CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Evaluating Machine Learning Algorithms in Design Pattern Recognition

Exploring the Performance of Classification and Clustering Algorithms in Design Pattern Recognition Utilising Large Language Models

Master's thesis in Computer science and engineering

SIMON ANDERSSON
VIKTOR BERGGREN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Evaluating Machine Learning Algorithms in Design Pattern Recognition

Exploring the Performance of Classification and Clustering Algorithms in Design Pattern Recognition Utilising Large Language Models

SIMON ANDERSSON
VIKTOR BERGGREN

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Evaluating Machine Learning Algorithms in Design Pattern Recognition
Exploring the Performance of Classification and Clustering Algorithms in Design Pattern Recognition Utilising Large Language Models
SIMON ANDERSSON
VIKTOR BERGGREN

Supervisor: Jennifer Horkoff, Computer Science and Engineering
Examiner: Hans-Martin Heyn, Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2024

Evaluating Machine Learning Algorithms in Design Pattern Recognition
Exploring the Performance of Classification and Clustering Algorithms in Design Pattern Recognition Utilising Large Language Models

SIMON ANDERSSON
VIKTOR BERGGREN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Design Pattern Recognition (DPR) is an ongoing research challenge in the field of software engineering for increasing software maintainability in code. Recent work has utilised Large Language Models (LLMs) for extracting semantic information from code. This study follows up on previous research and investigates, explores, and evaluates the performance of multiple classification and clustering algorithms when applied to embeddings extracted from LLMs. Performance is explored between contexts using different LLMs, design patterns, and programming languages. Data for design pattern implementations was gathered for Java, Python, and C# via GitHub and the P-MARt repository. Each algorithm was run with tuned hyperparameters, and their average performance across multiple runs was compared. The results indicate variance for the individual performance of the algorithms, but the overall performance order between the algorithms remains the same. Classification algorithms outperformed clustering algorithms, and clustering algorithms had low performance in the measured metrics across all tests. The results also showed a difference in performance between behavioral, creational, and structural design patterns. This study shows further promise for the use of LLMs for DPR and recognises the need for larger studies utilising LLMs for DPR.

Keywords: Computer science, design patterns, machine learning, large language models, software engineering, design pattern recognition, DPR, LLM.

Acknowledgements

We want to express our gratitude to our supervisor, Jennifer Horkoff, and our examiner, Hans-Martin Heyn, for their support and guidance throughout the study. They provided us with thoughtful feedback and advice that proved helpful with the study. We also want to thank Sushant Kumar Pandey for his support of the method and the technical aspects of the study.

Simon Andersson, Viktor Berggren, Gothenburg, June 2024

Contents

List of Figures

List of Tables

1

Introduction

Software maintainability is an important quality attribute to consider when developing code, and it accounts for the majority of the cost of software development [1]. Therefore, keeping the software maintainable reduces the cost of software development. Design patterns provide developers with tools for addressing recurring problems in code [2]. The impact of these design patterns on software maintainability has been demonstrated to be positively correlated [3], [4]. Consequently, identifying the use of design patterns in large-scale codebases is essential for software maintainability.

Manual inspection of code to identify design patterns is challenging and time-consuming [5]. Design patterns can also have different variations, known as variants [6], in order to fit the context of the code, which further complicates the problem. Incorrect identification of a design pattern can lead to incorrect implementations, which can affect the functionality of the system [7].

There are, however, other methods for automatically identifying design patterns in code using design pattern recognition (DPR) [8]. There are several different areas explored in DPR, from metric-based approaches like [9], [10], to machine learning-based approaches like [7], [11]. These methods have evaluated different classification and clustering techniques while focusing only on specific programming languages and design patterns. Research has shown that ML-based approaches show promising results by being able to understand variants of design patterns better than traditional metric-based approaches [12]. A consensus on which detection algorithms to use has yet to be reached [6]. Furthermore, capturing a wider range of design patterns is difficult because of the different methods for capturing features depending on the design pattern.

The most recent work on this topic has been using Large Language Models (LLMs) [7], [11]. The LLMs are used to extract an embedding from the code that is then used with another machine-learning algorithm to classify if and which design pattern a code is using. As there are currently no language models designed for the purpose of DPR, utilising LLMs will also require the use of a separate classifier or clusterer. The embeddings extracted from LLMs capture the semantic context of the source code, which can provide further useful information than previously metric-based approaches have used. This method of design pattern recognition using LLMs is a

very recent topic of research with many unexplored areas.

The previous research that has been done within LLM-based DPR has primarily focused on testing its feasibility. The most recent studies are exploring different areas of LLM-based DPR. They are testing different language models, the effects of pre-training the language models, the performance of different design patterns, and different combinations of these [11]. There has also been research on evaluating different supervised learning methods in non-DPR contexts [13]. But there is still a lack of research evaluating how different classification and clustering algorithms perform in the context of design pattern recognition when using other large language models, such as CodeBERT. This thesis will focus on exploring the performance of classification and clustering algorithms that are trained on embeddings generated by large language models. It is still unclear how the performance of learning algorithms varies when using them to classify embeddings generated by the LLMs.

Some possible methods for classification include Decision Trees, Support Vector Machine (SVM), and K-Nearest Neighbour (KNN) [14]. For clustering, K-Means, Hierarchical Clustering, and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) can be utilised [15]. However, it remains unclear which method provides the best performance [6]. Different programming languages have varying syntax and coding styles, making it important to evaluate which algorithm should be used. Furthermore, the choice of language model used for extracting embedded feature vectors will influence how the classification and clustering algorithms are trained. Therefore, a study is required to explore the feasibility of a range of classification and clustering algorithms.

1.1 Purpose of the study

The purpose of this study is to advance the field of software development by exploring and evaluating different classification and clustering algorithms in the context of identifying the use of design patterns in code. Specifically, this research aims to delve into the efficiency and accuracy of these algorithms when applied to embeddings extracted from code via LLMs. The study will also explore how different design patterns, language models, and programming languages affect the performance of the algorithms. The algorithms will be quantitatively analysed to measure their performance. Thereby determining effective algorithmic approaches for accurately classifying and clustering software design patterns. As a result, enhancing the ability of developers to maintain, understand, and improve large and complex codebases.

1.2 Research questions

The research questions are divided into two parts: the performance of the different algorithms (**RQ1**), and whether and how the choice of classification algorithm may change depending on the choice of design patterns, programming language, and

language model that is used (**RQ2**).

RQ1: *What are the performances of different classifying and clustering algorithms in the context of identifying design patterns in code?*

This is performed to find the differences in performances between different classifiers and clusterers and get a baseline of their performance in DPR.

RQ2: *Does the performance of different classifying and clustering algorithms vary depending on the context in which they are used?*

RQ2.1: *Does the performance vary between design patterns?*

RQ2.2: *Does the performance vary between language models?*

RQ2.3: *Does the performance vary between programming languages?*

This is performed to explore how the different parts, such as what language the code is written in and what type of LLM is being used, of a whole system affect the performance of the classifier. The goal of RQ2 is to find out if there are different classifiers with optimal performance depending on the context in which they are applied. It is done in order to study the effects on the performance of the classifiers when applied in different contexts.

1.3 Scope and Limitations

The study is limited to the assessment of classification and clustering algorithms in the context of DPR. The study will not examine all the design patterns identified by Gamma et al., nor will it examine their industrial adaptations [2]. This is due to the fact that there is not a lot of data available for this type of research. The main programming language that will be used in this study and used for RQ1 will be Java. Other programming languages will only be included for RQ2. This is due to the publicly available data for Java being much higher than other programming languages. The findings will, however, still be applicable to the patterns excluded from the study. The study will not use LLMs that require pre-training, and it will not look into how pre-training the LLMs affects the performance of the classifiers. This is due to the size of the models, and pre-training them requires a lot of time and hardware that is not readily available. The study will not explore the fine-tuning of the LLMs either, as it is outside the scope of the research questions and not feasible to be performed within the time schedule. Very large language models, such as models with multiple billions of parameters, will not be used due to hardware limitations.

2

Background

The focus of this study lies in the evaluation of a range of classification and clustering algorithms for DPR. The study will use implementations of design patterns as data, then use LLMs to extract important features, which will then be fed into classification and clustering algorithms that will predict which design pattern a given file consists of. This section provides the background and theoretical information necessary to understand the method and the findings of the study and will bring up the following topics: design patterns, large language models, and classification and clustering algorithms.

2.1 Design Patterns

A large part of software development is writing understandable and maintainable code. When codebases grow and developers add new functionality, the source code's complexity grow along with it. New developers learn about the importance of maintaining code and study various tools for this. One of the tools for keeping a structured and maintainable code base is design patterns. Design patterns are used in software development as a guide or description for how to solve commonly occurring problems [2].

Design patterns can be divided into three different types: creational, structural, and behavioral patterns [2]. Creational patterns are patterns that encapsulate knowledge of a class that should be instantiated, as well as hide the underlying structure of the class. Some examples of creational patterns are Singleton [16], Builder [17], and Prototype [18]. Structural patterns are patterns that focus on the structure of objects and classes while keeping their structure flexible. Some examples of structural patterns are Adapter [19], Bridge [20], and Composite [21]. Lastly, behavioral patterns are responsible for the communication between objects and classes and characterise complex control flows in programs. Some examples of behavioral patterns are Strategy [22], Observer [23], and State [24].

2.2 Large Language Models

Language Models (LMs) are models that can understand and generate human language [25]. LMs have the ability to predict a sequence of words and can generate new text based on a given input. The most common type of LM estimates what words may come next based on the prior words and their context. Large Language Models (LLMs) are more advanced language models that have a large number of parameters, often reaching multiple billions of parameters [26]. One key part of LLMs is their transformer architecture [27] and the underlying self-attention mechanism [28] it uses to determine the relevance of different parts of the input.

The transformer architecture presented by Vaswani et al. consists of an encoder and a decoder [28]. The encoder takes an input sequence of symbol representations and creates a sequence of continuous representations. The encoder creates a set of vectors for each word that describe the words' relationship with other words in the sequence. The decoder uses the output of the encoder to create a contextually relevant output of the model. Both the encoder and decoder contain self-attention mechanisms that help with extracting the contextual information of the input. This is done one-by-one by taking each word of the input and calculating and assigning weights to all the other words of the input based on their vector similarities that the model learns during training. Other important parts of LLMs are the ability to pre-train them on datasets and fine-tune them to perform specific tasks [29], as this has proven to improve their performance on multiple different tasks [30].

To represent values and inputs, LLMs use something called embeddings [31]. An embedding is a mathematical representation of objects like images, text, and audio. They represent these objects in the form of vectors. Figure 2.1 illustrates how embeddings are extracted from a document. Embeddings enable language models to find similarities and relationships between objects. As inputs to language models are objects like texts and images, the models will convert the inputs into embeddings. How a model generates the embeddings is different for each model, and the same input will generate different embeddings for different models. The generated embedding is then compared to the data the model was trained on to find similarities between the new input and what the model already knows. It will then use these similarities to perform its task.

This study will utilise existing LLMs to extract embeddings from source code, which in turn can be used to train classification and clustering algorithms. Since LLMs can capture the structure and semantic information from a text, classifiers can hopefully learn which semantic information is part of which design pattern and therefore get a higher accuracy when predicting.

Section 2.2.1 through 2.2.3 presents the different LLMs used in this study.

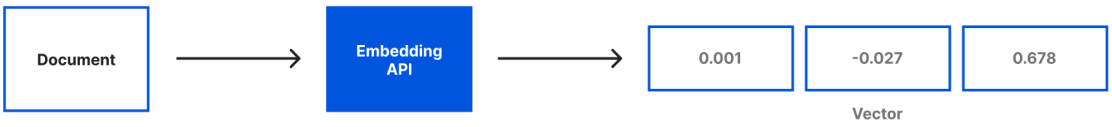

Figure 2.1: Document converted to embeddings [31].

2.2.1 CodeBERT

CodeBERT is a transformer-based bimodal pre-trained model that achieves state-of-the-art performance in natural language code search and code documentation generation [32]. Bimodal signifies that it is trained on Natural Language-Programming Language (NL-PL) pairs, which capture the semantic connection between natural language and programming language. Results show that it performs better than previous pre-trained models on NL-PL probing. The BERT in CodeBERT stands for Bidirectional Encoder Representations from Transformers, which is a framework that is used for natural language processing (NLP). This model is similar to other models using this framework, such as RoBERTa [33] and BERT [34]. The idea behind these models is to learn the contextual representations of huge natural language, which can, for example, be used to predict masked words in a sentence.

The novelty that CodeBERT presents is the fact that it is trained on several programming languages and is the first large NL-PL pre-trained model for multiple programming languages; those languages are Go, Java, JavaScript, PHP, Python, and Ruby [32]. Moreover, CodeBERT is not only trained on NL-PL pairs, that is, code paired with documentation, but also on a large amount of unimodal data, which is code that is not paired with documentation. For pre-training the model, a special separation token was used to divide the NL-PL pairs, where the natural language was represented as a sequence of words and the code was represented as a sequence of tokens. The output is represented as both the context vector representation of each token and as the aggregated sequence representation [32].

2.2.2 Instructor-xl

INSTRUCTOR is a sentence transformer-based model that is based on the single-encoder architecture that follows prior work [35]. INSTRUCTOR is designed to be applicable to many different tasks across different domains, not only within programming. The model takes text and task instructions as input. This makes it so INSTRUCTOR is aware of the domain and the task and generates an embedding that better fits the given task and domain. This was achieved by training the model on a large collection of datasets across multiple categories of tasks and domains.

Results show that INSTRUCTOR performs significantly better than other state-of-the-art models across all different types of tasks [35]. It outperforms the previous Sent-T5-XXL [36] model with a fraction of the parameters, with INSTRUCTOR having 335 million parameters compared to 4.8 billion parameters for Sent-T5-XXL

[35]. The model used in this study is Instructor-xl, which has 1.5 billion parameters.

2.2.3 GPT-2

GPT-2 is also a transformer-based model that follows the structure and details of the original GPT model [26]. When the GPT model was introduced, it was difficult for natural language classification models to perform well due to a scarce amount of task-specific labelled data [37]. The GPT model introduced a way for natural language models to realise gains on tasks that are unlabeled by performing generative pre-training on the data, followed by discriminative fine-tuning on the specific task. The GPT model also made use of task-aware transformations of the input, which were not used in previous work. The pre-training made the model gain significant knowledge and the ability to process long-range context and dependencies [37]. The training data for GPT-2 was obtained by scraping web pages from outbound links on Reddit [38]. GPT-2 made improvements to the original model by making detailed changes to the model architecture, increasing the number of parameters, increasing the size of the vocabulary, and increasing the context and batch size [26].

The GPT model proved to be effective and achieved state-of-the-art performance on the majority of the tested datasets [37]. The use of generative pre-training followed by discriminative fine-tuning proved to be a big step in machine learning and is now a method that is employed by a majority of LLMs [39].

2.3 Classifying Algorithms

Classification algorithms represent a category of machine learning methods where a model is trained to correctly identify the label of a given input [40]. This approach falls under supervised learning, necessitating labelled training data for the model to learn the pattern of the data. Classification problems involve two types of target variables: categorical and continuous. When the desired output is discrete, the task is referred to as classification. In contrast, when the output is continuous, the task is categorised as a type of regression.

There are two primary types of classification tasks: binary classification and multiclass classification. In binary classification, the target variable is binary, taking on values such as 0 or 1, true or false, etc. Meanwhile, in multiclass classification, the target variable can belong to one of multiple classes [41].

Multiple classification algorithms can be combined for a single classification task. This is known as an ensemble. Ensembles are models that use two or more algorithms to make their classifications. This is done to improve the performance, generalisability, and robustness of the model. There are three main types of ensembles: bagging, stacking, and boosting. Bagging fits many classifiers on different samples of the dataset and then averages all the predictions made. Stacking fits many different classifiers on the same data and then uses a separate classifier to combine the

predictions. Boosting is when multiple classifiers are fitted in sequence, where each algorithm fitted aims to minimise the errors made by the prior algorithms. The output of boosting is a weighted average of all the classifiers [42].

A variety of classification algorithms will be trained and evaluated in this study to compare their effectiveness when predicting design patterns. This paper will use multiclass classification as there are many different design patterns that will be predicted. Multiclass classification is often handled using a technique called One-Vs-Rest (OVR). The OVR technique means that a classifier will fit one classifier for each label [43]. Each classifier will then handle it as a binary classification problem, with its assigned label being the positive examples and everything else being negative. The classifiers will then produce a confidence score for the sample based on its assigned label, and the label with the highest confidence score is what the sample will be predicted to be.

The data used for training classification algorithms is structured as entities in a list, each with features, where each feature represents a property of the class. It is common to encounter an imbalanced dataset, which can negatively impact the performance of many classifiers. The next sections describe a subset of the available classification algorithms.

2.3.1 Random Forest

Random Forest is an ensemble learning technique where multiple decision trees are trained with a subset of the training data [44]. The classifier then predicts the data by aggregating the predictions of all the decision trees. This diversity of decision trees makes the model less prone to overfitting [45]. Furthermore, Random Forest is well-suited for handling high-dimensional feature spaces.

2.3.2 Naive Bayes

Naive Bayes is a supervised learning method that uses the principles of probability to determine the outcome of a class, specifically Bayes' theorem [46]. This algorithm assumes that each feature is independent and is often used in test classification problems. Naive Bayes is seen as a very simple and naive algorithm, as it does not care if features are correlated. Bayes' Theorem is represented by the following formula:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Here $P(A|B)$ is the probability of event A occurring given that event B has already occurred.

2.3.3 Gaussian Process

Gaussian Process (GP) is a supervised learning method that can be used for probabilistic classification problems [47]. This means that the model gives a measure of

how confident it is in its prediction. In short, GP works by assuming some underlying relationship in the data known as the prior and then using a link function (often logistic) to get the probabilities that the data is of a certain class. One disadvantage of the Gaussian Process is that they use the whole dataset when predicting, and they are less effective in high-dimensional data, i.e., when the feature vector is more than a few dozen dimensions.

2.3.4 Logistic Regression

Logistic Regression is a probabilistic discriminative model that is used for classification problems [48, Chapter 4.3]. The Logistic Regression model is at the core used for binary classification problems but can be used to classify multiclass problems by using multinomial logistic regression or by using the One-vs-Rest approach. This classifier uses a logistic function to describe the outcome of a class. The following formula describes the Logistic Regression, where $P(y = 1|x)$ stands for probability that y=1 given the input x.

$$P(y = 1|x) = \frac{1}{1 + e^{-(\beta_0+\beta_1 x_1+...+\beta_p x_p)}}$$

2.3.5 Support Vector Machine

Support Vector Machines (SVMs) are supervised learning methods that can be used for both regression and classification tasks [48, Chapter 7]. They are highly effective in high-dimensional spaces. SVMs work by constructing a set of hyperplanes that maximise the margin between the training data. Figure 2.2 illustrates how an SVM can separate the training data into two distinct classes. The data points closest to the separating hyperplane are called support vectors, as they critically define the margin boundaries.

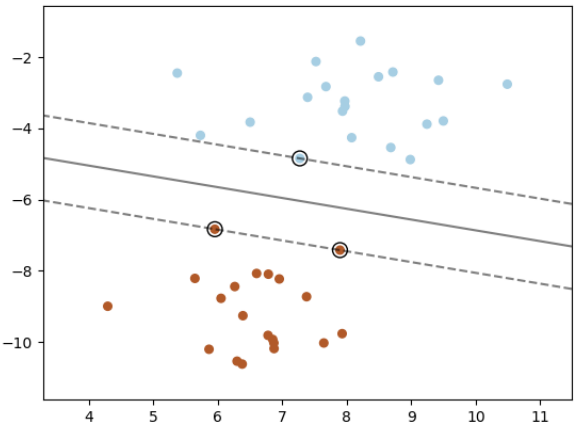

Figure 2.2: Decision boundaries for a linearly separable problem [49].

2.3.6 Nearest Neighbour

Nearest Neighbour (NN) is a simple algorithm that stores instances of the training data [50]. Classification of new data points is done by a majority vote of their nearest neighbours. There are several variations of the Nearest Neighbour algorithm, but the most common one is KNN. In KNN, the value of K determines the number of nearest data points considered when assigning a class to a new instance. Larger values of K reduce the impact of noise but can also blur boundaries between classes. KNN algorithms can become less effective on large datasets since every new point must be compared against all other points to determine its classification.

2.3.7 Gradient Boosting

Gradient Boosting is a powerful classification algorithm and one of the most popular ensemble methods [51]. This algorithm, a type of boosting, excels at capturing non-linear patterns in data. The model works by sequentially fitting simpler models, usually decision trees, that attempt to correct the errors made by previous models. It uses the negative gradient of the loss function to determine the direction of improvement for each subsequent model, repeating this process for the number of estimators used in the model.

2.3.8 Artificial Neural Networks

Artificial Neural Networks, or just Neural Networks, also known as Multi-Layer Perceptrons (MLPs), are a type of machine-learning algorithm that can be used for multiple machine-learning tasks, including classification [52]. The building blocks of a Neural Network are the neurons. A neuron has weighted input signals and an output signal that transforms the value using an activation function. Neurons are arranged into networks that consist of an input layer, one or more hidden layers, and one output layer. The input layer is what receives the input and passes it onto the first hidden layer. The hidden layers contain the neurons that transform the value they receive using the weights and activation functions of the neurons. The final layer is the output layer, which generates an output of the model according to the task. A simple model of a Neural Network with input, hidden, and output layers can be seen in Figure 2.3 below.

2.4 Clustering Algorithms

Clustering algorithms are tasks in machine learning that group similar objects together into clusters. This method is a form of unsupervised learning, which means it does not require labelled data for training [53]. Clustering methods are commonly used when the underlying structure of the data is unknown and to discover new patterns within the data. Unlike classification algorithms, clustering does not usually measure performance using accuracy, precision, or recall. If labelled data is available, it can be used to test the accuracy of the algorithm, but it can be tricky as the clustering method doesn't explicitly say which cluster belongs to a specific

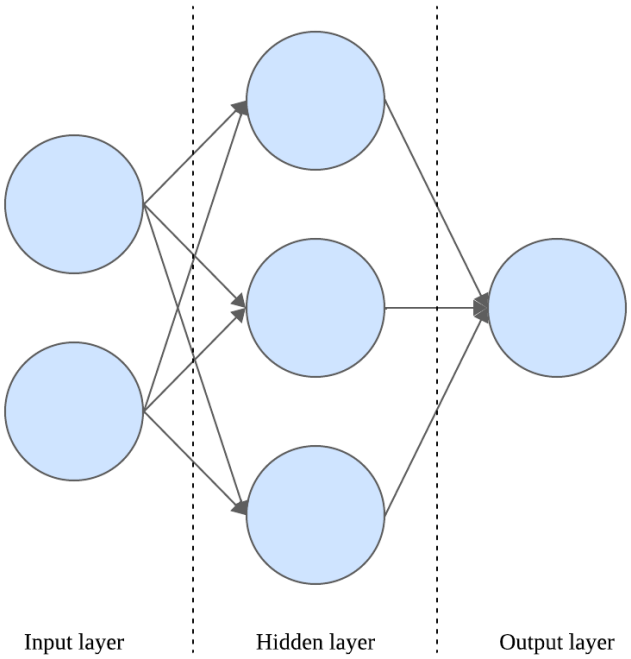

Figure 2.3: A simple model of a Neural Network.

class. Furthermore, clustering algorithms use a graphical aid called the Silhouette [54] to measure performance. Silhouettes are created by representing each cluster in a Silhouette diagram based on its tightness and separation. A mean Silhouette coefficient is then calculated by comparing the sample points to the generated Silhouette, providing a value between -1 and 1. Here, -1 indicates that a point has been incorrectly assigned to a cluster, while 1 represents the optimal assignment [55]. The following sections describe a subset of the available clustering algorithms.

For this study, clustering algorithms will be tested to see if they can find any clusters from the extracted embeddings.

2.4.1 K-Means

K-Means is a widely used clustering algorithm that tries to group data into a specified number of clusters based on their variance [56]. The main objective of the algorithm is to choose a centroid that minimises the within-cluster sum-of-squares, also known as inertia. The centroids selected are the means of the clusters. After a centroid has been chosen, the samples are assigned to their nearest centroid, which is subsequently updated in that cluster based on the mean of the assigned samples. This process repeats until the centroid stabilises and does not move significantly when adding new samples. Since K-means uses inertia to construct the cluster, it assumes that the clusters are more convex and isotropic-shaped and thus perform worse on elongated and irregular shapes.

2.4.2 DBSCAN

The DBSCAN algorithm clusters data by separating high-density areas from low-density areas [57]. The clusters are generated by finding a set of core samples given by a distance measure between data points. The neighbouring core samples are then measured to find all core samples of those; this is recursively done until no more core samples are found. These core samples are then surrounded by non-core samples, which are samples that are further away from the core samples given a threshold. The advantage of DBSCAN compared to K-means is that it can find clusters of any shape.

2.4.3 Mean Shift

Mean Shift clustering is a centroid-based algorithm that works by updating which centroid data points are considered the mean for a given region of data points [58]. The different candidates are then filtered to eliminate candidates that generate the same or very similar clusters. Centroid positions are calculated using hill climbing in order to find local maxima. The algorithm automatically calculates the number of clusters, but it can also be set manually. Mean Shift requires multiple nearest neighbour searches during the execution of the algorithm. This creates a complexity of $O(N^2)$ for the algorithm and also makes it not highly scalable. Application domains for Mean Shift include computer vision and image processing.

2.4.4 Gaussian Mixture

Gaussian Mixtures are a special type of clustering model called Gaussian Mixture Models [59]. Their probabilistic model assumes that all the data points are generated from a finite number of Gaussian distributions. There are different types of Gaussian mixture models, depending on the chosen estimation strategy. One estimation strategy uses the expectation-maximisation algorithm. The main difficulty of Gaussian mixture models is dealing with unlabeled data and knowing what data comes from which cluster. The expectation-maximisation algorithm uses an iterative process to get around this issue. It starts by assuming that clusters are random by centering them on randomly chosen data points or by using K-Means. The initialization method can be random or done by K-Means. It then calculates the probability of a data point being generated by each cluster and tweaks the parameters to maximise the likelihood of the data points, given their cluster assignments. This process is repeated and will be guaranteed to converge to a local optimum. Gaussian mixture models are often found in biometric systems, most notably in speech recognition [60], [61].

2.4.5 BIRCH

BIRCH is another clustering algorithm that works by building a tree based on the given data called a Clustering Feature Tree (CFT) [62]. The tree consists of Clustering Feature Nodes (CF Nodes) and the CF Nodes contain subclusters, and the subclusters can contain CF Nodes as children. The subclusters are what contain

the necessary information used to cluster the data. The algorithm works by inserting new samples into the root of the CFT, which is a CF Node. The new sample is then merged with the subcluster in the root that has the smallest radius after merging. If the subcluster it merged with contains a child CF Node, the process is repeated until it reaches a leaf. Splitting of nodes into branches occurs when the radius of a subcluster after merging is above a defined threshold value and the number of subclusters is greater than the branching factor. The number of clusters can be manually set; if it is not set manually, the clusters will be the subclusters at a leaf. Applications for BIRCH include image classification and image compression [63].

2.4.6 Affinity Propagation

Affinity Propagation works by passing messages between pairs of samples until the algorithm converges [64]. A dataset is described using exemplars, which are a small number of data points that are supposed to represent other data points. The messages that data points pass to each other represent how suitable a data point is to be the exemplar of another data point. This gets updated in response to values from other pairs. The updating continues iteratively until it converges. At convergence, the final exemplars are picked and the clusters are created. Like Mean Shift, Affinity Propagation is not highly scalable due to its complexity of $O(N^2T)$ where N is the number of data points and T is the number of iterations until convergence.

The next section will present some of the related work done in this field of design pattern recognition, where concepts and terms from this section will be widely used. Information from this section will also be used when discussing the results of the study.

3

Related Work

This section presents previous studies that utilised various methods for the purpose of DPR as well as the evaluation of classification algorithms in non-DPR contexts. It outlines the main components of these studies, including their evaluation methods and results.

3.1 Existing Approaches for DPR

Yarahmadi and Hasheminejad conducted an extensive literature review of papers about design pattern detection that were published between 2008 and 2019 [12]. In their study, they found that the majority of the approaches to DPR worked on creational, structural, and behavioral patterns. They did, however, note that some patterns were harder to identify or could not be identified at all due to their similar structure. Yarahmadi and Hasheminejad also found that a large majority of DPR approaches use source code as their type of input and found that there were many different ways of representing the data, including metrics and feature vectors. They also found that there were multiple different DPR approaches, but the most common were learning-based approaches that made use of artificial intelligence and machine learning. Furthermore, they found that the P-MARt repository [65] and the projects within it were used by a large majority of the papers for training and evaluation of their methods.

Yarahmadi and Hasheminejad also looked at the results of different DPR approaches in their study and presented advantages and disadvantages of the approaches [12]. They found that learning-based approaches performed well. It had a high recall and precision, was evaluated on large test cases, and could detect design patterns such as Singleton. But it had difficulties detecting patterns with similar structures. Metric-based approaches performed worse than learning-based approaches but could detect patterns with similar structure. They found that many of the approaches had small test cases, which questioned their ability to function in large systems. They also found that many approaches cannot detect different variants of design patterns, but that learning-based approaches are capable of this. They also state that many of the methods have weak performances and evaluations. Yarahmadi and Hasheminejad believe that many benefits can be gained by combining different methods.

As stated by Yarahmadi and Hasheminejad, there are many approaches that have been extensively researched when it comes to design pattern recognition [12]. Different approaches have different methods for how they represent their data. Some of these include metrics extracted from the source code, the Abstract Syntax Tree (AST), which is a tree that represents the structure of the code, and the Abstract Semantic Graph (ASG), which captures a higher level of abstraction compared to the AST. For ML-based approaches, the data representation may instead come from using language models. These language models are trained on a large set of data and can extract the semantic meaning of the source code in a more generalisable way. The next section outlines two different DPR approaches researched.

3.1.1 Metric-based Approaches

Tsantalis et al. explain a method for detecting Java design patterns from similarity scoring between graph vertices [9]. They evaluated their algorithm on three open source projects: JHotDraw, JRefactory, and JUnit. Each project was parsed, and various metrics were extracted to get a hierarchical tree structure of the classes, methods, and their invocations. These directed graphs were then mapped to matrices for easier manipulation. Known design patterns were also hard-coded as matrices, each with descriptive attributes. The graphs and matrices from the extracted code could then be used to measure the similarity score to find the best match. The authors also note that some design patterns have identical structures and cannot be individually identified without further context; these are Adapter/Command, and State/Strategy. Their results showed that their method could correctly identify every design pattern in their scope with only two false negatives and no false positives.

Dwivedi et al. utilise a metric-based approach where they extracted 67 metrics using the JBuilder software [10]. Some of these metrics are familiar, such as lines of code or the number of parameters. But it also includes more advanced metrics such as the polymorphism factor and coupling between objects. They evaluated the performance of three different supervised learning methods: Artificial Neural Networks, Support Vector Machine, and Random Forest. Five design patterns were used: Abstract Factory, Adapter, Bridge, Composite, and Template Method. For each pattern, all the related classes were used to extract metrics from the pattern. The design patterns used were from the P-MARt repository [65]. Validation was done using cross-validation, and accuracy was used to measure the performance of the models. Their study showed that ANN performed the poorest, while random forest was the best, with an F1-measure of 100% on QuickUML and JUnit and 97.6% on JHotDraw, averaged over all design patterns focused on.

Zanoni et al. describe an approach for detecting design patterns in code [6]. Their approach makes use of graph matching and machine learning techniques. Their study builds on previous work by implementing enhancements in their analysis method. They introduce a tool called MAPRLE-DPD. This tool is divided into three main modules. The first module is the information detector engine. This module is responsible for building a model of the programme and collecting and

storing microstructures and metrics in the model. The second module is the joiner. The joiner works on the microstructures that the information detector engine has extracted and extracts all possible design patterns. The third module is the classifier. The classifier computes similarities between the design patterns extracted by the joiner by comparing them to known implementations of the design patterns.

In their study, Zanoni et al. also define two different categories of design patterns [6]. These categories include single-level patterns and multi-level patterns. They define patterns as trees of levels where each level contains at least one role. A role is a class that is needed to implement a design pattern. A single-level pattern is a design pattern that can be implemented using only one level. Multiple roles are allowed in single-level patterns as long as they are at the same level. Multi-level patterns are design patterns that consist of multiple levels. A pattern can consist of multiple levels if its implementation uses multiple classes and one of the classes inherits from another class. An example of a multi-level design pattern is Template Method. Template Method creates an abstract template class that is then inherited and used by concrete classes.

Zanoni et al. evaluated their model in the form of experiments [6]. The dataset of design patterns they used consisted of 10 different projects, and the training set totaled 2794 different pattern instances. One project was gathered from different design pattern examples found on the web. The nine other projects were analysed in the P-MARt repository [65]. They used multiple different classification and clustering algorithms to evaluate the performance of the model, including Decision Trees, Random Forest, SVMs with different kernel functions, Naive Bayes, SimpleKMeans, and CLOPE [6]. Their model and the algorithms were evaluated on single-level patterns and multi-level patterns separately, and they only used clustering algorithms for multi-level patterns. The single-level patterns experimented on were Singleton and Adapter. The multi-level patterns were Composite, Decorator and Factory Method. The algorithms were compared on three different metrics: accuracy, F1-measure, and Area Under Curve (AUC). For single-level patterns, Zanoni et al. concluded that all algorithms have similar performances, with the exception of Naive Bayes and one variation of SVM that performed significantly worse than the other algorithms. For multi-level patterns, they found the best performance with an SVM. It is worth noting, however, that the goal of the study was to evaluate their approach to DPR, and there was no extensive testing or comparison of different algorithms.

3.1.2 LLM-based Approaches

Pandey et al. proposed an approach for DPR that utilised language models that have been trained on programming languages [11]. The study introduced a DPR technique called TransDPR. The language model used in their study was TransCoder. TransCoder is a language model developed by Meta [66], and the purpose of the model is to translate source code from and to Java, C++, and Python. Pandey et al. use TransCoder to extract the semantic information of a program [11]. This was done by extracting the embedding vector produced by TransCoder at the encoder

block of the TransCoder model.

Their model was evaluated over two patterns: Singleton and Prototype. The pattern instances were found in open-source projects on GitHub and were written in C++. For each instance of a pattern, they also created an artificial counter instance by multiplying the embedding vector of the instance with -1. The counterexamples were annotated as non-singleton and non-prototype. This was done to reduce overfitting and to make sure the model is not biassed towards any design pattern. Their training set was composed of 52 embedding vectors, of which half were artificial counterexamples. The model was assessed by splitting data into a training and test set, with 70% of the data being part of the training set. They only used one classifier, Logistic Regression, and the model gave 50% accuracy for Singleton and 90% accuracy for Prototype. Pandey et al. also performed an overall test where Singleton and Prototype were combined into positive examples and non-singleton and non-prototype were combined into negative examples. The accuracy for the overall test was 90%.

A study performed by Chand et al. tested the feasibility of using language models for DPR within industrial contexts [7]. Their approach is similar to the approach used by Pandey et al. in their study [11]. Chand et al. use two language models, Word2Vec and Code2Vec [7]. These models are used to generate an embedding of the source code. Both of the models were pre-trained on two open-source systems often utilised within the automotive industry: Genivi [67] and Android Auto [68]. The pre-training dataset consisted of 43,397 code examples, which were split up into test, validation, and training datasets. Both models had similar performance with the pre-training dataset, with Word2Vec performing slightly better.

Chand et al. evaluated the ability of the models to detect design patterns with 27 code examples across three design patterns: Singleton, Prototype, and Builder [7]. The algorithm they used was a K-means cluster algorithm. For the purpose of detecting design patterns, Code2Vec outperformed Word2Vec with a mean F1-score of 0.781 compared to 0.690 achieved by Word2Vec. The results showed that the models could store architectural information about the source code.

3.2 Existing evaluation of classification algorithms

There is also work in the field of evaluating classification techniques in a non-DPR context. The studies are mainly performed using different datasets, and the results show that classifiers are often task-dependent; in other words, some classifiers are good on some datasets while others are worse.

Caruana et al. conducted a large-scale study in which they compared ten supervised learning methods to determine which methods had the best performance across 11 different binary classification problems [13]. Their study included supervised learning algorithms such as logistic regression, SVMs, naive Bayes, KNN, random forest, decision trees, bagged trees, boosted trees, boosted stumps, and neural networks.

They evaluated these learning algorithms using eight performance metrics, including accuracy, F-score, and Lift Curves. Additionally, calibration methods such as Platt Scaling and Isotonic Regression were applied to compare the algorithms probabilistically. For datasets, they used 11 binary classification problems, including ADULT, COV_TYPE, and LETTER. Their results showed that boosted trees performed best when considering the average across all eight metrics, with random forests coming in second. They also noted that there is no universally best learning algorithm, as some algorithms that performed poorly on some problems excelled on others. For example, random forest, neural networks, and logistic regression were the best models on the MEDIS dataset. Overall, the worst models were naive Bayes and memory-based learning, such as KNN.

Soofi and Awan investigated several machine learning classification techniques and the applications for which they were used in [14]. Due to their effectiveness and simplicity, Decision Trees were the most commonly used algorithm for classification tasks [69]. One problem that Decision Trees excelled at was dealing with multi-valued attributes. Bayesian Networks were another classification technique investigated. This method graphs variables and their conditional dependencies using a Directed Acyclic Graph (DAG). One disadvantage of Bayesian Networks is that they require continuous variables to be discretized, which can cause classification issues [70], [71]. The benefits of Bayesian Networks include their ability to solve both regression and classification problems, as well as their ability to handle missing values. K-nearest neighbour (KNN) is another classification technique that is good at scaling over multimedia datasets and was used to address issues with space and time constraints. KNN is also effective when dealing with large amounts of training data and is resistant to noisy training data [72]. Some disadvantages include computation complexity, memory limitation, and poor run-time performance [73], [74]. Lastly, Support Vector Machines (SVM) is a technique considered to be one of the more convenient methods for solving classification problems [75]. SVMs excel at handling high-dimensional and non-linearly separable problems. However, a notable disadvantage is that achieving accurate results often requires setting key parameters correctly. SVM can be applied to issues involving multi-label classification and is effective at controlling the false-positive rate.

4

Method

The main method used in this project was an Mining Software Repository (MSR) experiment method [76]. The study used publicly available software repositories in order to understand and improve software maintainability. The study was also a knowledge-seeking study that aimed to evaluate and validate different methods and their performance [77]. The independent variables in this study were the classification and clustering algorithms. An overview of the process for RQ1 can be seen in Figure 4.1, and an overview of the process for RQ2 can be seen in Figure 4.2. The pipeline referenced in those figures is explained in Section 4.7.

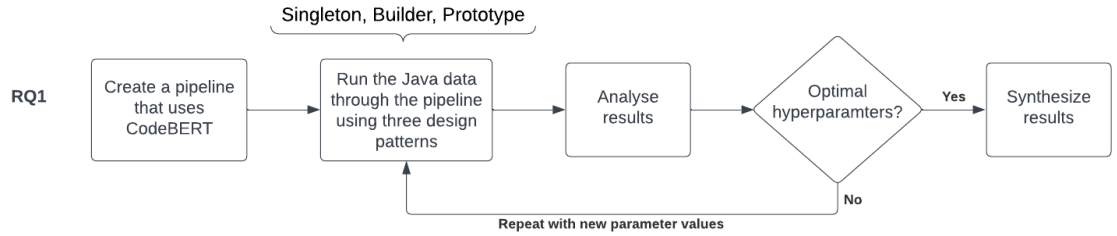

Figure 4.1: Visualisation of process for RQ1.

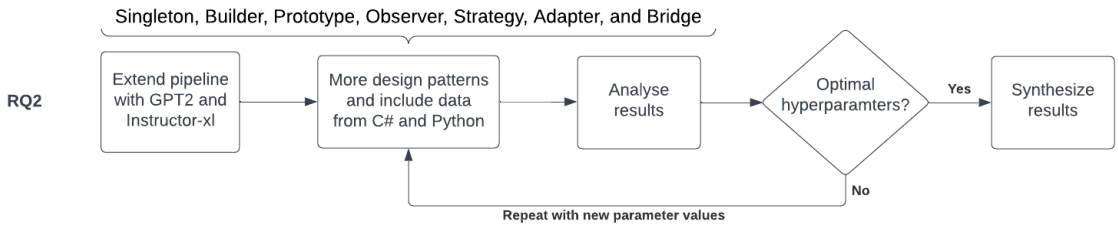

Figure 4.2: Visualisation of process for RQ2.

4.1 Data Collection

Java data for Singleton, Prototype, and Builder were gathered mainly from the P-MARt repository [65]. The P-MARt repository consists of labelled usage design patterns that have been extracted from different open source software that is written

in Java. The data gathered from P-MARt consisted of 32 examples of Prototype, 25 of Singleton, and 9 of Builder. All the examples were written in Java.

Data was also systematically gathered from public GitHub repositories. This is due to the P-MARt data gathered being imbalanced and having an unequal amount of data for the different patterns. An imbalanced dataset can lead to the machine-learning algorithms overfitting on the design pattern(s) that are most common in the dataset. This happens because the algorithm believes that the most common design patterns have a higher chance of appearing than the other design patterns. The P-MARt repository also only have Java data for Singleton, Prototype, and Builder and data had to be gathered for Adapter, Bridge, Strategy, and Observer as well as data for Python and C#. Examples of design pattern implementations were found by searching with specific search terms on GitHub. The search terms used to find data for all three programming languages were the following:

- "design patterns language:Java"
- "design patterns language:Python"
- "design patterns language:C#"

The results were then sorted by "Most stars". It was sorted by "Most stars", as it usually indicates a repository being popular and of high quality. The repositories were then manually inspected one-by-one for design pattern implementations. For data to be included in a repository, it had to satisfy a list of requirements. The requirements can be seen below.

- The repository should be in English.
- The repository should have at least one example of the design patterns part of this study (Singleton, Prototype, Builder, Observer, Strategy, Bridge, and Adapter).
- The design patterns should be explicitly annotated by the author(s) of the repository.

The annotation of a design pattern can be: the file(s) are named after the design pattern, the folder the file(s) are in is named after the design pattern, or the comments in the code explicitly state the design pattern implemented. All the gathered design pattern examples were then manually reviewed to see if the implementation of the pattern looked correct when compared to the implementation presented by Gamma et al. [2]. The number of files and their source for the Singleton, Prototype, and Builder patterns written in Java are shown in Table 4.1 below. All other patterns in Java had the same number of files and were all from GitHub. Python and C# had the same number of files, all from GitHub.

Table 4.1: Table showing the amount of files and their source for the Singleton, Prototype, and Builder design patterns written in Java.

Design Pattern	P-MARt	GitHub	Total
Singleton	25	7	32
Prototype	32	0	32
Builder	9	23	32

4.1.1 Data Preprocessing

The gathered data went through a manual preprocessing stage. All files used for a single pattern implementation were merged into one file, so each file in the final data set represented a full implementation and usage of a design pattern. Parts of the code that were deemed to be irrelevant to the implementation of the pattern were also removed. Examples of these were: package names, comments explaining the license the code was part of, the author of the code, and educational comments explaining how the pattern works. Comments related to the functionality of the code were not removed as they are relevant for the language model's understanding of the code. It was especially relevant for CodeBERT, as that model has been trained on pairs of natural languages and programming languages. Meaning that comments in the code can help the model understand the code better.

4.2 Google Colab

Google Colab is a hosted Jupyter Notebook service that allows anybody to write and execute Python code through the browser [78]. The pipeline used in this study was written in Google Colab. Colab was opted for as it is free to use and it provides access to computing resources on which the code is executed. Colab gives access to execute code on Graphical Processing Units (GPUs), which are required for running machine learning and AI models. This sped up the process of generating embeddings and helped us train classifiers and clusterers. Colab also supports sharing and collaborating on projects.

4.3 Language Models Used

The study utilised language models in order to generate embeddings from the given code that the classifying and clustering algorithms then used as input to detect the presence of a design pattern. Each model was also responsible for tokenizing the input if the model required it to generate an embedding. The model used for RQ1 was CodeBERT. As it is trained on Java data, is well-known, and outperforms other well-known models, CodeBERT was chosen as the language model to be used in RQ1. For RQ2, multiple language models were used to study the effects of the choice of language model on the performance of the classifiers and clusterers. In RQ2, CodeBERT, GPT-2, and Instructor-xl were used.

CodeBERT is a variant of the BERT language model that has been pre-trained on programming languages. CodeBERT was chosen as it is a variant of a well-known language model and its purpose is to aid with coding, making it suitable for this study. As CodeBERT is trained on programming languages, it is known as a Programming Language Model (PLM). It is trained on Java and Python data, which are two of the programming languages used in this study, and achieves state-of-the-art performance. CodeBERT is also the only PLM included in this study, and PLMs are what previous studies utilising language models for DPR have used [7], [11], [79]. CodeBERT also outperformed other language models on C# even though it was not part of its training data [32]. As its performance translates over to C# as well, it was also chosen to be used for RQ2.

GPT-2 was included as a model to use in RQ2 to investigate how and if the choice of language model impacts the performance of the machine learning algorithms. Later versions of the GPT LLM were not used as they were not available to download locally for free. The model is also available for free, well-known, and showed state-of-the-art performance when it was released [26]. GPT-2 and CodeBERT are similar in their functionality but different in what they are designed for. CodeBERT is designed and focused around code and aiding with coding [32]. GPT-2 is a more general model that aims to aid with many different tasks in many different fields, such as translation, summarization, and question answering [26]. GPT-2 also generates an embedding for each token instead of one embedding for all tokens. The models are similar and both relevant for the study, but still different enough to discuss results and performance differences.

Instructor-xl was included as a third model in RQ2 due to it being more recent (late 2022), and a lot of advancements have been made in the area of machine learning and AI between the release of CodeBERT and GPT-2 and the release of Instructor-xl [35]. GPT-2 and CodeBERT are also not designed for the purpose of finding the presence of design patterns in code. Instructor-xl is also not designed for the purpose of DPR, but the model can be given instructions for the task and therefore generate an embedding that takes the task of DPR into account. It is therefore also of interest how a more recent model that can generate an embedding more appropriate for the task changes the performance of the classifiers or clusterers.

4.4 Design Patterns

The design patterns used in RQ1 were Singleton, Prototype, and Builder. These patterns were selected for their widespread use in object-oriented programming and their frequent appearance in previous DPR work [12]. Singleton, Prototype, and Builder are all categorised as creational design patterns. As the focus of RQ1 was to explore the performance of different classifiers and clusterers, we chose to select patterns that are of the same type. The difference in performance between the individual design patterns can then also be studied without having to make inferences about design pattern types. Differences between pattern types will be explored in RQ2.

For RQ2, we expanded the number of patterns from three to seven, including at least two from each design pattern category: creational, behavioral, and structural. The patterns included for RQ2 were Singleton, Prototype, Builder, Observer, Strategy, Adapter, and Bridge. The patterns were chosen as they were often part of previous work in DPR and are often used within object-oriented programming [12]. This also meant that it was not too difficult to find examples of their implementations to use as training and testing data. At least two patterns for each classification were chosen in order to study how machine-learning algorithms perform when classifying or clustering patterns both within and across design pattern classifications and how their performance varies between them.

4.5 Classifying and Clustering Algorithms

The classification algorithms used in this study were Logistic Regression, Random Forest, K-Nearest Neighbours, Support Vector Machine, Decision Tree, Gradient Boosting, Naive Bayes, Gaussian Process, and a Neural Network. The Neural Network will be a Multilayer Perceptron classifier, where the mapping between input and output is non-linear by using a non-linear activation function. The clustering algorithms used in the study were K-Means, DBSCAN, Mean Shift, Gaussian Mixture, BIRCH, and Affinity Propagation. K-Means, Gaussian Mixture and BIRCH allow you to specify the number of classes present, and this will be done for those algorithms. This is done as this is a classification task and this information is available. The embeddings are of dimension 768, and reducing this size can lead to a loss of information. Specifics about training and testing the classifiers and clusterers are presented in Section 4.7.3.

A wide range of classifiers and clusterers was chosen to explore the performance of different algorithms and methods of classification. The chosen algorithms were opted for as they are well-known and have been used in previous work within DPR [6], [7], [11].

4.6 Metrics

The metrics calculated for each classifier were accuracy, precision, recall, and F1-score. Accuracy is the simplest metric here, and it shows how many of the predictions made were correct without considering any other information. It is calculated by dividing the number of correct predictions by the number of total predictions made. It is a value between zero and one, and the higher it is, the better the model is at making correct predictions.

TP: True Positives

TN: True Negatives

FP: False Positives

FN: False Negatives

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision is a metric that shows the proportion of how many of the predictions it makes for a certain class are correct. For example, it will show how many of the programmes it predicted as being Singleton are correct compared to how many Singleton programmes there are. It is a value between 0 and 1, and a higher value is better.

$$precision = \frac{TP}{TP + FP}$$

Recall shows how good the model is at finding all positive examples. It shows how many of the data points that should be classified as true were classified as true. In this case, it shows how many of the programmes that should be predicted as Singleton were predicted as Singleton. It is a value between 0 and 1, and a higher value is better.

$$recall = \frac{TP}{TP + FN}$$

The F1-score metric shows the accuracy of a model, and it takes precision and recall into account. Compared to just calculating the normal accuracy, the F1-score considers class imbalance and better reflects the true performance of a model. It is a value between 0 and 1, and a higher value means a better-performing model.

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

The data used in the study is labelled. Thus, F1, precision, and recall can be calculated for the clusterers as well. This does, however, require testing different mappings between patterns and clusters. Mapping patterns to clusters was only performed for RQ1 to be able to compare the clusterers and classifiers with the same metrics. It was not done for the other RQs as their performance in RQ1 did not show promise of the clusterers being able to compete with or outperform the classifiers. Mapping the patterns to clusters also takes time and was not feasible at all for RQ2.1 due the large amount of unique mappings. F1-score, precision, and recall were calculated by testing all different mappings of pattern to cluster and taking the results from the mapping that gave the highest F1-score.

The metrics used for RQ2 were the Adjusted Rand Index (ARI) and Silhouette score. These metrics were included in RQ1 as well for comparison purposes. The

Silhouette and ARI scores will tell whether the clusters are good in terms of how similar the points in a cluster are to each other. If a clusterer performs well in terms of these metrics, it can be used to classify code, but it would require the extra effort of mapping the clusters to the patterns. The clusterers will also be tuned around the Adjusted Rand Index.

The Silhouette score for a whole data set is calculated by calculating the Silhouette score for each individual data point, adding them, and dividing by the total number of data points. The Silhouette score ranges from -1 to 1. A negative score means that the data point is likely in the wrong cluster. A positive score indicates that the data point is assigned to the correct cluster. If clustering performs well, it should have a Silhouette score near 1.

a_i: Average distance of point i to all other points within the same cluster

b_i: Average distance of point i to all points in the nearest cluster

$$Silhouette\ score(i) = \frac{max(a_i, b_i)}{b_i - a_i}$$

$$Silhouette\ score = \frac{\sum_{i=1}^{n} Sillhouette\ Score(i)}{n}$$

The Rand Index (RI) is a metric that measures the similarity between two different clusterings of the same data. RI takes into account all different pairs of data and checks if the pairs have been assigned the same cluster or a different cluster when comparing to the true labels. It checks whether a pair within the same cluster has the same true label or whether a pair with the same true label is in different clusters. The Rand Index is calculated by the following formula:

$$Rand\ Index = \frac{number\ of\ agreeing\ pairs}{number\ of\ pairs}$$

The Adjusted Rand Index has the same value, but it adjusts the value according to the chance of making a correct clustering at random. It calculates an expected RI value that would have been achieved by random clustering and then adjusts the RI score according to the expected RI. This adjustment also gives ARI a baseline where scores near 0.0 have the same performance as random clustering. The ARI score ranges from -0.5 to 1, where a higher score is better.

4.7 Pipeline

For testing various classifiers and clusterers, a pipeline was created in Google Colab. An overview of the pipeline can be seen in Figure 4.3. This pipeline takes a directory of source code files as input.

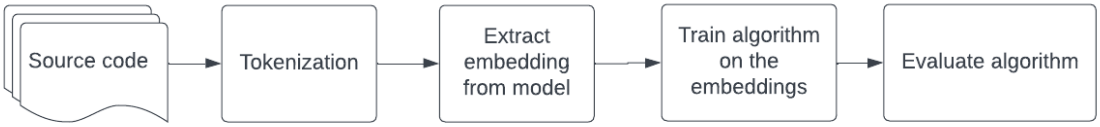

Figure 4.3: Visualisation of the pipeline.

4.7.1 Tokenization

The first step in the pipeline was to tokenize the code files. Much of the data used in this study would generate more tokens than the maximum input sequence length of the models. This issue was handled by splitting the code files into chunks, where each chunk was as big as the maximum input sequence length of the model. If a chunk or a file was smaller than the maximum input sequence length, it was padded to make the size equal to the maximum input size. This was done to make all inputs to the model have the same size. The padding does not add any semantic value and does not affect the embedding generated by the models. The chunks also had an overlap of 10% where the final 10% of a chunk would be the first 10% of the next chunk. This was done to help the models retain the context between chunks.

4.7.2 Extracting Embeddings

The tokenized chunks were then individually sent into the models, and their embeddings were extracted. All files that had multiple chunks would receive multiple embeddings. These embeddings were put into one final embedding by averaging all of their values. After this, all files would have one singular embedding as their representation. These embeddings were then saved as a comma-separated file, allowing them to be used later for training various classifiers and clusterers without the need to rerun the source code through the models.

The saved embeddings are then split into training and test data, where 80% was used for training the algorithms and 20% was used for testing. The data is split in this way in order to have sufficient data for the classifiers to train and learn from, as well as sufficient data to test and evaluate them. It is also done to reduce overfitting of the algorithms [80]. All patterns were split separately to ensure that there was an equal amount of data for each pattern in the training and test sets. It also removes any bias that can be introduced with imbalanced data sets. The clusterers did not use the separated data and instead created clusters from the full data set.

4.7.3 Training of Classifiers and Clusterers

The next step in the pipeline was to train and test all the classifiers and clusterers on the data. All algorithms used the same data, and they were trained and tested once per model and per programming language. Different hyperparameters were also tested for each algorithm by performing a grid search with selected hyperparameters. The grid search was also done per model and per programming language. This was done to ensure that the best possible result was achieved for all algorithms. What

hyperparameters were tested and how many combinations of each algorithm were tested can be seen in Table B.1 in Appendix B. The classifiers were optimised based on their F1-score, and the clusterers were optimised based on their ARI score. The values for the hyperparameters used in the different tests can also be found in the Appendix.

The classifiers and clusterers ran 200 times each with different training and test sets. This was done to reduce the effect that randomness has on the performance of the classifiers and clusterers and to see their average performance across multiple runs and how much it varies between runs [81]. Results before and after hyperparameter tuning were generated separately to see how much it affected the performance of the algorithms.

4.7.4 LLM Settings

For RQ1, we utilised CodeBERT [32] to extract the embeddings. RQ1 focused exclusively on the Singleton, Prototype, and Builder design patterns. All the design patterns analysed were code written in Java. Java was chosen as the language for RQ1, as the P-MARt data is all written in Java, and P-MARt has also been used in multiple previous studies within DPR [12]. The data used in RQ1 consisted of examples found in P-MARt and public GitHub repositories. The P-MARt dataset and the combination of P-MARt and public GitHub repositories were tested separately.

CodeBERT has a maximum input sequence length of 512 and also uses special classification and end-of-sequence tokens. So, the data for CodeBERT was split into chunks of size 510, and a classification token and end-of-sequence token were added as the first and last elements, respectively, making each chunk have a size of 512. As the maximum length is 512, the overlap for each chunk was 51 tokens.

RQ2 used two more models for the extraction of the embeddings: Instructor-xl [35], and GPT-2 [26]. The number of patterns was extended from three to seven and now included multiple patterns from all three types of design patterns. Two other programming languages were also tested: Python and C#. The pipeline was modified and extended to include GPT-2 and Instructor-xl. GPT-2 has a maximum input sequence length of 1024. The chunks created for GPT-2 were of size 1024 with an overlap size of 102. Instructor-xl has a maximum input sequence length of 512 and uses an overlap of 51 tokens. None of them use any special classification or end-of-sequence tokens. Instructor-xl, however, can be given instructions on the task, and the instructions given to the model were: *"Identify the software design pattern (Singleton, Prototype, Builder, Observer, Adapter, Bridge, Strategy) in the following code:"*.

5

Results

Quantitative data was obtained from the predictions the classifiers made on the test set and the clusters generated by the clusterers. Results relevant for RQ1 are presented in Section 5.1. Results relevant for RQ2 are presented in Section 5.2.

5.1 RQ1: Classifier Performance on Design Pattern Identification

This section presents results related to answering RQ1. The results are divided into baseline performance, which is the performance with default hyperparameters, presented in Section 5.1.1 and performance with tuned hyperparameters presented in Section 5.1.2. The design patterns used for RQ1 were Singleton, Prototype, and Builder. All design patterns were written in Java. All results presented in this section use embeddings generated by CodeBERT. The default hyperparameters for the algorithms can be found at scikit-learn [82].

5.1.1 Baseline Performance

The results of training and testing each classifier across 200 different runs with different training and test splits can be seen in the box plot in Figure 5.1 and in Table 5.1. Multiple classifiers were able to reach an F1-score of 1 in at least one run, even without any hyperparameter tuning. The classifiers that did not achieve a F1-score of 1 in any of the runs were Decision Tree, KNN, Naive Bayes, and Gaussian Process. The best lowest score was 0.651, which was achieved by Logistic Regression. Gaussian Process is the standout classifier here, with significantly worse performance than any other classifier. The mean F1-score it got was 0.219, with its highest being 0.328. It did have the lowest variance of all classifiers at 0.003, with the next lowest being 0.005, which was achieved by Logistic Regression. The second worst performance was attained by Decision Tree with a mean F1-score of 0.664.

Furthermore, Table 5.2 shows the F1-score (F1), Precision (P), and Recall (R) for each design pattern, Singleton, Prototype, and Builder averaged over 200 runs. These results are also visualised in confusion matrices, as can be seen in Figure 5.2.

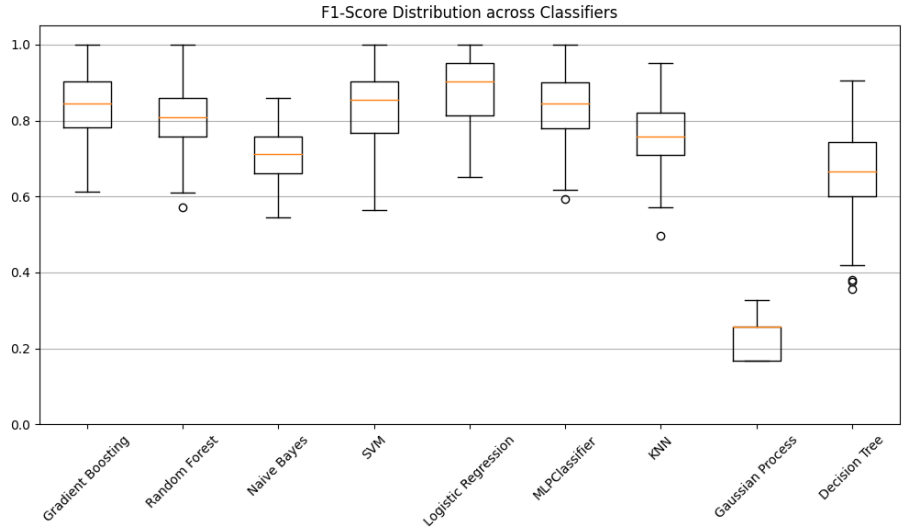

Figure 5.1: Box plot showing the F1-scores of the classifiers across 200 runs with default hyperparameters on CodeBERT embeddings, averaged across all patterns.

Table 5.1: Table depicting mean, variance, highest, and lowest F1-scores obtained from running each classifier 200 times with different training and test sets and with default hyperparameters on CodeBERT embeddings, results averaged across all design patterns.

Classifier	**Mean**	**Variance**	**Highest**	**Lowest**
MLPClassifier	0.830	0.005	**1.000**	0.593
Gradient Boosting	0.826	0.006	**1.000**	0.613
Random Forest	0.815	0.006	**1.000**	0.573
Decision Tree	0.664	0.011	0.905	0.356
KNN	0.763	0.008	0.952	0.498
Naive Bayes	0.707	0.005	0.860	0.544
Gaussian Process	0.219	**0.003**	0.328	0.167
Logistic Regression	**0.883**	0.005	**1.000**	**0.651**
SVM	0.832	0.005	**1.000**	0.563

The confusion matrices show which patterns were often predicted correctly and which ones were predicted wrong. The y-axis are the true labels, which indicate which design pattern the file contains, and the x-axis are the predicted labels, which indicate what the classifier predicted for that file. The diagonal in the matrices shows the percentage where the classifiers predicted the class correct, and the other cells show how the algorithm misclassified which design pattern the file consisted of. The matrices show that Singleton is the design pattern that gets predicted wrong most often. Singleton is also the design pattern that Prototype and Builder get wrongly predicted as being. Gaussian process is the classifier that stands out here, as it almost always predicts everything as being Singleton.

Table 5.2: Table depicting mean F1-score, precision, and recall obtained from running each classifier 200 times with different training and test sets across the Singleton, Prototype, and Builder design patterns.

Classifier	**Singleton** F1, P, R	**Prototype** F1, P, R	**Builder** F1, P, R
MLPClassifier	0.74, 0.78, 0.73	0.89, 0.92, 0.89	0.86, 0.85, 0.89
Gradient Boosting	0.75, 0.76, 0.76	0.88, 0.92, 0.85	0.85, 0.85, 0.88
Random Forest	0.76, 0.76, 0.78	0.84, 0.89, 0.82	0.85, 0.86, 0.86
Decision Tree	0.58, 0.62, 0.57	0.69, 0.71, 0.71	0.72, 0.73, 0.73
KNN	0.67, 0.77, 0.62	0.81, 0.78, 0.85	0.81, 0.80, 0.84
Naive Bayes	0.67, 0.71, 0.67	0.67, 0.67, 0.70	0.78, 0.81, 0.77
Gaussian Process	0.51, 0.34, **1.00**	0.00, 0.00, 0.00	0.15, 0.51, 0.09
Logistic Regression	**0.82**, **0.85**, 0.83	**0.93**, **0.97**, **0.90**	**0.90**, **0.88**, **0.93**
SVM	0.77, 0.75, 0.83	0.87, **0.97**, 0.81	0.85, 0.86, 0.87

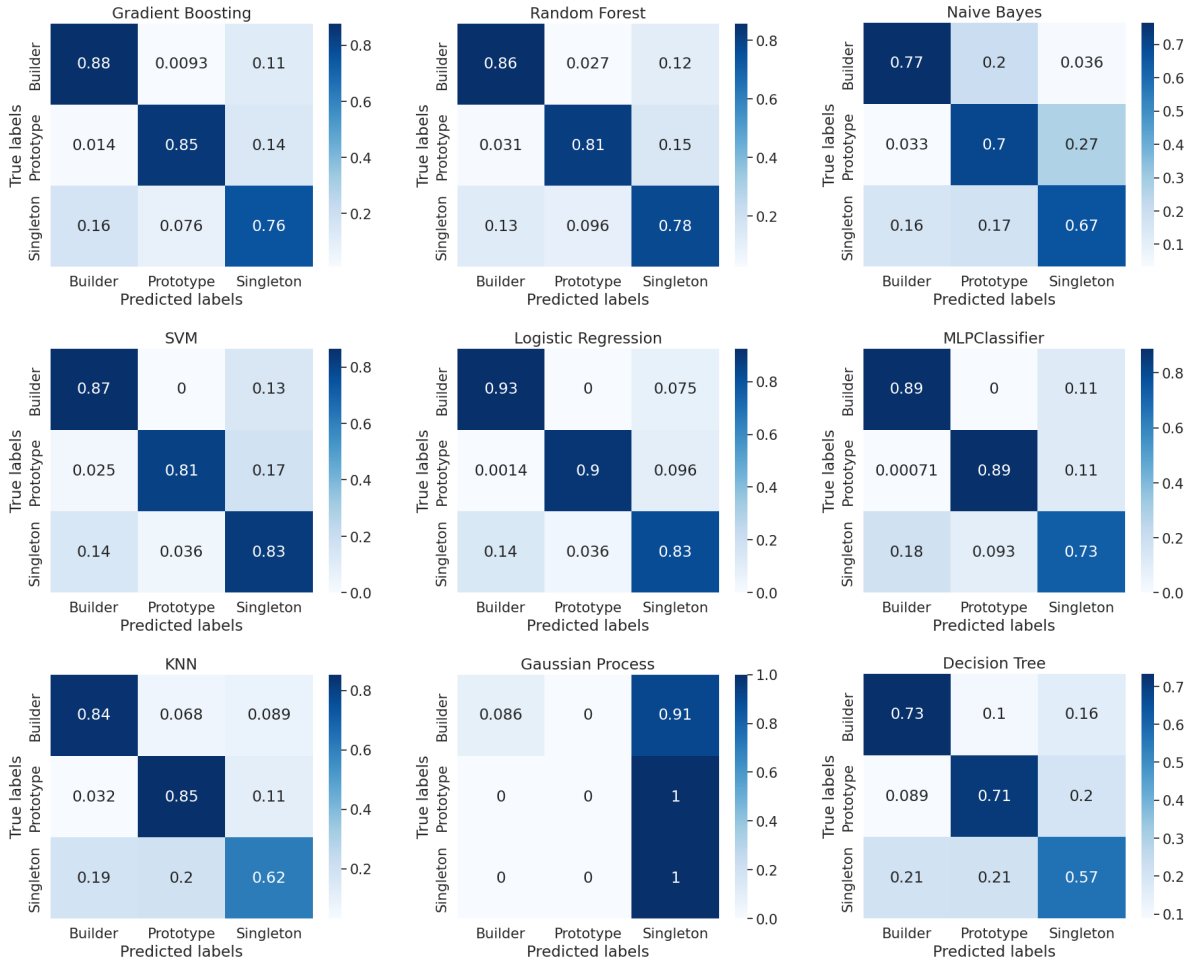

Figure 5.2: Confusion matrices showing how the baseline classifiers predicted each of the design patterns, averaged across 200 runs on CodeBERT embeddings.

Table 5.3 shows the baseline performance for different clustering algorithms. These algorithms were evaluated with 200 iterations each, and the mean was recorded for ARI, Silhouette score, F1, precision, and recall. ARI has a baseline score of 0 which is equal to clustering at random. Silhouette score measures the similarity of data points in the same cluster, and a positive score means that most data points are likely in the correct cluster. Both ARI and Silhouette score want to achieve a score close to 1 as 1 is the highest possible score for both metrics. From this baseline performance, K-Means got the highest ARI score at 0.283, and Mean Shift achieved the highest Silhouette score at 0.326. Mean Shift did, however, also get an ARI score of 0.000, meaning it is no better than random clustering. Mean Shift also got the next lowest F1-score at 0.167. BIRCH achieved the highest F1-score with a score of 0.638. The next highest clusterer, in terms of F1-score, was K-Means with an F1 of 0.335. The lowest mean F1-score was achieved by Affinity Propagation with a score of 0.069.

Table 5.3: ARI, Silhouette, F1, precision, and recall scores of the clustering algorithms with default hyperparameters on CodeBERT embeddings, averaged across all design patterns over 200 runs.

Clusterer	**ARI**	**Silhouette**	**F1**	**Precision**	**Recall**
K-Means	**0.283**	0.221	0.335	0.349	0.339
Mean Shift	0.000	**0.326**	0.163	0.109	0.323
Gaussian Mixture	0.290	0.222	0.313	0.331	0.331
BIRCH	0.263	0.167	**0.638**	**0.668**	**0.644**
Affinity Propagation	0.130	0.137	0.069	0.330	0.040
DBSCAN	NaN	NaN	NaN	NaN	NaN

Figure 5.3 below shows a t-SNE visualisation of the data, where the data has been standardised and reduced from 768 dimensions to 2. It is important to remember that, due to the high dimensionality of the original data, the t-SNE visualisation has a loss of information. The figure does show two clusters in the top right and bottom left, which are the Builder and Prototype patterns. The Singleton pattern is spread out in the left and middle and not in one singular cluster. Singleton (green) is the pattern that is most intertwined between the other patterns, and it is also the pattern that most often gets predicted wrong by the classifiers. Running the clusterers only on Builder (blue) and Prototype (orange) does yield better results for all clusterers in terms of the clustering metrics. All clusterers, except for BIRCH, also saw an improvement in the classification metrics (F1, precision, and recall) with BIRCH instead achieving a significantly worse score in those metrics. The full results of running without Singleton can be seen in Table 5.4 below.

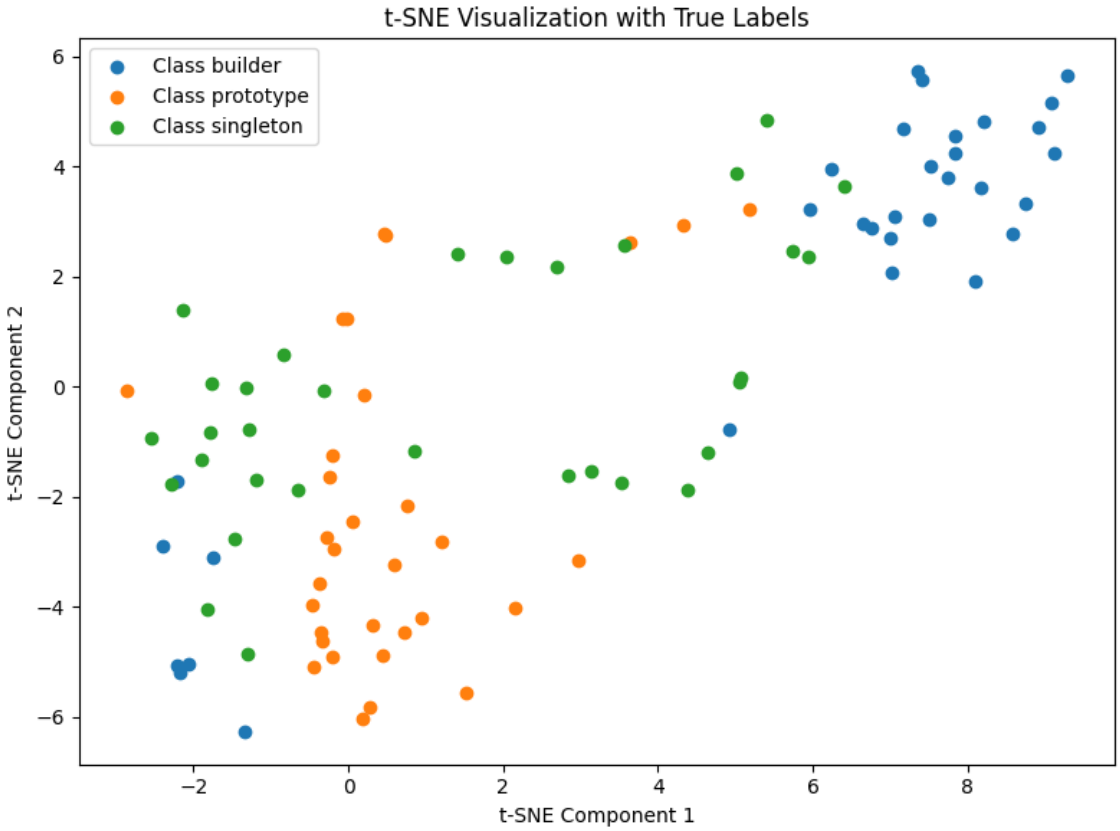

Figure 5.3: t-SNE visualisation of the CodeBERT embeddings.

Table 5.4: ARI, Silhoutte, F1, precision, and recall scores of the clustering algorithms with default hyperparameters on CodeBERT embeddings, without the Singleton pattern, averaged over 200 runs.

Clusterer	ARI	Silhouette	F1	Precision	Recall
K-Means	**0.490**	**0.278**	**0.502**	0.505	0.505
Mean Shift	0.008	0.164	0.451	**0.762**	**0.547**
Gaussian Mixture	0.484	0.278	0.475	0.479	0.479
BIRCH	0.421	0.274	0.167	0.164	0.172
Affinity Propagation	0.199	0.143	0.160	0.499	0.101
DBSCAN	NaN	NaN	NaN	NaN	NaN

5.1.2 Performance with Tuned Hyperparameters

This section presents the results of classifiers and clusterers after tuning their hyperparameters. The hyperparameters used in this section can be found in Appendix B Table B.2.

The results for 200 runs on tuned hyperparameters for the classifiers can be seen in Figure 5.4 and Table 5.5. They show that almost all classifiers increase their mean value slightly. Even with tuned hyperparameters, the performance is very similar to the baseline result, except for Gaussian Process which increased its mean from

0.219 to 0.811. It was also able to get a high F1-score of 1. Although it did achieve a much better mean score, it still had bad performances with many outliers and a low score of 0.167. This gave it the highest variance of all classifiers at 0.024, with the next highest being 0.009 from Decision Tree. The reason for the outliers is due to the algorithm overfitting. This is further explained in Section 6.1.1. Logistic Regression is still the best-performing classifier and increased its mean from 0.883 to 0.884. Moreover, Decision Tree is now the worst-performing classifier with a mean of 0.683 due to the high increase in performance of Gaussian Process. SVM and Naive Bayes had the same performance as with default hyperparameters.

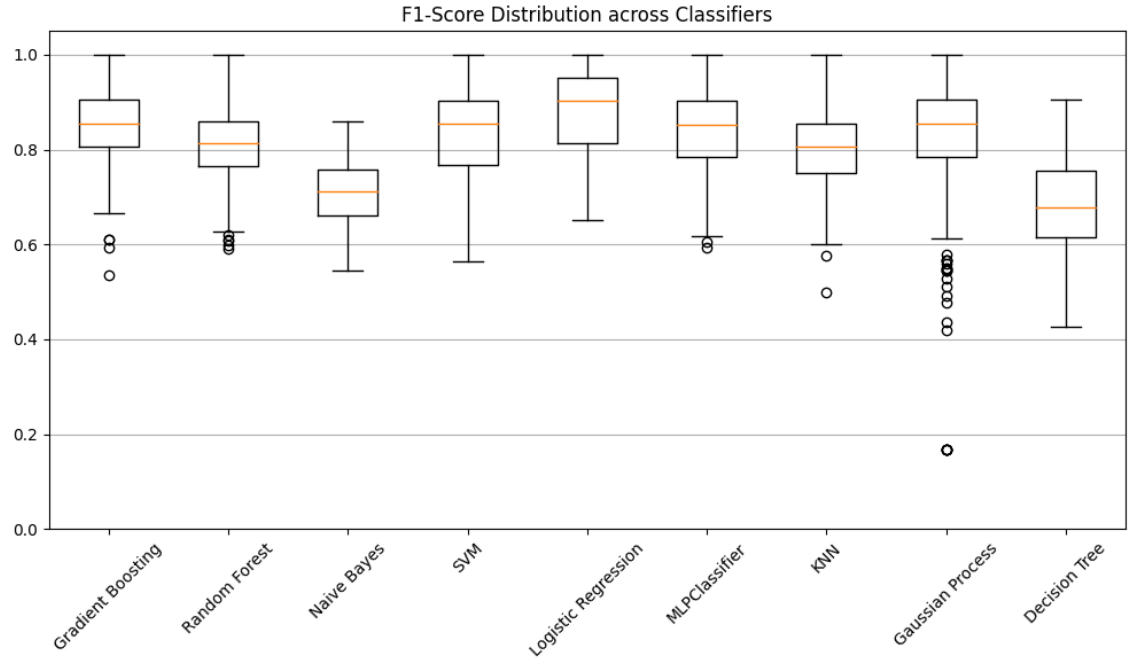

Figure 5.4: Box plot showing the F1-scores of the classifiers across 200 runs with tuned hyperparameters on CodeBERT embeddings, averaged across all patterns.

Table 5.5: Table depicting mean, variance, highest, and lowest F1-scores obtained from running each classifier with tuned hyperparameters on CodeBERT embeddings, averaged across all design patterns over 200 runs.

Classifier	**Mean**	**Variance**	**Highest**	**Lowest**
MLPClassifier	0.835	0.007	**1.000**	0.595
Gradient Boosting	0.851	0.007	**1.000**	0.535
Random Forest	0.818	0.006	**1.000**	0.592
Decision Tree	0.683	0.009	0.904	0.426
KNN	0.795	0.007	**1.000**	0.500
Naive Bayes	0.707	0.006	0.860	0.544
Gaussian Process	0.811	0.024	**1.000**	0.167
Logistic Regression	**0.884**	**0.005**	**1.000**	**0.651**
SVM	0.832	0.005	**1.000**	0.563

Table 5.6 and Figure 5.5 show the performance of the tuned classifiers on individual design patterns. The confusion matrices mostly look the same as baseline except for Gaussian Process which now correctly can identify Builder and Prototype to a much higher degree. Singleton is still the pattern that most often gets predicted wrong, and the other patterns get most often wrongly predicted as being.

Table 5.6: Table depicting F1-score, precision, and recall obtained from running each tuned classifier 200 times with different training and test sets across the Singleton, Prototype, and Builder design patterns.

Classifier	**Singleton** F1, P, R	**Prototype** F1, P, R	**Builder** F1, P, R
MLPClassifier	0.76, 0.84, 0.71	0.89, 0.87, **0.91**	0.86, 0.85, 0.89
Gradient Boosting	0.79, 0.79, 0.82	0.88, 0.94, 0.85	0.88, 0.88, 0.89
Random Forest	0.76, 0.74, 0.81	0.84, 0.89, 0.80	0.86, **0.89**, 0.85
Decision Tree	0.62, 0.65, 0.62	0.73, 0.76, 0.74	0.73, 0.76, 0.74
KNN	0.72, 0.79, 0.67	0.82, 0.80, 0.87	0.85, 0.84, 0.86
Naive Bayes	0.67, 0.71, 0.67	0.67, 0.67, 0.70	0.78, 0.81, 0.77
Gaussian Process	0.75, 0.75, 0.80	0.86, 0.93, 0.83	0.82, 0.85, 0.84
Logistic Regression	**0.83**, **0.85**, **0.83**	**0.93**, **0.97**, 0.90	**0.90**, **0.89**, **0.93**
SVM	0.77, 0.75, 0.83	0.87, **0.97**, 0.81	0.85, 0.86, 0.87

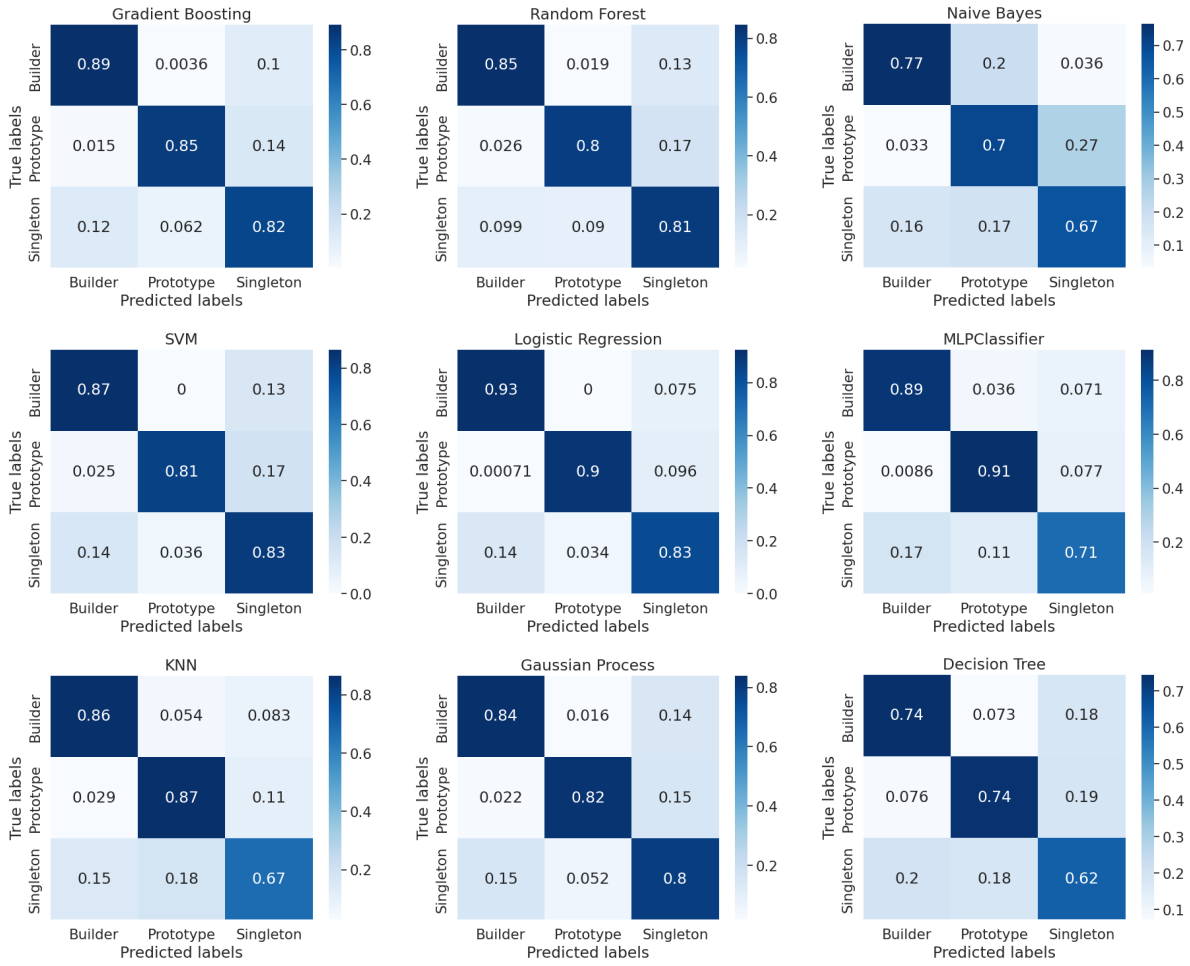

Figure 5.5: Confusion matrices showing how the tuned classifiers predicted each of the design patterns, averaged across 200 runs on CodeBERT embeddings.

Table 5.7 presents the results for the clusterers after hyperparameter tuning. Hyperparameter tuning did not have any major effect on performance. Mean Shift, BIRCH, and Affinity Propagation did not improve as their default hyperparameters were the best. Their default hyperparameters will remain the best hyperparameters for all future tests as well. K-Means and Gaussian Mixture are the only clusterers where hyperparameter tuning had an effect. Both of them had a slight improvement in ARI, with K-Means increasing from 0.283 to 0.286. Gaussian Mixture increased from 0.290 to 0.292. Both clusterers also achieved a higher F1-score, with K-Means going from 0.335 to 0.336 and Gaussian Mixture increasing from 0.313 to 0.328. Keep in mind that the clusterers were tuned around maximising the ARI score, meaning a decrease in other scores may occur.

Table 5.7: ARI, Silhouette, F1, precision, and recall scores of the clustering algorithms with hyperparameter tuning on CodeBERT embeddings, averaged across all design patterns over 200 runs.

Clusterer	**ARI**	**Silhouette**	**F1**	**Precision**	**Recall**
K-Means	0.286	0.221	0.336	0.350	0.341
Mean Shift	0.000	**0.326**	0.163	0.109	0.323
Gaussian Mixture	**0.292**	0.210	0.328	0.330	0.330
BIRCH	0.263	0.167	**0.638**	**0.668**	**0.644**
Affinity Propagation	0.130	0.137	0.069	0.330	0.040
DBSCAN	NaN	NaN	NaN	NaN	NaN

RQ1 Summary: Logistic Regression was the best-performing algorithm, with a mean F1-score of 0.884. Other algorithms that also achieved a high F1-score, compared to previous studies, were MLP-Classifier, Gradient Boosting, and SVM. Decision Tree and Naive Bayes were the two algorithms with the lowest scores, achieving a mean F1-score around 0.7. Hyperparameter tuning only gave a minor increase in performance to most classifiers. Clustering did not perform well and was outperformed by the classifiers.

5.2 RQ2: Influence of Context on Classifier Performance

This section presents the results relevant to answering RQ2. The section is divided into three subsections, one for each sub RQ, where the results relevant to each sub RQ are presented. The clusterers results will only be presented with the ARI and Silhouette metrics due to their overall bad performance in the classification metrics. The clustering algorithms are still kept and tested in case their performance is affected by the different input data tested for RQ2.

5.2.1 RQ2.1: Performance Variance Between Design Patterns

This section presents the results relevant to RQ2.1. All results presented in this section use data from seven different design patterns, including Singleton, Builder, Prototype, Adapter, Bridge, Observer, and Strategy. All design pattern examples were written in Java, and all classifiers and clusterers used embeddings generated by CodeBERT. Baseline performance is presented in Section 5.2.1.1 and performance with tuned hyperparameters is presented in Section 5.2.1.2.

5.2.1.1 Baseline Performance

Figure 5.6 and Table 5.8 show the results of nine different classifiers across 200 runs. These results are the average F1-score over the seven design patterns evaluated. Most of the classifiers are in the range of 0.43 to 0.74. The results here are worse than in RQ1 due to an increase in the number of classes. The best-performing classifier was Logistic Regression and the worst-performing was Gaussian Process. When using default hyperparameters, Gaussian Process achieved the lowest score by only predicting a single class, which results in most of the predictions being wrong. Furthermore, the overall F1-score is lower when training the classifiers on seven design patterns instead of three. None of the classifiers achieved an F1-score of 1 in any of the 200 runs. The highest F1-score achieved was 0.875 by Logistic Regression. The lowest was 0.036 by Gaussian Process.

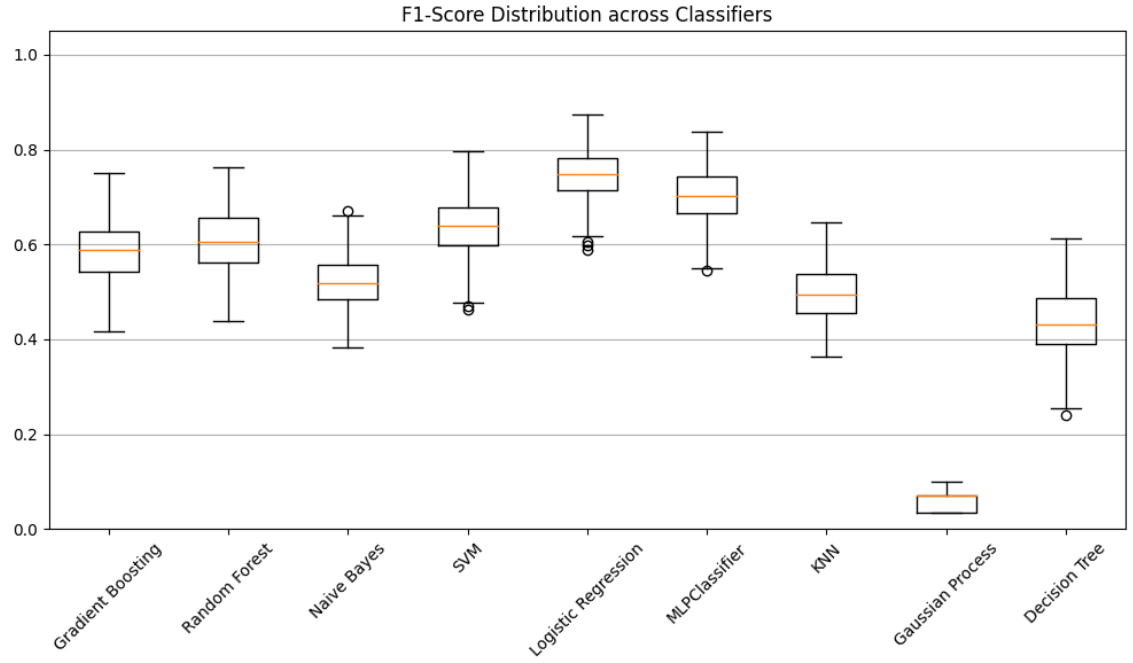

Figure 5.6: Box plot showing F1-scores of the classifiers across 200 runs with default hyperparameters on CodeBERT embeddings, averaged across all patterns.

Table 5.8: Table depicting mean, variance, highest, and lowest F1-scores obtained from running each classifier with default hyperparameters on CodeBERT embeddings, results averaged across all seven design patterns over 200 runs.

Classifier	Mean	Variance	Highest	Lowest
MLPClassifier	0.704	0.003	0.838	0.545
Gradient Boosting	0.585	0.004	0.752	0.418
Random Forest	0.608	0.004	0.763	0.438
Decision Tree	0.438	0.005	0.612	0.239
KNN	0.498	0.004	0.647	0.364
Naive Bayes	0.522	0.003	0.671	0.382
Gaussian Process	0.057	**0.000**	0.100	0.036
Logistic Regression	**0.745**	0.003	**0.875**	**0.588**
SVM	0.635	0.004	0.797	0.462

F1, precision, and recall score for the individual patterns can be seen in Appendix A in Table A.1 and Table A.2. The patterns average F1, precision, and recall score across all nine classifiers can be seen in Table 5.9 below. The pattern with the worst individual performance is Adapter, with an average F1-score of 0.292 across all classifiers. The pattern with the best individual performance is Prototype, with an average F1-score of 0.727 across all classifiers. The Structural patterns had the worst performance with an average F1-score of 0.376. The behavioral patterns with an average F1-score of 0.536, and the creational patterns had the overall best performance with an average F1-score of 0.632. The F1-score for Singleton, Prototype, and Builder is around 0.1 points lower when evaluating seven design patterns than

when evaluating three design patterns. This is due to the increase in classes in the classification task, as when the number of labels increases, the chance of a misclassification also increases.

Table 5.9: Average F1-score, precision, and recall for all seven design patterns across 200 runs and nine classifiers, with default hyperparameters on CodeBERT embeddings.

Pattern	**F1 Score**	**Precision**	**Recall**
Behavioral Patterns	**0.536**	**0.570**	**0.578**
Observer	0.657	0.686	0.668
Strategy	0.415	0.453	0.488
Creational Patterns	**0.632**	**0.678**	**0.626**
Builder	0.598	0.644	0.610
Prototype	0.727	0.756	0.722
Singleton	0.570	0.635	0.547
Structural Patterns	**0.376**	**0.376**	**0.400**
Adapter	0.292	0.296	0.309
Bridge	0.461	0.457	0.492

Table 5.10 below shows the ARI and Silhouette scores for the clusterers. All ARI scores got lowered compared to when running with only Singleton, Builder, and Prototype, with the highest ARI achieved here being 0.157 by Gaussian Mixture. All Silhouette scores decreased except for Mean Shift which increased from 0.326 to 0.372. DBSCAN only generated a single cluster and could therefore not generate any ARI or Silhouette scores.

Table 5.10: ARI and Silhouette score of clustering algorithms with default hyperparameters on CodeBERT embeddings, averaged across all seven design patterns over 200 runs.

Clusterer	**Adjusted Rand Index**	**Silhouette score**
K-Means	0.156	0.116
Mean Shift	0.000	**0.372**
Gaussian Mixture	**0.157**	0.116
BIRCH	0.154	0.105
Affinity Propagation	0.124	0.090
DBSCAN	NaN	NaN

Figure 5.7 below shows a t-SNE visualisation of the full data set. This is a two-dimensional visualisation of data that has 768 dimensions, meaning there is a loss of data in this visualisation. Prototype (purple), Singleton (brown), and Builder (green) have similar clustering as in Figure 5.3. They are all part of the left cluster, with some builder data in the top right, and they are all creational patterns. Observer (red), Strategy (pink), Bridge (orange), and Adapter (blue) are all clustered in the middle right. The Observer data points are part of the large cluster in the middle right, but they are clustered close together. Strategy, Bridge, and Adapter are all spread out in the large cluster and do not show a clear singular cluster. Strategy, Bridge, and Adapter are also the three patterns with the worst individual performance. The behavioral and structural patterns are not visually separated in the same way that the creational patterns are.

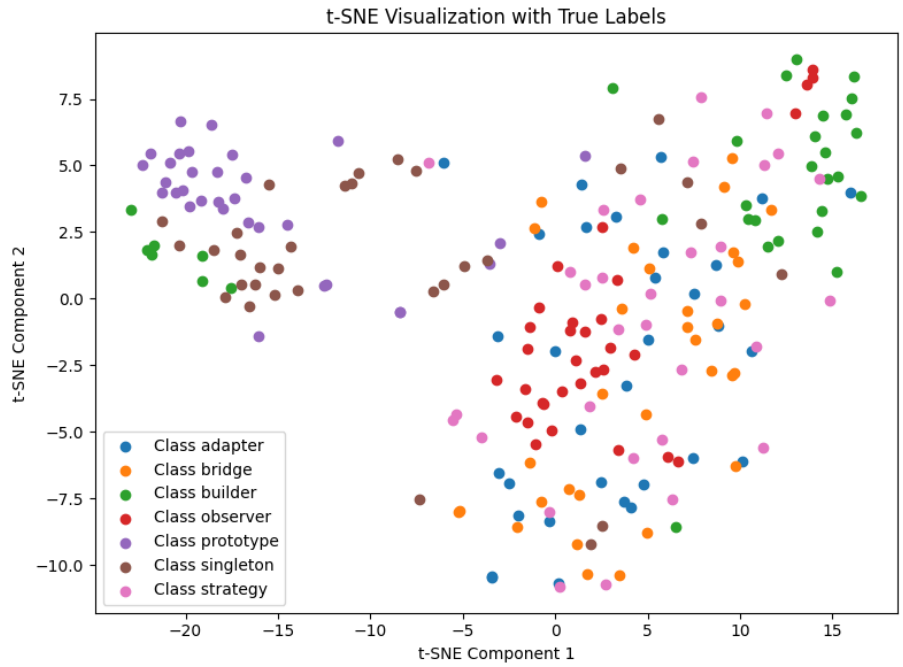

Figure 5.7: t-SNE visualisation of the CodeBERT embeddings of the seven design patterns.

5.2.1.2 Performance with Tuned Hyperparameters

Figure 5.8 and Table 5.11 show the result of the tuned hyperparameters across 200 runs. The hyperparameters used can be found in Table B.3 in Appendix B. All classifiers had an improvement in mean except for Naive Bayes, which has the same mean. Gaussian Process had the largest increase in performance, with a mean of 0.735, up from a mean of 0.057. The highest mean F1 was achieved by Logistic Regression with a mean of 0.751, a slight increase from the baseline of 0.745. SVM got the highest F1-score, with one of the runs achieving an F1 of 0.879, an increase of 0.004 from the baseline best. The worst-performing classifier is now Decision Tree, with an average F1-score of 0.461, a small increase from the previous 0.438. Gradient Boosting also saw a big increase in performance, going from an average F1 of 0.585 to 0.716.

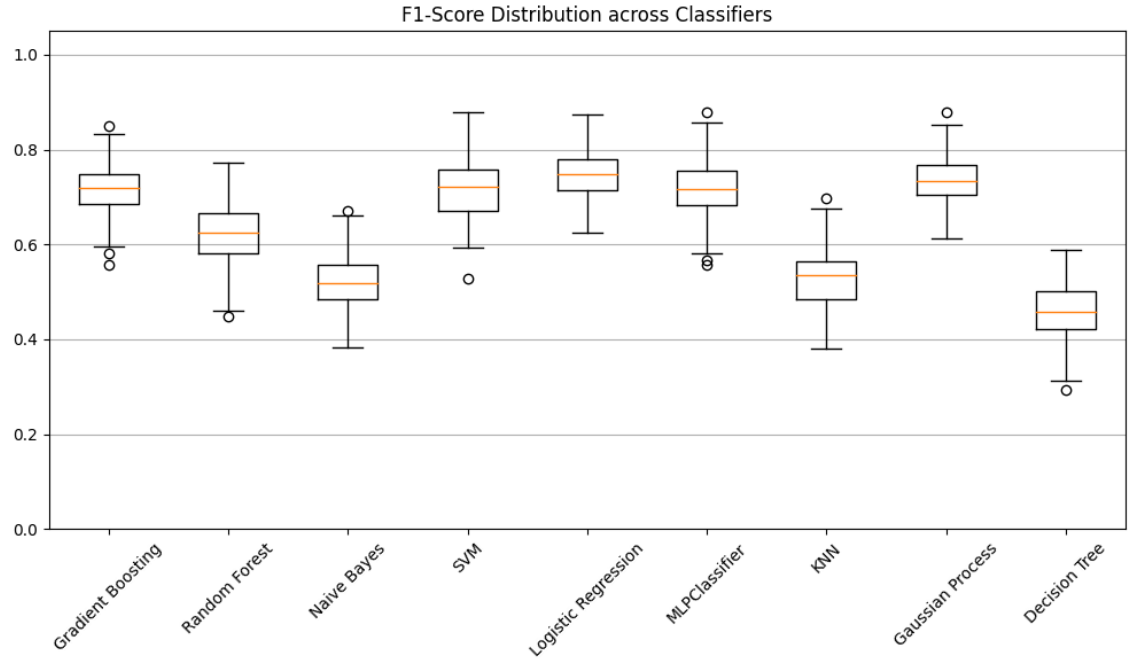

Figure 5.8: Box plot showing F1-scores of the classifiers across 200 runs with hyperparameter tuning on CodeBERT embeddings, averaged across all patterns.

Table 5.11: Table depicting mean, variance, highest, and lowest F1-scores obtained from running each classifier 200 times with tuned hyperparameters on CodeBERT embeddings, results averaged across all design patterns.

Classifier	Mean	Variance	Highest	Lowest
MLPClassifier	0.721	0.003	0.878	0.557
Gradient Boosting	0.715	0.003	0.849	0.558
Random Forest	0.623	0.004	0.774	0.448
Decision Tree	0.459	0.004	0.589	0.292
KNN	0.531	0.004	0.697	0.380
Naive Bayes	0.522	0.003	0.671	0.382
Gaussian Process	0.735	**0.002**	0.878	0.612
Logistic Regression	**0.751**	**0.002**	0.875	**0.626**
SVM	0.719	0.003	**0.879**	0.527

The individual pattern performance across all nine classifiers can be seen in Table A.3 and Table A.4 in Appendix A. The average F1, precision, and recall scores for all seven patterns across all nine classifiers can be seen in Table 5.12 below. Adapter remained the pattern with the lowest F1-score at 0.350; the next lowest also remained the same, with Strategy getting an F1-score of 0.523. The pattern with the highest and next-highest F1-score was also the same as baseline, with Prototype achieving a score of 0.839 and Observer achieving a score of 0.801. The performance order of the different pattern types remains the same. They did, however, all get an increase of about 0.1 average F1-score. The structural patterns had the lowest average F1 at 0.473. Behavioral patterns had an average F1-score of 0.662, and creational patterns had the highest average F1-score at 0.742.

Table 5.12: Average F1-score, precision, and recall for all seven design patterns across 200 runs and nine classifiers, with tuned hyperparameters on CodeBERT embeddings.

Pattern	F1 Score	Precision	Recall
Behavioral Patterns	**0.662**	**0.684**	**0.677**
Observer	0.801	0.800	0.841
Strategy	0.523	0.569	0.514
Creational Patterns	**0.742**	**0.731**	**0.743**
Builder	0.700	0.584	0.721
Prototype	0.839	0.862	0.837
Singleton	0.686	0.746	0.671
Structural Patterns	**0.473**	**0.498**	**0.484**
Adapter	0.350	0.405	0.342
Bridge	0.596	0.591	0.626

ARI and Silhouette scores for the clusterers after hyperparameter tuning can be seen in Table 5.13. The only clusterer whose score changed was Gaussian Mixture. Its ARI increased by 0.004 to 0.161, and its Silhouette score decreased by 0.003 to

0.113. DBSCAN could not generate any results. Some clusterers could not use their best combination of hyperparameters, as that was achieved by creating one cluster for each data point.

Table 5.13: ARI and Silhouette score of the clustering algorithms with hyperparameter tuning on CodeBERT embeddings, averaged across all seven design patterns over 200 runs.

Clusterer	Adjusted Rand Index	Silhouette score
K-Means	0.156	0.116
Mean Shift	0.000	**0.372**
Gaussian Mixture	**0.161**	0.113
BIRCH	0.154	0.105
Affinity Propagation	0.124	0.090
DBSCAN	NaN	NaN

RQ2.1 Summary: Increasing the number of patterns and possible classifications did lower the performance of the classification and clustering algorithms. Logistic Regression was the best-performing algorithm again, with an F1-score of 0.751. MLPClassifier, SVM, Gradient Boosting, and Gaussian Process also performed well, compared to results from previous studies [7], [11]. Decision Tree had the worst performance with an F1-score of 0.459. Hyperparameter tuning gave a slight increase in performance, more than in RQ1, but not a large amount. The structural patterns achieved the lowest F1-score, followed by behavioral with creational patterns achieving the highest F1-score. Prototype, a creational pattern, got the highest F1-score out of all seven patterns. The pattern with the lowest F1-score was Adapter, a structural pattern. The clustering algorithms did not show good performance with or without tuned hyperparameters, compared to the classification algorithms.

5.2.2 RQ2.2: Performance Variance of Language Models

This section presents the results relevant to answering RQ2.2. All results presented in this section are with tuned hyperparameters. Baseline results with default hyperparameters are not presented as interesting results to compare between CodeBERT, GPT-2, and Instructor-xl; they are the best possible results, and that is achieved with tuned hyperparameters. DBSCAN will no longer be part of the clusterer results as it could never generate any ARI or Silhouette scores. This research question only focuses on the creational design patterns: Singleton, Prototype, and Builder. The design pattern examples were written in Java. Results for GPT-2 are presented in Section 5.2.2.1, and results for Instructor-xl are presented in Section 5.2.2.2. The comparison of results between GPT-2, Instructor-xl, and CodeBERT is presented in Section 5.2.2.3.

5.2.2.1 GPT-2

This section presents the results obtained from the GPT-2 embeddings of the Singleton, Builder, and Prototype data written in Java. The values for the hyperparameters can be found in Table B.4 in Appendix B.

Figure 5.9 and Table 5.14 show the results for nine classifiers trained on embeddings from the GPT-2 LLM. In these results, SVM was the best-performing classifier, with a mean F1-score of 0.883, followed by the next two classifiers, Random Forest (0.875) and Gradient Boosting (0.870). The worst-performing classifier was Decision Tree with a F1-score of 0.741, which also had the highest variance with 0.010.

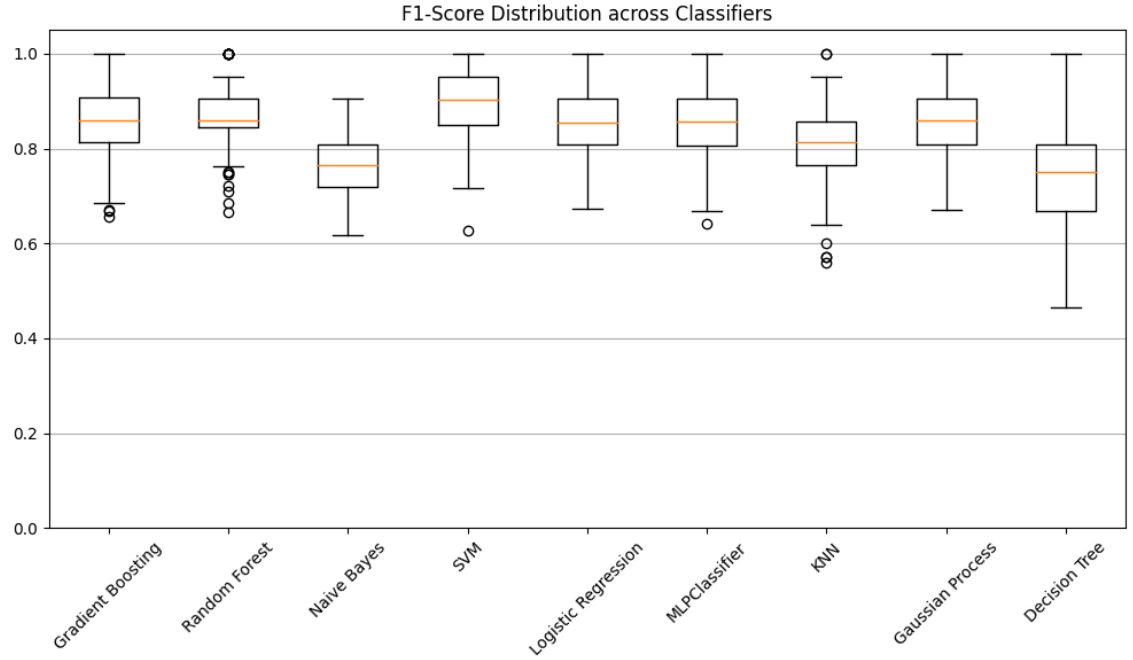

Figure 5.9: Box plot showing F1-scores of the classifiers across 200 runs with hyperparameter tuning on GPT-2 embeddings, averaged across all patterns.

Table 5.14: Table depicting mean, variance, highest, and lowest F1-scores obtained from running each classifier with tuned hyperparameters on GPT-2 embeddings, results averaged across all design patterns over 200 runs.

Classifier	**Mean**	**Variance**	**Highest**	**Lowest**
MLPClassifier	0.855	0.006	**1.000**	0.642
Gradient Boosting	0.870	0.005	**1.000**	0.656
Random Forest	0.875	0.004	**1.000**	0.665
Decision Tree	0.741	0.010	**1.000**	0.465
KNN	0.818	0.006	**1.000**	0.560
Naive Bayes	0.764	**0.003**	0.905	0.619
Gaussian Process	0.865	0.006	**1.000**	0.670
Logistic Regression	0.854	0.006	**1.000**	**0.672**
SVM	**0.883**	0.005	**1.000**	0.628

Table 5.15 and Figure 5.10 show the F1-score, precision, recall, and accuracy for each individual design pattern. The table shows that SVM performs best on Singleton and Builder, while Random Forest achieves the highest score on Prototype. Moreover, the confusion matrix shows that all classifiers perform very similar to each other in terms of miss-classification. Singleton is the pattern that most often gets predicted wrong by most classifiers. When Builder and Prototype get predicted wrong, it is most often that they are predicted as being Singleton.

Table 5.15: Table depicting F1-score, precision, and recall obtained from running each tuned classifier 200 times with different training and test sets across the Singleton, Prototype, and Builder embeddings from GPT-2.

Classifier	**Singleton** F1, P, R	**Prototype** F1, P, R	**Builder** F1, P, R
MLPClassifier	0.78, **0.82**, 0.77	0.92, 0.94, **0.90**	0.87, 0.85, **0.90**
Gradient Boosting	0.81, **0.82**, 0.83	0.92, 0.96, 0.89	0.88, 0.88, 0.89
Random Forest	0.82, **0.82**, 0.85	**0.93**, **0.98**, **0.90**	0.87, 0.88, 0.88
Decision Tree	0.65, 0.70, 0.64	0.82, 0.84, 0.83	0.75, 0.76, 0.77
KNN	0.75, 0.81, 0.73	0.87, 0.90, 0.85	0.84, 0.81, 0.88
Naive Bayes	0.69, 0.69, 0.72	0.85, 0.93, 0.79	0.75, 0.74, 0.78
Gaussian Process	0.80, **0.82**, 0.80	0.92, 0.95, **0.90**	0.88, 0.88, **0.90**
Logistic Regression	0.78, 0.81, 0.77	0.92, 0.95, **0.90**	0.87, 0.85, **0.90**
SVM	**0.84**, 0.79, **0.91**	0.92, 0.97, 0.89	**0.89**, **0.94**, 0.85

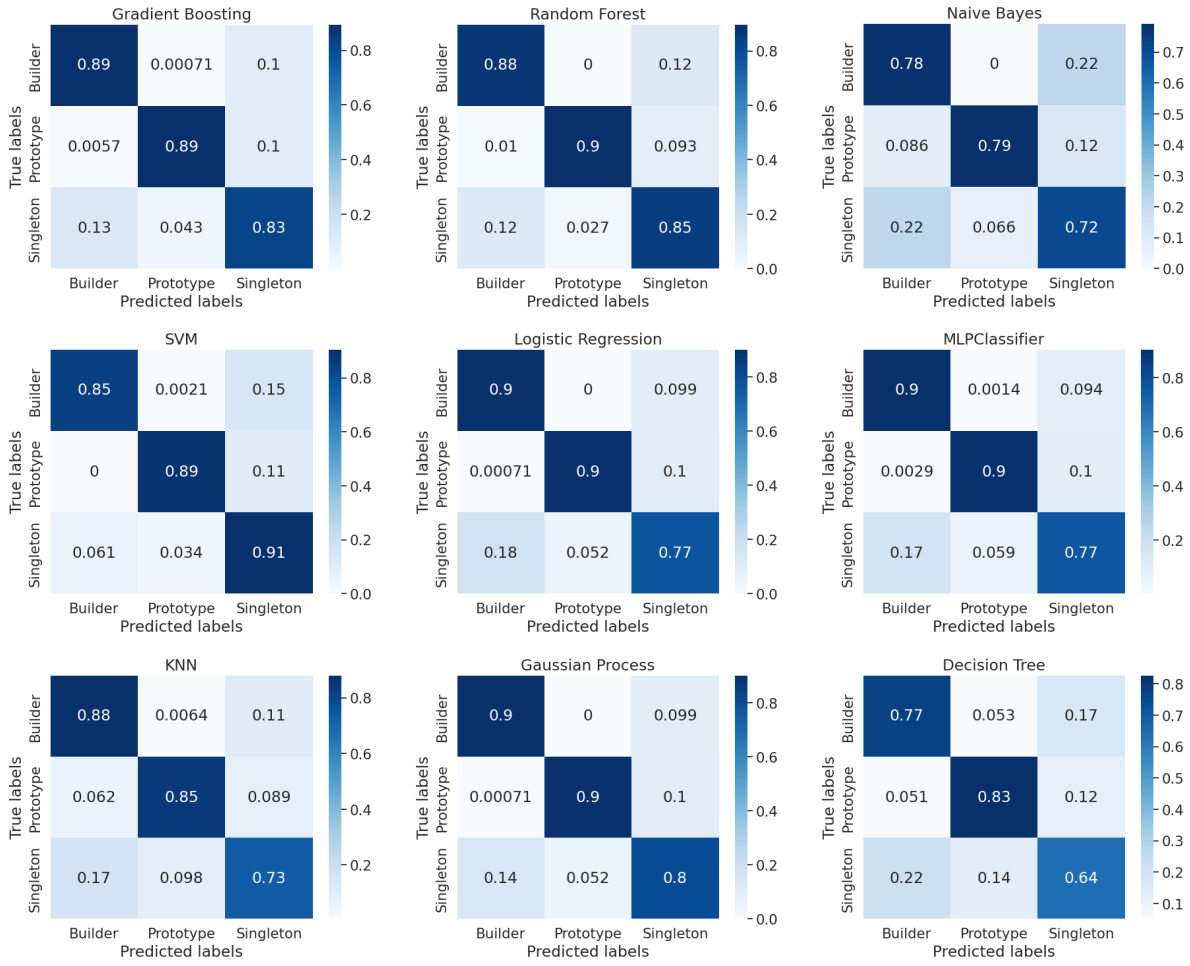

Figure 5.10: Confusion matrices showing how the tuned classifiers predicted each of the design patterns, averaged across 200 runs on GPT-2 embeddings.

Table 5.16 below presents the results for the clusterers on the GPT-2 embeddings. All clusterers had tuned hyperparameters, but Mean Shift, BIRCH, and Affinity Propagation had their default values as the best hyperparameters. The highest ARI score was achieved by Affinity Propagation, with a score of 0.380. The highest Silhouette score was 0.245 by Mean Shift. The lowest ARI score was Mean Shift with 0.164, and the lowest Silhouette score was 0.145 by Affinity Propagation.

Table 5.16: ARI and Silhouette score of the clustering algorithms with hyperparameter tuning on GPT-2 embeddings, averaged across all design patterns over 200 runs.

Clusterer	Adjusted Rand Index	Silhouette score
K-Means	0.348	0.232
Mean Shift	0.164	**0.245**
Gaussian Mixture	0.294	0.226
BIRCH	0.358	0.233
Affinity Propagation	**0.380**	0.145

Figure 5.11 shows a t-SNE visualisation of the GPT-2 embeddings. Like the CodeBERT embeddings, this is 768-dimensional data that is reduced down to two dimensions, so there is a loss of information. The visualisation shows that there are two clusters of data, one on top left and one on bottom right. The majority of the Prototype data (orange) is in the bottom right cluster, with a few data points in the other cluster. The Builder data (blue) is mostly in the top left cluster, with a few data points in the bottom right. Singleton (green) has the majority of its data in the bottom right cluster but still has a significant amount in the top left cluster and is more spread out compared to Prototype and Builder.

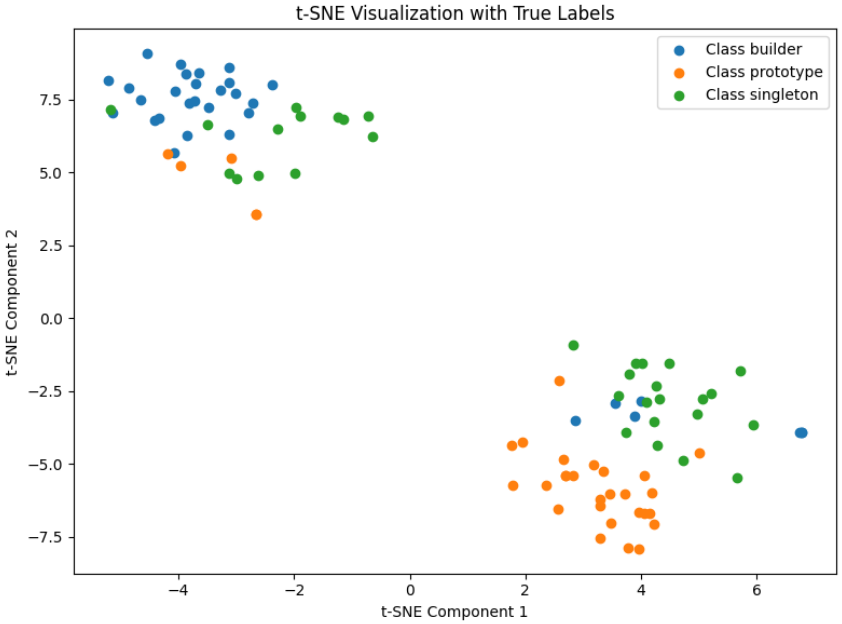

Figure 5.11: t-SNE visualisation of the GPT-2 embeddings.

5.2.2.2 Instructor-xl

This section presents the results acquired for the Instructor-xl embeddings of the Singleton, Builder, and Prototype data written in Java. The values for the hyperparameters can be found in Table B.5 in Appendix B.

Figure 5.12 and Table 5.17 illustrate the results for the classifiers trained on embeddings from Instructor-xl. Random Forest achieves the highest mean F1-score of 0.856. The two next-best classifiers were Logistic Regression with a score of 0.836 and Naive Bayes with a score of 0.833. The worst-performing classifier was Decision Tree with a F1-score of 0.625, with the next lowest mean being 0.740 by KNN.

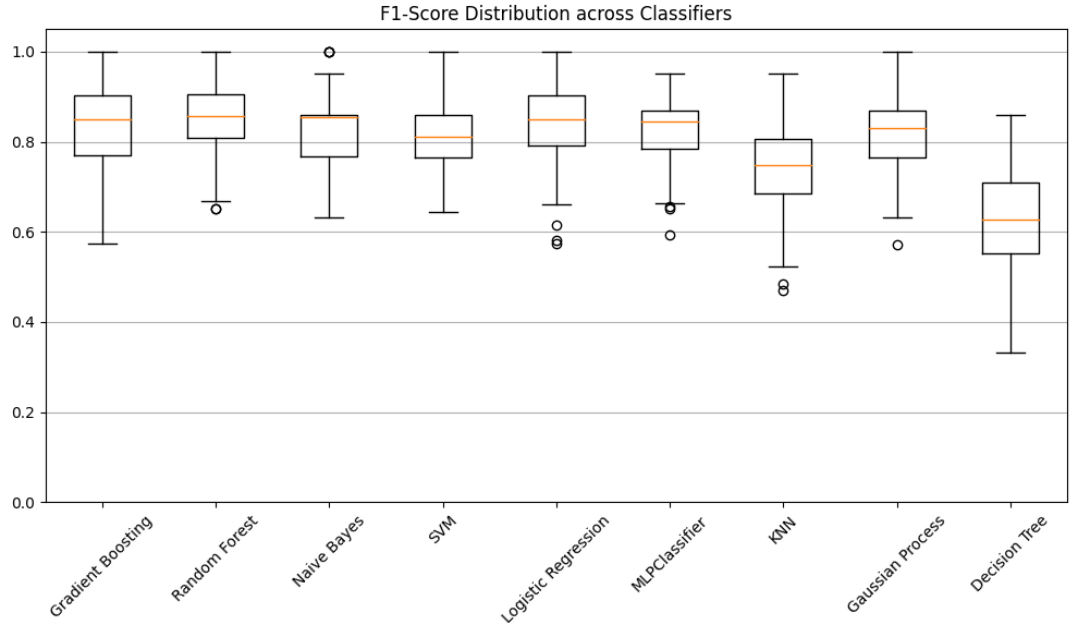

Figure 5.12: Box plot showing the F1-scores of the classifiers across 200 runs with hyperparameter tuning on Instructor-xl embeddings, averaged across all patterns.

Table 5.17: Table depicting mean, variance, highest, and lowest F1-scores obtained from running each classifier with tuned hyperparameters on Instructor-xl embeddings, results averaged across all design patterns over 200 runs

Classifier	Mean	Variance	Highest	Lowest
MLPClassifier	0.828	**0.005**	0.952	0.593
Gradient Boosting	0.828	0.007	**1.000**	0.574
Random Forest	**0.856**	**0.005**	**1.000**	**0.652**
Decision Tree	0.625	0.013	0.860	0.333
KNN	0.740	0.008	0.952	0.470
Naive Bayes	0.833	**0.005**	**1.000**	0.633
Gaussian Process	0.824	0.006	**1.000**	0.571
Logistic Regression	0.836	0.006	**1.000**	0.574
SVM	0.821	**0.005**	**1.000**	0.644

Table 5.18 and Figure 5.13 show how the classifiers performed on the individual patterns. Prototype is the pattern that was the easiest for the classifiers to identify with all classifiers, except for Decision Tree, which achieved an F1-score over 0.90 on it. Singleton was the hardest pattern to identify from the Instructor-xl embeddings, with most classifiers achieving an F1 around 0.75. Builder is in the middle between Singleton and Prototype, with most classifiers achieving an F1-score around 0.80.

Table 5.18: Table depicting F1-score, precision, and recall obtained from running each tuned classifier 200 times with different training and test sets across the Singleton, Prototype, and Builder embeddings from Instructor-xl.

Classifier	**Singleton** F1, P, R	**Prototype** F1, P, R	**Builder** F1, P, R
MLPClassifier	0.75, 0.81, 0.72	0.92, 0.92, **0.94**	**0.82**, **0.82**, 0.84
Gradient Boosting	0.74, 0.79, 0.73	0.94, 0.97, 0.92	0.80, 0.79, 0.84
Random Forest	**0.80**, **0.82**, **0.81**	**0.95**, 0.99, 0.91	**0.82**, 0.81, **0.85**
Decision Tree	0.56, 0.60, 0.56	0.75, 0.76, 0.76	0.57, 0.59, 0.58
KNN	0.63, 0.72, 0.60	0.92, 0.92, 0.93	0.68, 0.67, 0.73
Naive Bayes	0.78, 0.80, 0.79	**0.95**, **1.00**, 0.90	0.78, 0.78, 0.81
Gaussian Process	0.74, 0.77, 0.73	0.94, 0.97, 0.92	0.80, 0.79, 0.84
Logistic Regression	0.76, 0.80, 0.75	0.94, 0.97, 0.92	0.81, 0.81, 0.84
SVM	0.75, 0.77, 0.77	0.94, 0.99, 0.91	0.78, 0.80, 0.79

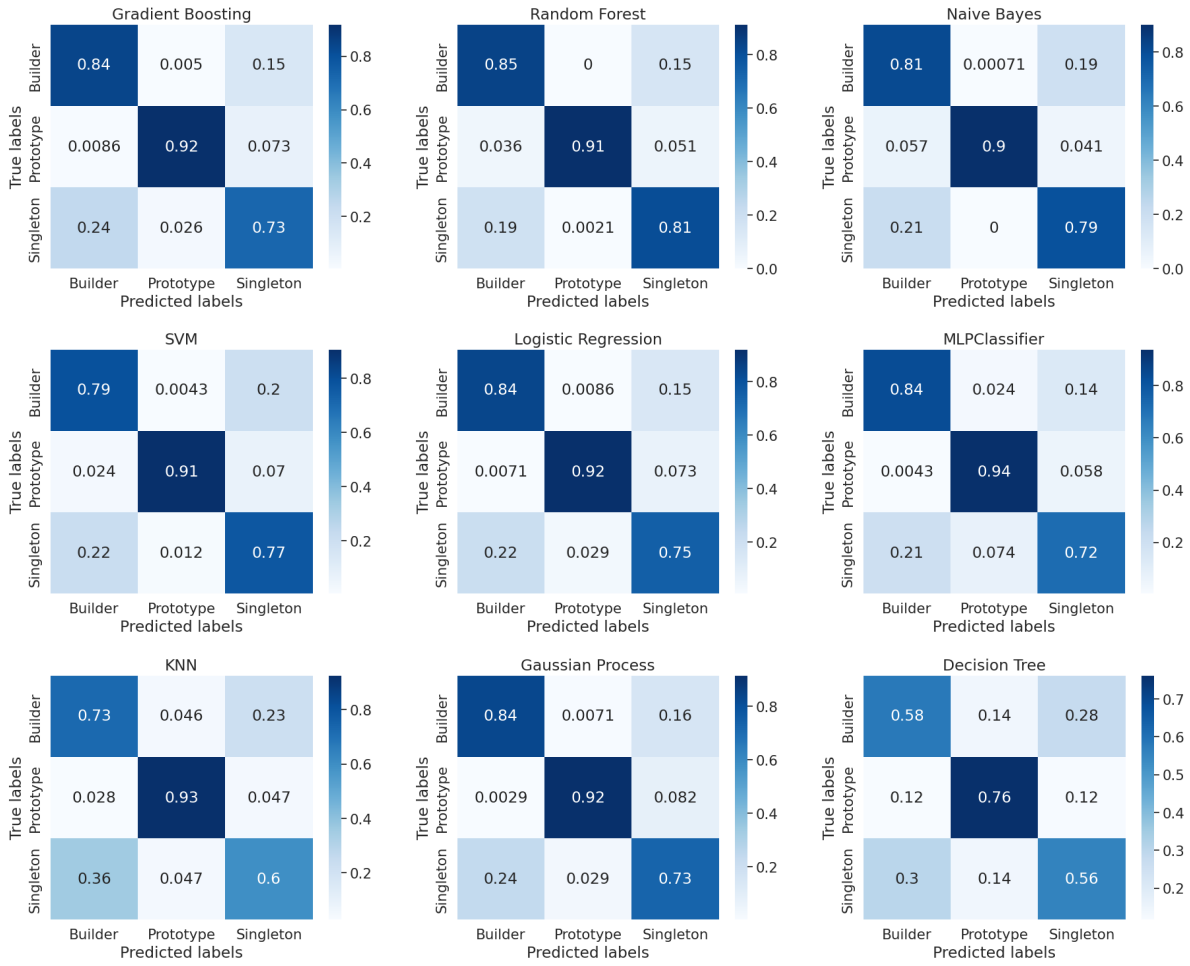

Figure 5.13: Confusion matrices showing how the tuned classifiers predicted each of the design patterns from the Instructor-xl embeddings, results averaged across 200 runs.

Table 5.19 shows the ARI and Silhouette scores for the clusterers on the Instructor-xl embeddings. K-Means and BIRCH got the highest ARI scores at 0.494. The highest Silhouette score was achieved by Mean Shift, with a score of 0.160. Mean Shift also got the lowest ARI score at 0.000, meaning it is no better than just random clustering. The lowest Silhouette score was attained by Gaussian Mixture, with a score of 0.058.

Table 5.19: ARI and Silhouette score of the clustering algorithms with hyper-parameter tuning on Instructor-xl embeddings, averaged across all design patterns over 200 runs.

Clusterer	Adjusted Rand Index	Silhouette score
K-Means	**0.494**	0.074
Mean Shift	0.000	**0.160**
Gaussian Mixture	0.413	0.058
BIRCH	**0.494**	0.076
Affinity Propagation	0.125	0.098

Figure 5.14 below shows the t-SNE visualisation of the Instructor-xl embeddings. The Instructor-xl embeddings are also of 768 dimensions, and the visualisation has a loss of information. The visualisation shows that Prototype (orange) is in its own cluster in the bottom right. Singleton (green) and Builder (blue) are at the top and left parts and are more spread out than Prototype. They do, however, still occupy their own space in the graph, with some data points intertwined in the middle.

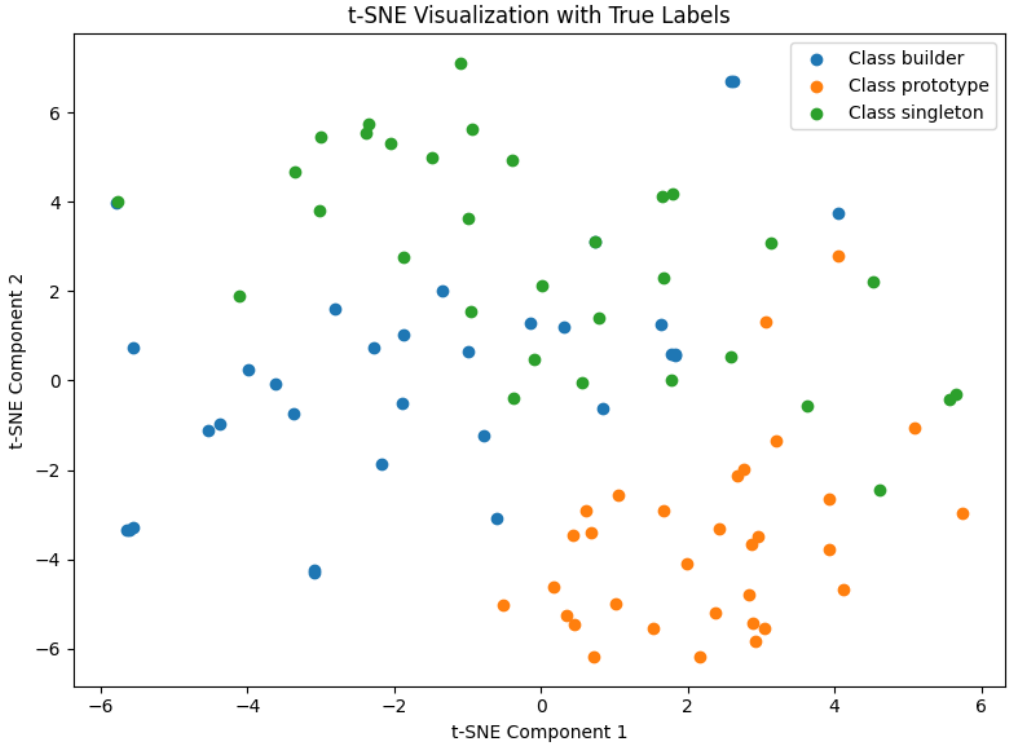

Figure 5.14: t-SNE visualisation of the Instructor-xl embeddings.

5.2.2.3 Comparison Between Language Models

Table 5.20 below presents the mean F1-scores achieved by the tuned classification algorithms across the three LLMs tested in this study. The best result was obtained by Logistic Regression with the CodeBERT embeddings, with a mean F1-score of 0.884. The best algorithm for GPT-2 was SVM, with a score of 0.883. The best classifier for Instructor-xl was Random Forest, with a mean F1-score of 0.856. While the best result was achieved with CodeBERT, GPT-2 achieved the highest mean across all classifiers with a score of 0.836. CodeBERT's mean across all classifiers was 0.802, and Instructor-xl had a mean of 0.799. The LLMs all have varying performance across the different classifiers, and there is no singular best-performing classifier for all LLMs. Decision Tree was the worst-performing classifier in all three language models.

Table 5.20: Comparison of mean F1-scores achieved by the classifiers with tuned hyperparameters between CodeBERT, GPT-2, and Instructor-xl.

Classifier	**CodeBERT**	**GPT-2**	**Instructor-xl**
MLPClassifier	0.835	0.855	0.828
Gradient Boosting	0.851	0.870	0.828
Random Forest	0.818	0.875	**0.856**
Decision Tree	0.683	0.741	0.625
KNN	0.795	0.818	0.740
Naive Bayes	0.707	0.764	0.833
Gaussian Process	0.811	0.865	0.824
Logistic Regression	**0.884**	0.854	0.836
SVM	0.832	**0.883**	0.821
Mean across all classifiers	0.802	0.836	0.799

Table 5.21 below shows the average F1-score the different embeddings achieved for each pattern across all nine classifiers. Singleton was the pattern with the lowest F1-score across all three LLMs. Prototype had the highest score in GPT-2 and Instructor-xl. Builder had the highest score for CodeBERT at 0.836, but Prototype also got a high score for CodeBERT at 0.832.

Table 5.21: Comparison of mean F1-scores for Singleton, Prototype, and Builder.

Design Pattern	**CodeBERT**	**GPT-2**	**Instructor-xl**
Singleton	0.741	0.769	0.723
Prototype	0.832	**0.897**	**0.917**
Builder	**0.836**	0.844	0.762

Table 5.22 below shows the ARI and Silhouette scores achieved by the clusterers across CodeBERT, GPT-2, and Instructor-xl. The overall highest ARI score was achieved by both K-Means and BIRCH using the Instructor-xl embeddings, and they achieved a score of 0.494. The highest Silhouette score was 0.326 and was achieved by Mean Shift using CodeBERT embeddings. GPT-2 achieved the highest average

result for both the ARI and Silhouette scores. The performance for all clusterers varies between the language models, except for Mean Shift. Mean Shift got the lowest ARI score and the highest Silhouette score across all LLMs.

Table 5.22: Comparison of mean ARI and Silhouette scores achieved by the tuned clusterers between CodeBERT, GPT-2, and Instructor-xl.

Clusterer	CodeBERT		GPT-2		Instructor-xl	
	ARI	**Silhouette**	**ARI**	**Silhouette**	**ARI**	**Silhouette**
K-Means	0.286	0.221	0.348	0.232	**0.494**	0.074
Mean Shift	0.000	**0.326**	0.164	**0.245**	0.000	**0.160**
Gaussian Mixture	**0.292**	0.210	0.294	0.226	0.413	0.058
BIRCH	0.263	0.167	0.358	0.233	**0.494**	0.076
Affinity Propagation	0.130	0.137	**0.380**	0.145	0.125	0.098
Mean across all clusterers	0.194	0.212	0.309	0.216	0.305	0.093

RQ2.2 Summary: The three different large language models, CodeBERT, GPT-2, and Instructor-xl, achieve varying performance across classifiers, clusterers, and design patterns. All three models had a different classifier and clusterer as the best performers. All three language models achieved a high F1-score in identifying the Prototype pattern, with it being the highest for GPT-2 and Instructor-xl and the second highest for CodeBERT. The highest F1-score was achieved by Logistic Regression with CodeBERT embeddings, but GPT-2 had the best average performance for both the classifiers and clusterers.

5.2.3 RQ2.3: Performance Variance of Programming Languages

This section presents the results relevant to answering RQ2.3. All results presented in this section are with tuned hyperparameters using the CodeBERT LLM. Results for Python are presented in Section 5.2.3.1, and results for C# are presented in Section 5.2.3.2. Comparisons between results for Java, Python, and C# are presented in Section 5.2.3.3.

5.2.3.1 Python

This section presents the results acquired for the Python data. The values for the hyperparameters can be found in Table B.6 in Appendix B.

Figure 5.15 and Table 5.23 demonstrate the results for the classification algorithms trained on design patterns written in Python. Logistic Regression was the highest scoring algorithm, with a mean F1-score of 0.875 and a variance of 0.005. The worst classification algorithm was Decision Tree, which achieved a mean F1-score of 0.605 with a variance of 0.013.

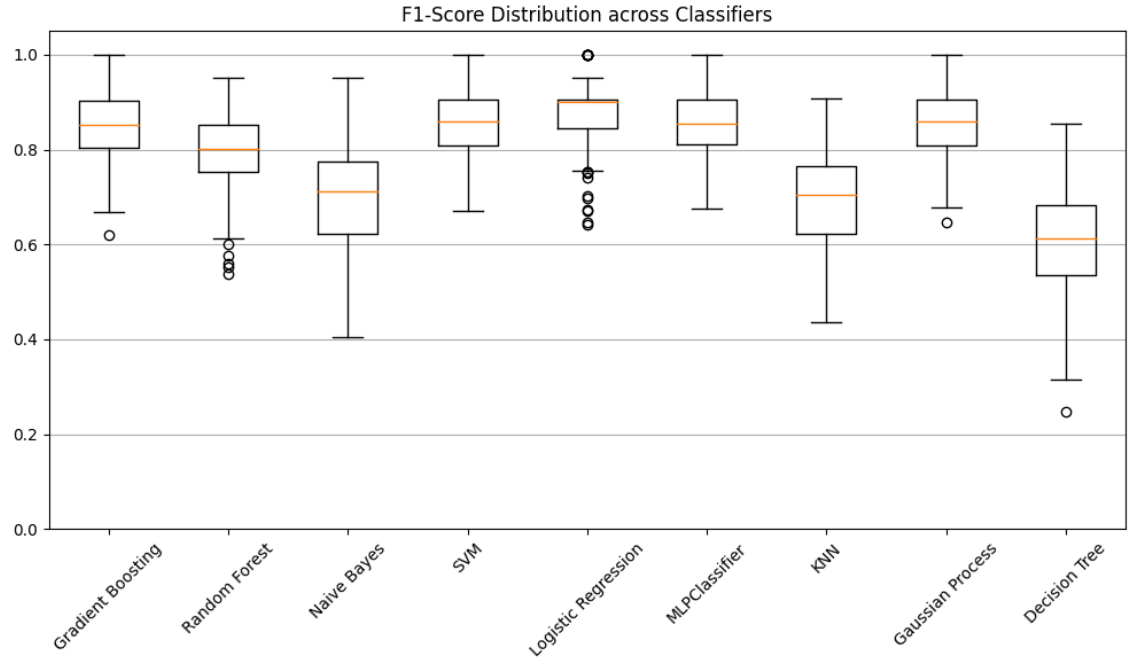

Figure 5.15: Box plot the F1-scores of the classifiers across 200 runs with hyperparameter tuning of the Python code on CodeBERT embeddings, averaged across all patterns.

Table 5.23: Table depicting mean, variance, highest, and lowest F1-scores obtained from running each classifier with tuned hyperparameters 200 times on the CodeBERT embeddings of the Python code, results averaged across all design patterns.

Classifier	**Mean**	**Variance**	**Highest**	**Lowest**
MLPClassifier	0.863	**0.005**	**1.000**	**0.676**
Gradient Boosting	0.839	**0.005**	**1.000**	0.621
Random Forest	0.786	0.007	0.952	0.539
Decision Tree	0.605	0.013	0.855	0.249
KNN	0.690	0.009	0.907	0.437
Naive Bayes	0.707	0.011	0.952	0.404
Gaussian Process	0.867	**0.005**	**1.000**	0.647
Logistic Regression	**0.875**	**0.005**	**1.000**	0.642
SVM	0.868	**0.005**	**1.000**	0.670

Table 5.24 and Figure 5.16 show the individual classification results for Python. The confusion matrix shows that Singleton and Builder are easier to classify than Prototype. Logistic Regression has an accuracy of 0.92 on Singleton and Builder while only achieving 0.86 on Prototype. Prototype is, in this case, miss-classified as Builder with an accuracy of 0.11.

Table 5.24: F1-score, precision, and recall obtained from running each tuned classifier 200 times across the Singleton, Prototype, and Builder embeddings from CodeBERT on Python.

Classifier	**Singleton** F1, P, R	**Prototype** F1, P, R	**Builder** F1, P, R
MLPClassifier	0.90, 0.93, 0.89	0.81, **0.90**, 0.76	0.88, 0.83, 0.96
Gradient Boosting	0.89, 0.93, 0.87	0.78, 0.84, 0.76	0.84, 0.81, 0.90
Random Forest	0.82, 0.89, 0.78	0.73, 0.74, 0.75	0.81, 0.80, 0.84
Decision Tree	0.66, 0.73, 0.64	0.52, 0.53, 0.54	0.64, 0.66, 0.66
KNN	0.74, 0.84, 0.68	0.57, 0.59, 0.59	0.76, 0.73, 0.81
Naive Bayes	0.72, 0.76, 0.72	0.64, 0.68, 0.63	0.76, 0.77, 0.79
Gaussian Process	**0.91**, 0.93, **0.91**	0.80, **0.90**, 0.74	0.89, 0.83, **0.97**
Logistic Regression	0.89, **0.94**, 0.87	**0.83**, 0.89, **0.80**	**0.90**, **0.85**, **0.97**
SVM	0.90, 0.93, 0.89	0.81, 0.89, 0.77	0.89, **0.85**, 0.96

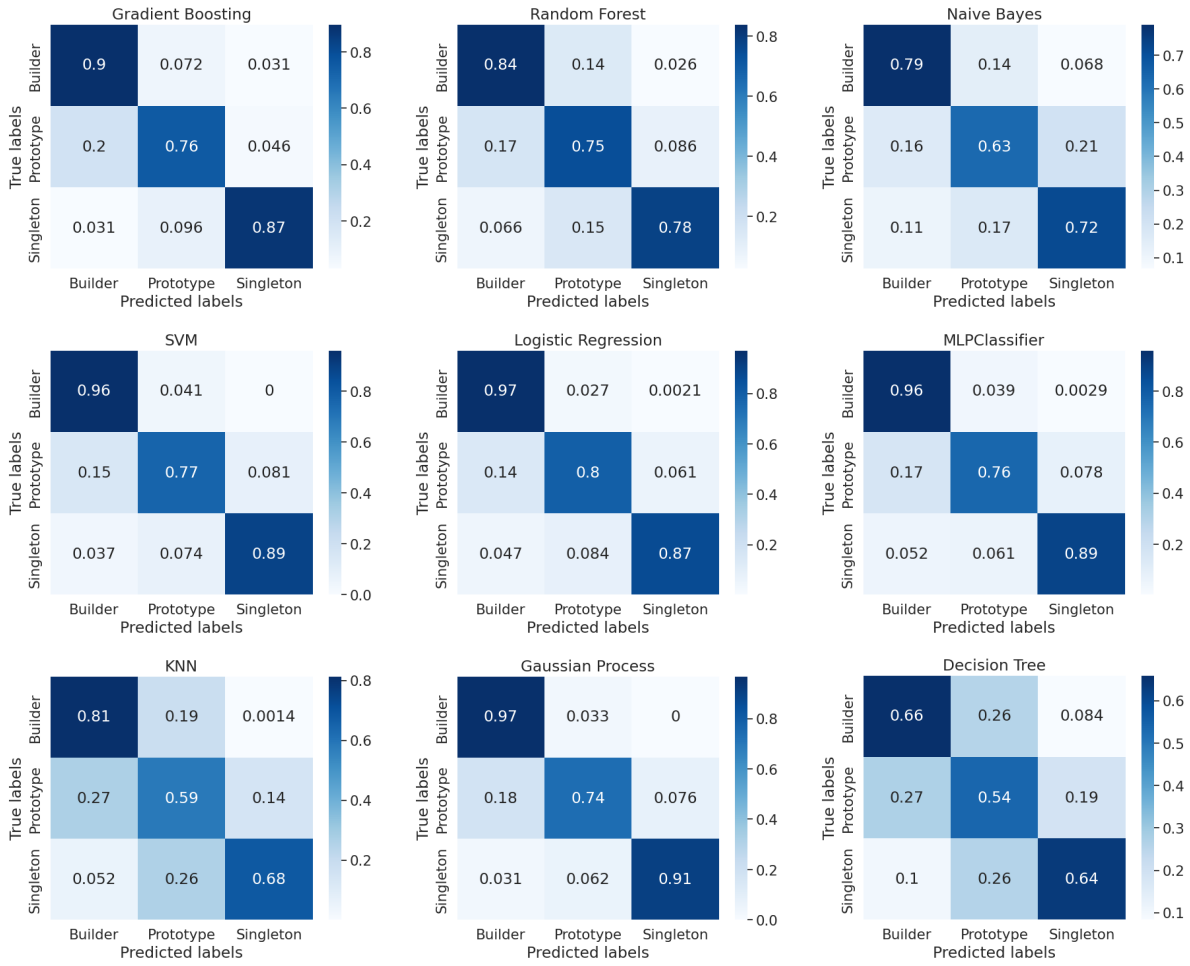

Figure 5.16: Confusion matrices showing how the tuned classifiers predicted each of the design patterns on the CodeBERT embeddings of the Python code, results averaged across 200 runs.

Table 5.25 below depicts the ARI and Silhouette scores obtained from the Python data. Mean Shift got the lowest ARI at 0.000, meaning it is no better than random clustering. It also got the highest Silhouette score at 0.151. All other clusterers got an ARI just above 0.1 and a Silhouette score around 0.1. Their scores are overall low and near the ARI random clustering baseline score of 0.

Table 5.25: ARI and Silhouette score of the tuned clustering algorithms running on CodeBERT embeddings of the Python data, averaged over 200 runs.

Clusterer	Adjusted Rand Index	Silhouette score
K-Means	0.102	0.109
Mean Shift	0.000	**0.151**
Gaussian Mixture	0.106	0.104
BIRCH	**0.119**	0.098
Affinity Propagation	0.118	0.086

Figure 5.17 shows the t-SNE visualisation of the Python data. The visualisation reduces the dimensionality of the data from 768 to 2, so there is a loss of information. The visualisation shows that the Python data is very spread out and only has minor discernible clusters. The Singleton data (green) is mainly in the lower half, spread out from left to right, with most of the data points being in the bottom left. The Builder data (blue) is mainly in the top middle of the graph. The Prototype data (orange) is mainly in the middle part of the graph, spread out from the top to the bottom.

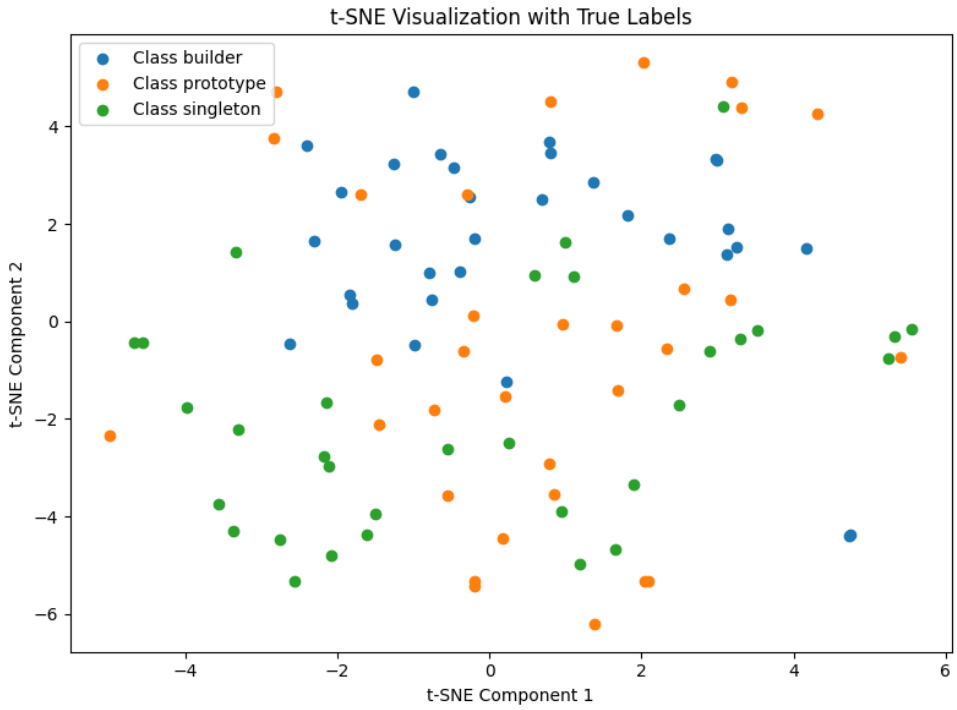

Figure 5.17: t-SNE visualisation of the CodeBERT embeddings of the Python data.

5.2.3.2 C#

This section presents the results acquired for the C# data. The values for the hyperparameters can be found in Table B.7 in Appendix B.

Figure 5.18 and Table 5.26 show the results for the nine different classifiers when trained on design patterns written in C#, from embeddings extracted with CodeBERT. Logistic Regression achieved the highest mean F1-score of 0.899 with a variance of 0.004. The worst-performing classifier was Decision Tree, with a mean F1-score of 0.622 and a variance of 0.009.

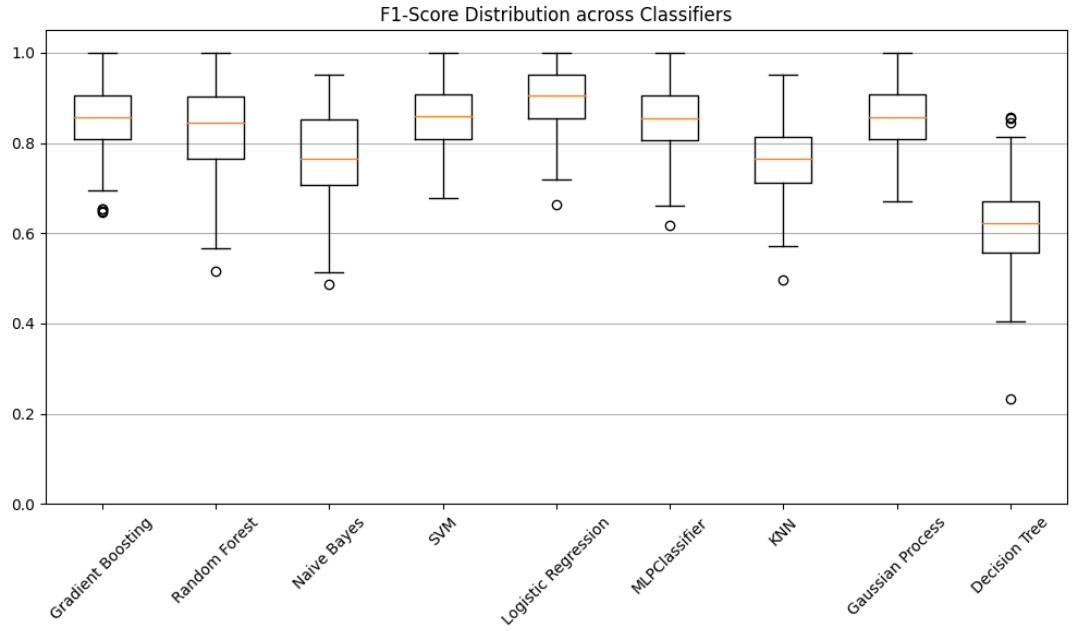

Figure 5.18: Box plot showing the F1-scores of the classifiers across 200 runs with hyperparameter tuning on CodeBERT embeddings of the C# data, averaged across all patterns.

Table 5.26: Table depicting mean, variance, highest, and lowest F1-scores obtained from running each classifier with tuned hyperparameters 200 times on CodeBERT embeddings of the C# data, results averaged across all design patterns.

Classifier	Mean	Variance	Highest	Lowest
MLPClassifier	0.850	0.006	**1.000**	0.618
Gradient Boosting	0.861	0.006	**1.000**	0.647
Random Forest	0.827	0.007	**1.000**	0.515
Decision Tree	0.622	0.009	0.857	0.232
KNN	0.768	0.007	0.952	0.497
Naive Bayes	0.765	0.010	0.952	0.487
Gaussian Process	0.862	0.006	**1.000**	0.670
Logistic Regression	**0.899**	**0.004**	**1.000**	0.663
SVM	0.866	0.005	**1.000**	**0.678**

Table 5.27 and Figure 5.19 demonstrate how the classifiers identified individual design patterns. Logistic Regression outperformed other classifiers across all design patterns, achieving an accuracy of 0.92 on both Builder and Singleton and 0.86 on Prototype. The Prototype pattern was the most challenging to classify accurately, with accuracies ranging from 0.56 when using Decision Tree to 0.86 using Logistic Regression.

Table 5.27: Table depicting F1-score, precision, and recall obtained from running each tuned classifier 200 times across the Singleton, Prototype, and Builder embeddings from CodeBERT on C#.

Classifier	**Singleton** F1, P, R	**Prototype** F1, P, R	**Builder** F1, P, R
MLPClassifier	0.89, 0.90, 0.89	0.81, 0.86, 0.80	0.85, 0.84, 0.87
Gradient Boosting	0.90, 0.89, **0.92**	0.82, 0.86, 0.80	0.86, 0.87, 0.87
Random Forest	0.86, 0.87, 0.86	0.78, 0.80, 0.79	0.84, 0.87, 0.84
Decision Tree	0.67, 0.70, 0.68	0.55, 0.58, 0.56	0.65, 0.69, 0.65
KNN	0.78, 0.87, 0.74	0.69, 0.71, 0.70	0.83, 0.81, 0.87
Naive Bayes	0.80, 0.81, 0.83	0.74, 0.72, 0.78	0.75, 0.86, 0.69
Gaussian Process	0.90, 0.92, 0.90	0.82, 0.87, 0.80	0.86, 0.85, 0.89
Logistic Regression	**0.93**, **0.95**, **0.92**	**0.88**, **0.91**, **0.86**	**0.89**, **0.88**, **0.92**
SVM	0.90, 0.94, 0.87	0.82, 0.84, 0.83	0.88, 0.87, 0.90

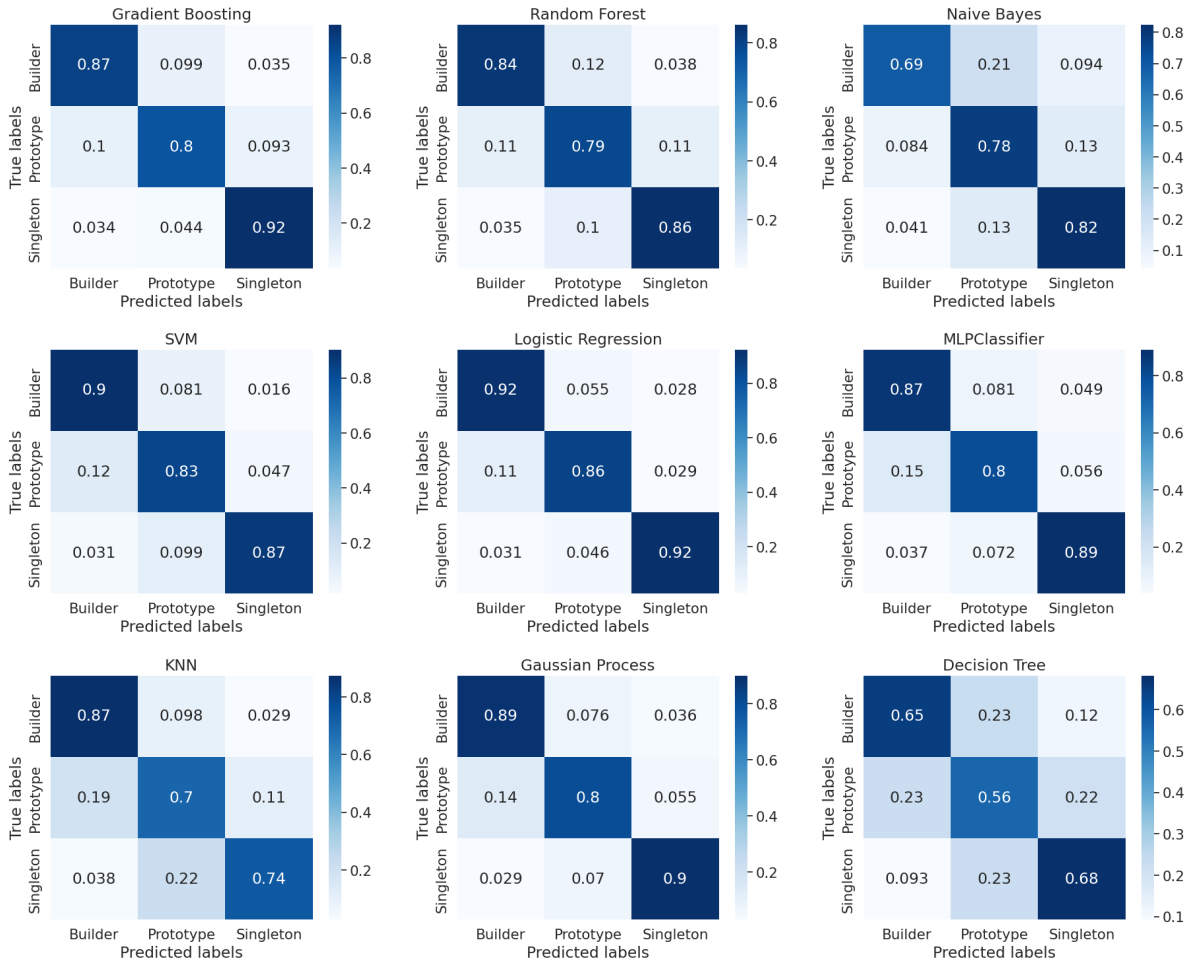

Figure 5.19: Confusion matrices showing how the tuned classifiers predicted each of the design patterns on CodeBERT embeddings of the C# data, results averaged across 200 runs.

Table 5.28 below shows the ARI and Silhouette scores achieved by the clusterers on the C# data. Mean Shift did again get the lowest ARI at 0.000 and the highest Silhouette score at 0.159. All other clusterers got an ARI below 0.1, with BIRCH being the exception, getting an ARI of 0.206. The Silhouette scores for the clusterers were around 0.1. The clusterers had an overall bad performance with low ARI and Silhouette scores.

Table 5.28: ARI and Silhouette score of the tuned clustering algorithms running on CodeBERT embeddings of the C# data, averaged over 200 runs.

Clusterer	Adjusted Rand Index	Silhouette score
K-Means	0.086	0.102
Mean Shift	0.000	**0.159**
Gaussian Mixture	0.098	0.097
BIRCH	**0.206**	0.106
Affinity Propagation	0.055	0.068

Figure 5.20 shows a two-dimensional t-SNE visualisation of the C# data. The original data has 768 dimensions, so the visualisation does have a loss of information. The visualisation shows that the data for all design patterns is very spread out, and there are only minor discernible. The Singleton data (green) has a cluster in the top middle part but also has multiple data points spread out at the bottom. The Prototype data (orange) is mainly in the centre of the graph but has data points on the left, right, and top of the graph as well. The Builder data (blue) does not have a discernible cluster, but it is wrapped around the Prototype data like a roof, with some data points in the top right and bottom parts of the graph as well.

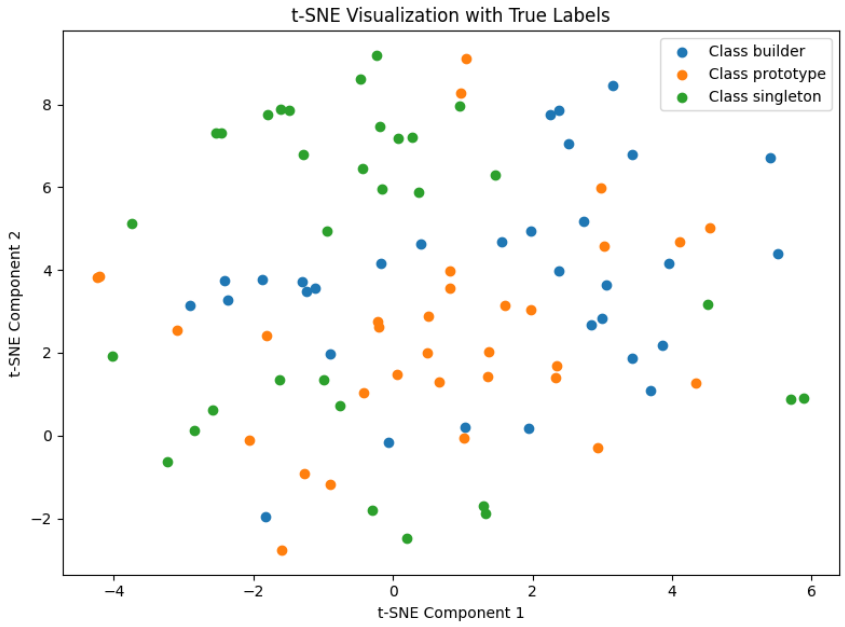

Figure 5.20: t-SNE visualisation of the CodeBERT embeddings of the C# data.

5.2.3.3 Comparison Between Programming Languages

Table 5.29 shows a comparison of the mean F1-scores achieved by the classifiers on the Java, Python, and C# data. The language with the highest mean F1 across all classifiers was C# with an average of 0.813. Java was the second highest at 0.802, and Python had the lowest average F1-score at 0.789. The best classifier was Logistic Regression for all languages. The worst classifier for all programming languages was Decision Tree. The performance ordering of the rest of the classifiers is not the same, but they are similar, mainly swapping a single position up or down.

Table 5.29: Comparison of mean F1-scores achieved by the classifiers with tuned hyperparameters between Java, Python, and C#.

Classifier	**Java**	**Python**	**C#**
MLPClassifier	0.835	0.863	0.850
Gradient Boosting	0.851	0.839	0.861
Random Forest	0.818	0.786	0.827
Decision Tree	0.683	0.605	0.622
KNN	0.795	0.690	0.768
Naive Bayes	0.707	0.707	0.765
Gaussian Process	0.811	0.867	0.862
Logistic Regression	**0.884**	**0.875**	**0.899**
SVM	0.832	0.868	0.866
Mean across all classifiers	0.802	0.789	0.813

Table 5.30 below depicts the F1-score achieved for the Singleton, Prototype, and Builder patterns averaged across all nine classifiers for Java, Python, and C#. For Java, the Builder pattern had the highest F1-score at 0.837, and Prototype was second at 0.832. Singleton had the lowest F1-score at 0.741. For both Python and C#, Singleton had the highest F1-score, with Builder being second and Prototype achieving the lowest score.

Table 5.30: Comparison of mean F1-scores for Singleton, Prototype, and Builder.

Design Pattern	**Java**	**Python**	**C#**
Singleton	0.741	**0.825**	**0.848**
Prototype	0.832	0.721	0.768
Builder	**0.837**	0.812	0.823

Table 5.31 below presents the ARI and Silhouette scores achieved for all the clusterers for Java, Python, and C#. The comparison shows varying performance between the programming languages. Mean Shift had the lowest ARI at 0.000 for all languages, and it also had the highest Silhouette score for all languages. Gaussian Mixture had the highest ARI for Java, and BIRCH achieved the highest ARI for both Python and C#. Java achieved the highest mean ARI and Silhouette scores for all clusterers at 0.194 and 0.212, respectively. Python and C# both achieved an average ARI of 0.089 and an average Silhouette score of just above 0.1.

Table 5.31: Comparison of mean ARI and Silhouette scores achieved by the tuned clusterers between CodeBERT, GPT-2, and Instructor-xl.

Clusterer	Java		Python		C#	
	ARI	**Silhouette**	**ARI**	**Silhouette**	**ARI**	**Silhouette**
K-Means	0.286	0.221	0.102	0.109	0.086	0.102
Mean Shift	0.000	**0.326**	0.000	**0.151**	0.000	**0.159**
Gaussian Mixture	**0.292**	0.210	0.106	0.104	0.098	0.097
BIRCH	0.263	0.167	**0.119**	0.098	**0.206**	0.106
Affinity Propagation	0.130	0.137	0.118	0.086	0.055	0.068
Mean across all clusterers	0.194	0.212	0.089	0.109	0.089	0.106

RQ2.3 Summary: The performance for the classifiers is similar for Java, Python, and C# when averaging across all design patterns. Logistic Regression was the best-performing classifier for all languages, and Decision Tree was the worst-performing classifier for all languages. C# had the highest mean F1-score across all classifiers, followed by Java and then Python. Performance for individual patterns varies between the languages. Singleton achieved the lowest F1-score for Java but got the highest for Python and C#. Builder had the highest in Java and was the second highest for Python and C#. Java had the best performance for the clusterers, with a 0.105 higher average ARI than Python and C# and over a 0.1 higher average Silhouette score. Mean Shift got the lowest ARI and highest Silhouette score for all languages. BIRCH achieved the highest ARI for Python and C#. Gaussian Mixture had the highest ARI for Java.

6

Discussion

This section discusses the results presented in the Results chapter. It discusses the reasons behind the results and what could have affected the outcome. A comparison of the results with previous literature and arguments for how these findings can be of use in the field of software engineering and in the research of design pattern recognition are also discussed. A discussion of the machine-learning algorithms and their performance is presented in Section 6.1. A discussion of the different contexts explored in this study is presented in Section 6.2.

6.1 Algorithm Performance on Design Pattern Identification

This section discusses the results of the classifiers and clusterers, compares them with previous work, and presents reasons and arguments for the obtained results and their determinants. A discussion of classifiers is presented in Section 6.1.1. A discussion of clusterers is presented in Section 6.1.2.

6.1.1 Classifiers

The majority of classification algorithms exhibited minimal changes in mean F1-score between baseline results and tuned hyperparameter results. This suggests either that the default parameters for most classifiers are well-balanced or that the parameters have a limited impact on classifier training within the context of DPR. However, the Gaussian Process classifier demonstrated a low baseline result of 0.219, which increased significantly to 0.811 after hyperparameter tuning. This drastic change is due to the choice of kernel and the *length_scale* parameter. The *length_scale* parameter controls the smoothness of the decision boundary; a lower value increases the risk of overfitting, while a higher value can result in underfitting. Grid search results indicate that a *length_scale* of 2.0 significantly outperforms the default value of 1.0. Gaussian Process also had the highest variance of all classifiers, 0.024, compared to the next highest of 0.009. It had several low-scoring outliers. A possible explanation for this can be due to Gaussian Process's ineffectiveness in high-dimensional spaces, which can lead to overfitting.

In this study, Logistic Regression was the overall best-performing classifier, achieving

an F1-score of 0.884 on three patterns written in Java with CodeBERT embeddings. The work done by Pandey et al. also used Logistic Regression and achieved a similar F1-score of 0.88 [11]. Zanoni et al. did not use Logistic Regression, instead opting for several different SVC, Decision Tree, and Random Forest algorithms [6]. Their best-performing classifier, with an F1-score of 0.91, was J48 with Reduced Error Pruning (a Decision Tree algorithm designed to handle overfitting). Even though J48 with Reduced Error Pruning achieved a higher F1-score than Logistic Regression, a definitive conclusion about which classifier is superior cannot be drawn. This is because the studies used different methods and datasets. Chaturvedi et al. compared four different algorithms and found that Support Vector Regression was the best performing, better than Linear Regression, Polynomial Regression, and Artificial Neural Networks [83]. Compared to this study's results, SVM and Artificial Neural Networks were very similar, achieving scores of 0.832 and 0.835, respectively.

There was also a decrease in performance for Singleton, Prototype, and Builder design patterns in RQ1, where we evaluated only them, compared to RQ2.1, where we added four extra design patterns. When the number of classes increases, so does the complexity of the classification problem. With the increase in complexity, the classification algorithms may have a harder time identifying important decision boundaries, which can lead to more misclassifications and therefore lower performance. There are different methods used by the classifiers to handle multiclass classification, but most of them rely on either confidence score or distance to a decision boundary [43]. With more classes, the chance of an incorrect class gaining a higher confidence score than the correct class is likely to be higher as there are more classes to compare the input with.

There are several factors that explain why Naive Bayes and Decision Tree performed worse than other classifiers. Decision Trees can easily overfit and create overly complex models on high-dimensional data. This also explains why the J48 with Reduced Error Pruning, designed to handle overfitting, performs well in the study by Zanoni et al. Additionally, small variations in the data can significantly change the results of the trees, making them unstable [84]. These disadvantages are often mitigated by using Decision Trees within an ensemble method, as evidenced by the superior performance of Random Forest in this study. Naive Bayes, on the other hand, assumes that each feature is independent, which likely is not the case with the dataset [46]. This assumption could explain its poor performance.

Some classifiers are more effective on high-dimensional datasets; these include SVM, Gradient Boosting, and Random Forest. All of these classifiers achieved good results in our study. Given that Logistic Regression and SVM performed well, it can be assumed that the embeddings extracted by the LLMs are somewhat linearly separable.

6.1.2 Clusterers

Clustering did not perform as well as classification; the average F1-score achieved overall by the clustering algorithms was 0.307 compared to the 0.802 achieved by

the classification algorithms. The performance differences between the classifiers and clusterers are mainly due to the type of problem that design pattern recognition is and the methods used to handle it. It is a supervised learning problem, meaning the data is labelled and we want to classify the data according to the labels, and clusterers are meant for unsupervised learning [53]. A classification algorithm approximates a function that takes an input and maps it to an output. Clustering algorithms create different groups of the data with the objective of building a representation of the data, not classifying it based on different labels. Clustering algorithms do not even know the number of classes (labels) present in the data [85]. For certain algorithms, this can be specified, but not for all of them.

K-Means, Gaussian Mixture, and BIRCH are clustering algorithms where the number of clusters to create can be specified. They were also the clustering algorithms that most often had the best performance. Mean Shift and Affinity Propagation could not specify the number of clusters and often had bad performance, with Mean Shift only getting an ARI above 0.000 when running without the Singleton pattern in RQ1. This is also likely the reason that DBSCAN was never able to generate any results. DBSCAN does not have the ability to specify the number of clusters to create. The algorithm could either find no clusters to create or it could only generate a single cluster. The reason that Mean Shift, Affinity Propagation, and BIRCH have the same baseline and tuned performance is a combination of what parameters are available to tune and that changing these parameters often leads to the clusterer creating one cluster per data point. This is also an explanation of why the Silhouette score was often higher than the ARI for Mean Shift and Affinity Propagation. The Silhouette score only checks the point distance between two clusters, while the ARI compares cluster labels with true labels. Creating a lot of clusters would give a low ARI score as there would be fewer pairs of data points that share the same true label and cluster label.

Previous work in DPR also mainly used classification algorithms [12]. Clustering algorithms have been used in previous work within DPR [7]; however, one of the authors of that paper recommended against using them for this task. The utility of using clustering for DPR is low, but it can help with visualisation.

6.2 Influence of Context on Algorithm Performance

This section discusses the different contexts explored in this study and how they affect the performance of the algorithms. Design patterns are discussed in Section 6.2.1, language models are discussed in Section 6.2.2, and programming languages are discussed in Section 6.2.3.

6.2.1 Performance Variance Between Design Patterns

Overall, the performance of the individual design patterns varies depending on the programming language and the language model used. When using three design patterns in Java with CodeBERT, Builder had the best performance, with Prototype

being second and Singleton last. With GPT-2 and Instructor-xl, Prototype had the best performance. For Python and C#, Singleton had the best performance.

Comparing with previous work, the results and performance are similar to what was achieved in this study. Chand et al., who also compared Singleton, Builder, and Prototype with the Word2Vec and Code2Vec LLMs, also found varying performance between the LLMs [7]. For Code2Vec, Prototype had the best performance, followed by Singleton and then Builder. For Word2Vec, Builder and Singleton shared the best performance, followed by Prototype. Pandey et al., who used TransCoder on Singleton and Prototype patterns, achieved the highest performance for the Prototype pattern, with Singleton getting a significantly lower F1-score than Prototype [11].

The overall performance of individual design patterns is lower when increasing the number of patterns and including patterns from all pattern types (behavioral, creational, and structural). The lower performance is due to how the classifiers handle multiclass classification problems with the One-Vs-Rest technique. With more labels, there are more classifiers that can produce a faulty prediction with a confidence score higher than what the correct classifier produces. Across the seven patterns, Prototype got the highest F1-score, and Adapter got the lowest. Adapter achieving a lower score compared to other design patterns are also found in multiple previous studies [6], [12], [86].

The reasons behind the performance of individual patterns could vary. Prototype and Observer were the two patterns with the highest F1-score. They are both similar to each other and have other patterns in their structure requiring multiple classes and similar structures in their implementation [2]. They do, however, both also contain a method that can be used to easily identify them. The Prototype pattern can be identified as it will contain a *clone* method. The Observer pattern can be identified by its *update* method. Other patterns with a similar structure to Observer and Prototype are Builder, Strategy, Bridge, and Adapter. They all have a similar structure, but they miss the unique identifier present in Observer and Prototype patterns, which can explain their lower F1-score. Singleton is unique here as it only requires a single class to implement. But its F1-score is the lowest of all the creational patterns. This is because Singleton is also a common pattern to use in other design patterns, such as Builder and Prototype. This means that there could be Builder and Prototype examples that use Singleton as well. This would also explain that when Builder and Prototype are misclassified, they are most often misclassified as being Singleton. From the data used in this study, it seems to be Builder that would most often contain the Singleton pattern, as Singleton is often misclassified as being Builder, and it also explains the high recall that the Builder pattern has compared to its precision.

The results indicate that there is a performance difference between behavioral, creational, and structural patterns. This can be due to the LLMs being able to identify patterns dealing with the creation or behaviour of code easier than they identify patterns dealing with the structure of code. However, the performance differences

between the pattern types obtained in this study may be due to other factors. One factor that it can be is the individual pattern performances within each pattern type. For example, the structural patterns had the lowest F1-score, and the Adapter and Bridge patterns both have a similar structure to other patterns and do not have a unique identifier. Gamma et al. also explicitly state that Bridge and Adapter are similar in their structure [2]. Performance variations in behavioral, creational, and structural patterns are likely caused by the specific design patterns used rather than inherent differences in the LLMs' ability to identify pattern types. There are many more patterns within the three pattern types to explore, and it would require a study of its own to see how LLMs and machine-learning algorithms vary in their performance and understanding of the different pattern types. The results indicate that there are performance differences between the pattern types, but further research and data are needed to conclude what the main factor behind their performance is.

6.2.2 Performance Variance Between Language Models

The three LLMs tested in this study showed varying performance and had different best-performing algorithms and different pattern performance. The t-SNE visualisations also show a variance in the LLMs semantic understanding of the code. The difference between the LLMs is not significant, but it is notable as it shows that the LLM does have an impact on the final outcome.

Comparing all three LLMs and their properties shows what could be the reason behind their overall performance. Comparing the performance between the models used, the number of parameters does not seem to affect performance. The CodeBERT and GPT-2 models used in this study have 125 and 124 million parameters, respectively, and Instructor-xl has 1.5 billion parameters, but it does not show a significant difference in its performance. They were all released around the same time as well, with GPT-2 being released in 2019, CodeBERT in 2020, and Instructor-xl in 2021 [26], [32], [35]. This means that they all had similar technologies, hardware, and techniques available.

The factor that seems to have the most effect on their performance in DPR is the data the model was trained on. GPT-2, which had the best mean performance across all classifiers and clusterers, was trained on data that was gathered by scraping web pages [38]. The fourth most common domain part of the training set was GitHub. This means that the training data very likely consists of GitHub repositories that are very similar, if not the same, as the ones used in this study. Instructor-xl training data is similar to that of GPT-2, as they are both trained on different types of data and tasks. However, Instructor-xl has fewer datasets related to code, coding, or programming when compared to GPT-2 [35], [87]. Although CodeBERT and Instructor-xl have very similar performance, CodeBERT is only trained on code gathered from open source repositories on GitHub. This is likely due to the difference in model parameters. Instructor-xl has over ten times more parameters, which makes it able to learn more and capture more complex patterns than CodeBERT [88]. CodeBERT's training data does come from GitHub, but the GitHub repositories

were all packages or libraries and not just any open source repositories on GitHub [89]. It is likely, then, that the projects in P-MARt were part of the training data but not the other GitHub repositories used in this study. GPT-2 did not have this requirement for what repositories to include, meaning it likely included the same or very similar data to what was used in this study. The t-SNE visualisations between the three LLMs also show that CodeBERT and GPT-2 have a more similar semantic understanding of the data than when they are compared to Instructor-xl.

Along with previous work that utilises LLMs for DPR [7], [11], this study also shows promising results for LLMs within DPR. The results show that all three LLMs semantic understanding of the code can be used to identify design patterns in the code. Metric-based approaches still have more research behind them and provide both tools and more established methods than LLM-based approaches [6], [12]. One disadvantage of metric-based approaches is that they require a tool to extract the metrics, whereas in an LLM-based approach, the LLM extracts the semantic meaning of a code. LLM-based approaches can be more robust against variation in design patterns since they do not look at specific parts of the code. But the results speak for continued research on using LLMs in DPR and that it would be valuable to perform studies with larger datasets and to develop methods and tools utilising LLMs.

6.2.3 Performance Variance Between Programming Languages

This study compared and evaluated classification and clustering algorithms across three programming languages: Java, Python, and C#. No significant performance differences were found between the languages. Overall, C# exhibited the highest F1-score among the languages. Logistic Regression was the best-performing classifier for all languages, achieving a mean F1-score above 0.87. In clustering, Python had a much lower ARI than Java and C#, suggesting that clusterers struggled to generate meaningful clusters for Python code. One notable difference between the languages was Python's higher variance compared to Java and C#. Additionally, some classifiers produced low outliers for Python, potentially caused by lower-quality data for its design patterns.

Furthermore, results for individual design patterns indicate that Java performed poorly in classifying Singleton compared to Prototype and Builder. Conversely, Python and C# performed poorly on Prototype. These differences could stem from variations in how design patterns are implemented within each language and the data collection methods used in the study. Java's dataset (P-MARt and GitHub) may have contained noisier Singleton examples, while Python and C# relied on more textbook-like Singleton patterns. Despite these data discrepancies, all classifiers achieved relatively similar mean F1-scores across all programming languages. Decision Tree, KNN, and Naive Bayes were the least effective classifiers, with Logistic Regression emerging as the most successful.

Most of the previous work has used a Java dataset (P-MARt) to test design pattern recognition, including Chand et al., Tsantalis et al., and Dunlop, which all achieved

similar F1-scores to this study [9], [11], [86]. Pandey et al. and Dunlop also evaluated classifiers in C++. Dunlop used a metric-based approach and found that C++ outperformed Java in F1-score, while Pandey et al. used a LLM-based approach and found that Singleton only achieved a F1-score of 0.55 in C++, but Prototype gained a 0.923 F1-score. The results can vary significantly from study to study, and most datasets are quite small. Consequently, more research is needed to determine if any differences exist between programming languages in the context of design pattern recognition.

6.3 Threats to Validity

This section discusses the internal, external, and construct validity threats of the study.

6.3.1 Internal Validity

The data used in the study and how it is split into training and test sets can affect the algorithm's performance. The design patterns in the P-MARt repository are examples extracted from tools and libraries written in Java, while the design patterns gathered from GitHub is more representative of how a design pattern would be presented in a textbook. The study mitigates the effect of different training and test data by running each classifier on 200 different training and test sets. Multiple tests were also performed using both data from P-MARt and data gathered from GitHub. Java data was a combination of P-MARt and GitHub data, and Python and C# data were all from GitHub.

The amount of data can also affect the performance of the classifiers. More data to train on can make the classifiers learn more about the design patterns and better be able to identify and distinguish between them. Labelled data for this task is scarce, and most studies opt for P-MARt or gathering their own data from GitHub, both of which were used in this study.

Another validity threat is the design patterns used in the study. Certain design patterns can be easier to identify and distinguish from other patterns, which can affect the performance of the algorithms. The test in RQ2.1 did, however, show that while the performance of the individual classifiers can change, the general performance order will mostly remain the same.

Another factor that could affect the outcome is language models. Different models will produce different embeddings, which can change the results and performance of the algorithms. This was explored and discussed in RQ2.2, where multiple different language models were tested and compared.

The programming language can also affect the outcome of the tests, as the syntax of different programming languages can be easier for the LLMs and algorithms to understand and identify design patterns in. This was explored in RQ2.3 with

Python and C#. The results were also compared with previous studies that used other languages, and they also found similar results to what was found in this study.

6.3.2 External Validity

The data used in the study for most tests was a combination of data from P-MARt and GitHub. The data from P-MARt is extracted from real code from industry software and is also the data that the majority of the studies within DPR use. The data from GitHub is more similar to textbook implementations of design patterns, which can affect the generalisability of the findings in this study to industry code. The majority of the data was, however, from P-MARt. The F1, precision, and recall scores were also similar to results found in previous studies in DPR.

How context affects the performance of the algorithms was also explored by including more patterns, testing different language models, and using different programming languages. The results of the study show that it is generalisable between language models and programming languages. Previous studies that used different language models and programming languages also showed similar performance in terms of F1, precision, and recall. RQ2.1 did, however, show that different design patterns are easier to identify than others and that the individual performance of the algorithms is not generalisable across design patterns. The general order of performance between the algorithms should, however, mostly remain the same. The design patterns in this study are those described by Gamma et al., and the performance of the classifiers might not be generalisable to design patterns from other standards, such as AUTOSAR.

6.3.3 Construct Validity

The performance metrics used in this study were F1-score, precision, and recall. Those metrics are used in a large majority of previous studies that utilise classification algorithms and provide an understandable value for comparison. Other measures, such as the AUC-ROC curve, were not used as they are harder to interpret and not used a lot in previous studies, making it therefore harder to compare and know what would be considered a good performance in terms of these metrics for DPR.

For clustering, Adjusted Rand Index, and Silhouette score were used in combination with the F1-score. The Silhouette score is a widely used metric in clustering algorithms, and although it can be used to compare scores from classifiers, it provides a useful metric for comparing the clustering algorithms. The F1-score was calculated for RQ1 to get a good comparison between clusterers and classification, but as stated in the method, it was not viable when the combinations of clusterers increased. Furthermore, they had a very poor performance in terms of F1-score, and it was decided not to continue measuring F1-score in RQ2. The ARI score does, however, compare cluster labels with true labels and can therefore be a representative measure of the algorithm's effectiveness in classifying patterns as well. The

t-SNE visualisations have a loss of information and do not show the actual clustering of the data but do show a representation of it; this is also stated in the results where a t-SNE visualisation is presented.

The variance in the results was mitigated by running each algorithm 200 times with different training and test data splits. For classification algorithms, a box plot was used to visualise the aggregated results, including variance, median, lower quartile, and upper quartile scores. These visualisations helped in comparing the performances. For clustering algorithms, the aggregated results were presented in a table. The clustering algorithms were not measured as extensively as the classification algorithms. This is due to DPR mainly being a classification task, and the purpose of clustering algorithms is not to classify. Their overall performance in RQ1 also showed it would not be valuable to do extensive measuring of their performance. Clustering algorithms were, however, still kept in case their performance was dependent on the context explored in RQ2.

7

Conclusion

This study evaluated and tested several classification and clustering algorithms in design pattern recognition (DPR), utilising semantic information from large language models (LLMs). The performance of the algorithms was also analysed across diverse contexts, including a broader range of design patterns, programming languages, and language models. This study used the P-MARt dataset with a combination of GitHub repositories to extract a set of design patterns used in training and testing. A pipeline was created to systematically extract, train, and evaluate the classifiers and clusterers. To find the optimal hyperparameters, a grid search was performed with cross-validation to exhaustively test a set of parameters.

The results indicated that the order of classifier performance remained relatively consistent, with Logistic Regression generally outperforming other algorithms and Decision Tree often ranking lower. Furthermore, the results indicate that Logistic Regression was among the top-performing classifiers, often achieving a high F1-score and performing similarly to results in previous DPR studies. However, no single classifier exhibited a consistently significant advantage. Clustering algorithms, however, performed less favourably and are not for classification purposes in DPR.

Overall, the results indicate that there is no single classification algorithm that should be used in design pattern recognition. Logistic Regression, Gradient Boosting, Multi-layer Perceptron, and SVM were classifiers that performed well in all the tests and contexts. Subsequently, this study also showed that LLMs such as CodeBERT, GPT-2, and Instructor-xl were able to extract semantic information from written code and aid with the task of design pattern recognition. Finally, more research is required to fully utilise LLM-based DPR in software development.

7.1 Future Work

Utilising large language models for DPR is a very recent area of research, and there are a lot of opportunities and areas to explore. Previous studies that use LLMs for DPR have been small studies about testing their feasibility and using small datasets to do it. One opportunity for a future study is to do the same thing done in this study or other previous studies, but using bigger datasets. This will give a better comparison of how using LLMs performs compared to using metrics and give a better

idea of the potential available when utilising LLMs for DPR.

Another idea to further establish LLMs for DPR is to develop a tool for an IDE that utilises them and puts the research into practice. Zanoni et al. developed MARPLE-DPD as a tool for the Eclipse IDE using a metric-based approach [6]. Tools already exist for metric-based approaches, meaning it should be possible to do it as well for an LLM-based approach. This will allow further comparison between LLM-based approaches and metric-based approaches. It will also provide data and ideas on how LLMs can be implemented and utilised for other areas of research and practice.

Along with trying to further establish LLMs in DPR, there are also research areas to explore that can be done in smaller studies. Pre-training and fine-tuning of the LLMs are yet to be fully explored. Pre-training alone has been explored in a previous study [7], but fine-tuning and its effects have not been studied, and combining pre-training and fine-tuning has not been explored either. Both pre-training and fine-tuning of LLMs require access to good hardware, as they require a lot of resources and time for computation. It is not currently feasible to perform without access to hardware. As time goes on, however, the language models will continue to evolve, and the hardware and software will become faster, making the overall usage of LLMs for DPR better and more accessible.

Bibliography

[1] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 367–377. DOI: 10.1109/ESEM.2009.5314233.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0201633612.

[3] P. Hegedűs, D. Bán, R. Ferenc, and T. Gyimóthy, "Myth or reality? analyzing the effect of design patterns on software maintainability," in *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 138–145, ISBN: 978-3-642-35267-6.

[4] F. M. Alghamdi and M. R. Qureshi, "Impact of design patterns on software maintainability," *International Journal of Intelligent Systems and Applications*, vol. 6, no. 10, pp. 41–46, 2014. DOI: 10.5815/ijisa.2014.10.06.

[5] S. Chand, S. K. Pandey, M. Staron, and J. Horkoff, *Design patterns understanding and use in the automotive industry: An interview study.*

[6] M. Zanoni, F. Arcelli Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *Journal of Systems and Software*, vol. 103, May 2015. DOI: 10.1016/j.jss.2015.01.037.

[7] S. Chand, S. K. Pandey, M. Staron, M. Ochodek, and D. Durisic, "Comparing word-based and ast-based models for design pattern recognition," ser. PROMISE 2023, San Francisco, CA, USA: Association for Computing Machinery, 2023, ISBN: 9798400703751. DOI: 10.1145/3617555.3617873. [Online]. Available: https://doi.org/10.1145/3617555.3617873.

[8] H. Yarahmadi and S. M. H. Hasheminejad, "Design pattern detection approaches: A systematic review of the literature," *Artificial Intelligence Review*, vol. 53, Dec. 2020. DOI: 10.1007/s10462-020-09834-5.

[9] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 896–909, 2006. DOI: 10.1109/TSE.2006.112.

[10] A. K. Dwivedi, A. Tirkey, and S. K. Rath, "Software design pattern mining using classification-based techniques," *Frontiers of Computer Science*, vol. 12, pp. 908–922, 5 Oct. 2018, heergterg, ISSN: 20952236. DOI: 10.1007/s11704-017-6424-y.

[11] S. Pandey, M. Staron, J. Horkoff, M. Ochodek, N. Mucci, and D. Durisic, "Transdpr: Design pattern recognition using programming language models," Oct. 2023, pp. 1–7. DOI: 10.1109/ESEM56168.2023.10304862.

[12] H. Yarahmadi and S. M. H. Hasheminejad, "Design pattern detection approaches: A systematic review of the literature," *Artificial Intelligence Review*, vol. 53, Dec. 2020. DOI: 10.1007/s10462-020-09834-5.

[13] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," *Proceedings of the 23rd international conference on Machine learning - ICML '06*, 2006. DOI: 10.1145/1143844.1143865.

[14] A. Soofi and A. Awan, "Classification techniques in machine learning: Applications and issues," *Journal of Basic & Applied Sciences*, vol. 13, pp. 459–465, Aug. 2017. DOI: 10.6000/1927-5129.2017.13.76.

[15] P. Rai and S. Shubha, "A survey of clustering techniques," *International Journal of Computer Applications*, vol. 7, Oct. 2010. DOI: 10.5120/1326-1808.

[16] Refactoring Guru, *Singleton*. [Online]. Available: https://refactoring.guru/design-patterns/singleton (accessed 7/2/2024).

[17] Refactoring Guru, *Builder*. [Online]. Available: https://refactoring.guru/design-patterns/builder (accessed 7/2/2024).

[18] Refactoring Guru, *Prototype*. [Online]. Available: https://refactoring.guru/design-patterns/prototype (accessed 7/2/2024).

[19] Refactoring Guru, *Adapter*. [Online]. Available: https://refactoring.guru/design-patterns/adapter (accessed 7/2/2024).

[20] Refactoring Guru, *Bridge*. [Online]. Available: https://refactoring.guru/design-patterns/bridge (accessed 7/2/2024).

[21] Refactoring Guru, *Composite*. [Online]. Available: https://refactoring.guru/design-patterns/composite (accessed 7/2/2024).

[22] Refactoring Guru, *Strategy*. [Online]. Available: https://refactoring.guru/design-patterns/stratagy (accessed 7/2/2024).

[23] Refactoring Guru, *Observer*. [Online]. Available: https://refactoring.guru/design-patterns/observer (accessed 7/2/2024).

[24] Refactoring Guru, *State*. [Online]. Available: https://refactoring.guru/design-patterns/state (accessed 7/2/2024).

[25] Y. Chang, X. Wang, J. Wang, *et al.*, "A survey on evaluation of large language models," *ACM Transactions on Intelligent Systems and Technology*, 2023.

[26] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[27] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[28] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[29] E. Kasneci, K. Sessler, S. Küchemann, *et al.*, "Chatgpt for good? on opportunities and challenges of large language models for education," *Learning and Individual Differences*, vol. 103, p. 102 274, 2023, ISSN: 1041-6080. DOI: https://doi.org/10.1016/j.lindif.2023.102274. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1041608023000195.

[30] B. Min, H. Ross, E. Sulem, *et al.*, "Recent advances in natural language processing via large pre-trained language models: A survey," *ACM Computing Surveys*, vol. 56, no. 2, pp. 1–40, 2023.

[31] Cloudflare, *What are embeddings in machine learning?* [Online]. Available: https://www.cloudflare.com/learning/ai/what-are-embeddings/ (accessed 30/1/2024).

[32] Z. Feng, D. Guo, D. Tang, *et al.*, *Codebert: A pre-trained model for programming and natural languages*, 2020. arXiv: 2002.08155 [cs.CL].

[33] Y. Liu, M. Ott, N. Goyal, *et al.*, *Roberta: A robustly optimized bert pretraining approach*, Jul. 2019.

[34] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds., Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. [Online]. Available: https://aclanthology.org/N19-1423.

[35] H. Su, W. Shi, J. Kasai, *et al.*, "One embedder, any task: Instruction-finetuned text embeddings," 2022. [Online]. Available: https://arxiv.org/abs/2212.09741.

[36] J. Ni, G. H. Ábrego, N. Constant, *et al.*, *Sentence-t5: Scalable sentence encoders from pre-trained text-to-text models*, 2021. arXiv: 2108.08877 [cs.CL].

[37] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[38] OpenAI, *Gpt-2*. [Online]. Available: https://huggingface.co/openai-community/gpt2 (accessed 4/4/2024).

[39] R. Toews, "The next generation of large language models," *Forbes*, Feb. 2023. [Online]. Available: https://www.forbes.com/sites/robtoews/2023/02/07/the-next-generation-of-large-language-models/?sh=3daab5ff18db (accessed 2/2/2024).

[40] Google, *What is supervised learning?* [Online]. Available: https://cloud.google.com/discover/what-is-supervised-learning (accessed 30/1/2024).

[41] Z. Keita, *Classification in machine learning: An introduction*. [Online]. Available: https://www.datacamp.com/blog/classification-machine-learning (accessed 30/1/2024).

[42] J. Brownlee, "A gentle introduction to ensemble learning algorithms," Apr. 2021. [Online]. Available: https://machinelearningmastery.com/tour-of-ensemble-learning-algorithms/ (accessed 7/2/2024).

[43] scikit-learn developers, *Multiclass and multioutput algorithms*. [Online]. Available: https://scikit-learn.org/stable/modules/multiclass.html (accessed 2/4/2024).

[44] L. Breiman, *Classification and regression trees*. CRC Press, 2017.

[45] scikit-learn developers, *Sklearn.ensemble.randomforestclassifier*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html (accessed 12/2/2024).

[46] H. Zhang, "The optimality of naive bayes," vol. 2, Jan. 2004.

[47] C. E. Rasmussen and C. K. Williams, *Gaussian processes for machine learning*, 2005. DOI: 10.7551/mitpress/3206.001.0001.

[48] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006, ISBN: 0387310738.

[49] scikit-learn developers, *1.4.7. mathematical formulation*. [Online]. Available: https://scikit-learn.org/stable/modules/svm.html#mathematical-formulation (accessed 14/2/2024).

[50] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967. DOI: 10.1109/TIT.1967.1053964.

[51] J. H. Friedman, "Greedy function approximation: A gradient boosting machine.," *The Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, 2001. DOI: 10.1214/aos/1013203451. [Online]. Available: https://doi.org/10.1214/aos/1013203451.

[52] J. Brownlee, "Crash course on multi-layer perceptron neural networks," Aug. 2022. [Online]. Available: hhttps://machinelearningmastery.com/neural-networks-crash-course/ (accessed 20/2/2024).

[53] M. All, *Clustering in machine learning: 5 essential clustering algorithms*. [Online]. Available: https://www.datacamp.com/blog/clustering-in-machine-learning-5-essential-clustering-algorithms (accessed 30/1/2024).

[54] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, 1987, ISSN: 0377-0427. DOI: https://doi.org/10.1016/0377-0427(87)90125-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0377042787901257.

[55] scikit-learn developers, *Sklearn.metrics.silhouette_score*. [Online]. Available: https://bit.ly/3UyagTc (accessed 30/1/2024).

[56] scikit-learn developers, *K-means*. [Online]. Available: https://scikit-learn.org/stable/modules/clustering.html#k-means (accessed 30/1/2024).

[57] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96, Portland, Oregon: AAAI Press, 1996, pp. 226–231.

[58] D. Comaniciu and P. Meer, "Mean shift: A robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 5, pp. 603–619, 2002. DOI: 10.1109/34.1000236.

[59] D. A. Reynolds, "Gaussian mixture models," in *Encyclopedia of Biometrics*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:1063711.

[60] C. Müller, *Speaker Classification II*. Springer, 2007.

[61] J. Benesty, M. M. Sondhi, Y. Huang, *et al.*, *Springer handbook of speech processing*. Springer, 2008, vol. 1.

[62] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: An efficient data clustering method for very large databases," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '96, Montreal, Quebec, Canada: Association for Computing Machinery, 1996, pp. 103–114, ISBN: 0897917944. DOI: 10.1145/233269.233324. [Online]. Available: https://doi.org/10.1145/233269.233324.

[63] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: A new data clustering algorithm and its applications," *Data mining and knowledge discovery*, vol. 1, pp. 141–182, 1997.

[64] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *Science*, vol. 315, no. 5814, pp. 972–976, 2007. DOI: 10.1126/science.1136800. eprint: https://www.science.org/doi/pdf/10.1126/science.1136800. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.1136800.

[65] Y.-G. Guéhéneuc, "P-mart: Pattern-like micro architecture repository," *1st EuroPLoP Focus Group on Pattern Repositories*, Jan. 2007.

[66] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, *Unsupervised translation of programming languages*, 2020. arXiv: 2006.03511 [cs.CL].

[67] COVESA, *Genivi alliance*. [Online]. Available: https://github.com/GENIVI/ (accessed 25/1/2024).

[68] Android, *Android auto*. [Online]. Available: https://android.googlesource.com/platform/packages/services/Car/ (accessed 25/1/2024).

[69] M. D. Twa, S. Parthasarathy, C. Roberts, A. M. Mahmoud, T. W. Raasch, and M. A. Bullimore, "Automated decision tree classification of corneal shape," *Optometry and Vision Science: Official Publication of the American Academy of Optometry*, vol. 82, p. 1038, 2005. DOI: 10.1097/01.opx.0000192350.01045.6f.

[70] Y. Yang and G. I. Webb, "Discretization for naive-bayes learning: Managing discretization bias and variance," *Machine Learning*, vol. 74, pp. 39–74, 2009. DOI: 10.1007/s10994-008-5083-5.

[71] N. Friedman and M. Goldszmidt, "Discretizing continuous attributes while learning bayesian networks," in *ICML*, 1996, pp. 157–165.

[72] K. Teknomo, *K nearest neighbor tutorial*. [Online]. Available: http://people.revoledu.com/kardi/tutorial/KNN/ (accessed 6/12/2023).

[73] P. Cunningham and S. J. Delany, "K-nearest neighbour classifiers," 2007.

[74] N. Bhatia, "Survey of nearest neighbor techniques," *arXiv preprint arXiv:1007.0085*, 2010.

[75] A. Nizar, Z. Dong, and Y. Wang, "Power utility nontechnical loss analysis with extreme learning machine method," *IEEE Transactions on Power Systems*, vol. 23, pp. 946–955, 2008. DOI: 10.1109/TPWRS.2008.926431.

[76] *Mining software repositories*. [Online]. Available: https://www.msrconf.org/ (accessed 6/12/2023).

[77] K.-J. Stol, M. Goedicke, and I. Jacobson, "Introduction to the special section—general theories of software engineering: New advances and implications for research," *Information and Software Technology*, vol. 70, pp. 176–180, 2016, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.07.010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584915001330.

[78] Google, *Google colab*. [Online]. Available: https://colab.google/ (accessed 22/1/2024).

[79] D. Parthasarathy, C. Ekelin, A. Karri, J. Sun, and P. Moraitis, "Measuring design compliance using neural language models: An automotive case study," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE 2022, Singapore, Singapore: Association for Computing Machinery, 2022, pp. 12–21, ISBN: 9781450398602. DOI: 10.1145/3558489.3559067. [Online]. Available: https://doi.org/10.1145/3558489.3559067.

[80] J. Brownlee, "How to identify overfitting machine learning models in scikit-learn," Nov. 2020. [Online]. Available: https://machinelearningmastery.com/overfitting-machine-learning-models/ (accessed 8/2/2024).

[81] A. SIGSOFT, *Optimization studies in se (including search-based software engineering)*, 2024. [Online]. Available: https://github.com/acmsigsoft/EmpiricalStandards.

[82] scikit-learn, *Scikit-learn machine learning in python*, 2024. [Online]. Available: https://scikit-learn.org/stable/index.html.

[83] S. Chaturvedi, A. Chaturvedi, A. Tiwari, and S. Agarwal, "Design pattern detection using machine learning techniques," in *2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, 2018, pp. 1–6. DOI: 10.1109/ICRITO.2018.8748282.

[84] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer New York, 2006, ISBN: 978-0-387-31073-2. DOI: https://doi.org/10.1007/978-0-387-45528-0.

[85] K. Pykes, *Introduction to unsupervised learning*. [Online]. Available: https://www.datacamp.com/blog/introduction-to-unsupervised-learning (accessed 1/4/2024).

[86] N. Dunlop, "Investigating the accuracy of metric-based and machine learning approaches in detecting design patterns," Bachelor's Thesis, University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden, 2023.

[87] Y. Wang, S. Mishra, P. Alipoormolabashi, *et al.*, *Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks*, 2022. arXiv: 2204.07705 [cs.CL].

[88] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[89] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, *Codesearchnet challenge: Evaluating the state of semantic code search*, 2020. arXiv: 1909.09436 [cs.LG].

A

Appendix 1

Table A.1: Baseline Performance Metrics for Each Design Pattern and Classifier for RQ2.1.

Classifier	DP	F1 Score	Precision	Recall
Decision Tree	Adapter	0.24	0.24	0.26
	Bridge	0.33	0.35	0.34
	Builder	0.47	0.50	0.48
	Observer	0.50	0.52	0.51
	Prototype	0.73	0.75	0.74
	Singleton	0.47	0.51	0.46
	Strategy	0.32	0.35	0.32
Gradient Boosting	Adapter	0.37	0.35	0.43
	Bridge	0.52	0.51	0.56
	Builder	0.64	0.68	0.63
	Observer	0.68	0.74	0.66
	Prototype	0.82	0.87	0.80
	Singleton	0.62	0.68	0.60
	Strategy	0.44	0.48	0.43
Gaussian Process	Adapter	0.00	0.00	0.00
	Bridge	0.00	0.00	0.00
	Builder	0.14	0.51	0.09
	Observer	0.00	0.00	0.00
	Prototype	0.00	0.00	0.00
	Singleton	0.00	0.00	0.00
	Strategy	0.25	0.14	1.00
KNN	Adapter	0.23	0.22	0.24
	Bridge	0.44	0.41	0.50
	Builder	0.62	0.63	0.63
	Observer	0.61	0.50	0.82
	Prototype	0.81	0.80	0.84
	Singleton	0.57	0.80	0.47
	Strategy	0.21	0.43	0.14
Logistic Regression	Adapter	**0.47**	0.50	0.47
	Bridge	**0.68**	0.66	0.74
	Builder	**0.77**	0.77	0.79
	Observer	0.89	0.88	0.92
	Prototype	**0.93**	0.98	0.90
	Singleton	**0.80**	0.85	0.78
	Strategy	**0.67**	0.72	0.66

Table A.2: Baseline Performance Metrics for Each Design Pattern and Classifier for RQ2.1.

Classifier	DP	F1 Score	Precision	Recall
MLPClassifier	Adapter	0.40	0.43	0.40
	Bridge	0.64	0.63	0.68
	Builder	**0.77**	0.77	0.79
	Observer	**0.90**	0.90	0.92
	Prototype	0.89	0.91	0.90
	Singleton	0.71	0.79	0.67
	Strategy	0.61	0.64	0.61
Naive Bayes	Adapter	0.31	0.31	0.34
	Bridge	0.46	0.50	0.45
	Builder	0.57	0.57	0.59
	Observer	0.70	0.88	0.60
	Prototype	0.67	0.67	0.70
	Singleton	0.59	0.62	0.60
	Strategy	0.35	0.35	0.37
Random Forest	Adapter	0.29	0.31	0.30
	Bridge	0.54	0.53	0.59
	Builder	0.71	0.69	0.75
	Observer	0.79	0.81	0.80
	Prototype	0.83	0.86	0.81
	Singleton	0.66	0.74	0.63
	Strategy	0.43	0.47	0.42
SVM	Adapter	0.32	0.31	0.35
	Bridge	0.54	0.53	0.57
	Builder	0.70	0.68	0.74
	Observer	0.85	0.95	0.79
	Prototype	0.87	0.97	0.81
	Singleton	0.71	0.73	0.72
	Strategy	0.46	0.50	0.45

Table A.3: Performance metrics for each design pattern and classifier for RQ2.1, with tuned hyperparameters.

Classifier	DP	F1 Score	Precision	Recall
Decision Tree	Adapter	0.24	0.24	0.25
	Bridge	0.41	0.45	0.41
	Builder	0.50	0.53	0.51
	Observer	0.55	0.56	0.57
	Prototype	0.75	0.82	0.72
	Singleton	0.49	0.54	0.49
	Strategy	0.29	0.31	0.29
Gradient Boosting	Adapter	0.41	0.46	0.40
	Bridge	0.66	0.63	0.71
	Builder	0.78	0.79	0.79
	Observer	0.88	0.89	0.89
	Prototype	0.90	0.94	0.87
	Singleton	0.75	0.78	0.75
	Strategy	0.64	0.67	0.64
Guassian Process	Adapter	0.38	0.59	0.30
	Bridge	**0.70**	0.66	0.76
	Builder	0.78	0.77	0.82
	Observer	0.88	0.82	0.97
	Prototype	**0.91**	0.94	0.89
	Singleton	**0.81**	0.83	0.82
	Strategy	0.68	0.70	0.69
KNN	Adapter	0.16	0.21	0.15
	Bridge	0.50	0.47	0.56
	Builder	0.67	0.70	0.67
	Observer	0.65	0.52	0.91
	Prototype	0.82	0.80	0.85
	Singleton	0.61	0.79	0.52
	Strategy	0.29	0.47	0.23
Logistic Regression	Adapter	**0.47**	0.58	0.43
	Bridge	0.69	0.71	0.70
	Builder	**0.79**	0.81	0.81
	Observer	**0.91**	0.86	0.98
	Prototype	0.86	0.84	0.91
	Singleton	0.79	0.80	0.81
	Strategy	**0.72**	0.79	0.69

Table A.4: Performance metrics for each design pattern and classifier for RQ2.1, with tuned hyperparameters.

Classifier	DP	F1 Score	Precision	Recall
MLPClassifier	Adapter	0.45	0.51	0.44
	Bridge	0.68	0.67	0.72
	Builder	0.73	0.73	0.76
	Observer	0.90	0.89	0.92
	Prototype	**0.91**	0.94	0.89
	Singleton	0.73	0.79	0.71
	Strategy	0.66	0.68	0.67
Naive Bayes	Adapter	0.31	0.31	0.34
	Bridge	0.46	0.50	0.45
	Builder	0.57	0.57	0.59
	Observer	0.70	0.88	0.60
	Prototype	0.67	0.67	0.70
	Singleton	0.59	0.62	0.60
	Strategy	0.35	0.35	0.37
Random Forest	Adapter	0.29	0.32	0.29
	Bridge	0.56	0.55	0.60
	Builder	0.71	0.70	0.75
	Observer	0.83	0.86	0.82
	Prototype	0.84	0.90	0.81
	Singleton	0.68	0.74	0.66
	Strategy	0.46	0.47	0.47
SVM	Adapter	0.44	0.43	0.48
	Bridge	0.69	0.68	0.72
	Builder	0.77	0.77	0.79
	Observer	0.91	0.92	0.91
	Prototype	0.89	0.91	0.89
	Singleton	0.73	0.82	0.68
	Strategy	0.61	0.68	0.58

B

Appendix 2

Table B.1: The hyperparameters tested for each algorithm and how many combinations of each were tested.

Algorithm	**Parameters tested**	**Combinations**
Gaussian Process	*kernel, n_restarts_optimizer, max_iter_predict*	45
Logistic Regression	*penalty, C, solver, tol, max_iter, class_weight*	4000
Random Forest	*n_estimators, max_depth, min_samples_split, min_samples_leaf, criterion, max_features, bootstrap, class_weight*	576
KNN	*n_neighbors, weights, metric, algorithm, leaf_size, p*	600
SVC	*C, kernel, gamma, degree, class_weight, decision_function_shape*	800
Decision Tree	*max_depth, min_samples_split, min_samples_leaf, criterion, max_features, class_weight, ccp_alpha*	3456
Gradient Boosting	*n_estimators, learning_rate, max_depth, min_samples_split, min_samples_leaf, max_features, loss*	432
Naive Bayes	*var_smoothing*	9
MLP	*hidden_layer_sizes, activation, solver, alpha, learning_rate, learning_rate_init, max_iter, early_stopping, tol*	1080
DBSCAN	*eps, min_samples*	40
KMeans	*init, max_iter, tol*	70
Mean Shift	*bandwidth, bin_seeding, cluster_all, min_bin_freq*	96
Affinity Propagation	*damping, max_iter, convergence_iter, preference*	162
Gaussian Mixture	*covariance_type, reg_covar, n_init, covariance_type, max_iter, init_params*	384
BIRCH	*threshold, branching_factor, compute_labels*	32

Table B.2: Hyperparameters for Classifiers in RQ1.

Classifier	**Hyperparameters**
Logistic Regression	C=0.8, class_weight=None, penalty='l2', solver='lbfgs'
Random Forest	n_estimators=50, criterion='gini', max_depth=5, max_features='sqrt', min_samples_leaf=1, min_samples_split=2, bootstrap=False, class_weight='balanced'
K-Nearest Neighbors	n_neighbors=5, algorithm='auto', leaf_size=5, metric='manhattan', p=1, weights='distance'
SVM	kernel='rbf', C=1, class_weight='balanced', decision_function_shape='ovo', degree=1, gamma='scale'
Decision Tree	ccp_alpha=0.05, class_weight=None, criterion='entropy', max_features=0.3, min_samples_leaf=2, min_samples_split=2
Gradient Boosting	learning_rate=0.5, loss='log_loss', max_depth=3, max_features='log2', min_samples_leaf=4, min_samples_split=2, n_estimators=200, subsample=0.75, init=LogisticRegression(random_state=42, max_iter=10000, C=1, penalty='l1', solver='liblinear')
Naive Bayes	var_smoothing=1e-10
Gaussian Process	copy_X_train=True, kernel=1**2 * RBF(length_scale=2), max_iter_predict=100, multi_class='one_vs_rest', n_restarts_optimizer=0, optimizer='fmin_l_bfgs_b', warm_start=True
MLPClassifier	activation='identity', alpha=0.0001, early_stopping=False, hidden_layer_sizes=(50, 50), learning_rate='invscaling', learning_rate_init=0.1, max_iter=200, solver='sgd', tol=0.0001
Clusterers	**Hyperparameters**
KMeans	n_init='auto', init='k-means++', n_clusters=3, max_iter=200, tol=0.1
Affinity Propagation	convergence_iter=15, damping=0.5, max_iter=100
Mean Shift	bandwidth=None, bin_seeding=False, cluster_all=True, min_bin_freq=1
BIRCH	n_clusters=3, branching_factor=50, compute_labels=True, threshold=0.1
Gaussian Mixture	n_components=3, covariance_type='diag', init_params='random_from_data', n_init=1, max_iter=100, reg_covar=1e-07

Table B.3: Hyperparameters for classifiers and clusterers in RQ2.1.

Classifier	**Hyperparameters**
Logistic Regression	C=0.1, class_weight=None, max_iter=100 penalty='l2', solver='liblinear', tol=0.01
Random Forest	n_estimators=200, criterion='gini', max_depth=None, max_features='sqrt', min_samples_leaf=2, min_samples_split=2, bootstrap=False, class_weight=None
K-Nearest Neighbors	n_neighbors=5, algorithm='auto', leaf_size=5, metric='manhattan', p=1, weights='distance'
SVM	kernel='poly', C=10, class_weight='balanced', decision_function_shape='ovo', degree=1, gamma='scale'
Decision Tree	ccp_alpha=0.05, class_weight=None, criterion='entropy', max_features=None, min_samples_leaf=2, min_samples_split=2, max_depth=10
Gradient Boosting	learning_rate=0.1, loss='log_loss', max_depth=3, max_features='log2', min_samples_leaf=4, min_samples_split=2, n_estimators=100, init=LogisticRegression(random_state=42, max_iter=10000, C=1, penalty='l1', solver='liblinear')
Naive Bayes	var_smoothing=1e-13
Gaussian Process	copy_X_train=True, kernel=1**2 * DotProduct(sigma_0=1), max_iter_predict=100, optimizer='fmin_l_bfgs_b'
MLPClassifier	activation='identity', alpha=1e-05, early_stopping=True, hidden_layer_sizes=(100, 100), learning_rate='adaptive', learning_rate_init=0.01, max_iter=1000, solver='lbfgs', tol=0.001
Clusterers	**Hyperparameters**
KMeans	n_init='auto', init='k-means++', n_clusters=7, max_iter=500, tol=1e-05
Affinity Propagation	convergence_iter=15, damping=0.5, max_iter=100
Mean Shift	bandwidth=None, bin_seeding=False, cluster_all=True, min_bin_freq=1
BIRCH	n_clusters=7, branching_factor=50, compute_labels=True, threshold=0.1
Gaussian Mixture	n_components=7, covariance_type='spherical', init_params='kmeans', max_iter=100, n_init=1, reg_covar=1e-06

Table B.4: Hyperparameters for classifiers and clusterers for GPT-2 in RQ2.2.

Classifier	**Hyperparameters**
Logistic Regression	C=1, class_weight=None, max_iter=1000 penalty=None, solver='sag', tol=0.01
Random Forest	n_estimators=200, criterion='entropy', max_depth=None, max_features='sqrt', min_samples_leaf=2, min_samples_split=5, bootstrap=False, class_weight='balanced
K-Nearest Neighbors	n_neighbors=7, algorithm='auto', leaf_size=5, metric='manhattan', p=1, weights='distance'
SVM	kernel='poly', C=1.2, class_weight='balanced', decision_function_shape='ovo', degree=3, gamma='scale'
Decision Tree	ccp_alpha=0.0, class_weight='balanced', criterion='gini', max_features=0.3, min_samples_leaf=1, min_samples_split=5, max_depth=10
Gradient Boosting	learning_rate=0.1, loss='log_loss', max_depth=3, max_features='log2', min_samples_leaf=2, min_samples_split=2, n_estimators=500, init=LogisticRegression(random_state=42, max_iter=10000, C=1, penalty='l1', solver='liblinear')
Naive Bayes	var_smoothing=1e-13
Gaussian Process	copy_X_train=True, kernel=1**2 * Matern(length_scale=1.5, nu=1.5), max_iter_predict=200, optimizer='fmin_l_bfgs_b', n_restarts_optimizer=5
MLPClassifier	activation='tanh', alpha=1e-05, early_stopping=True, hidden_layer_sizes=(100,), learning_rate='constant', learning_rate_init=0.01, max_iter=1000, solver='lbfgs', tol=0.001
Clusterers	**Hyperparameters**
KMeans	n_init='auto', init='random', n_clusters=3, max_iter=300, tol=1e-05
Affinity Propagation	convergence_iter=15, damping=0.5, max_iter=100
Mean Shift	bandwidth=None, bin_seeding=False, cluster_all=True, min_bin_freq=1
BIRCH	n_clusters=3, branching_factor=50, compute_labels=True, threshold=0.1
Gaussian Mixture	n_components=7, covariance_type='spherical', init_params='kmeans', max_iter=150, n_init=2, reg_covar=1e-06

Table B.5: Hyperparameters for classifiers and clusterers for Instructor-xl in RQ2.2.

Classifier	**Hyperparameters**
Logistic Regression	C=1, class_weight='balanced', max_iter=10000 penalty='l1', solver='saga', tol=0.01
Random Forest	n_estimators=200, criterion='gini', max_depth=5, max_features='log2', min_samples_leaf=2, min_samples_split=2, bootstrap=True, class_weight=None
K-Nearest Neighbors	n_neighbors=5, algorithm='auto', leaf_size=5, metric='euclidean', p=1, weights='distance'
SVM	kernel='poly', C=1.2, class_weight='balanced', decision_function_shape='ovo', degree=1, gamma='auto'
Decision Tree	ccp_alpha=0.01, class_weight='balanced', criterion='entropy', max_features=0.3, min_samples_leaf=1, min_samples_split=2, max_depth=10
Gradient Boosting	learning_rate=0.5, max_depth=5, max_features='sqrt', min_samples_leaf=2, min_samples_split=5, n_estimators=200, init=LogisticRegression()
Naive Bayes	var_smoothing=1e-13
Gaussian Process	copy_X_train=True, kernel=1**2 * RBF(length_scale=5), max_iter_predict=100, optimizer='fmin_l_bfgs_b'
MLPClassifier	activation='logistic', alpha=1e-05, early_stopping=True, hidden_layer_sizes=(100, 100), learning_rate='constant', learning_rate_init=1, max_iter=1000, solver='lbfgs', tol=1e-05
Clusterers	**Hyperparameters**
KMeans	n_init='auto', init='random', n_clusters=3, max_iter=300, tol=0.1
Affinity Propagation	convergence_iter=15, damping=0.5, max_iter=100
Mean Shift	bandwidth=None, bin_seeding=False, cluster_all=True, min_bin_freq=1
BIRCH	n_clusters=3, branching_factor=50, compute_labels=True, threshold=0.1
Gaussian Mixture	n_components=3, covariance_type='spherical', init_params='k-means++', max_iter=100, n_init=2, reg_covar=1e-06

Table B.6: Hyperparameters for classifiers and clusterers for Python in RQ2.3.

Classifier	**Hyperparameters**
Logistic Regression	C=100, class_weight='balanced', max_iter=10000 penalty='l1', solver='liblinear', tol=0.01
Random Forest	n_estimators=100, criterion='gini', max_depth=5, max_features='sqrt', min_samples_leaf=2, min_samples_split=2, bootstrap=False, class_weight='balanced'
K-Nearest Neighbors	n_neighbors=3, algorithm='auto', leaf_size=5, metric='manhattan', p=1, weights='distance'
SVM	kernel='linear', C=0.1, class_weight='balanced', decision_function_shape='ovo', degree=1, gamma='scale'
Decision Tree	ccp_alpha=0.0, class_weight=None, criterion='entropy', max_features=0.3, min_samples_leaf=4, min_samples_split=5, max_depth=10
Gradient Boosting	learning_rate=0.1, max_depth=3, max_features='log2', min_samples_leaf=2, min_samples_split=2, n_estimators=100, init=LogisticRegression()
Naive Bayes	var_smoothing=1e-13
Gaussian Process	kernel=1**2 * RBF(length_scale=5) max_iter_predict=100, optimizer='fmin_l_bfgs_b'
MLPClassifier	activation='tanh', alpha=1e-05, early_stopping=True, hidden_layer_sizes=(50,), learning_rate='invscaling', learning_rate_init=1, max_iter=1000, solver='lbfgs', tol=1e-05
Clusterers	**Hyperparameters**
KMeans	n_init='auto', init='k-means++', n_clusters=3, max_iter=400, tol=0.01
Affinity Propagation	convergence_iter=15, damping=0.5, max_iter=100
Mean Shift	bandwidth=None, bin_seeding=False, cluster_all=True, min_bin_freq=1
BIRCH	n_clusters=3, branching_factor=50, compute_labels=True, threshold=0.1
Gaussian Mixture	n_components=3, covariance_type='spherical', init_params='k-means++', max_iter=100, n_init=1, reg_covar=1e-07

Table B.7: Hyperparameters for classifiers and clusterers for C# in RQ2.3.

Classifier	**Hyperparameters**
Logistic Regression	C=10, class_weight='balanced', max_iter=100 penalty='l2', solver='liblinear', tol=0.01
Random Forest	n_estimators=100, criterion='gini', max_depth=5, max_features='sqrt', min_samples_leaf=2, min_samples_split=2, bootstrap=False, class_weight='balanced'
K-Nearest Neighbors	n_neighbors=5, algorithm='auto', leaf_size=5, metric='manhattan', p=1, weights='distance'
SVM	kernel='linear', C=0.1, class_weight='balanced', decision_function_shape='ovo', degree=1, gamma='scale'
Decision Tree	ccp_alpha=0.07, class_weight='balanced', criterion='gini', max_features=0.3, min_samples_leaf=1, min_samples_split=2, max_depth=20
Gradient Boosting	learning_rate=0.1, max_depth=5, max_features='sqrt', min_samples_leaf=4, min_samples_split=5, n_estimators=200, init=LogisticRegression()
Naive Bayes	var_smoothing=1e-13
Gaussian Process	kernel=1**2 * DotProduct(sigma_0=1), max_iter_predict=100, optimizer='fmin_l_bfgs_b'
MLPClassifier	activation='identity', alpha=1e-05, early_stopping=True, hidden_layer_sizes=(100, 50), learning_rate='adaptive', learning_rate_init=0.1, max_iter=1000, solver='lbfgs', tol=1e-05
Clusterers	**Hyperparameters**
KMeans	n_init='auto', init='k-means++', n_clusters=3, max_iter=100, tol=1e-07
Affinity Propagation	convergence_iter=15, damping=0.5, max_iter=100
Mean Shift	bandwidth=None, bin_seeding=False, cluster_all=True, min_bin_freq=1
BIRCH	n_clusters=3, branching_factor=50, compute_labels=True, threshold=0.1
Gaussian Mixture	n_components=3, covariance_type='spherical', init_params='k-means++', max_iter=150, n_init=2, reg_covar=1e-07