Supplementary Material

This file contains the following information. The first part provides details of the used software fault datasets including details of the software metrics. The second part provides details of the GAN models with the used values of the control parameters. The third part contains details of the machine learning techniques with the values of the control parameters.

1. Used software fault datasets

Eclipse datasets were collected by D'Ambros et al. [1], PROMISE datasets were prepared by Jureczko and Madeyski [2], and JIRA is a new repository of highly-curated defect datasets collected by Yatish et al. [3].

Table 1: Description of the used software fault datasets

Project	**Release**	**Description**	**Programming language**
Prop 1	V4, V40, and V185	Proprietary software projects that provide custom build solutions and successfully installed in the customer environment.	Object-oriented programming language (Class-level)
Prop 2	V9, V44, V128, V164, and V192		
Prop 3	V225, V236, V245, and V265		
Prop 4	V285, V292, and V305		
Prop 5	V347 and V362		
Prop 6	V452 and V453	A standard tool that supports quality assurances in software development.	Object-oriented programming language (Class-level)
JDT_core	Eclipse	JDT Core is the Java infrastructure of the Java IDE.	Java programming language
Equinox	Eclipse	It is a module runtime that allows developers to implement an application as a set of "bundles" using the common services infrastructure.	
Mylyn	Eclipse	It is the task and application lifecycle management (ALM) framework for Eclipse.	
PDE_UI	Eclipse	It is a Plug-in Development Environment (PDE) that provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and RCP products.	
Lucene	Apache	It is a Java library providing powerful indexing and search features, as well as spell checking, hit highlighting and advanced analysis/tokenization capabilities.	
ActiveMQ	5.0.0	Messaging and Integration Patterns server	Java programming language
Derby	10.5.1.1	Relational Database	
Groovy	1_6_Beta_1	Java-syntax-compatible OOP for JAVA	
HBase	0.94.0	Distributed Scalable Data Store	
Hive	0.9.0	Data Warehouse System for Hadoop	
JRuby	1.1	Ruby Programming Language for JVM	
Wicket	1.3.0-beta2	Web Application Framework	

Table 2: Description of the software metrics

Datasets	Software metrics	Source
Prop datasets (20 software metrics)	LOC (Lines of Code), WMC (Weighted Methods per Class), NPM (Number of Public Methods), AMC (Average Method Complexity), Max cc and Avg cc (Max and Average McCabe's Cyclomatic Complexity), MOA (Measure of Aggregation), CBO (Coupling between Object Classes), RFC (Response for a class), CA (Afferent Couplings), EC (Efferent Couplings), IC (Inheritance coupling), CBM (Coupling between methods), LCOM (Lack of Cohesion in Methods) and LCOM 3, CAM (Cohesion among methods of classes), DIT (Depth of Inheritance Tree), NOC (Number of Children), MFA (Measure of Functional Abstraction), DAM (Data Access Metric)	[2]
Eclipse datasets (18 software metrics)	CBO (Coupling between Object Classes), DIT (Depth of Inheritance Tree), fanIn, fanOut, LCOM (Lack of Cohesion in Methods), NOC (Number of Children), numberOfAttributes, numberOfAttributesInherited, numberOfLinesOfCode, numberOfMethods, numberOfMethodsInherited, numberOfPrivateAttributes, numberOfPrivateMethods, numberOfPublicAttributes, numberOfPublicMethods, RFC (Response for a class), WMC (Weighted method count)	[1]
JIRA datasets (64 software metrics)	CountDeclMethodPrivate, AvgLineCode, CountLine, MaxCyclomatic, CountDeclMethodDefault, AvgEssential, CountDeclClassVariable, SumCyclomaticStrict, AvgCyclomatic, AvgLine, CountDeclClassMethod, AvgLineComment, AvgCyclomaticModified, CountDeclFunction, CountLineComment, CountDeclClass, CountDeclMethod, SumCyclomaticModified, CountLineCodeDecl, CountDeclMethodProtected, CountDeclInstanceVariable, MaxCyclomaticStrict, CountDeclMethodPublic, CountLineCodeExe, SumCyclomatic, SumEssential, CountStmtDecl, CountLineCode, CountStmtExeRatio, CommentToCode, CountLineBlank, CountStmt, MaxCyclomaticModified, CountSemicolon, AvgLineBlank, CountDeclInstanceMethod, AvgCyclomaticStrict, PercentLackOfCohesion, MaxInheritanceTree, CountClassDerived, CountClassCoupled, CountClassBase, CountInput_Max, CountInput_Mean, CountInput_Min, CountOutput_Max, CountOutput_Mean, CountOutput_Min, CountPath_Max, CountPath_Mean, CountPath_Min, MaxNesting_Max, MaxNesting_Mean, MaxNesting_Min, COMM, ADEV, DDEV, Added_lines, Del_lines, OWN_LINE, OWN_COMMIT, MINOR_COMMIT, MINOR_LINE, MAJOR_COMMIT, MAJOR_LINE	[3]

2. GAN models parameters

2.1 Wasserstein Distance

The Wasserstein Distance is a metric for comparing the distance between two probability distributions. Since it can be defined informally as the least amount of energy needed to transfer and transform a pile of dirt in the shape of one probability distribution into the shape of the other, it is also known as Earth Mover's distance, or EM distance. The cost is the multiplication of the amount of dirt transported and the distance travelled. The Wasserstein-1 distance is difficult to calculate, but it can be approximated by switching the GAN objective to the equation.

$$\min_G \max_D \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[D(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}}[D(G(\boldsymbol{z}))]$$

As long as D is a k-Lipschitz function. Clipping D's weight to lie in a compact space [-c, c], e.g. c = 0.01, satisfies this restriction. To implement WGAN with weight-clipping, we must make two basic changes to Vanilla GAN: the loss function logs must be deleted, and the discriminator weights must be clipped. WGAN with weight-clipping usually works better than Vanilla GAN. Weight-clipping, on the other hand, limits the discriminator's power, causes the weights to be moved to the two extremes of the allowed range [-c, c], and can result in exploding or disappearing gradients.

2.2 GAN parameters used in the source code

Table 3: Parameter values set for different GAN models

Model	Parameter values
Vanilla GAN	noise_dim = 32, dim = 128, batch_size = 32, log_step = 100, epochs =100, learning_rate = 5e-4
WGANGP	gan.fit(X_train, y=y_train.values, condition=True, epochs=100, batch_size=64, <Generator parameter> netG_kwargs = {'hidden_layer_sizes': (128,64), 'n_cross_layers': 1, 'cat_activation': 'gumbel_softmax', 'num_activation': 'none', 'condition_num_on_cat': False, 'noise_dim': 32, 'normal_noise': False, 'activation': 'leaky_relu', 'reduce_cat_dim': True, 'use_num_hidden_layer': True, 'layer_norm':False,}, <Discremenator parameter> netD_kwargs = {'hidden_layer_sizes': (128,64,32), 'n_cross_layers': 2, 'embedding_dims': None, 'activation': 'leaky_relu', 'sigmoid_activation': False, 'noisy_num_cols': True, 'layer_norm':True,})

CTGAN	model = CTGAN(epochs = 100, batch_size=32, generator_dim=(256, 256, 256), discriminator_dim=(256, 256, 256))

3. Machine Learning Models

In this paper, we have used five different machine learning techniques to build the prediction models that classify the software fault module as faulty or non-faulty in the given software systems. The description of these techniques are given below.

3.1. K-nearest neighbor (KNN): KNN is a supervised machine learning algorithm. It uses statistical inferences to predict the label of data samples. We can use KNN to solve classification as well as regression problems. It is a non-parametric technique. It uses a simple voting scheme to predict the class of a test sample. The majority class among the k nearest neighbors of a given test sample is predicted as the class of that test sample. It uses distance measures like Euclidean distance to calculate the distance between samples and to select the k neighbors of the given test sample. Usually, k is a small positive integer like k = 3. Value of k determines the efficiency of KNN.

3.2. Random Forest (RF): It is an ensemble technique and we can use it to solve classification as well as regression tasks. RF works as follows: It generates many decision trees on the different subsets of input training data. Then, it combines these trees to predict on the unseen testing data samples. RF uses many decision trees for the purpose of training and testing and the mode and mean of the prediction from these trees is reported as the actual prediction value. It can be said of as the correction for the decision tree and it does not overfit the decision tree of the training set.

3.3. Decision tree (DT): It is a classification technique, which creates a tree type structure for classification. It represents attributes of the dataset as internal nodes of the tree and class labels of the data as leaf nodes in the tree. The root of the tree is selected as the best attribute of the data. Then, training data is split into subsets over attributes, where subsets are made in such a way that each subset only contains data that has the same value for an attribute. Until all branches of the tree find leaf nodes, the above two steps are repeated. Concept of weighted tree is used by DT classifier, where the internal nodes are marked with features used for classification and edges of the tree created are marked as a trial with dataset weight. Class labels are used for naming of tree leaves. Using this method, going through the branches, the complete dataset can be classified from root until the leaf node is reached. DT adopts a tree-like structure for learning, which eventually maps information available in the dataset features to its expected value.

3.4. Naive Bayes (NB): It's a statistical inference based technique that takes use of Bayes' theorem. It updates the probability of a hypothesis continuously as more supporting evidence for that hypothesis becomes available. Naive Bayes assumes that dataset features are independent and one feature does not affect the importance of another feature. NB uses conditional probability to predict it's output. It evaluates the probability of an unseen event by using the prior probability of available events.

3.5. Logistic regression (LR): LR uses statistical inference. With the help of some given data, it evaluates the probability of occurring of an unseen event. LR is generally used for binary class classification (either 0 or 1). It does an estimation on the natural logarithm function to find the relation between the variables. It calculates the probability of occurrence of an event by using odds ratio, in which a ratio between odds of an event happening to it's not happening.

Table 4: Parameter values set for different machine learning techniques

Technique	**Parameter values**
K-Nearest Neighbour (KNN)	n_neighbors = range(1, 21, 2), weights = ['uniform', 'distance'], metric = ['euclidean', 'manhattan', 'minkowski']
Naïve Bayes (NB)	With default parameter in scikit learn
Logistic Regression (LR)	solvers = ['liblinear'], penalty = ['l2'], c_values = [100, 10, 1.0, 0.1, 0.01]
Decision Tree (DT)	depth=np.arange(1,40), grid="criterion":["gini","entropy"], "max_depth":mdepth}
Random Forest (RF)	'bootstrap': [True], 'max_depth': [2,4,5], 'max_features': ["log2","sqrt"], 'min_samples_leaf': [10,30], 'min_samples_split': [10,12], 'n_estimators': [100, 350, 500]

4. Used class imbalance techniques

We use six different class imbalance techniques for the comparative analysis.

4.1 Synthetic Minority Oversampling Techniques (SMOTE): SMOTE is the most common oversampling technique to handle class imbalance problem. It is based on the KNN algorithm. SMOTE generates new synthetic data instances by first selecting the □ nearest neighbors of a minority class instance and then generate new instance by interpolating *K* neighbors. The use of the KNN algorithm in SMOTE ensures that the newly generated instance lies in the region of the minority class. Generally, Euclidean distance is used to select the nearest neighbors.

4.2 Borderline-SMOTE: It is an improved version of SMOTE. Instead of treating every instance equally, Borderline-SMOTE puts more focus on those instances that are hard for prediction models to classify. These instances are referred to as borderline instances because they are closer to the borderline between the majority and minority classes. This technique selects as the initial instances those borderline instances that have more majority class than minority class nearest neighbor instances.

4.3 Adaptive Synthetic Sampling Method for Imbalanced Data (ADASYN): The ADASYN technique uses a weighted distribution for different minority class instances according to their level of difficulty in learning. In this way, more synthetic data is generated for minority class instances that are harder to learn compared to those minority examples that are easier to learn. As a result, the ADASYN approach improves learning with respect to the data distributions as compared to the SMOTE.

4.4 Random Oversampling (ROS): It is a simple oversampling technique. It involves randomly selecting examples from the minority class, with replacement, and adding them to the training dataset. This technique simply duplicates the population of the minority class instances.

4.5 Random Undersampling (RUS): It is a simple undersampling technique, which reduces the population of the majority class instances. It involves randomly selecting examples from the majority class and deleting them from the training dataset. This technique results in information loss.

4.6 AdaBoost: It is a boosting ensemble method. The learning of the technique is based on incremental learning, where a set of intermediate models is generated in different iterations, and incrementally, these intermediate models are refined by updating example (observations) weights in every iteration for the learning technique. This ensures that the incorrectly predicted examples are counted heavily in the next iterations. The final prediction is performed by combining the outputs of all intermediate generated models. AdaBoost is an adaptive version of the classic boosting method.

Table 5: Parameter values set for different class imbalance techniques

Technique	**Parameter values**
SMOTE	sampling_strategy='auto', random_state=None, k_neighbors=5, n_jobs=None
Borderline-SMOTE	sampling_strategy='auto', random_state=None, k_neighbors=5, n_jobs=None, m_neighbors=10, kind='borderline-1'
ADASYN	sampling_strategy='auto', random_state=None, n_neighbors=5, n_jobs=None
ROS	sampling_strategy='auto', random_state=None, shrinkage=None
RUS	sampling_strategy='auto', random_state=None, replacement=False
AdaBoost	base_estimator='Random Forest', *, n_estimators=50, learning_rate=1.0, algorithm='SAMME.R', random_state=None

References

[1] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison,"Empirical Softw. Engg., vol. 17, no. 4-5, pp. 531–577, Aug. 2012.

[2] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in Proceedings of the 6th International Conference on Predictive Models in Software Engineering, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 9:1–9:10.

[3] S. Yatish, J. Jiarpakdee, P. Thongtanunam, and C. Tantithamthavorn,"Mining software defects: Should we consider affected releases?" in The International Conference on Software Engineering (ICSE), 2019.