Time-series Approaches to Change-prone Class Prediction Problem

Cristiano Sousa Melo, Matheus Mayron Lima da Cruz, Antônio Diogo Forte Martins, José Maria da Silva Monteiro Filho and Javam de Castro Machado

Department of Computing, Federal University of Ceará, Fortaleza-Ceará, Brazil

Keywords: Change-prone Class, Machine Learning, Deep Learning, Recurrent Algorithm, Time-series.

Abstract: During the development and maintenance of a large software project, changes can occur due to bug fix, code refactoring, or new features. In this scenario, the prediction of change-prone classes can be very useful in guiding the development team since it can focus its efforts on these pieces of software to improve their quality and make them more flexible for future changes. A considerable number of related works uses machine learning techniques to predict change-prone classes based on different kinds of metrics. However, the related works use a standard data structure, in which each instance contains the metric values for a particular class in a specific release as independent variables. Thus, these works are ignoring the temporal dependencies between the instances. In this context, we propose two novel approaches, called Concatenated and Recurrent, using time-series in order to keep the temporal dependence between the instances to improve the performance of the predictive models. The Recurrent Approach works for imbalanced datasets without the need for resampling. Our results show that the Area Under the Curve (AUC) of both proposed approaches has improved in all evaluated datasets, and they can be up to 23.6% more effective than the standard approach in state-of-art.

1 INTRODUCTION

During the development phase of a large software project, it can have multiple versions due to changes that are necessary, like bug fix or refactoring, for example. The distribution of a specific version of an application is called release. In general, along this process, the number of classes increases, making arduous software maintenance over the years.

In this context, a change-prone class is a piece of software that probably will change in the next release. Overall, a new software release contains previously existing classes that have been changed, besides new and dropped classes. The change-prone classes require a particular attention (Malhotra and Khanna, 2013). Identifying those classes can enable the developers to focus preventive action on testing, inspecting, and restructuring efforts (Koru and Liu, 2007). Also, when a new development team needs to maintain a software code, it necessary a considerable effort to find the most important classes, that is, those that change more frequently.

Several empirical studies deal with the change-prone class prediction problem. Altogether, these works usually propose different kind of metrics, i.e., independent variables, such as: oriented object metrics (Zhou et al., 2009), C&K metrics (Chidamber and Kemerer, 1994), code smells (Khomh et al., 2009), design patterns (Poisnet et al., 2011) and evolution metrics (Elish and Al-Rahman Al-Khiaty, 2013), in order to maximize performance metrics in Machine Learning algorithms.

However, as a real-world problem, most of the studied datasets are imbalanced. Thus, they can skew the predictive model if not handled correctly (Prati et al., 2009). The most recent works use two different approaches to avoid this problem: oversampling and undersampling. The former generates synthetic data of the minority dependent variable while the latter removes data of the majority dependent variable. Nevertheless, both change the real data history.

Besides, the related works use a standard structure for the dataset, in which each instance contains the metric values for a particular class in a specific release as independent variables. Figure 1 illustrates an example of this standard structure. Note that each line, i.e., instance, represents a class from the software project in a particular release, and its columns contain k metrics, independent variables, M, and one last column representing the dependent variable D. Observe that in this example, the C_w class has three instances, one for each release in which it appears. In practice, the Machine Learning algorithms use the k columns M and the dependent variable D to generate
their models, ignoring the class identifier $C$ and the release number $R$. It means this usual approach is just engaging in learning a pattern to predict which class will suffer modification in the next release, ignoring their history, i.e., the temporal dependence between the instances.

Thus, to the best of the authors’ knowledge, although many works are studying the change-prone class prediction problem, none of them take the temporal dependence between the instances into account to generate predictive models.

In this work, motivated by these issues previously mentioned, we have proposed two time-series approaches, called Concatenated and Recurrent, in order to investigate the importance of keeping the temporal dependence between the instances to improve the quality of the performance metrics of the predictive models. A time-series is simply a series of data points ordered in time, usually for a single variable. However, in our experiments, the time-series is multi-variable, i.e., all independent variables have their time-series to generate the predictive models. These proposed time-series approaches have been compared to the standard structure for designing the dataset. The Recurrent Approach works for imbalanced datasets without the need for resampling. With the purpose of validating the proposed approaches, we have used four open-source projects: a C# project presented by (Melo et al., 2019), and three JAVA Apache projects: Ant, Beam, and Cassandra. Classification models are built using the standard structure, and both the time-series approach proposed in this work. The performance of the predictive models was measured using the Area Under the Curve (AUC).

2 RELATED WORKS

A considerable amount of works in the change-prone class prediction problem has been proposed involving different aspects, like the predictive model for imbalanced or unlabeled datasets, novel predictive metrics, or guidelines in order to support the set of steps.

(Yan et al., 2017) have proposed a predictive model on unlabeled datasets. They have tackled this task by adopting the unsupervised method, namely CLAMI. Besides, they have proposed CLAMI+ by extending CLAMI. CLAMI+, compared to state-of-the-art, aims to learn from itself. They have tested their experiments in 14 open source projects, and they have shown that CLAMI+ slightly improves the results compared to CLAMI.

The work presented by (Malhotra and Khanna, 2017) shows that most datasets in change-prone class prediction problems are imbalanced since they have a minority dependent variable. Then, they evaluate many strategies for handling imbalanced datasets using some resample techniques in six open-source datasets. The results have shown that resample techniques tend to improve the quality of performance metrics.

(Catolino et al., 2018) investigate the use of developer-related factors approach. For this, they use these factors as predictors and compare them with existing models. It has shown the improvement of the performance metrics. Besides, combining their proposal with Evolution metrics (Elish and Al-Rahman Al-Khiaty, 2013) can improve up to 22% more effective than a single model. Their empirical study has been executed in 20 datasets of public repositories.

(Choudhary et al., 2018) have proposed the use of new metrics in order to avoid the commonly used independent variable to generate models in change-prone class prediction problems, like (Chidamber and Kemerer, 1994) metrics or Evolution Metrics. These newly proposed metrics are execution time, frequency or method call, run time information of methods, popularity, and class dependency. They have evaluated their proposed metrics using various versions of open-source software.

(Melo et al., 2019) have proposed a guideline in order to support the change-prone class prediction problem. This guideline helps in decision making related to designing datasets, doing statistical analysis, checking outliers, removing dimensionality with feature engineering, resampling imbalanced datasets, tuning, showing the results, and become them reproducible. They have shown that careful decision making can harshly improve performance metrics like the Area Under the Curve. Furthermore, they have shown that a predictive model that uses an imbalanced dataset has the worst results compared to resampled datasets.

All these studies aforementioned confirm the im-
The importance of creating different strategies to improve the quality of performance metrics. However, none of them take the temporal dependence between the instances into account during the development phase of models. In the next section, we describe two time-series approaches in order to keep the temporal history of each class in their respective release order.

3 TIME-SERIES APPROACHES TO CHANGE-PRONE CLASS PREDICTION PROBLEM

As previously mentioned, the related works use a standard structure for the dataset, in which each instance contains the metric values for a particular class in a specific release as independent variables. So, it ignores the temporal dependence between the instances, since the same class can change in different releases. Thus, by convention, we are going to refer to the standard structure as One Release per Instance (ORI).

In this work, we have proposed two time-series approaches, called Concatenated and Recurrent, in order to exploit the temporal dependence between the instances to improve the quality of the performance metrics of the predictive models. In these new time-series approaches, depending on window size, each instance will contain the total - or partial - history of the metrics values of a class, following its release order.

Thus, initially, to design a dataset according to those novel proposed approaches is necessary to define a window size. The value of the window size parameter will be fixed and will define the number of releases that will compose each instance. In general, we do not know which is the best value for this hyperparameter, so the recommended is to try a range of values and check their performance metrics in order to investigate what window size retrieves the best results. During the dataset building, if the number of releases that a certain class $C$ appears is less than the window size, the data about this class $C$ will be dropped. However, the opposite is not true. If the number of releases that a certain class $C$ appears is greater than the window size, the data about class $C$ will be split, as shown in Figure 2. Note that, in Figure 2, there is a class $C$ that appears in three different releases. Now, suppose that the window size was set as two. In this scenario, the data about class $C$ will be split into two instances, i.e., from release $i$ to release $i+1$ and from release $i+1$ to release $i+2$. For our experiments, the window size is sliding.

The following subsections will present in detail the two time-series approaches proposed in this work, called Concatenated Approach (CA) and Recurrent Approach (RA). The former refers to a structure in which the dataset must be used in Machine Learning algorithms to generate predictive models, while the latter refers to a structure in which must be used in recurrent algorithms, like Gated Recurrent Units (GRU) (Cho et al., 2014), an advanced neural network derived from Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) in Deep Learning algorithms.

3.1 Concatenated Approach (CA)

To generate a dataset according to the Concatenated Approach is necessary, initially, to compute the values of the $k$ metrics for each release $R$ in which a specific class $C$ appears. So, this process will produce $R$ instances, each one with $k$ columns. After that, for each class $C$, it needs to concatenate its $n$ sequential instances, where $n$ is the window size, according to the split process. Note that in this step, the produced instances will contain $n \times k$ columns, as the independent variable. For example, if a class $C$ appears in $R = 4$ releases and the window size $n = 3$, two instances, each one with $k \times 3$ columns will be generated.

Figure 3 illustrates a dataset built according to the Concatenated Approach. Note that $R$ represents the chosen metrics (they can be Oriented Object, Evolution, or Code Smells). Thus, an instance in this dataset contains $n \times k$ columns, i.e., independent variables, respecting the releases order $i, i+1, \ldots, n$ to ensure the use of time-series information.

3.2 Recurrent Approach (RA)

The Recurrent Approach consists of storage the partial or total history into a single instance of a class,
forming a three-dimensional matrix \((x, k, n)\), where \(x\) means the number of classes, \(k\) means the number of metrics, and \(n\) means the window size.

Figure 4 depicts an example of the data structure used in the recurrent approach. \(R\) represents the releases of a certain class \(C\), and it will follow the release order \(i, i+1, \ldots, n\). Each one of these releases contains \(k\) metrics \(M\). In short, this dataset contains \(x\) classes \(C\), i.e., there are \(x\) instances of window size \(n\), where each release has \(k\) metrics. Note that if the window size is less than the number of the release of a class \(C\), this last one will be split, generating more instances into the dataset.

The recurrent approach differs from the concatenated one because, in the former, the recurrent algorithms contain cell state, i.e., it keeps the temporal dependence between the independent variables. In the latter, although the window size increases the number of independent variables into an instance, the Machine Learning algorithms treat each one of them independently.

### 4 EXPERIMENTAL SETTING

Our research methods consist of developing different kinds of predictive models and compare them. For this, we have generated predictive models using three different approaches: one release per instance (ORI), concatenated approach (CA) and recurrent approach (RA), as defined in Section 3.

#### 4.1 Designing the Datasets

In order to evaluate the Time-Series Approaches proposed in this work, we have chosen four software projects. The first dataset was presented in (Melo et al., 2019), and it results from a large C# project. It was used to evaluate a guideline designed to support the different tasks involved in the change-prone class prediction problem. The remaining datasets were built from three JAVA open-source projects: Apache Ant, Apache Beam, and Apache Cassandra, respectively. Each one of these datasets was modeled following three different approaches: one release per instance (ORI), concatenated approach (CA), and recurrent approach (RA).

#### 4.1.1 Independent Variable

In this work, we have chosen as independent variables the same metrics used in (Melo et al., 2019): C&K metrics (Chidamber and Kemerer, 1994), Cyclomatic Complexity (McCabe, 1976), and Lines of Code. Logically, these independent variables were used in all used datasets. To collect all these oriented object metrics, we have used Scitools Understand. The definition of each chosen metric follows:

- **Class Between Object (CBO):** It is a count of the number of non-inheritance related couples with other classes. Excessive coupling between objects outside of the inheritance hierarchy is detrimental to modular design and prevents reuse. The more independent an object is, the easier it is to reuse it in another application;

- **Cyclomatic Complexity (CC):** It has been used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program’s source code;

- **Depth of Inheritance Tree (DIT):** It is the length of the longest path from a given class to the root class in the inheritance hierarchy. The deeper a class is in the hierarchy, the higher the number of methods it is likely to inherit making it more complex;

- **Lack of Cohesion in Methods (LCOM):** It is the methods of the class that are cohesive if they use the same attributes within that class, where 0 is strongly cohesive, and one is lacking cohesive. The cohesiveness of methods within a class...
is desirable since it promotes encapsulation of object, meanwhile, lack of cohesion implies classes should probably be split into two or more sub-classes;

- **Line of Code (LOC):** It is the number of line of code, except blank lines, imports or comments;
- **Number of Child (NOC):** It is the number of immediate sub-classes subordinated to a class in the class hierarchy. It gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class;
- **Response for a Class (RFC):** It is the number of methods that an object of a given class can execute in response to a received message; if a large number of methods can be invoked in response to a message, the testing and debugging of the object becomes more complicated;
- **Weighted Methods per Class (WMC):** The number of methods implemented in a given class where each one can have different weights. In this dataset, each method has the same weight (value equal to one), except getters, setters and constructors, which have weight equal to zero. This variable points that a larger number of methods in an object implies in the greater the potential impact on children since they will inherit all the methods defined in the object.

### 4.1.2 Dependent Variable

In this work, we have adopted as the dependent variable the metric proposed by (Lu et al., 2012). Thus, following its definition, an instance, i.e., an oriented object class, will be labeled as 1 if in the next release occurs alteration in its LOC independent variable, and 0 otherwise.

### 4.1.3 Software Projects Overview

An overview of the software projects used to build the datasets that were employed in order to evaluate the time-series approaches proposed in this work can be seen in Table 1. It shows (i) the project names, (ii) the period in which these projects were developed, (iii) the average percentage of change-prone classes, (iv) the number of releases, and (v) the number of object-oriented classes present in the first and in last releases.

The ratio of the imbalanced dataset is calculated by the cardinality of minority class over the majority class. In this problem, the minority class tends to be that we want to predict. According to Table 1, Apache Ant is the only balanced dataset, while the other datasets are imbalanced.

### 4.2 Generating One Release per Instance (ORI) Models

In order to generate and evaluate predictive models using the One Release per Instance (ORI) approach, we have followed the guideline proposed by (Melo et al., 2019). Initially, we have split each dataset into training and test sets. The following tasks were performed only on the training set. Lastly, we have used the test set to check if the generated models were overfitted.

Four different classification algorithms were used to evaluate the ORI models: Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), and MLP. The source code was developed in Python with the support of the Scikit-Learn libraries (Pedregosa et al., 2012), besides it can be found in our public GitHub repository, which will be detailed in Section 7.

The following subsections will describe in detail the tasks performed over each dataset in order to generate the predictive models based on the ORI approach and following the guideline proposed by (Melo et al., 2019):

#### 4.2.1 Descriptive Statistics

Tables 2, 3, 4, and 5 show the descriptive statistics of the (Melo et al., 2019), Apache Ant, Apache Beam, and Apache Cassandra datasets, respectively. All these tables show the minimum, maximum, mean, median, and standard deviation for each chosen independent variable.

#### 4.2.2 Outlier Detection

Outliers are extreme values that deviate from other observations on the dataset. Outliers are candidates for aberrant data, which may lead to model misspecification, biased parameter estimation, and incorrect results (Liu et al., 2004).

In this work, we have used the Interquartile Range (IQR) to detect outliers. The IQR is the length of the box in the boxplot, i.e., $Q_3 - Q_1$. Here, outliers are defined as instances that are below $Q_1 - 1.5 \times IQR$ or above $Q_3 + 1.5 \times IQR$.

Table 6 shows an overview of the datasets before and after outliers removal.

#### 4.2.3 Normalization

The goal of normalization is to put the values of each column in the same scale. It is essential to avoid that the gradients may end up taking a long time to find the
Table 1: Characteristics of the Software Projects Used to Generate the Datasets.

<table>
<thead>
<tr>
<th>Software Project</th>
<th>Period</th>
<th>Imb. Rate</th>
<th>Releases</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Melo et al., 2019</td>
<td>Jan 2013 - Dec 2018</td>
<td>7.8%</td>
<td>8</td>
<td>762-1693</td>
</tr>
<tr>
<td>Apache Ant</td>
<td>Jun 2016 - May 2019</td>
<td>70.1%</td>
<td>10</td>
<td>1020-19611</td>
</tr>
<tr>
<td>Apache Beam</td>
<td>Oct 2017 - Jun 2019</td>
<td>22.4%</td>
<td>15</td>
<td>36595-73923</td>
</tr>
<tr>
<td>Apache Cassandra</td>
<td>Mar 2010 - Feb 2019</td>
<td>30.3%</td>
<td>49</td>
<td>4244-30308</td>
</tr>
</tbody>
</table>

Table 2: Descriptive Statistics for (Melo et al., 2019).

<table>
<thead>
<tr>
<th>Metric</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Med</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>0</td>
<td>1369</td>
<td>36.8</td>
<td>12</td>
<td>91.2</td>
</tr>
<tr>
<td>CBO</td>
<td>0</td>
<td>162</td>
<td>7.10</td>
<td>3</td>
<td>12.3</td>
</tr>
<tr>
<td>DIT</td>
<td>0</td>
<td>7</td>
<td>0.78</td>
<td>0</td>
<td>1.76</td>
</tr>
<tr>
<td>LCOM</td>
<td>0</td>
<td>100</td>
<td>0.17</td>
<td>0</td>
<td>0.28</td>
</tr>
<tr>
<td>NOC</td>
<td>0</td>
<td>189</td>
<td>0.61</td>
<td>0</td>
<td>6.54</td>
</tr>
<tr>
<td>RFC</td>
<td>0</td>
<td>413</td>
<td>9.96</td>
<td>1</td>
<td>25.6</td>
</tr>
<tr>
<td>WMC</td>
<td>0</td>
<td>56</td>
<td>1.55</td>
<td>0</td>
<td>4.24</td>
</tr>
<tr>
<td>CC</td>
<td>0</td>
<td>488.0</td>
<td>15.9</td>
<td>8</td>
<td>30.3</td>
</tr>
</tbody>
</table>

Table 2: Descriptive Statistics for Apache Ant.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Med</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>0</td>
<td>1586</td>
<td>79.4</td>
<td>37</td>
<td>127</td>
</tr>
<tr>
<td>CBO</td>
<td>0</td>
<td>40</td>
<td>3.21</td>
<td>2</td>
<td>4.26</td>
</tr>
<tr>
<td>DIT</td>
<td>0</td>
<td>7</td>
<td>2.16</td>
<td>2</td>
<td>1.14</td>
</tr>
<tr>
<td>LCOM</td>
<td>0</td>
<td>100</td>
<td>32.5</td>
<td>19</td>
<td>35.2</td>
</tr>
<tr>
<td>NOC</td>
<td>0</td>
<td>166</td>
<td>0.52</td>
<td>0</td>
<td>4.99</td>
</tr>
<tr>
<td>RFC</td>
<td>0</td>
<td>210</td>
<td>23.4</td>
<td>12</td>
<td>27.6</td>
</tr>
<tr>
<td>WMC</td>
<td>0</td>
<td>125</td>
<td>7.79</td>
<td>4</td>
<td>10.0</td>
</tr>
<tr>
<td>CC</td>
<td>0</td>
<td>99</td>
<td>36.2</td>
<td>33</td>
<td>29.7</td>
</tr>
</tbody>
</table>

Table 2: Descriptive Statistics for Apache Beam.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Med</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>0</td>
<td>2749</td>
<td>57.7</td>
<td>22</td>
<td>117</td>
</tr>
<tr>
<td>CBO</td>
<td>0</td>
<td>106</td>
<td>5.33</td>
<td>5.33</td>
<td>7.23</td>
</tr>
<tr>
<td>DIT</td>
<td>0</td>
<td>6</td>
<td>1.62</td>
<td>1</td>
<td>0.75</td>
</tr>
<tr>
<td>LCOM</td>
<td>0</td>
<td>100</td>
<td>17.5</td>
<td>0</td>
<td>28.5</td>
</tr>
<tr>
<td>NOC</td>
<td>0</td>
<td>901</td>
<td>0.44</td>
<td>0</td>
<td>11.2</td>
</tr>
<tr>
<td>RFC</td>
<td>0</td>
<td>100</td>
<td>8.67</td>
<td>6</td>
<td>8.96</td>
</tr>
<tr>
<td>WMC</td>
<td>0</td>
<td>100</td>
<td>4.28</td>
<td>2</td>
<td>5.61</td>
</tr>
<tr>
<td>CC</td>
<td>0</td>
<td>179</td>
<td>5.89</td>
<td>3</td>
<td>9.42</td>
</tr>
</tbody>
</table>

Table 2: Descriptive Statistics for Apache Cassandra.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Med</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>0</td>
<td>45517</td>
<td>103</td>
<td>32</td>
<td>754</td>
</tr>
<tr>
<td>CBO</td>
<td>0</td>
<td>145</td>
<td>5.7</td>
<td>3</td>
<td>8.65</td>
</tr>
<tr>
<td>DIT</td>
<td>0</td>
<td>6</td>
<td>1.6</td>
<td>2</td>
<td>0.74</td>
</tr>
<tr>
<td>LCOM</td>
<td>0</td>
<td>100</td>
<td>20.5</td>
<td>0</td>
<td>30.6</td>
</tr>
<tr>
<td>NOC</td>
<td>0</td>
<td>116</td>
<td>0.27</td>
<td>0</td>
<td>2.3</td>
</tr>
<tr>
<td>RFC</td>
<td>0</td>
<td>294</td>
<td>10.9</td>
<td>4</td>
<td>17.9</td>
</tr>
<tr>
<td>WMC</td>
<td>0</td>
<td>294</td>
<td>6.5</td>
<td>3</td>
<td>12.0</td>
</tr>
<tr>
<td>CC</td>
<td>0</td>
<td>843</td>
<td>13.7</td>
<td>5</td>
<td>31.1</td>
</tr>
</tbody>
</table>

In this work, we have used the min-max normalization presented in equation 1.

\[ x' = \lambda_1 + (\lambda_2 - \lambda_1) \left( \frac{x - \text{min}_A}{\text{max}_A - \text{min}_A} \right) \]  

In the equation 1, \( \lambda_1 \) and \( \lambda_2 \) are values to adjust the data domain to a desired range. The \( \text{min}_A \) and \( \text{max}_A \) are the minimum, and the maximum values of the feature A. The original and the normalized value of the attribute are represented by \( x \) and \( x' \), respectively. The standard range is \([0,1]\).

4.2.4 Resample Techniques

In real-world problems, some datasets can be imbalanced, i.e., a majority class containing most samples while the other class contains few samples, this one generally of our interest (Prati et al., 2009). Using imbalanced datasets to train models leads to higher misclassifications for the minority class. It occurs because there is a lack of information about it. Some strategies to avoid this misclassification are resampling techniques, categorized into:

**Undersampling (US):** refers to the process of reducing the number of samples in the majority class. Three US techniques used in this work are Random Under-Sampler (RUS) (Batista et al., 2004), which consists of randomly choose an instance from the majority class and remove it. Tomek’s Link (TL) (Tomek, 1976), which consists of finding two instances, one from the majority class and the other from the minority class, that are the nearest neighbor of each other and then remove one of the instances. Edited Nearest Neighbours (ENN) (Wilson, 1972),
which consists in removing the nearest instances that do not belong to the class of the instance.

**Oversampling (OS):** consists of generating synthetic data in the minority class in order to balance the proportion of data. Three OS techniques used in this work are Random Over-Sampler (ROS) (Batista et al., 2004), which consists of randomly choose an instance from the minority class and repeat it. Synthetic Minority Oversampling Technique (SMOTE) (Chawla et al., 2002), which consists of generating new synthetic instances obtained by combining the features of an instance and its k-nearest neighbors. Adaptive Synthetic (ADASYN) (He et al., 2008), which consists in creating synthetic instances of the minority class considering the difficult to classify this instance with a KNN algorithm, then it will focus on the harder instances.

### 4.2.5 Cross-validation

In order to validate the stability of the generated predictive models, we use a cross-validation technique to give information on how well the learner will generalize to an unseen dataset (Arlot and Celisse, 2009). As specified before, the model will be generated over the train set; and cross-validation will split it into train and validation sets. In the end, the test set will serve as data that represents the real scenario. The test set is original, i.e., outlier removal or resamples techniques were not performed on it.

The cross-validation technique used in this work was stratified k-fold since some datasets are imbalanced. Hence, a proportional amount of minority labels is necessary during the phase of the split in k-fold. According to (James et al., 2013), a suitable k value in k-fold is k = 5 or k = 10.

### 4.2.6 Performance Metrics

A confusion matrix gives an overview of output were sorted correctly and incorrectly, classifying them as True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN). The following metrics are widely used to validate performance results:

\[
\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2}
\]

\[
\text{precision} = \frac{TP}{TP + FP} \tag{3}
\]

\[
\text{sensitivity or recall} = \frac{TP}{TP + FN} \tag{4}
\]

\[
specificity = \frac{TN}{TN + FP} \tag{5}
\]

To evaluate a predictive model for the classification problem is necessary to select the appropriate performance metrics according to two possible scenarios: for a balanced or imbalanced dataset. Since most of our datasets are imbalanced, metrics like accuracy can lead to dubious results (Akosa, 2017).

The more suitable metrics for imbalanced datasets is the Area Under the Curve (AUC) and F-score (Equation 6). These performance metrics are suitable because they take the minority classes correctly classified into account, unlike the accuracy.

\[
F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}} \tag{6}
\]

For these reasons, we have compared the predictive models using AUC. However, we have also provided other performance metrics like accuracy, specificity, sensitivity, and F1 (F-Score).

### 4.3 Generating Time-series Models

In this subsection, we are going to detail the process performed to obtain our best results in the concatenated and recurrent approaches, presented in Section 3.

#### 4.3.1 Concatenated Approach

For concatenated approach, we have evaluated three different values for the window sizes: 2, 3, and 4. Thus, for each previously selected dataset, we have generated three distinct versions, according to the window size. Next, we have performed the guideline proposed by (Melo et al., 2019). However, for the concatenated approach, we neither perform the outlier detection nor the feature selection steps.

Besides, four different classification algorithms were used to evaluate the concatenated approach: Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), and MLP. The source code was developed in Python with the support of the Scikit-Learn libraries (Pedregosa et al., 2012), besides it can be found in our public GitHub repository, which will be detailed in Section 7.

#### 4.3.2 Recurrent Approach

For recurrent approach, we have evaluated three different values for the window sizes: 2, 3, and 4. Thus, for each previously selected dataset, we have generated three distinct versions, according to the window size. However, unlike the concatenated approach, we
did not use the guideline proposed by (Melo et al., 2019). We have just built the data structure into a Tensor using Keras and uses it with the Gated Recurrent Unit (GRU) algorithm. So, we have used imbalanced datasets. The GRU structure used in recurrent approach is presented next:

```python
Begin
model = Sequential()
model.add(GRU(256, input_shape=input_shape,
    return_sequences=True))
model.add(Dropout(0.3))
model.add(GRU(128, return_sequences=True))
model.add(Dropout(0.3))
model.add(GRU(64))
model.add(Dropout(0.3))
model.add(Dense(output_dim,
    activation='softmax'))
model.summary()
End
```

In this approach, we have used three layers of GRU, from 256 entry to 1, and its activation function is softmax. We also added three Dropout layers (set as 0.3) in order to avoid overfitting. The number of epochs was set by 20, the number of batches was set by 512, and the number of k, in stratified k-fold, was set by 5.

5 EXPERIMENTAL EVALUATION

In order to evaluate the proposed approaches, they have been compared to the one release per instance (ORI) approach. Classification models are built using the four selected datasets and three different values for the window size (2, 3, and 4). The performance of the predictive models was measured using the Area Under the Curve (AUC).

5.1 Analyzing ORI Approach vs Time-Veries Approaches

Table 7 shows the best predictive model found for each dataset, Melo (Melo et al., 2019), Apache Ant, Apache Beam, and Apache Cassandra, using the One Release per Instance (ORI) Approach. So, Table 7 illustrates the (i) resample technique used, (ii) the algorithm performed, (iii) if outlier removal was used, (iv) the spent time for the execution, and (v) the values of performance metrics. For the (Melo et al., 2019) dataset, only AUC has been provided in their paper, so the other performance metrics are missing. Note that, for each dataset was necessary to run some resample technique since an imbalanced dataset tends to provides the worst results. Besides, another interesting result was that no dataset presented the best result using outlier removal.

Table 8 shows the best predictive model found for each dataset using the Concatenated Approach (CA). It is important to note that for (Melo et al., 2019) dataset, the window size 4 was not used due to the small number of C# classes present in 4 or more releases.

Table 9 depicts the best predictive model found for each dataset and for each window size using the Recurrent Approach (RA). In this scenario, we have used only the imbalanced datasets and a recurrent algorithm.

5.2 Discussion of Results

One of the goals of this research is to experimentally evaluate how time-series approaches tend to improve the performance metrics of predictive models used to solve the change-prone class prediction problem over imbalanced datasets.

Given the results discussed in the last subsection, we have decided to get the best AUC metric obtained from each dataset and compare them according to different approaches. Figure 5 shows the results of this comparison, which x-axis means the AUC and y-axis means the best scenario from each dataset in different approaches.

For all evaluated datasets, the proposed time-series approaches outperformed the ORI approach. Note that, in Apache Cassandra, we had the best performance in the ORI approach with the AUC 0.7432 using SMOTE resample with MLP while the Recurrent Approach provided the AUC 0.9188, up to 23.6% more effective than the standard approach in literature. Besides this improvement, we highlight the reduction of execution time in this dataset. For the ORI approach, the execution time was 37 minutes while in CA we spent nearly 34 minutes, and RA only took 8 minutes.

We want to highlight the difference of AUC between the baselines in all approaches, i.e., neither in ORI nor in Concatenated we perform outlier detection or resample technique. According to Figure 6, time-series approaches have the best result overall ORI approach. In this figure, the x-axis means the AUC and y-axis means the best scenario from each dataset in different approaches. The results in the ORI approach have been the worst since no treatment was performed. It shows that time-series approaches, specially Recurrent ones, deal with imbalanced scenarios.
6 THREATS TO VALIDITY

We identify some threats to validity in this experimental analysis:

1. **Amount of releases.** For our experiments, all datasets described in Table 1 contain a reasonable amount of releases. It allowed us to test different window sizes. When we have decided to use time-series approaches, it is interesting to check more than one window size to investigate different values.

2. **Dropped releases.** It is necessary to keep in mind that when we have a large window size, all classes that contain less release that window size is dropped. Even that the result can be harshly improved, the large window size can drop a considerable amount of releases, and it can compromise the history of the project.

3. **Amount of datasets.** In our experiment, we test our novel approaches in four datasets. We cannot generalize the results of using these four datasets to ensure that time-series is always going to retrieve...
7 REPRODUCIBILITY

To reproduce all experiments presented in this research, we provide public access in a GitHub repository\(^1\). All scripts have used Jupyter Notebook in order to keep track of how this research was designed, run, and analyzed. We set random seed as 42, in order to fix the randomness. Besides, we also available the following content:

1. Scripts to generate raw datasets from Apache repositories.
2. All generated datasets (release-by-release or final versions) in CSV.

Information containing the exact version of all libraries or external programs used in this research is available in a README file in our repository.

---

8 CONCLUSIONS

In this work, we presented two novels time-series approaches, called Concatenated and Recurrent, to solve the change-prone class prediction problem over imbalanced datasets. The former aims to concatenate releases side by side in order to generate predictive models using ordinary Machine Learning algorithms, and the latter designs a data structure into a tensor to generate predictive models using recurrent algorithms. The Recurrent Approach uses the concept of cell state to keep the dependencies between the releases. Besides, the Recurrent Approach works well for imbalanced datasets without the need for resampling.

In order to evaluate the proposed time-series approaches, they have been compared to the one release per instance (ORI) approach. Classification models are built using the four large open-source projects and three different values for the window size (2, 3, and 4). The performance of the predictive models was measured using the Area Under the Curve (AUC). The experimental results show that the Area Under the Curve (AUC) of both proposed approaches has improved in all evaluated datasets, and they can be up to 23.6% more effective than the standard approach in state-of-art. Furthermore, in some cases, the execution time harshly fell. We highlight that for the recurrent approach, resample techniques have not been performed, i.e., besides the gain in execution time, we can keep the original data history.

As future works, we plan to perform an extensive experimental analysis of the proposed approaches using a wide range of public imbalanced and balanced datasets. Moreover, we will investigate other data layouts that can be used in recurrent or deep learning algorithms, like Long Short-Term Memory (LSTM), in order to improve the performance metrics.

ACKNOWLEDGEMENTS

This research was funded by LSBD/UFC.

REFERENCES


James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning with Applications in R*. Springer.


