Towards Predicting Architectural Design Patterns: A Machine Learning Approach

Article *in* Computers · October 2022

DOI: 10.3390/computers11100151

Swati Megha
Innopolis University
13 PUBLICATIONS **116** CITATIONS

Manuel Mazzara
Innopolis University
557 PUBLICATIONS **8,066** CITATIONS

computers

MDPI

Article

Towards Predicting Architectural Design Patterns: A Machine Learning Approach

Sirojiddin Komolov *,†, Gcinizwe Dlamini †, Swati Megha † and Manuel Mazzara †

Institute of Software Development and Engineering, Innopolis University, 420500 Innopolis, Russia
* Correspondence: s.komolov@innopolis.ru
† These authors contributed equally to this work.

Abstract: Software architecture plays an important role in software development, especially in software quality and maintenance. Understanding the impact of certain architectural patterns on software quality and verification of software requirements has become increasingly difficult with the increasing complexity of codebases in recent years. Researchers over the years have proposed automated approaches based on machine learning. However, there is a lack of benchmark datasets and more accurate machine learning (ML) approaches. This paper presents an ML-based approach for software architecture detection, namely, MVP (Model–View–Presenter) and MVVM (Model–View–ViewModel). Firstly, we present a labeled dataset that consists of 5973 data points retrieved from GitHub. Nine ML methods are applied for detection of software architecture from source code metrics. Using precision, recall, accuracy, and F1 score, the outstanding ML model performance is 83%, 83%, 83%, and 83%, respectively. The ML model's performance is validated using *k*-fold validation (*k* = 5). Our approach outperforms when compared with the state-of-the-art.

Keywords: object oriented frameworks; software architecture; machine learning; software system design

check for updates

Citation: Komolov, S.; Dlamini, G.; Megha, S.; Mazzara, M. Towards Predicting Architectural Design Patterns: A Machine Learning Approach. *Computers* **2022**, *11*, 151. https://doi.org/10.3390/computers11100151

Academic Editor: Dae-Kyoo Kim

Received: 31 August 2022
Accepted: 7 October 2022
Published: 12 October 2022

1. Introduction

Software architecture (SA) plays a crucial role in software projects because its purpose is to represent functional and non-functional requirements. However, designing SA is a complex process that requires identifying the appropriate quality attributes with respect to the functional requirements, and this process is performed in multiple phases [1,2]. Therefore, conducting a systematic approach on the SA process is fundamental. Buschmann et al. [3] and Shaw and Garlan [4] presented the architectural styles and patterns that can be applied in several general contexts to ensure certain quality attributes when followed on the SA process.

In most of the software products, source code (SC) can be considered as an implementation of the SA. Since SA and SC represent the solutions from different abstraction levels (i.e., high level and low level, respectively), there is a gap between these entities [5], which can lead to several problems with maintaining the consistency and synchronization between software architecture and source code. Tian et al. [6] identified five key relationships that are considered and practiced in industry [7]:

- Transformability, where SA can be implemented by SC;
- Traceability, where SA and its elements can be identified by SC and vice versa;
- Consistency, where SA and SC are consistent with each other—that is, changes made to SA should be reflected on SC and vice versa;
- Interplay, where the quality of SA affects the SC and vice versa;
- Recovery, where information about SA (and its elements) can be obtained by SC.

Hence, we can conclude that the relationship between SA and SC is an important factor that should be explored further.

Especially, we will focus on the Traceability and Recovery relationships between SA and SC by utilizing code metrics to see how information about SA can be extracted by SC, and what the most important features of SC that are reflected by SA are.

Mostly, architects are responsible for choosing architectural patterns from their experience, existing patterns being used in the industry, or the presented patterns in several studies. Architects make decisions considering the business context, functional and non-functional requirements, and constraints from stakeholders. This decision-making process is efficiently practiced in the industry and has been discussed in academia as well [8,9]. As architects use a combination of their experience and available literature and existing practices for the decision, they tend to use their own customized architectural design rather than implementing a well-defined systematic approach.

On the other hand, artificial intelligence techniques have been effectively applied in many problem domains including the software engineering process [10]. The study shows that outstanding results can be obtained when a machine learning approach, which is the sub-field of artificial intelligence, is utilized in software engineering processes addressing the following issues [11]:

- **Representation.** Based on the target goal, what should be the format and representation of the data and knowledge?
- **Characteristics of learning process.** Based on the characteristics of the data, knowledge, and environment, should the learning process be supervised or unsupervised or query or reinforcement learning?
- **Properties of training data and domain theories.** Collecting enough data to achieve satisfactory results. The learning process should consider the quality and correctness of the data.
- **Target function output.** Depending on the output value, the learning process can be binary classification, multi-value classification, and regression.
- **Theoretical underpinnings and practical considerations.** How should the output from the learning process be interpreted?

Based on the foregoing, we can conclude that machine learning techniques can be applied in the software architecture process. As previously stated, the software architecture process is large and complex, since it requires several considerations that should be taken into account, such as the environment in which the problem is being solved, the stakeholder's expectations, and functional and non-functional requirements that affect the final architecture choice [12,13]. However, these considerations can be incorporated one by one to build the architecture, following the divide-and-conquer principle. The main research objective of this study is extracting knowledge from source code metrics, specifically, discovering the extent to which source code metrics can be used together with ML to detect architectural patterns, such as the Model–View–Presenter (MVP) and Model-View–ViewModel (MVVM).

As software architects make decisions about the architecture of a system, the decision gets embedded or implemented in source code. Our goal is using the code metrics to detect the decisions made by the expert. We aim to reach our goal with minimum human interaction. Then, the detected decisions from the source code can be represented as software architectural patterns [3,4].

For the sake of simplicity, the experiments were conducted in a relatively small platform. For a start, we decided to conduct the experiments on a relatively small platform; therefore, the Android platform was selected [14]. Most Android platform projects over the years have been implemented using the Java programming language. However, in recent years, programmers have been gradually shifting to Kotlin programming language for Android application development. The transition from Java to Kotlin is due to the announcement made by Google on becoming the main language for the Android development [15]. Despite the announcement from the tech giant, the Java programming language is still used for software development and the number of open-source Android projects is still growing. Moreover, the Java programming language is an object-oriented and static typed language, which make it easy for static code analysis that might be used for evaluating software architecture [16,17]. Therefore, the Java programming language is chosen for experiments.

This study considers MVP and MVVM as target architectural patterns, since these architectural patterns have become the mainstream for Android projects on architectural decisions and several research works that have tried to predict these architectural patterns, which the results of the current work can be compared with [18–20]. Thus, the dataset for the experiment consists of open-source Android projects written in Java. This study selected GitHub [21], which is one of the largest open-source platforms for creating this project dataset. The source code for open-source Android projects were retrieved from GitHub.

The main contributions of this paper are as follows:

- Machine-learning-based approach for detecting software architecture, namely, MVP and MVVM from source code metrics.
- A dataset consisting of 5973 Java projects retrieved from GitHub.
- The analysis of retrieved dataset using nine popular machine learning models. The dataset and code are freely available for the research community (https://github.com/Rhtyme/architectural-design-patterns-prediction, accessed on 29 August 2022).
- Establishing a set of important source code metrics that can be used to detect MVVM and MVP architectural patterns using machine learning.

The rest of the paper is organized as follows: In Section 3, we discuss the review of the existing literature in this domain. Section 4 presents the dataset, machine learning algorithms, and their evaluation methods being used in the experimentation. Section 5 presents and discusses the results from experiments. Finally, Section 7 concludes the research by demonstrating the main findings, limitations, and future work.

2. Background

This section presents background on software architectural patterns considered in this paper. This section can be skipped by readers already familiar with the matter.

2.1. Model–View–Presenter Architecture

The Model–View–Presenter (MVP) architecture is derived from the Model–View–Controller (MVC) pattern and mostly followed by projects that have a user interface [22]. It facilitates the separation of concerns (i.e., user interface and data processing are developed separately) and simplifies overall development including testing. Figure 1 presents the design of MVP architecture. The three main components/layers that make up MVP architecture are as follows [23]:

(A) Model—represents the data layer that should be displayed and contains business logic;
(B) View—represents user interface, displays the passed data, delegates user-commands to controller layer and does not hold any logic;
(C) Presenter—acts as a controller between model and view; requests data from model layer and passes the received data to View; and handles user-interaction events from the View layer accordingly.

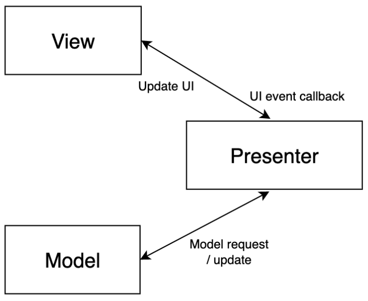

Figure 1. The MVP architecture.

2.2. Model–View–ViewModel Architecture

Model–View–ViewModel (MVVM) architecture is also implemented in projects where user interface (UI) and business logic are the main concerns of development. Figure 2 presents the visual structure of MVVM architecture. The UI is separated from the business logic, and the following three layers compose the MVVM architecture [24]:

(A) Model layer—represents domain or data layer which represents the content;
(B) View layer—responsible for user interface and, compared to MVP, separated by ViewModel layer (which can be considered as an alternative to the Presenter layer of MVP) more, and the interaction between View and ViewModel is handled through so-called data bindings that trigger an event when data are changed or on user input;
(C) ViewModel layer acts as a middle-ware between View and Model, handles user-interactions that it receives by View through binders, and provides a model via data-state whose changes can be observed.

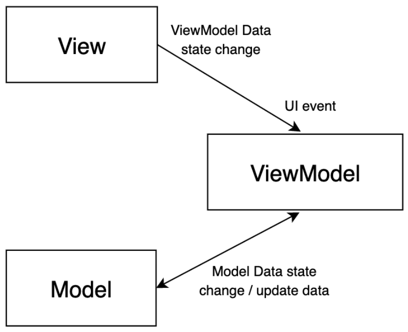
View
ViewModel Data state change
UI event
ViewModel
Model
Model Data state change / update data

Figure 2. The MVVM architecture.

3. Related Work

This section provides information about studies related to architectural or design pattern detection using machine learning. This section can be skipped by readers already familiar with the matter. To the best of our knowledge, limited research has been conducted towards detecting architectural patters from source code metrics using machine learning techniques. Several research works present ways to identify design patterns, anti-patterns, and code smells; nevertheless, only a few of them have tried to detect architectural patterns.

3.1. Architectural Patterns

Daoudi et al. [19] proposed an approach to identify architectural patterns of an Android app through extracting class files from the .apk file, by using heuristic method. A heuristic method involves static code analysis, which examines the significant classes (they filtered the significant classes by eliminating library source code, and helper classes) in the project by their types (i.e., package names) and the model classes, which are responsible for data in apps and are manipulated by input events. Nevertheless, their heuristic method involves manual work of identifying characteristics for each of the architectural patterns. Moreover, the MVVM pattern is not evaluated in their validation phase.

Similarly, Chekhaba et al. [18] proposed an approach to predict architectural patterns in Android projects. The researchers extracted code metrics from open-source projects and used them as input in the several ML algorithms. The main limitation of their work is that they have trained the ML algorithms on a smaller dataset of only 69 projects. Moreover, the researchers manually categorized the classes of the trained dataset into the components of architectural patterns (such as classifying the classes into model class or view class, etc.) and labeled them as specific patterns. In light of the presented results, room for improvement in terms of increasing the dataset and performance of the machine learning models proposed by the researchers still exists.

Dobrean et al. [20] proposed a hybrid approach for detecting architectural patterns, especially MVC. The hybrid approach includes static analysis of source code and unsupervised machine learning methods for clustering. Although the authors achieved quite high accuracy of 85%, the approach depends highly on a particular SDK (a set of libraries) to identify the components of the architecture, which makes the solution useless in the areas where projects are not heavily dependent on the SDKs or libraries. Moreover, the experiments are conducted only on nine open-source and private codebases, which are not sufficient.

3.2. Design Patterns

Other related work is the prediction of design patterns [25] using code metrics and machine learning [26–28]. However, design patterns are quite different since they are implemented to solve common problems and mostly applied to some part of the projects, but we aim to predict architectural patterns that have a global impact on the whole project.

Another relevant work conducted by N. Nazar et al. [29] proposed an approach called DPD_F that applied machine learning algorithms to detect software design patterns. This approach aims to create a semantic representation of Java source code using the code features and applies the Word2Vec algorithm [30] to obtain the numerical representation for feeding into machine learning classifier models. The main contribution of the research work presented by [30] is the large-scale dataset that consists of 1300 Java source code files, which implement one of the software design patterns (or do not implement as False samples). The dataset is equally balanced within the existing labels. The machine learning models achieve around 80% accuracy, which is one of the best performances in this area.

In light of all the existing approaches, one of the contributions of this study is a dataset that minimizes the issue of lack of data. In addition, we propose an approach based on using CK metrics to detect MVVM and MVP patterns. The proposed approach aims at increasing the accuracy for detection of architectural patterns with a possibility of drawing the explanation for classification and relationship to other aspects of software maintenance.

4. Methodology

Figure 3 presents the proposed approach for detecting architectural patterns from the source code metrics. This approach comprises four main stages:

(A) Data Extraction;
(B) Data Preprocessing;
(C) Machine Learning Models;
(D) Performance Evaluation.

The following sections present the details of each phase in the methodology.

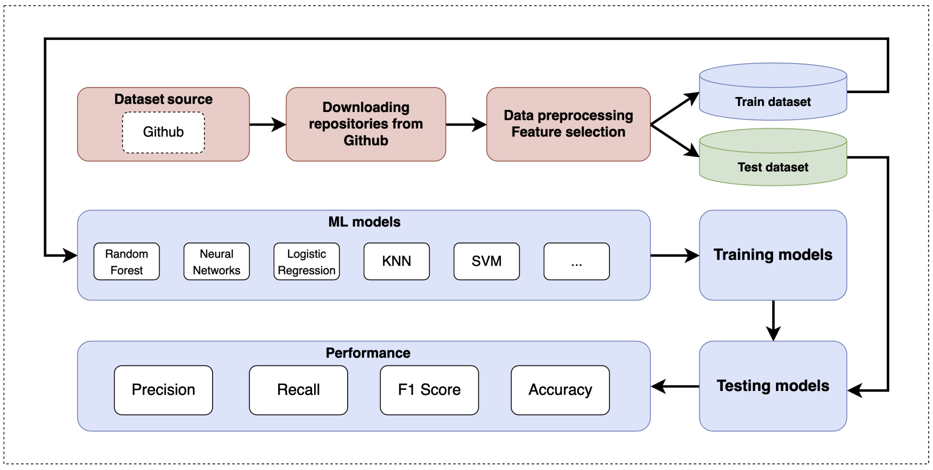

Figure 3. The pipeline of the process execution.

4.1. Data Extraction

Data extraction is one of the challenging tasks that machine learning practitioners and data engineers face. For this work, data were retrieved from the open-source codebase named GitHub. The Github platform has a convenient Application Programming Interface (API) through which repositories with different tags can be retrieved [31]. The API provides an interface to fill the retrieved repositories based on different features, such as programming language and some other different tags. Hence, repositories were retrieved based on the architecture label (i.e., Model–View–ViewModel and Model–View–Presenter). The following queries were used to fetch repositories with the tag Android using MVP [23] and MVVM [24] architectures:

- MVP architecture: ***language:Java Android MVP***
- MVVM architecture: ***language:Java Android MVVM***

GitHub provides a convenient API for searching through queries; however, there are limitations to regulate the usage of the API. Two of the limitations that raised challenges in the our data retrieval from GitHub are as follows:

(A) The limit on the number of requests per minute;
(B) The maximum number of repositories retrieved per query, which is fixed to 1000.

The period starting from 1 January 2013 (this is the date that Android repositories started being uploaded into the GitHub platform) to 16 March 2022 was divided into date segments of 180 days (the inner experiments showed that during 180 days, less than 1000 repositories were created). Further, for each date segment, the GitHub search query was performed. Thus, the final query was as follows: ***Android MVP language:Java created: 2013-01-01..2013-06-29***.

To comply with the restriction of 30 requests per minute, the script was paused for certain seconds using Python's built-in sleep method in the time package. Eventually, 3492 projects in the MVP architecture and 2481 projects in the MVVM architecture were downloaded. The above-explained process can be repeated, and all the scripts with their instructions are presented in the Github repository created by the authors for this research work: https://github.com/Rhtyme/architectural-design-patterns-prediction, (accessed on 29 August 2022).

4.2. Data Preprocessing and Feature Selection

Data preprocessing is a crucial step in machine learning. Using an open-source tool *CK* [32], we extract CK metrics for each retrieved repository from GitHub. The output of the metrics extraction are .csv files, which contain metrics related to method, class, field, and variables. The set of extracted metrics are presented in Appendix A. The metrics are aggregated by summing. Figure 4 presents a visual pipeline of how the data for serving as input to the machine learning models are created.

Most machine learning models such as distance-based methods (i.e., KNN) and deep learning (neural networks) require the data to be normalized. However, some other models, such as decision tree and random forest, require less data curation to learn the underlying structure of the data. For training data-sensitive models, we scale all the numerical values by subtracting the mean value and dividing by the standard deviation. For the tree-based and ensemble learning methods, the data are not scaled.

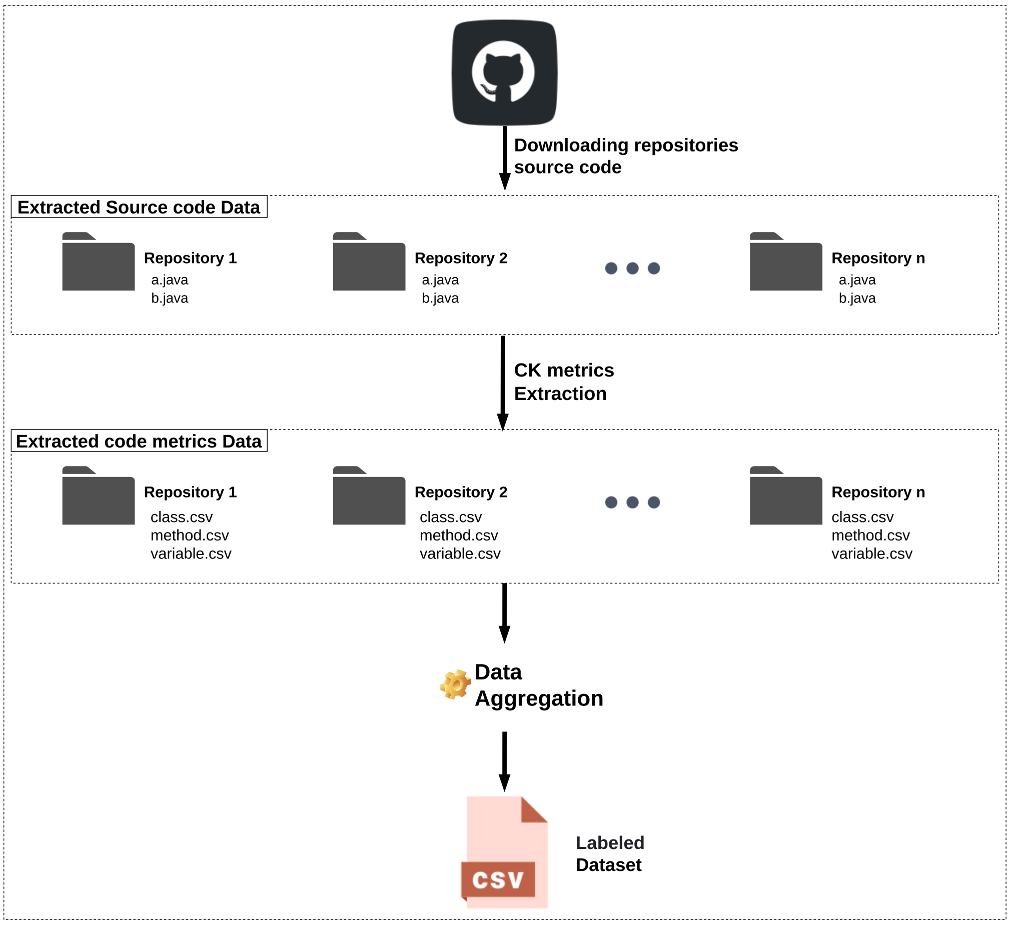

Figure 4. The pipeline for dataset creation.

4.3. Machine Learning Models

The machine learning models have the capability of automatically extracting relevant knowledge from the given datasets by utilizing mathematical models [33,34]. For our work, we chose the nine most popular ML algorithms, which belong to different families starting from simple logistic regression to neural networks. The selected ML models cover almost all the ML algorithms families, starting from simple linear to neural networks. The selected models are as follows: neural network [35], naïve Bayes [36], logistic regression (LR), support vector machine (SVM) [37], decision tree (DT), random forest (RF), K-nearest neighbor (K-NN), catboost (CB) [38], and explainable boosting machine (EBM) [39]. The neural networks architecture and training parameters are presented in Table 1. The number of neurons for each of the hidden layers is the same (number of neurons equals 2).

Table 1. Neural network model architecture and parameters.

Parameter	Value
Number of hidden layers	3
Activation function	rectified linear unit function
Optimizer	adam
Learning rate	0.0001
Batch size	200
Epochs	200

4.4. Performance Metrics

To evaluate the performance of each of the aforementioned machine learning models and validate our proposed approach, four standard evaluation metrics in machine learning were selected: accuracy, precision (P), recall (R), and F1 score ($F1$). The four performance evaluation metrics are computed from a confusion matrix of values, namely, true negatives (TN), true positives (TP), false positives (FP), and false negatives (FN). The metrics are computed as follows:

$$Precision = \frac{TP}{(TP + FP)} \tag{1}$$

$$Recall = \frac{TP}{(TP + FN)} \tag{2}$$

$$F1\text{-}score = 2 * \frac{Precision * Recall}{(Precision + Recall)} \tag{3}$$

$$Weighted\ \ F1\text{-}score = \frac{\sum_{i=1}^{K} Support_i \cdot F1_i}{Total} \tag{4}$$

$$Accuracy = \frac{TP + TN}{(TP + FN + FP + TN)} \tag{5}$$

where $Support$ is the number of samples with a given label i and $F1_i$ is the $F1\text{-}score$ of a given label i. F1 score is an ideal measure since it calculates the harmonic mean between recall and precision. Over the years, the four ML models' performance metrics selected in our research have proved to be reliable and provided a true ML model's performance (i.e., in [18,40]). To validate the performance of the machine learning models, K-fold cross-validations were used where $k = 5$.

5. Results and Discussions

This section presents the implementation details of the proposed approach; machine learning models' performance; and lastly, the discussions of the overall results achieved in this study. We implemented our approach using the Python programming language, open-source Python libraries—sklearn (version 1.1.1) [41] and interpret (version 0.2.7) [39]—for training and testing the ML models. For data extraction, which is the extraction of the CK metrics, we used an open-source Java-based tool called *CK* [32]. The list and description of the extracted CK metrics are presented in Appendix A, Table A1. All the metrics presented in Table A1 are used as input to all the ML models. The experiments were carried out on an Intel Core i5 CPU with 8 GB of RAM.

Figure 5 presents the data retrieved from GitHub and the distribution in the training and evaluation of the machine learning models. It can be seen that our data are balanced, which enhances the learning of the machine learning and avoids the accuracy paradox [42].

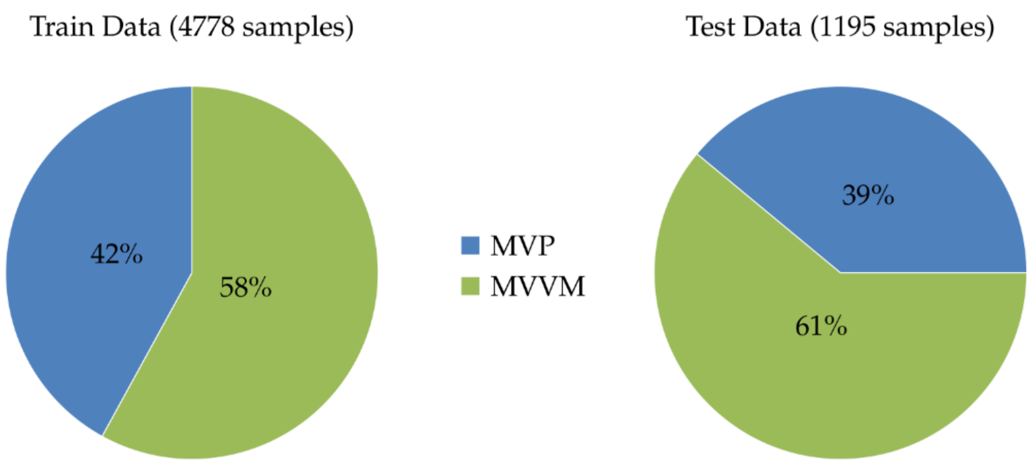

Figure 5. Train and Test data distribution.

This study seeks to identify architectural design patterns using source code metrics as input to a supervised machine learning algorithm. Consequently, the aim is to examine the relationship between source code metrics and the architectural design patterns. Therefore, the following are the research questions (RQs):

RQ1: In the context of machine learning, what are the contributing metrics that enhance the prediction of architectural design patterns?

RQ2: What could be the interpretation of the feature importance concept in the domain of software design?

RQ3: What is the performance of machine learning models in detection of MVVM and MVP architectural patterns in GitHub repositories?

RQ4: What is the complexity and performance of the proposed approach in this paper compared with state-of-the-art?

The following subsections present the discussion of the aforementioned RQs and related results.

5.1. RQ1: About the Selection of Important Features for Predicting Architectural Design Patterns Using Machine Learning Models

With the continuously increasing deployment of machine learning in domains ranging from healthcare to security, the rationale of how the ML models make decisions has become significantly important to understand. This study entails thorough understanding of the most important source code metrics combination that embeds the information about architectural design patterns, specifically MVVM and MVP. Tree-based machine learning models in their nature are interpretable models. Therefore, we select the three tree-based models (RF, CB, and EBM) that achieved outstanding accuracy in our experiments to identify the important source code metrics.

Table 2 presents the ranking of the top 10 important source code metrics for predicting MVVM or MVP pattern using tree-based machine learning models. Based on the results presented in Table 2, it is evident that the number of class default Methods (class_defMethodsQty) and number of returns in method (method_returnsQty) are the most important metrics. This evidence is in line with the theory and semantics of implementing MVVM or MVP using Java programming language. The results presented in Table 2 can serve as a starting point for researchers in designing machine-learning-based methods for detecting architectural design patterns.

Table 2. Features with high importance. Description of each feature name can be found in Appendix A.

Rank	RF	CB	EBM
1	class_defaultMethodsQty	class_defaultMethodsQty	class_defaultMethodsQty
2	method_cbo	class_dit	method_returnsQty
3	method_returnsQty	method_returnsQty	class_dit
4	class_dit	method_cbo	class_defaultFieldsQty
5	method_constructor	method_modifiers	method_cbo
6	method_modifiers	class_innerClassesQty	class_innerClassesQty
7	class_tcc	class_defaultFieldsQty	method_modifiers
8	class_lcc	class_anonymousClassesQty	class_noc
9	class_cbo	class_privateMethodsQty	class_lambdasQty
10	class_lcom	class_noc	method_constructor

5.2. RQ2: About Interpretation of ML Models' Feature Importance in the Context of Software Engineering

The scientific literature of the last three decades has concentrated on the phenomenon of *software architecture degradation* [43,44]. It is intended as the process of continuous divergence between the prescriptive and the descriptive software architecture of a system, i.e., "as intended" vs. "as implemented" software architecture. Previous studies have shown that source code metrics could be used for identifying classes contributing to architectural inconsistencies [45]. This work shows the feasibility of using a machine-learning-based approach for detecting architectural design patterns starting from source code metrics. This process is fundamentally automatic; autonomous; and to a large extent, accurate and reliable.

Upon analyzing the final software artifacts, and comparing the results with expectations and requirements (functional and non-functional), the software designer is able to automatically flag the system for ”architecture degradation“. Further manual inspection may be necessary. Furthermore, maintainability is highly enhanced when architectural patterns are maintained consistent and technical debt is reduced and kept under control, or ideally eliminated [46]. In light of research conducted by other researchers such as [7,45], it would be interesting to study the relationship between less important features (as presented in Table 3) and software architecture degradation or architectural inconsistencies.

Table 3. Features with low ranking based on RF, CB, and EBM feature importance scores. Description of each feature name can be found in Appendix A.

Feature Name	RF Rank	CB Rank	EBM Rank
method_comparisonsQty	67	64	73
class_nosi	72	63	71
class_comparisonsQty	69	68	75
class_finalMethodsQty	78	71	63
method_maxNestedBlocksQty	71	65	77
class_loc	73	72	76
method_logStatementsQty	80	79	79
class_logStatementsQty	79	80	80
method_innerClassesQty	81	81	81
class_synchronizedFieldsQty	82	82	82

5.3. RQ3: About Accuracy of Machine Learning in Detecting MVVM and MVP

Table 4 presents the binary classification of architectural design patterns. For each classifier, precision, recall, accuracy and F1 score are presented. Based on Table 4, the machine learning models are tools that can be used to detect MVVM and MVP architectural patterns from the source code. As presented in Table 4, even simple linear models such as LR can achieve acceptable accuracy above 82%. The cross-validation results prove the stability (since the standard deviation is less than 0.02) and provides confidence that the ML models are not overfitting. However, room for improvement still exists.

Table 4. MVP and MVVM binary classification.

Classifier	Precision	Recall	Accuracy	F1 Score	Cross-Validation F1 Score
LR	0.82	0.82	0.82	0.82	0.82 ± 0.01
Naive Bayes	0.56	0.60	0.60	0.52	0.50 ± 0.02
SVM	0.83	0.83	0.83	0.83	0.82 ± 0.02
Decision Tree	0.70	0.69	0.69	0.69	0.70 ± 0.01
Random Forest	0.81	0.81	0.81	0.81	0.80 ± 0.01
Neural Network	0.83	0.83	0.83	0.83	0.82 ± 0.01
k-NN	0.78	0.78	0.78	0.78	0.77 ± 0.00
CB	0.82	0.82	0.82	0.82	0.82 ± 0.01
EBM	0.82	0.82	0.82	0.82	0.81 ± 0.02

This work can serve as a starting point for many other researchers. Depending on the objective of researchers and software engineers, a model can be chosen from the list in Table 4. For example, if the goal of a programmer is to understand another programmer’s source code and make a mapping to the requirements, any of the high-performing models could be selected. However, to explain the conclusion that a specific pattern is implemented, an explainable model, such as EBM or CB, can be used since these models are, in some sense, easy to interpret compared with black box models (i.e., neural network).

5.4. RQ4: Comparison with the State-of-the-Art

Table 5 presents the results of comparisons of the proposed approach with the state-of-the-art (the comparisons can visually be viewed in Figure 6). We compared our approach with two prominent and recent works ([18,19]). For comparisons, the best-performing ML models were selected according to the results presented in Table 4. The obtained results in Table 5 depict that our approach outperforms the state-of-the-art. However, it has some drawbacks since, in our dataset, only two pattern labels are present while the dataset proposed by Chekhaba et al. [18] has three labels.

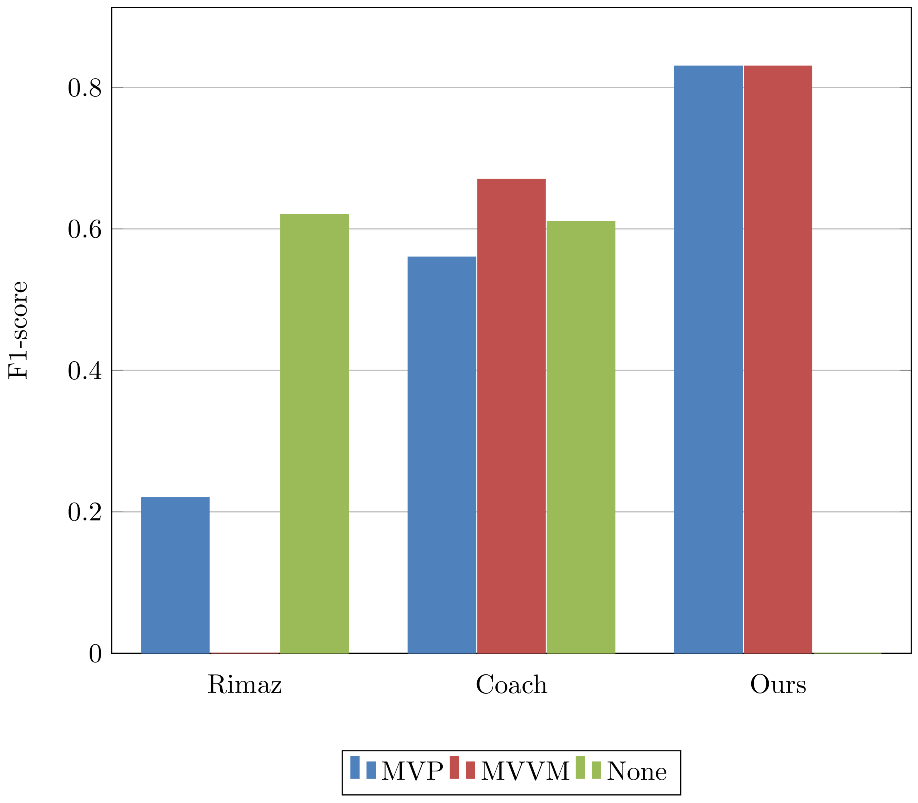

Figure 6. F1–score comparison with state-of-the-art.

Table 5. Comparison with state-of-the-art.

Pattern	Rimaz [19]			Coach [18]			Ours		
	P	R	F1	P	R	F1	P	R	F1
MVP	0.55	0.17	0.22	0.73	0.45	0.56	0.86	0.94	**0.83**
MVVM	0	0	0	0.97	0.51	0.67	0.86	0.94	**0.83**
None	0.52	0.96	**0.62**	0.47	0.84	0.61	0	0	0

According to the results in Tables 4 and 5, it is worth noting that our approach includes a deep learning model, which serves as a foundation for the most recent artificial intelligence (AI)-based solutions or breakthroughs. However, AI models such as neural networks have an issue of consuming more resources, such as time, and require more data for better performance while reducing the interpretability of the results. Researchers in different domains (cyber security [47], healthcare [48,49], finance [50], etc.) have proposed lightweight deep-learning approaches and methods to interpret the results despite the nature of the ML model. Our proposed approach is more inclined to the state-of-the-art interpretable AI and application in the software engineering domain.

Pattern classification results are reported in Table 6 (the report can visually be viewed in the Figure 7). The results show that our proposed approach and dataset can be a contribution to the research community. Most of the machine learning models trained on our dataset correctly classify MVVM and MVP patterns, which are manually annotated. The results presented in the fourth column in Table 6, the ML models were trained on our dataset and evaluated on the dataset from Coach [18]. The catboost model correctly classified all the data samples in the test dataset. This is a good indicator since our study aims at attaining interpretable AI whereby most of approaches are tree-based. One of the drawbacks of the dataset provided by [18] is the size; however, this problem can be solved by the already existing data generation approaches researchers have proposed over the years (i.e., statistical [51] and deep learning [52,53]). The oversampling can be more beneficial in enhancing the training data for data-hungry models such as deep neural networks.

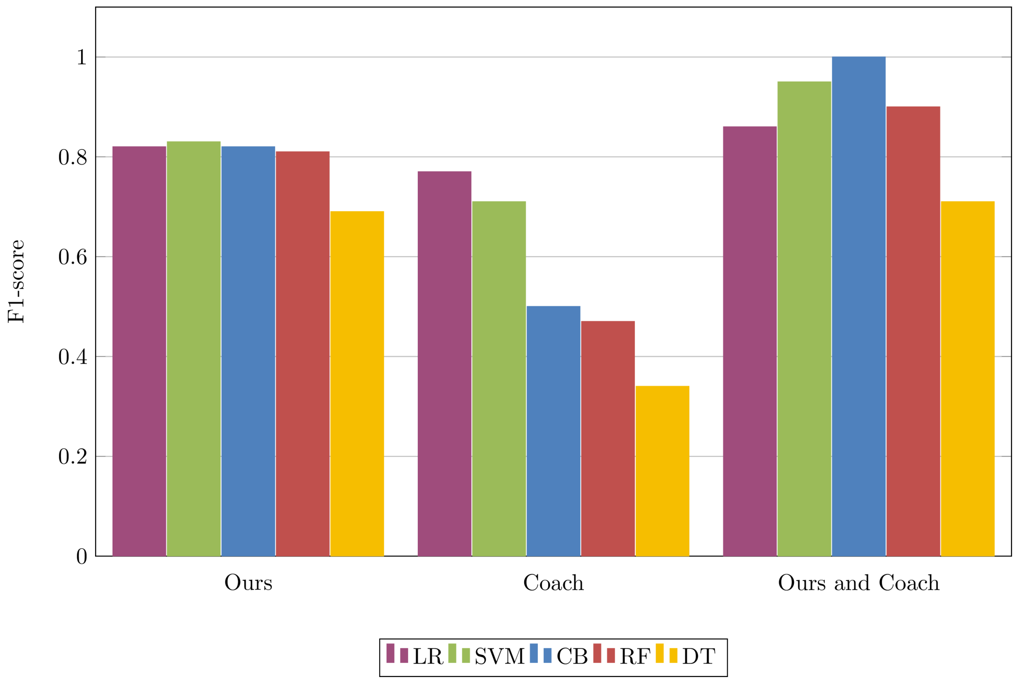

Figure 7. F1–score comparison using dataset from Coach [18] and data proposed in this paper.

Table 6. The results obtained by using the dataset from Coach and data proposed in this paper.

Classifier	**Our Data**			**Coach Data [18]**			**Ours + Coach [18]**		
	P	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
LR	0.82	0.82	0.82	0.79	0.77	0.77	**0.86**	**0.86**	**0.86**
Naive Bayes	**0.56**	**0.60**	**0.60**	0.25	0.31	0.26	0.46	0.52	0.45
SVM	0.83	0.83	0.83	0.77	0.69	0.71	**0.96**	**0.95**	**0.95**
Decision Tree	0.70	0.69	0.69	0.27	0.46	0.34	**0.71**	**0.71**	**0.71**
Random Forest	0.81	0.81	0.81	0.58	0.53	0.47	**0.90**	**0.90**	**0.90**
CB/AdaBoost	0.82	0.82	0.82	0.42	0.61	0.50	**1.00**	**1.00**	**1.00**
EBM	**0.82**	**0.82**	**0.82**	—	—	—	0.81	0.81	0.81

6. Limitations and Threats to Validity

It goes without mentioning that our proposed approach has some limitation and threats to validity. This section presents the prominent threats to validity and limitations.

6.1. External Validity

External validity emerges when the results are generalized to real-world scenarios. This, in machine-learning-based approaches, could refer to the imbalanced dataset and the insufficient number of data points. In this study, such a threat is minimized by splitting the data in such a way that both MVVM and MVP data points are equally represented in the training and testing set (details are in Figure 5).

6.2. Reliability Validity

If the presented approach or experiments cannot be replicated, the approach has reliability threats. The data and source code have been made publicly available, and provided the experiment setups to mitigate the threat in this study. In addition to making the approach publicly available, using public tools has been opted for source code metric extraction tools.

6.3. Limitations

The main limitation of the proposed approach is related to the dataset. The retrieved dataset covers only two architectural design patterns, which are mainly met for Android development, while there are other different types of design patterns such as the ones proposed by the Gang of Four [54]. However, the essence of our approach is the use of source code metrics and exploring the relationship of architectural patterns and code metrics. Despite the main limitation, the approach of using CK metrics to detect design patterns can be easily adopted for even non-architectural design pattern detection.

7. Conclusions and Future Work

This paper presents a machine-learning-based approach for detecting architectural design patterns from source code metrics. First, we automatically retrieved data from GitHub using the GitHub API and made it available to the research community. The data retrieved contain 5973 samples (1195 for testing and 4778 for training). Secondly, we used nine popular machine learning algorithms to detect MVVM and MVP architectures in each sample of the retrieved dataset.

Among the selected machine learning models, an ensemble learning method named Catboost, SVM, and neural Network showed outstanding performance based on F1 score, recall, precision and accuracy. The best-performing ML model achieved precision, recall, accuracy, and F1 score of 83%, 83%, 83%, and 83%, respectively. To validate the performance of the machine learning models, we use cross validation with $k = 5$. The achieved results prove the relationship between source code metrics and architectural design patterns (specifically, MVVM and MVP). In addition to detection of architectural patterns, explainable ML models were used to explore the relationship between architectural patterns and source code metrics. These findings can further be used to identify the correlation between software architecture and other aspects, such as functional requirements, non-functional requirements, architecture degradation, etc.

The future work will first further perform experiments on a set of code metrics and other artifacts such as quality attributes, functional requirements, etc. that can have an impact on software architectural decisions. Secondly, we plan to enrich our dataset by adding more manually annotated data points to create a benchmark dataset. Finally, this study can be the ground for examining the relationships between the less-important features, as found by the machine learning models as in [7].

Author Contributions: Conceptualization, S.K. and M.M.; writing—original draft preparation, S.K. and S.M.; writing—review and editing, G.D. and M.M.; methodology, G.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data used in this research: https://github.com/Rhtyme/architectural-design-patterns-prediction, accessed on 29 August 2022.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Table A1 presents the description of the source code metrics used as predictors for MVVM and MVP architectural design pattern.

Table A1. Descriptions of source code metrics used as ML model features.

Feature Name	Description
class_cbo	Coupling between objects—the number of dependencies of the class
class_cboModified	Modified coupling between objects—this metric is about the number of dependencies of the class, but it also considers the dependencies from the class as being both the reference the type makes to others and the references that it receives from other types
class_fanin	The number of references from other classes to a particular class, i.e., the number of classes that use this class
class_fanout	The number of references from a particular class, i.e., the number of classes referenced by a particular class
class_wmc	Weight Method Class or McCabe's complexity—the number of branch instructions in a class
class_dit	Depth Inheritance Tree—the number of parent classes that a class has
class_noc	Number of Children—the number of classes that inherit a particular class
class_rfc	Response for a Class—the number of unique method executions in a class
class_lcom	Lack of Cohesion of Methods—the number of unconnected method pairs in a class representing independent parts having no cohesion or, in other words, the measure of how well the methods of a class are related to each other
class_lcom	Modified Lack of Cohesion of Methods—this metric just normalizes the LCOM value within 0 and 1
class_tcc	Tight Class Cohesion—the cohesion of a class, normalized within 0 and 1. This metric measures the cohesion of a class by counting the direct connections between visible methods

Table A1. *Cont.*

Feature Name	Description
class_lcc	Loose Class Cohesion—the approach of this metric is similar to TCC, but it also considers indirect connections between visible classes
class_totalMethodsQty	The number of methods in a class
class_staticMethodsQty	The number of static methods in a class
class_publicMethodsQty	The number of public methods in a class
class_privateMethodsQty	The number of private methods in a class
class_protectedMethodsQty	The number of protected methods in a class
class_defaultMethodsQty	The number of methods with default modifier in a class
class_visibleMethodsQty	The number of visible methods in a class
class_abstractMethodsQty	The number of abstract methods in a class
class_finalMethodsQty	The number of final methods in a class
class_synchronizedMethodsQty class_totalFieldsQty	The number of synchronized methods in a class The number of fields in a class
class_staticFieldsQty	The number of static fields in a class
class_publicFieldsQty	The number of public fields in a class
class_privateFieldsQty	The number of private fields in a class
class_protectedFieldsQty	The number of protected fields in a class
class_defaultFieldsQty	The number of fields with default modifier in a class
class_finalFieldsQty	The number of final fields in a class
class_synchronizedFieldsQty	The number of synchronized fields in a class
class_nosi	Number of static invocations—the number of static methods executions
class_loc	Lines of code ignoring the empty lines and comments
class_returnQty	The number of return statements
class_loopQty	The number of loops
class_comparisonsQty	The number of comparisons
class_tryCatchQty	The number of try-catch statements
class_parenthesizedExpsQty	The number of expressions inside parentheses
class_stringLiteralsQty	The number of string literals including repeated literals
class_numbersQty	The number of number literals
class_assignmentsQty	The number of value assignments to an object
class_mathOperationsQty	The number of math operations (i.e., addition, division, multiplication, etc.)
class_variablesQty	The number of variables within a class
class_maxNestedBlocksQty	The highest number of blocks nested together

Table A1. *Cont.*

Feature Name	Description
class_anonymousClassesQty class_innerClassesQty	The number of declared anonymous classes The number of inner classes
class_lambdasQty	The number of lambda expressions
class_uniqueWordsQty	The number of unique words within a class
class_modifiers	The number of declared modifiers (i.e., public, private, protected, etc.)
class_logStatementsQty	The number of log statements
field_usage	The number of field usages
method_constructor	The number of constructor methods
method_line	The number of method lines
method_cbo method_cboModified method_fanin	Coupling between objects—the number of dependencies a method has Modified coupling between objects—this metric is about the number of dependencies of a method, but it also considers the dependencies from the method as being both the reference to others and the references that it receives from others The number of cases that referenced to a method
method_fanout	The number of cases referenced from a method
method_wmc	Weight Method Class or McCabe's complexity—the number of branch instructions in a method
method_rfc	Response for a Class—the number of unique method executions in a method
method_loc	The number of code lines of a method ignoring empty lines and comments
method_returnsQty	The number of return statements in a method
method_variablesQty	The number of variables in a method
method_parametersQty	The number of parameters a method accepts
method_methodsInvokedQty	The number of all direct invocations of methods
method_methodsInvokedLocalQty	The number of all direct invocations of methods of the same class
method_methodsInvokedIndirectLocalQty	The number of all indirect invocations of methods of the same class
method_loopQty	The number of loops in a method
method_comparisonsQty	The number of comparisons in a method
method_tryCatchQty	The number of try-catch statements in a method
method_parenthesizedExpsQty	The number of expressions inside parentheses in a method

Table A1. *Cont.*

method_stringLiteralsQty	The number of string literals including repeated literals in a method
method_numbersQty	The number of number literals in a method
method_assignmentsQty	The number of value assignments to an object in a method
method_mathOperationsQty	The number of math operations (i.e., addition, division, multiplication, etc.) in a method
method_maxNestedBlocksQty	The highest number of blocks nested together in a method
method_anonymousClassesQty	The number of declared anonymous classes in a method
method_innerClassesQty	The number of inner classes in a method
method_lambdasQty	The number of lambda expressions in a method
method_uniqueWordsQty	The number of unique words within a method
method_modifiers	The number of declared modifiers (i.e., public, private, protected, etc.) in a method
method_logStatementsQty	The number of log statements in a method
method_hasJavaDoc	Whether a method has JavaDoc
variable_usage	The number of variable usages
method_innerClassesQty	The number of inner classes within a method
class_nosi	The number of static method invocations within a class

References

1. Garlan, D.; Bass, L.; Stafford, J.; Nord, R.; Ivers, J.; Little, R. Documenting software architectures: Views and beyond. In Proceedings of the 25th International Conference on Software Engineering, Portland, OR, USA, 3–10 May 2003.
2. Bosch, J.; Molin, P. Software architecture design: Evaluation and transformation. In Proceedings of the IEEE Conference and Workshop on Engineering of Computer-Based Systems, Nashville, TN, USA, 7–12 March 1999; pp. 4–10. [CrossRef]
3. Buschmann, F.; Henney, K.; Schmidt, D.C. *Pattern-Oriented Software Architecture, on Patterns and Pattern Languages*; John Wiley & Sons: Hoboken, NJ, USA, 2007; Volume 5.
4. Shaw, M.; Garlan, D. *Characteristics of Higher Level Languages for Software Architecture*; Technical Report CMU/SEI-94-TR-023; Software Engineering Institute, Carnegie Mellon University: Pittsburgh, PA, USA, 1994.
5. Fairbanks, G. *Just Enough Software Architecture: A Risk-Driven Approach*; Marshall & Brainerd: Brainerd, MN, USA, 2010.
6. Murta, L.G.; van der Hoek, A.; Werner, C.M. Continuous and automated evolution of architecture-to-implementation traceability links. *Autom. Softw. Eng.* **2008**, *15*, 75–107. [CrossRef]
7. Tian, F.; Liang, P.; Babar, M.A. Relationships between software architecture and source code in practice: An exploratory survey and interview. *Inf. Softw. Technol.* **2022**, *141*, 106705. [CrossRef]
8. Sahlabadi, M.; Muniyandi, R.C.; Shukur, Z.; Qamar, F. Lightweight Software Architecture Evaluation for Industry: A Comprehensive Review. *Sensors* **2022**, *22*, 1252. [CrossRef] [PubMed]
9. Kazman, R.; Bass, L.; Klein, M.; Lattanze, T.; Northrop, L. A basis for analyzing software architecture analysis methods. *Softw. Qual. J.* **2005**, *13*, 329–355. [CrossRef]
10. Meinke, K.; Bennaceur, A. Machine Learning for Software Engineering: Models, Methods, and Applications. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), Gothenburg, Sweden, 27 May–3 June 2018; pp. 548–549.
11. Zhang, D.; Tsai, J.J. Machine learning and software engineering. *Softw. Qual. J.* **2003**, *11*, 87–119. [CrossRef]
12. Bass, L.; Clements, P.; Kazman, R. *Software Architecture in Practice*; Addison-Wesley Professional: Boston, MA, USA, 2003.
13. Garlan, D. Software architecture: A roadmap. In Proceedings of the Conference on the Future of Software Engineering, Limerick, Ireland, 4–11 June 2000; pp. 91–101.
14. Documentation. Android Development. Available online: https://developer.android.com/docs (accessed on 24 March 2022).
15. Google I/O 2019: Empowering Developers to Build the Best Experiences on Android + Play. Available online: https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html (accessed on 16 September 2022).

16. Dahse, J.; Holz, T. Simulation of Built-in PHP Features for Precise Static Code Analysis. *NDSS* **2014**, *14*, 23–26.
17. Ebad, S.A.; Ahmed, M.A. Measuring stability of object-oriented software architectures. *IET Softw.* **2015**, *9*, 76–82. [CrossRef]
18. Chekhaba, C.; Rebatchi, H.; ElBoussaidi, G.; Moha, N.; Kpodjedo, S. Coach: Classification-Based Architectural Patterns Detection in Android Apps. In Proceedings of the 36th Annual ACM Symposium on Applied Computing, Online, 22–26 March 2021; pp. 1429–1438. [CrossRef]
19. Daoudi, A.; ElBoussaidi, G.; Moha, N.; Kpodjedo, S. An Exploratory Study of MVC-Based Architectural Patterns in Android Apps . In Proceedings of the SAC '19: The 34th ACM/SIGAPP Symposium on Applied Computing, Limassol, Cyprus, 8–12 April 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 1711—1720. [CrossRef]
20. Dobrean, D.; Diosan, L. A Hybrid Approach to MVC Architectural Layers Analysis. In Proceedings of the ENASE, Online, 26–27 April 2021; pp. 36–46.
21. github. *GitHub*; GitHub: San Francisco, CA, USA, 2020.
22. Humeniuk, V. Android Architecture Comparison: MVP vs. VIPER. 2019. Available online: http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1291671&dswid=-8436 (accessed on 29 August 2022).
23. Potel, M. *MVP: Model-View-Presenter the Taligent Programming Model for C++ and Java*; Taligent Inc.: Cupertino, CA, USA, 1996; Volume 20.
24. Gossman, J. Introduction to Model/View/ViewModel Pattern for Building WPF apps. 2005. Available online: https://learn.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps (accessed on 29 August 2022).
25. Gamma, E. *Design Patterns: Elements of Reusable Object-Oriented Software*; Pearson Education: New Delhi, India, 1995.
26. Uchiyama, S.; Washizaki, H.; Fukazawa, Y.; Kubo, A. Design pattern detection using software metrics and machine learning. In Proceedings of the First international Workshop on Model-Driven Software Migration (MDSM 2011), Oldenburg, Germany, 1–4 March 2011; p. 38.
27. Zanoni, M.; Arcelli Fontana, F.; Stella, F. On applying machine learning techniques for design pattern detection. *J. Syst. Softw.* **2015**, *103*, 102–117. [CrossRef]
28. Thaller, H. Towards Deep Learning Driven Design Pattern Detection/submitted by Hannes Thaller. Ph.D. Thesis, Universität Linz, Linz, Austria, 2016.
29. Nazar, N.; Aleti, A.; Zheng, Y. Feature-based software design pattern detection. *J. Syst. Softw.* **2022**, *185*, 111179. [CrossRef]
30. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. *arXiv* **2013**, arXiv:1301.3781.
31. Github Rest Api. Available online: https://docs.github.com/en/rest (accessed on 24 March 2022).
32. Aniche, M. Java Code Metrics Calculator (CK). 2015. Available online: https://github.com/mauricioaniche/ck/ (accessed on 29 August 2022).
33. Crawford, M.; Khoshgoftaar, T.M.; Prusa, J.D.; Richter, A.N.; Al Najada, H. Survey of review spam detection using machine learning techniques. *J. Big Data* **2015**, *2*, 23. [CrossRef]
34. Jacob, S.S.; Vijayakumar, R. Sentimental analysis over twitter data using clustering based machine learning algorithm. *J. Ambient. Intell. Humaniz. Comput.* **2021**. [CrossRef]
35. Hastie, T.; Tibshirani, R.; Friedman, J. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2009.
36. Rish, I. An empirical study of the naive Bayes classifier. In Proceedings of the IJCAI 2001 Workshop on Empirical Methods in Artificial Intelligence, Seattle, WA, USA, 4–10 August 2001; pp. 41–46.
37. Cortes, C.; Vapnik, V. Support-vector networks. *Mach. Learn.* **1995**, *20*, 273–297. [CrossRef]
38. Dorogush, A.V.; Ershov, V.; Gulin, A. CatBoost: Gradient boosting with categorical features support. *arXiv* **2018**, arXiv:1810.11363.
39. Nori, H.; Jenkins, S.; Koch, P.; Caruana, R. InterpretML: A Unified Framework for Machine Learning Interpretability. *arXiv* **2019**, arXiv:1909.09223.
40. Magán-Carrión, R.; Urda, D.; Díaz-Cano, I.; Dorronsoro, B. Towards a reliable comparison and evaluation of network intrusion detection systems based on machine learning approaches. *Appl. Sci.* **2020**, *10*, 1775. [CrossRef]
41. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.
42. He, H.; Ma, Y. *Imbalanced Learning: Foundations, Algorithms, and Applications*; John Wiley & Sons: Hoboken, NJ, USA, 2013.
43. Perry, D.E.; Wolf, A.L. Foundations for the Study of Software Architecture. *SIGSOFT Softw. Eng. Notes* **1992**, *17*, 40–52. [CrossRef]
44. Medvidovic, N.; Taylor, R.N. Software architecture: Foundations, theory, and practice. In Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, South Africa, 1–8 May 2010; Volume 2; pp. 471–472.
45. Lenhard, J.; Blom, M.; Herold, S. Exploring the suitability of source code metrics for indicating architectural inconsistencies. *Softw. Qual. J.* **2019**, *27*, 241–274. [CrossRef]
46. Holvitie, J.; Licorish, S.A.; Spínola, R.O.; Hyrynsalmi, S.; MacDonell, S.G.; Mendes, T.S.; Buchan, J.; Leppänen, V. Technical debt and agile software development practices and processes: An industry practitioner survey. *Inf. Softw. Technol.* **2018**, *96*, 141–160. [CrossRef]

47. Shaukat, K.; Luo, S.; Chen, S.; Liu, D. Cyber threat detection using machine learning techniques: A performance evaluation perspective. In Proceedings of the 2020 International Conference on Cyber Warfare and Security (ICCWS), Online, 20–21 October 2020; pp. 1–6.
48. Ahmad, M.A.; Eckert, C.; Teredesai, A. Interpretable machine learning in healthcare. In Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics, Washington, DC, USA, 29 August–1 September 2018; pp. 559–560.
49. Nayak, S.R.; Nayak, J.; Sinha, U.; Arora, V.; Ghosh, U.; Satapathy, S.C. An automated lightweight deep neural network for diagnosis of COVID-19 from chest X-ray images. *Arab. J. Sci. Eng.* **2021**. [CrossRef]
50. Patron, G.; Leon, D.; Lopez, E.; Hernandez, G. An Interpretable Automated Machine Learning Credit Risk Model. In Proceedings of the Workshop on Engineering Applications, Bogota, Colombia, 7–10 October 2020; pp. 16–23.
51. Chawla, N.V.; Bowyer, K.W.; Hall, L.O.; Kegelmeyer, W.P. SMOTE: Synthetic minority over-sampling technique. *J. Artif. Intell. Res.* **2002**, *16*, 321–357. [CrossRef]
52. Douzas, G.; Bacao, F. Effective data generation for imbalanced learning using conditional generative adversarial networks. *Expert Syst. Appl.* **2018**, *91*, 464–471. [CrossRef]
53. Dlamini, G.; Fahim, M. DGM: A data generative model to improve minority class presence in anomaly detection domain. *Neural Comput. Appl.* **2021**, *33*, 13635–13646. [CrossRef]
54. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design patterns: Abstraction and reuse of object-oriented design. In Proceedings of the European Conference on Object-Oriented Programming, Kaiserslautern, Germany, 26–20 July 1993; pp. 406–431.