Object-oriented programming is a paradigm used by a significant number of developers in the design, development, and implementation of software systems. It facilitates the maintenance phase of developed applications. Unfortunately, this phase is increasingly jeopardized by developers introducing development defects that negatively impact software quality. Poor maintenance hinders system evolution, the ease of changes that software engineers could make, program understanding, and increases the tendency for errors. In summary, poor maintenance leads to the deterioration of software quality and reduces the lifespan of systems. These anomalies, which are not bugs or technically incorrect codes and do not immediately disrupt program operation, indicate weaknesses in design and can slow development or increase the risk of bugs or failures in the future. They are identifiable based on the taxonomy of detection approaches presented by Hadj-Kacem *et al*.from four sources of information: structural, semantic, behavioral, and historical. Structural defects essentially refer to code structure that violates object-oriented design principles, such as modularity, encapsulation, data abstraction, etc. Blob and Long Method are two structural defects that align with this assertion and are widespread in source code.
They are classified according to several abstractions which we group into two categories: traditional heuristic approaches and machine learning-based approaches. Due to the challenges of finding threshold values for metric identification, the lack of consistency between different identification techniques, the subjectivity of developers in defining defects, and the difficulty of manually constructing optimal heuristics, research has shifted towards machine learning. These models are mathematical techniques that use historical data to automatically identify complex patterns and make informed and intelligent decisions. However, this new paradigm is mainly built by learning models taken individually, putting aside the adage that there is strength in numbers and rarely takes into account the imbalance of data in the context of source codes.
Knowing that pre-trained models have the ability to extract nuanced features and contextual information, they could enable better discrimination between minority and majority classes in the context of imbalanced datasets.
What machine learning method is suitable for imbalanced datasets in the context of structural development defect detection?
Our main objective is therefore to build a model based on ensemble learning whose basic estimator is composed of a pre-trained model having the capacity to contribute to the balance of classes and a classifier.
Underlying questions arise based on the above objective.
RQ1: What is the optimal number of base estimators for a model?
RQ2: Do pre-trained models alleviate class imbalance?
RQ3: Is an ensemble learning method using a pre-trained model for vector representations incorporating a deep learning classifier the state-of-the-art for ensemble methods?
To address these concerns, we organize this article as follows: In the second section, we present a literature review on detection approaches and on ensemble learning methods. Section 3 presents our approach. In Section 4, we present and discuss our results, and finally, we conclude the article.

2. Related Works

In the field of structural development defect detection approaches, several classifications have been defined. In this work, we summarize the classification into two groups, as indicated by Yue *et al*: traditional heuristic approaches and those based on machine learning.

2.1. Traditional Heuristic Approaches

The process of heuristic approaches generally unfolds in two steps. A set of metrics associated or not with specific indicators on code instances is calculated, characterizing the considered defect. Then, thresholds are applied to these metrics. Following this framework, Peldszus *et al*. proposed a model that associates software metrics and various indicators of code smells to allow the system not to deteriorate as it evolves.

However, the subjectivity of code smell indicators can render detection tools unusable in certain contexts.
Chen *et al*. simultaneously implemented the Pysmell tool, whose detection strategy involved applying a set of metrics associated with parameterized thresholds to relevant code excerpts. Similarly, Hammad *et al*. designed a plugin named JFly (Java Fly) integrated into the Eclipse environment based on a set of rules characterizing the defects to be detected, including software metrics associated with thresholds. All these approaches use threshold values, posing the recurring problem of the subjectivity of optimal choice.
Traditional heuristic approaches are increasingly abandoned in defect detection methods due, among other reasons, to the subjectivity in defining threshold values, steering research towards machine learning.

2.2. Machine Learning-Based Approaches

Machine learning is a field of study in artificial intelligence that aims to give machines the ability to "learn" from data through mathematical models. Two methods of using machine learning algorithms represent the state of the art in research.

2.2.1. Individual Model Cases

Hamdy *et al*. experimented with two recurrent neural networks, LSTM (Long short term memory) and GRU (Gated recurrent unit), and a convolutional neural network CNN to detect blob. They concluded that neural networks outperform commonly used machine learning models like Naïve Bayes, Random Forests, and Decision Trees.
Although adding the lexical and syntactic features of the source code to the software metrics has provided a set of relevant information to the training data, these features do not take into account the semantics of the code.
Kacem *et al*. conducted a study using a hybrid method, coupling an unsupervised learning phase using a deep autoencoder whose purpose is to transform code snippets into vector representations of reduced dimensions and supervised learning (artificial neural network) to classify these codes based on their vector characterizations.
Sharma *et al*. compared three types of models: Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and autoencoders with hidden layers composed of dense neural networks (DNN), CNN, and RNN. The authors used the open-source Tokenizer tool to generate vector representations of code.
In these two aforementioned works, the process of vector representation of code snippets does not take into account the context of the code, which defines its meaning. To take into account the semantics of the source code, Kacem *et al*. designed an approach that generates vector representations from abstract syntax trees from code excerpts used as input parameters for a variational autoencoder (VAE) whose produce semantic information through learning Finally a logistic regression classifier is applied to this semantic information to determine whether a code snippet is a defect or not. The limitation of this work lies in the procedure for extracting representative vectors from code snippets, which involves several steps, potentially making the entire system more comple.