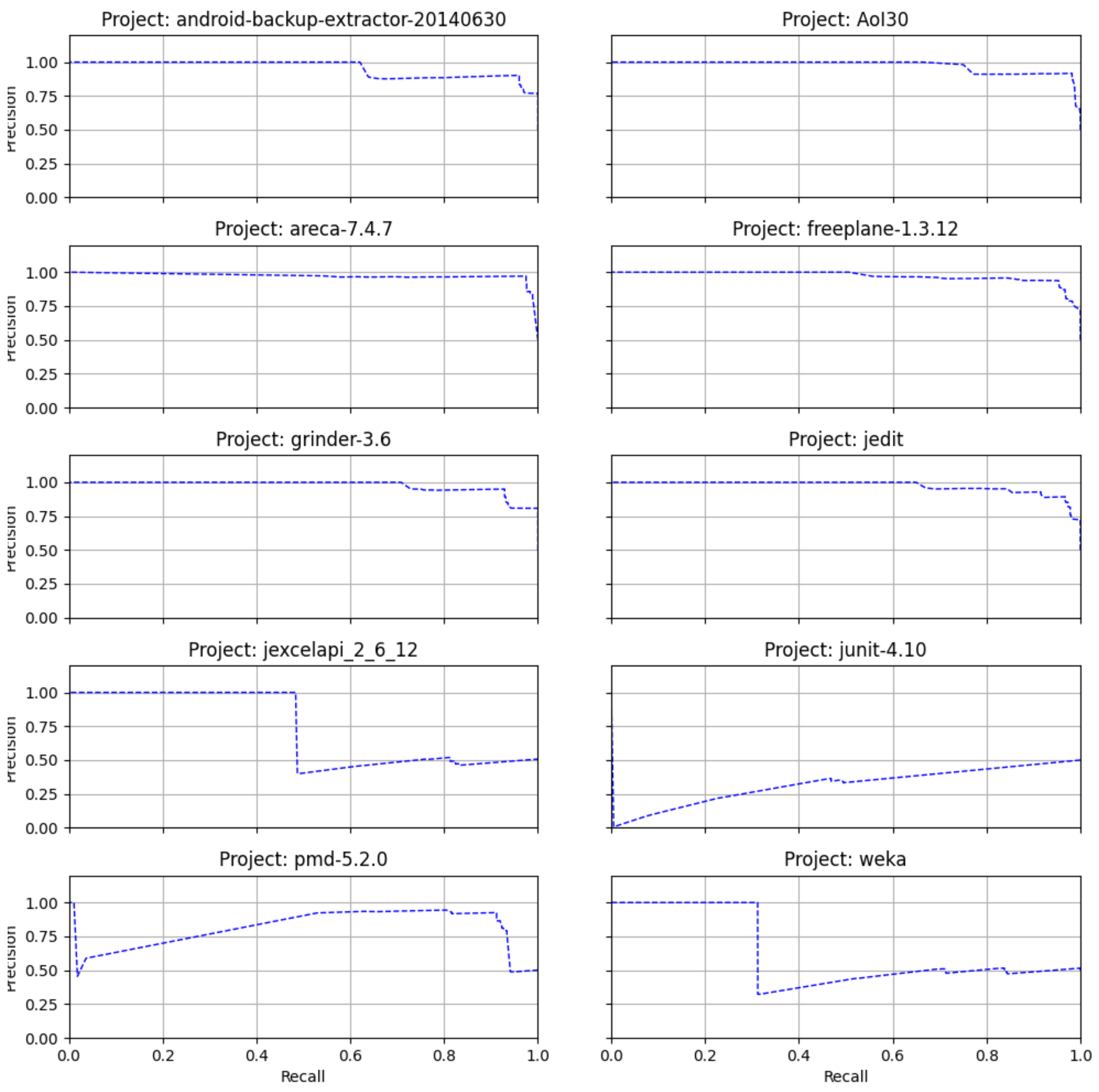


Figure 10.8: Performance scores of precision vs recall of a non-parametric SPN model for every project.

# Appendix D: Results Feature Envy Experiments

In this Appendix, the results are shown of the experiments detecting the code smell Feature Envy using non-parametric SPN models. The first experiment incorporates word embeddings and distance metrics as classification features to detect the code smell. Therefore, in Table 10.6 the results are shown for the SPN model that is also trained with word embeddings produced by *word2vec*. Figure 10.9 shows the precision vs recall curves. From Table 10.6 it is clear that the SPN model under-performs the DL approach both regarding the F1 metric and the ROC AUC metric. The SPN model gives a recall of 100% and an ROC AUC of 50% or less.

Table 10.6: Results of the SPN model with word embeddings and distance metrics as classification features.

| Applications | SPN models | | | | DL Approach NNs | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | F1 | AUC | Precision | Recall | F1 | AUC |
| Areca | 9.33% | 100% | 17.08% | 50% | 50.00% | 87.96% | 63.76% | 91.93% |
| Freeplane | 11.86% | 100% | 21.20% | 50% | 36.24% | 94.14% | 52.33% | 83.10% |
| jEdit | 10% | 100% | 18.19% | 50% | 38.18% | 91.30% | 53.85% | 84.29 |
| jUnit | 17.50% | 100% | 29.79% | 50% | 50.00% | 82.22% | 62.18% | 85.72% |
| PMD | 15.52% | 100% | 26.87% | 50% | 37.37% | 86.05% | 52.11% | 84.90% |
| Weka | 8.34% | 100% | 15.40% | 50% | 38.24% | 87.00% | 53.13% | 78.75% |
| Android | 10.64% | 100% | 19.23% | 50% | 29.70% | 76.67% | 42.82% | 74.81% |
| Grinder | 8.78% | 100% | 16.15% | 50% | 31.20% | 78.64% | 46.15% | 85.21% |
| Art of Illusion | 38.99% | 100% | 7.51% | 50% | 36.68% | 96.97% | 53.22% | 93.63% |
| jExcelAPI | 3.29% | 100% | 6.37% | 50% | 34.04% | 88.89% | 49.23% | 89.98% |
| **Average** | **13.43%** | **100%** | **17.78%** | **50%** | **36.79%** | **88.11%** | **51.91%** | **84.90%** |

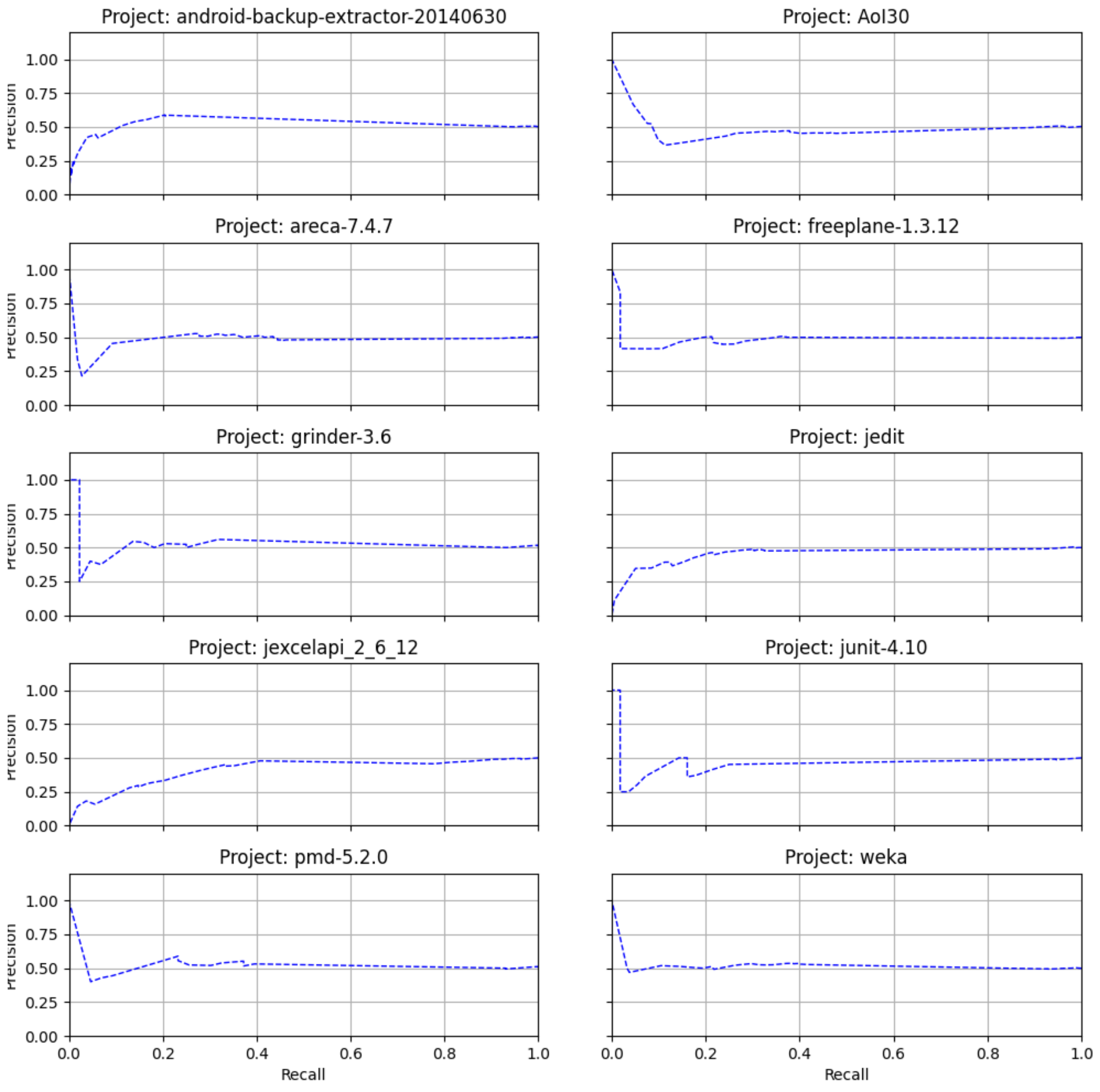


Figure 10.9: Performance scores of precision vs recall of the SPN model to detect code smells Feature Envy.

Secondly, in Table 10.7 the results are shown of the experiments to detect the code smell Feature Envy with only distance metrics as classification features. Figure 10.10 shows the precision vs recall curves. From Table 10.7 it is clear that the SPN model outperforms the DL approach regarding the F1 metric. However, the DL approach outperforms the SPN model with regards to the ROC AUC metric for all projects. A point to note is that the ROC AUC for the SPN model hovers around 50%.

Table 10.7: Results after training the SPN model with only code metrics as classification features.

| Applications | SPN models | | | | DL Approach NNs | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | F1 | AUC | Precision | Recall | F1 | AUC |
| Areca | 49.26% | 92.59% | 64.31% | 48.79% | 50.00% | 87.96% | 63.76% | 91.93% |
| Freeplane | 49.45% | 96.75% | 65.45% | 50.02% | 36.24% | 94.14% | 52.33% | 83.10% |
| jEdit | 49.11% | 92.69% | 64.20% | 44.38% | 38.18% | 91.30% | 53.85% | 84.29 |
| jUnit | 49.07% | 94.64% | 64.63% | 44.89% | 50.00% | 82.22% | 62.18% | 85.72% |
| PMD | 50.00% | 93.02% | 65.04% | 53.24% | 37.37% | 86.05% | 52.11% | 84.90% |
| Weka | 49.46% | 92.00% | 64.34% | 53.53% | 38.24% | 87.00% | 53.13% | 78.75% |
| Android | 50.13% | 94.76% | 65.56% | 50.48% | 29.70% | 76.67% | 42.82% | 74.81% |
| Grinder | 50.60% | 93.18% | 65.08% | 53.63% | 31.20% | 78.64% | 46.15% | 85.21% |
| Art of Illusion | 49.58% | 89.39% | 63.78% | 50.52% | 36.68% | 96.97% | 53.22% | 93.63% |
| jExcelAPI | 45.65% | 77.77% | 57.52% | 41.23% | 34.04% | 88.89% | 49.23% | 89.98% |
| **Average** | **49.23%** | **91.68%** | **63.99%** | **49.09%** | **36.79%** | **88.11%** | **51.91%** | **84.90%** |

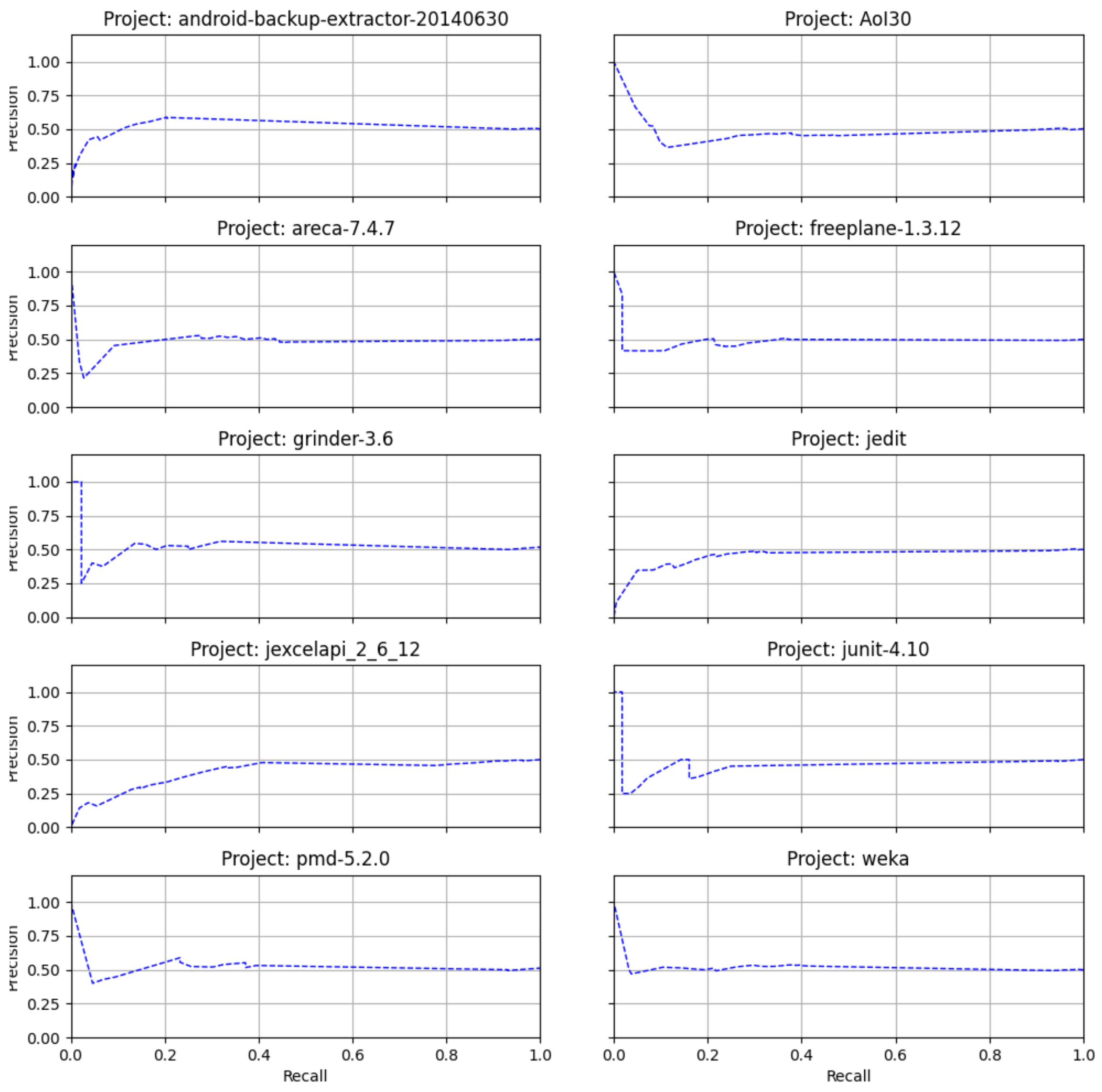


Figure 10.10: Performance scores of precision vs recall of the SPN model to detect code smells Feature Envy with only distance metrics as classification features.

# Appendix E: Results Large Class Experiments

In this Appendix, the results are shown of the experiments detecting the code smell Large Class using non-parametric SPN models. The first experiment incorporates word embeddings and code metrics as classification features to detect the code smell. Firstly, the detailed results are shown in Table 10.8. Figure 10.11 shows the precision vs recall curves. From Table 10.8, it is clear that the SPN model only produced results that do not seem skewed for the projects Freeplane, Android, and Grinder. For these projects, the SPN model outperforms the DL approach by large margins. However, without judging the quality of the results of the DL approach, it does seem that the DL approach has a better performance on average.

Table 10.8: Results after training the SPN model with word embeddings and code metrics as classification features.

| Applications | SPN models | | | | DL Approach NNs | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | F1 | AUC | Precision | Recall | F1 | AUC |
| Areca | 80.00% | 100% | 88.89% | 75.00% | 11.43% | 80.00% | 20.00% | 68.22% |
| Freeplane | 68.42% | 81.25% | 74.29% | 64.85% | 11.97% | 70.00% | 20.44% | 72.76% |
| jEdit | 100% | 0.00% | 100% | 0.00% | 15.00% | 75.00% | 25.00% | 77.39 |
| jUnit | 100% | 0.00% | 100% | 0.00% | 11.76% | 40.00% | 18.18% | 71.75% |
| PMD | 100% | 0.00% | 100% | 0.00% | 16.67% | 100% | 28.57% | 83.49% |
| Weka | 100% | 0.00% | 100% | 0.00% | 10.06% | 94.44% | 18.18% | 68.55% |
| Android | 80.65% | 92.89% | 86.21% | 73.11% | 16.46% | 79.41% | 27.27% | 79.75% |
| Grinder | 83.33% | 62.50% | 71.43% | 73.44% | 12.73% | 70.00% | 21.54% | 79.15% |
| Art of Illusion | 100% | 0.00% | 0.00% | 0.00% | 12.33% | 81.82% | 21.43% | 78.49% |
| jExcelAPI | 61.54% | 80.00% | 69.37% | 53.00% | 22.00% | 84.62% | 34.92% | 80.89% |
| **Average** | **87.39%** | **41.68%** | **79.02%** | **39.29%** | **12.95%** | **80.95%** | **22.33%** | **75.77%** |

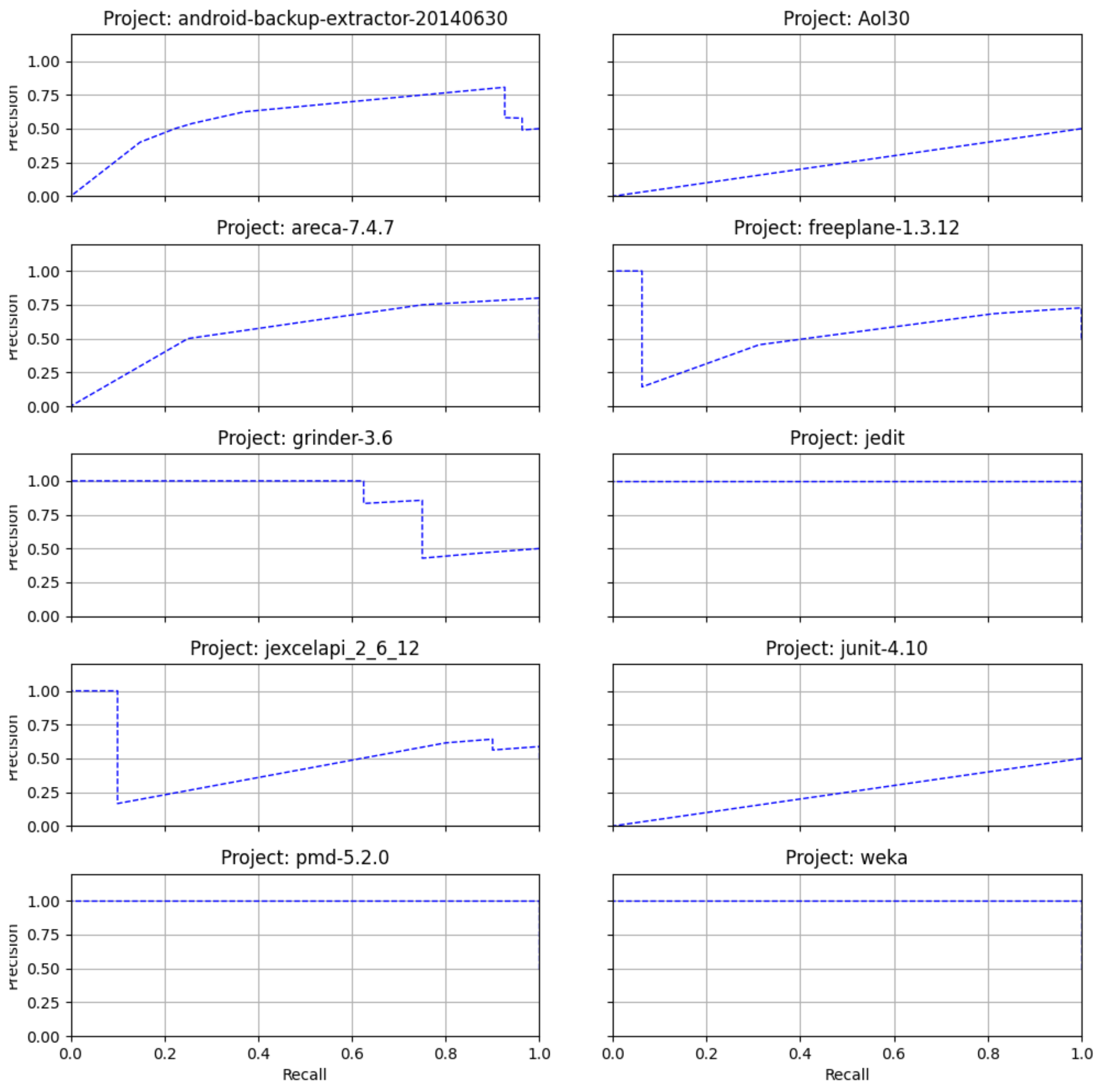


Figure 10.11: Performance scores of precision vs recall of the SPN model to detect code smells Large Class.

Secondly, in Table 10.9 the results are shown of the experiments to detect the code smell Large Class with only code metrics as classification features. Also, Figure 10.12 shows the precision vs recall curves. From Table 10.9 it is clear that the SPN model outperforms the DL approach based on the F1 score by a large margin. Regarding the ROC AUC metric, the DL approach outperforms the SPN model for all projects by a large margin. The ROC AUC for most projects is below 50%. Furthermore, the ROC AUC for jUnit is 0.00%.

Table 10.9: Results after training the SPN model with only code metrics as classification features.

| Applications | SPN models | | | | DL Approach NNs | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | F1 | AUC | Precision | Recall | F1 | AUC |
| Areca | 80.00% | 100% | 88.89% | 79.00% | 11.43% | 80.00% | 20.00% | 68.22% |
| Freeplane | 47.85% | 68.75% | 56.45% | 18.02% | 11.97% | 70.00% | 20.44% | 72.76% |
| jEdit | 55.55% | 83.39% | 66.67% | 77.78% | 15.00% | 75.00% | 25.00% | 77.39 |
| jUnit | 50.00% | 100% | 66.67% | 0.00% | 11.76% | 40.00% | 18.18% | 71.75% |
| PMD | 66.67% | 100% | 80.00% | 63.24% | 16.67% | 100% | 28.57% | 83.49% |
| Weka | 61.54% | 85.71% | 71.64% | 53.44% | 10.06% | 94.44% | 18.18% | 68.55% |
| Android | 65.71% | 85.19% | 74.19% | 56.48% | 16.46% | 79.41% | 27.27% | 79.75% |
| Grinder | 36.67% | 50.18% | 42.08% | 9.38% | 12.73% | 70.00% | 21.54% | 79.15% |
| Art of Illusion | 70.00% | 87.50% | 77.78% | 54.52% | 12.33% | 81.82% | 21.43% | 78.49% |
| jExcelAPI | 50.00% | 80.00% | 61.54% | 24.00% | 22.00% | 84.62% | 34.92% | 80.89% |
| **Average** | **59.55%** | **84.07%** | **69.37%** | **41.03%** | **12.95%** | **80.95%** | **22.33%** | **75.77%** |

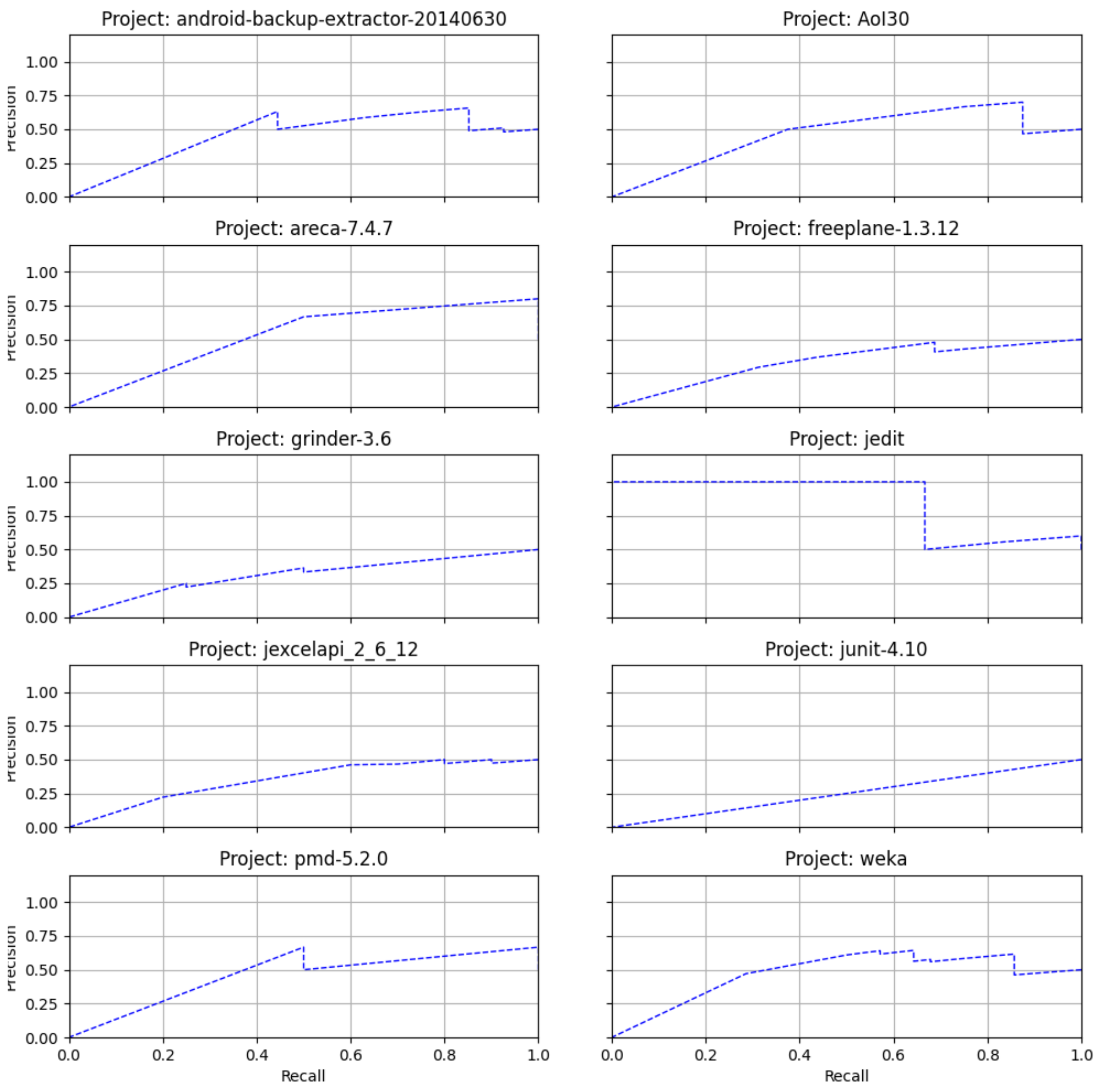
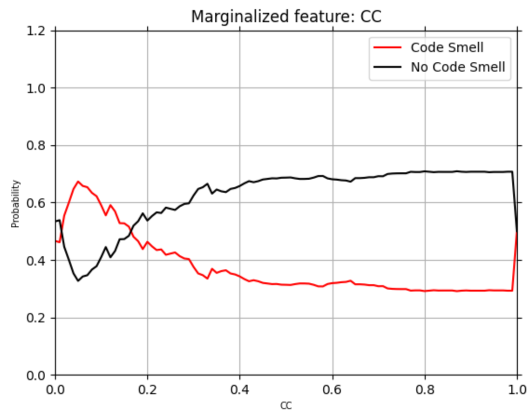
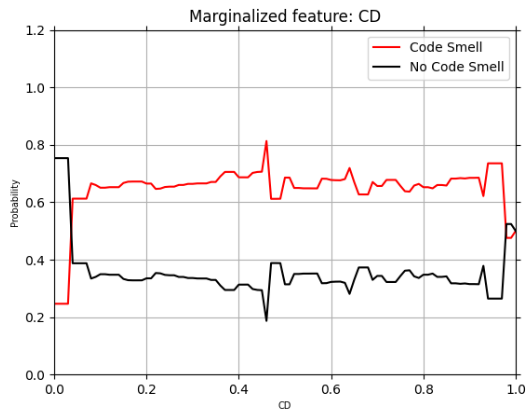


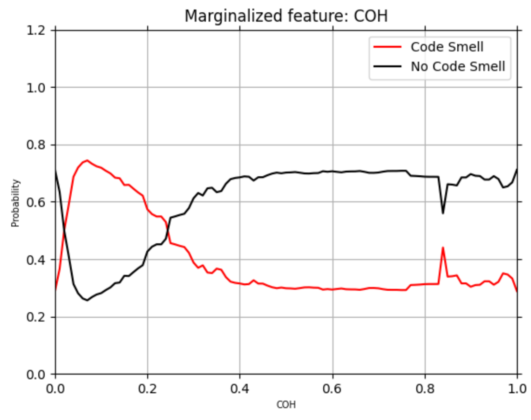
Figure 10.12: Performance scores of precision vs recall of the SPN model to detect code smell Large Class with only code metrics as classification features.

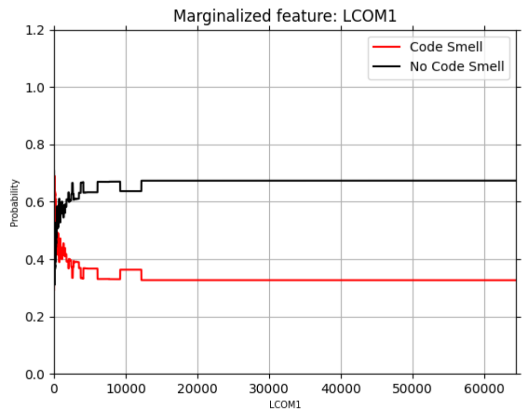
# Appendix F: Results Marginal Inference Experiments

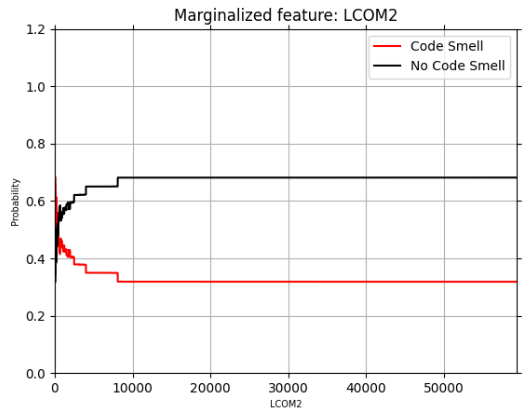
Below the significance is shown, marginal inference probabilities, of every code metric regarding the three code smells: Long Method, Feature Envy, and Large Class.

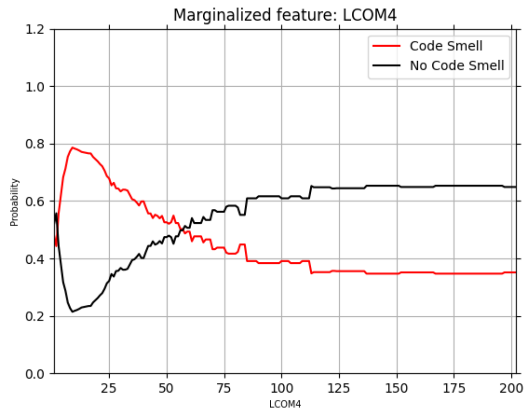












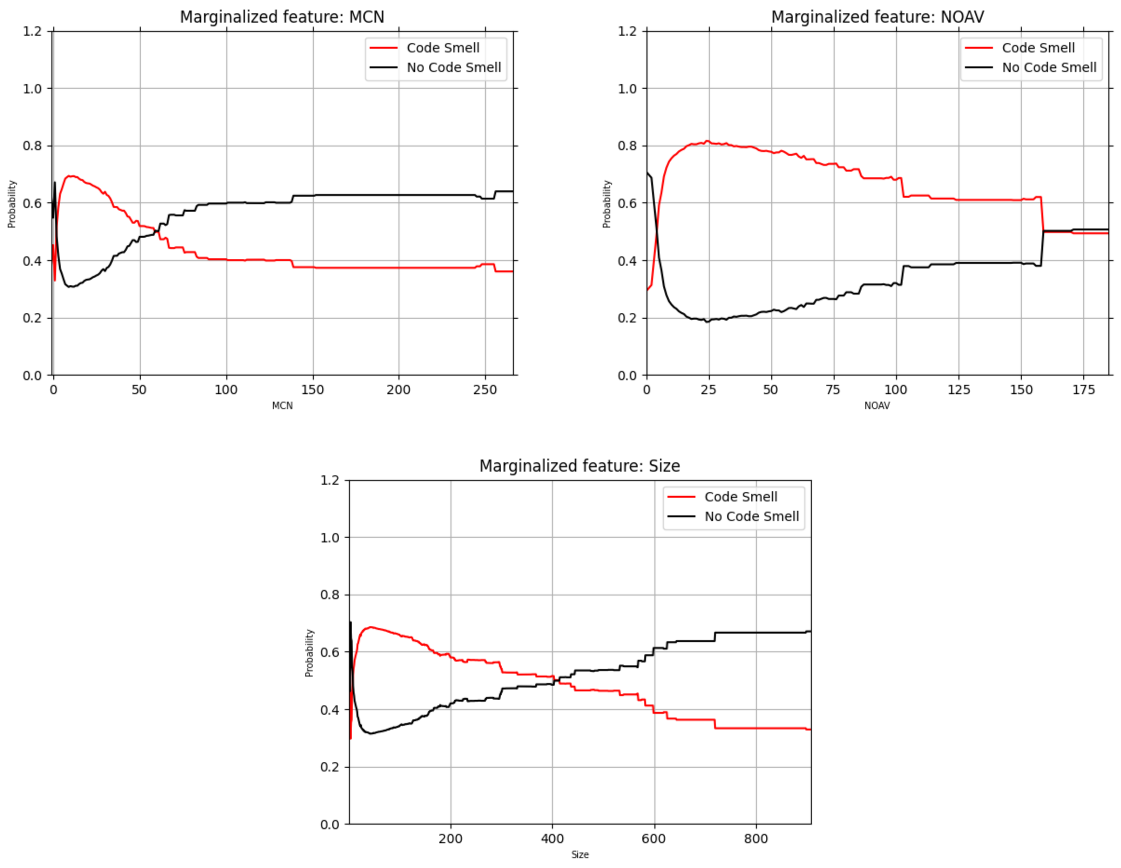


Figure 10.13: Marginal inference output per feature to show the influence on detection of code smell Long Method.

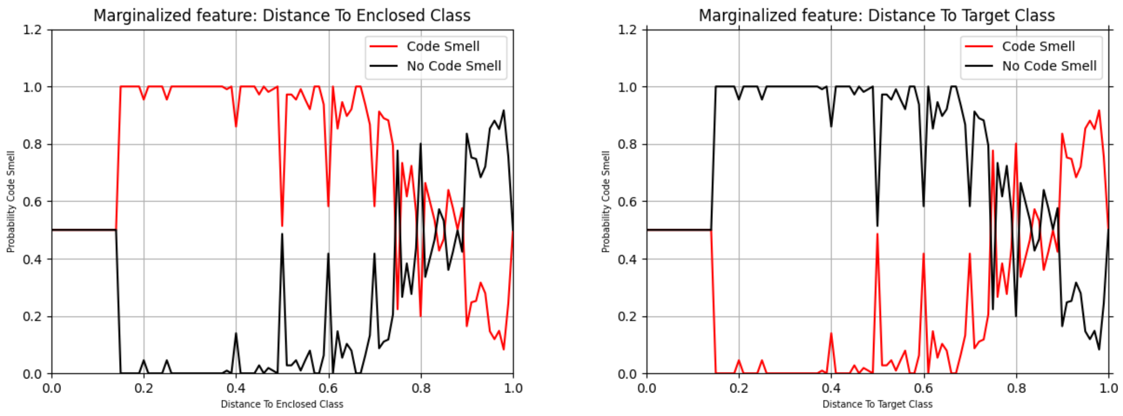
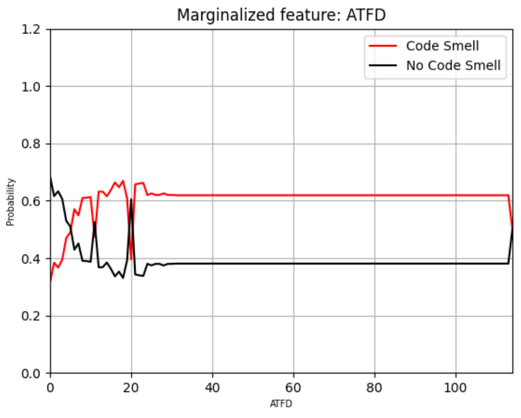
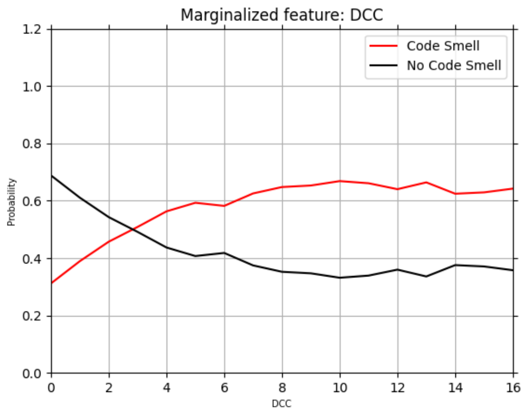
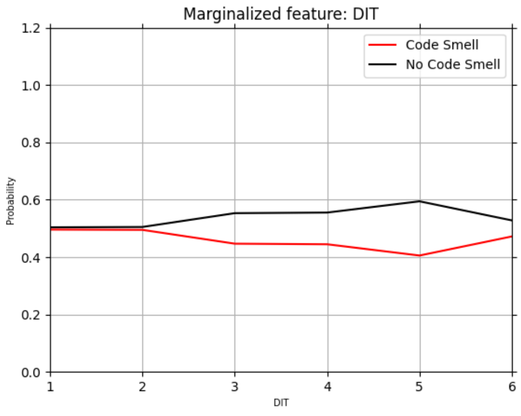
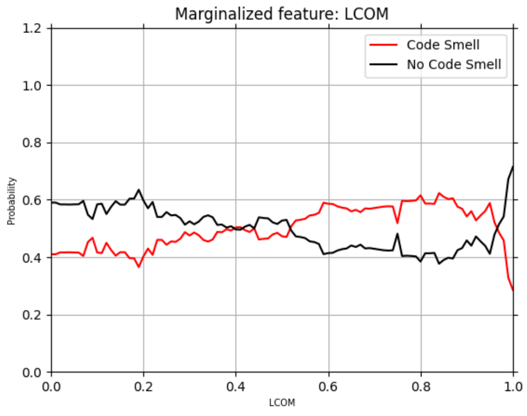
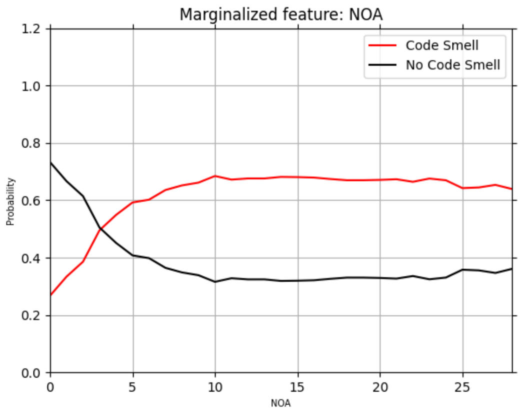
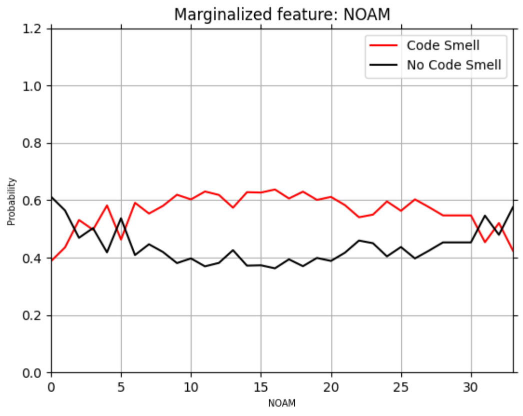
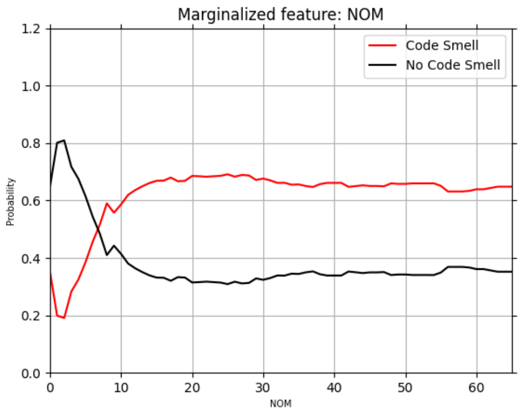
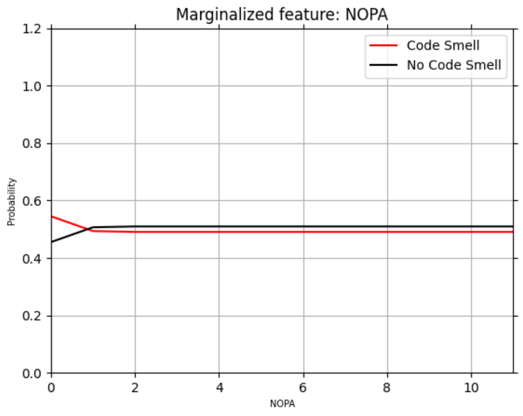


Figure 10.14: Marginal inference output per feature to show the influence on detection of code smell Feature Envy.


Marginalized feature: ATFD
Code Smell
No Code Smell
Probability
ATFD

Marginalized feature: DCC
Code Smell
No Code Smell
Probability
DCC

Marginalized feature: DIT
Code Smell
No Code Smell
Probability
DIT

Marginalized feature: LCOM
Code Smell
No Code Smell
Probability
LCOM

Marginalized feature: NOA
Code Smell
No Code Smell
Probability
NOA

Marginalized feature: NOAM
Code Smell
No Code Smell
Probability
NOAM

Marginalized feature: NOM
Code Smell
No Code Smell
Probability
NOM

Marginalized feature: NOPA
Code Smell
No Code Smell
Probability
NOPA

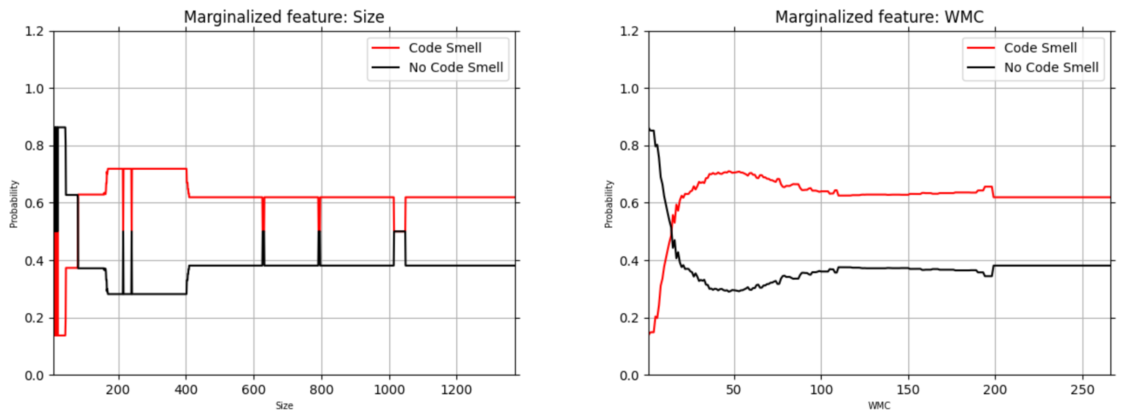


Figure 10.15: Marginal inference output per feature to show the influence on detection of code smell Large Class.