Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

On the Data Quality Characteristics of Software Engineering Defect Prediction Datasets

Autor: Pablo Acereda García

Tutor: Daniel Rodríguez García

2020

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Fin de Grado

On the Data Quality Characteristics of Software Engineering Defect Prediction Datasets

Autor: Pablo Acereda García

Director: Daniel Rodríguez García

Tribunal:

Presidente: Name of the tribunal president

Vocal 1º: Name of the first vocal

Vocal 2º: Name of the second vocal

Calificación: .....

Fecha: .....

A quien le haga falta saberlo...

"Si tienes un problema que no tiene solución, ¿para qué te preocupas? Y si tiene solución, ¿para qué te preocupas?."

Proverbio chino, y de mi abuela

Acknowledgements

Sin entrenamiento, no existe el conocimiento. Sin conocimiento, no existe la confianza. Sin confianza, la victoria no existe.

Julio César

Tras muchas horas de esfuerzo, querer tirarme de los pelos, dar cabezazos contra la pared o querer gritar... simplemente han pasado 4 años. Sin saberlo, he tenido junto a mí el apoyo de mi familia; también he conocido a quienes los considero como tal; quienes han venido y se han ido - pero todo ello tiene algo en común: no podría haber hecho esto solo.

De mi familia, especialmente mis padres y hermana siempre he sacado el apoyo moral que me ha hecho falta. Me han dado la oportunidad de ser yo el que vaya cometiendo mis propios errores y aprendiendo de ellos. Gracias por dar vuestro mejor esfuerzo para que yo haya podido llegar hasta aquí, por haberme educado para ser siempre curioso y no conformarme con nada, por haberme regañado cuando he errado y animado cuando lo he conseguido. Imposible haberlo conseguido sin vosotros. Otra pequeña mención para mis abuelos, por haberme enseñado cosas sobre la vida que no están escritas en los libros.

A mi pareja, por hacerme descansar después de inacabables jornadas de trabajo, por aquellas largas noches hablando sobre la vida y riéndonos de nosotros mismos hasta casi llorar de la risa. Gracias por hacerme ver el mundo y ser mi hogar. Agradezco a mis amigos de toda la vida por hacerme darme cuenta de que las cosas no siempre son tan malas como parecen.

Gracias a mi *hermanita* por enseñarme que sin importar lo difícil que sea la vida, siempre se puede seguir adelante con la cabeza alta. Me mantuviste despierto allí donde la noche se hacía larga.

Resumen

Este documento ha sido generado con una plantilla para memorias de trabajos fin de carrera, fin de máster, fin de grado y tesis doctorales. Está especialmente pensado para su uso en la Universidad de Alcalá, pero debería ser fácilmente extensible y adaptable a otros casos de uso. En su contenido se incluyen las instrucciones generales para usarlo, así como algunos ejemplos de elementos que pueden ser de utilidad. Si tenéis problemas, sugerencias o comentarios sobre el mismo, dirigidlas por favor a Pablo Acereda García <pablo.acereda@edu.uah.es>.

Palabras clave: Plantillas de trabajos fin de carrera/máster/grado y tesis doctorales, LaTeX, soporte de español e inglés, generación automática.

Abstract

This document has been generated with a template for Bsc and Msc Thesis (trabajos fin de carrera, fin de máster, fin de grado) and PhD. Thesis, specially thought for its use in Universidad de Alcalá, although it should be easily extended and adapted for other use cases. In its content we include general instructions of use, and some example of elements than can be useful. If you have problemas, suggestions or comments on the template, please forward them to Pablo Acereda García <pablo.acereda@edu.uah.es>.

Keywords: Bsc., Msc. and PhD. Thesis template, LaTeX, English/Spanish support, automatic generation.

Contents

List of Figures

List of Tables

List of source code listings

List of Algorithms

Chapter 1

Introduction

%

1.1. Introduction to Machine Learning

Through time, humans wanted to keep all memories or thoughts precious to them for posterity. Thanks to some technological advancements, larges amounts of information can be stored for a relatively cheap price - videos, photographs, drawings, weather measures, financial information, academic data, etc. It can all be digitalized and kept *forever*.

These new possibilities also brought new challenges. *What to do with all that information?* Data scientist were born to try to manage and make some sense out of the overwhelming new information being created.

The figure of a data scientist is that who uses scientific methods, processes, algorithms and systems to extract knowledge from structured and unstructured data. The algorithms used in data science, are sequences of statistical processing steps. Then, there is Machine Learning which uses computer algorithms that improve through experience - it is also a subset of Artificial Intelligence (AI). In this case, the algorithms are *trained* to filter and separate patterns and features with massive amounts of data.

The training and learning processes allow to make predictions and decisions for new data, being advantageous for almost any field: commercial purposes, fraud prediction, plant caring, network traffic, and so on.

The machine learning tools applied to this project are numerous, and they are going to be further explained in the *Background* section (see Section 2). As a brief introduction to the techniques and material in this paper, it has been used *ECoL* R package to obtain the complexity metrics of several datasets; the Python's *sklearn* package for learning algorithms and some analytical metrics (used through the experiments); as well as the *imbalanced-learn* Python package to deal with imbalanced datasets. The resulting scripts allow the visualization between different tables and - plots for comparing results.

1.2. Aim and Objectives

Here we analyze the complexity metrics proposed by Ho and Basu [4] in a number of software defect datasets, that have been previously implemented in *ECoL* R package.

The aim of this work is to explore complexity metrics on software defect datasets. Also, to analyze how classification algorithms are affected by techniques that mitigate imbalance and how that affects the complexity metrics previously analyzed. To do so, we explore several objectives:

- RQ1 Which complexity metrics are related to miss-classification?
- RQ2 How complexity metrics are correlated to the outcome of supervised algorithms?
- RQ3 How complexity metrics and imbalance are related? A
- RQ4 Do complexity metrics tell us something about the quality of the datasets?

The purpose of this dissertation is to conduct a study of complexity metrics and imbalanced datasets. A comparison between the *raw* data and that data after performing some changes that should affect the results: K-folding Cross Validation, Imbalanced techniques, etc.

Chapter 2

Background

2.1. Data Complexity Metrics

There are many complexity metrics that could be used for the scope of this project, but it has been chosen the ones that are explained below; following the metrics obtained from [5], and surveys by Lorena et al [2, 6]. They are also implemented in the Extended Complexity Metrics Library (ECoL) for R.

These are a set of measures that help characterizing the complexity of classification and regression problems. The measures that are going to be computed for this project are:

Overlapping Evaluate how informative the available features are to separate the classes. See 2.1.1 for more details.

Neighborhood Characterize the presence and density of classes in local neighborhoods. See 2.1.2 for more details.

Linearity Quantify, if it is possible, whether classes are linear separable by a hyperplane or linear function. See 2.1.3 for more details.

Dimensionality Indicative of data sparsity, how smoothly samples are distributed within attributes. See 2.1.4 for more details.

Balance Capture the difference in the number of examples per class in the dataset. See 2.1.5 for more details.

Network Represents the dataset as a graph and extracts structural information out of it. See 2.1.6 for more details.

Correlation Relationship between the feature values and the outputs. See 2.1.7 for more details.

Smoothness In regression problems, the smoother the function to be fitted to the data, the simpler it shall be. See 2.1.8 for more details.

These complexity measures are going to be computed for certain datasets and comparisons between results are going to be made.

2.1.1. Measures of Overlap of Individual Feature Values

An overlapped dataset is defined as a multi-class dataset where the data is interlaced. The meaning of this statement is explained as follows: having a dataset whose geometrical expression have the domain, D_n[1], a second class will have a domain, D'_n, intersecting that of the first class.

2.1.1.1. Fisher's Discriminant Ratio (F1)

This metric is specific to one feature dimension. Even if the given problem is multidimensional, not necessarily all features have to contribute to class discrimination, only one of the features needs to be the discriminant. Fulfilling this statement would ease the complexity of the problem.

The measure uses the following mathematical expression:

$$f = \frac{(\mu_1 - \mu_2)^2}{\sigma_1^2 + \sigma_2^2}$$

Where $\mu_1, \mu_2, \sigma_1^2, \sigma_2^2$ are the means and variances, respectively, of the two classes[2].

2.1.1.2. Directional Vector Fishers Discriminant Ratio (F1v)

Complemented F1 (see Section 2.1.1.1). It does so by searching for a vector capable of separating two classes after the training samples have been projected into it.

2.1.1.3. Volume of Overlap Region (F2)

Another way to measure the complexity is by using the tails of the overlap of the classes. This is accomplished by taking the maximum and minimum values of each class. Afterwards, the length of overlap is calculated, normalized by range of the classes. This is done by:

$$f = \prod_i \frac{MIN(max(f_i, c_1), max(f_i, c_2)) - MAX(min(f_i, c_1), min(f_i, c_2))}{MAX(max(f_i, c_1), max(f_i, c_2)) - MIN(min(f_i, c_1), min(f_i, c_2))}$$

Where f_i indicates the feature; c_n indicates the class; and i indicates the dimensionality of the problem.

2.1.1.4. Feature Efficiency (F3)

In this measure, in high-dimensional problems, each feature is evaluated separately on how much it contributes to the separation of the classes. If there exists overlap within the range of each class for a certain feature, the class is considered to be ambiguous for that dimension in the specific region where the overlap takes place.

Therefore a problem will be more feasible if the exists at least one feature where the ranges for each class do not overlap. Thus, the *maximum feature efficiency* will be used as a measure. This measure is obtained with the entire training set, by obtaining the remaining points being separable by the feature.

[1] Being n the number of attributes for that class

[2] It has been used two classes for the definition of the metric, but more classes could be added to the problem.

2.1.1.5. Collective Feature Efficiency (F4)

It gives an overview on how different features may work together in data separation. The most discriminant feature (see F3 - 2.1.1.4) is selected and then all samples separable by this feature are removed from the dataset. This process is repeated with all the features until no samples are left. The resulting value of this metric comes from the ratio of samples that have been discriminated.

2.1.2. Measures of Neighborhood

For a certain set of target classes in a dataset, the neighborhood measures try to analyze the neighbor data items of every data point and try to capture class overlapping and the shape of the decision boundary. The work is done over a distance matrix which stores distances between all pairs of data points in the dataset.

2.1.2.1. Mixture Identifiability (N1, N2, N3)

The measure is defined as the mean Euclidean distance from each point in the dataset, to its nearest neighbors, not minding the class they are in [7]. The valuable rates that can be taken are the means for interclass neighbors and intraclass neighbors.

2.1.2.2. Non-linearity of the Nearest Neighbor Classifier (N4)

It builds a new dataset by interpolating pairs of training samples of the same class and the induce a 1NN classifier on the original data and measure the error rate in the new data points.

2.1.2.3. Fraction of Hyper-spheres Covering Data (T1)

It creates a hyper-sphere centered at each one of the training samples. Their radios are then growth until one hyper-sphere reaches a sample of other class. Then, smaller hyper-spheres contained in larger hyper-spheres are eliminated. This measure is the ratio between the number of remaining hyper-spheres and the total number of samples in the dataset.

2.1.2.4. Local Set Average Cardinality (LSC)

It is a set of points from the dataset whose distance of each sample is smaller than the distance from the samples of the different classes.

2.1.3. Measures of Separability of Classes

Given two, or more classes, inside a certain dataset, they can be called linear separable if a straight line between those classes can be drawn. As the previous statement suggests, the classes must be isolated clusters after the linear function is delimited.

2.1.3.1. Linear Separability (L1, L2, L3)

As a way to adapt to separable and non-separable problems, it is used the formulation proposed by Smith[8], which minimizes an error function:

$$\begin{aligned} &\text{minimize } a^t t \\ &\text{subject to } Z^t w + t \geq b \\ &t \geq 0 \end{aligned}$$

Implying that a and b are vectors; w is the weight vector; t is the error vector and Z is a matrix obtained by adding one dimension, with value one, to the input vector x.

2.1.4. Measures of Dimensionality

For any dataset, the dimensionality measures will indicate data sparsity. These measures capture how sparse a dataset tends to have regions of low density. These regions have been proven to be harder for the extraction of good classification and regression models.

2.1.4.1. Average Number of samples per dimension (T2)

It represents the average number of points per dimension. It is calculated with the ratio between the number of examples and dimensionality of the dataset.

2.1.4.2. Average intrinsic dimensionality per number of samples (T3)

Another way to measure the dimensionality of a dataset, is by computing the average of number of points per PCA. It is a similar measure to $T2$, which uses the number of PCA components needed to represent 95 variability as the base of a data sparsity assessment.

2.1.4.3. Intrinsic dimensionality proportion (T4)

It represents a ratio of PCA Dimensions to the Original. It returns an estimate of the proportion of relevant and original dimensions of the dataset.

2.1.5. Measures of Class Balance

Given a certain dataset, *balance* measures capture the differences in the number of samples per class for that dataset. When the imbalance ratio is too severe, problems related to generalization of classification techniques could happen.

2.1.5.1. Entropy of class proportions (C1)

This measure captures the imbalance of a dataset using proportions of samples per class.

2.1.5.2. Multi-class imbalance ratio (C2)

The ratio at hand represents an index calculated to measure a class balance. It is not only suited for binary class classification problems, but also is suited for multi-class classification problems.

2.1.6. Measures of Network

These measures create a graph representation of the dataset to extract structural information from it. The transformation between raw data and the graph representation is based on the epsilon-NN ($\varepsilon - NN$) algorithm. Afterwards, a post-processing step is applied to the graph, pruning edges between samples of opposite classes.

2.1.6.1. Average density of of network (Density)

It is a representation of the count of edges in the graph, divided by the maximum number of edges between pairs of data points.

2.1.6.2. Clustering Coefficient (ClsCoef)

Computes the average of the clustering tendency of the vertices by the ratio of existent edges between neighbors and the total number of edges that could possibly exist between them.

2.1.6.3. Average hub score (Hubs)

Is given by the number of connections to other nodes, weighted by the amount of connections the neighbors have.

2.1.7. Measures of Feature Correlation

A regression task that calculate the correlation of the values of the features to the outputs. In case one feature is highly correlated to the output, it is understandable that simpler functions can be fitted to the data.

2.1.7.1. Maximum/Average feature correlation to the output (C1, C2)

Representation of the maximum/average value of the Spearman correlations for each feature and the output.

2.1.7.2. Individual feature efficiency (C3)

Returns the number of samples to be removed from the dataset to reach a high Spearman correlation value to the output.

2.1.7.3. Collective feature efficiency (C4)

Gives the ratio of samples removed from the dataset based on an iterative process of linear fitting between the features and the target attribute.

2.1.8. Measures of Smoothness

It is a regression task, used for regression problems. In those problems, the smoother the function to be fitted, the simpler the task it becomes. Large variations in input/output are an indication of the existence of more intricate relationships between them.

2.1.8.1. Output distribution (S1)

Checks if the samples joined in then MST have similar output values. The lower the value, the simpler the problem - where outputs of similar samples in the input space are also next to each other.

2.1.8.2. Input distribution (S2)

Monitors the similarity in the input space of data items with similar outputs based on distance.

2.1.8.3. Error of a nearest neighbor regressor (S3)

Stands for the mean squared error of a 1-nearest neighbor regressor using leave-one-out.

2.1.8.4. Non-linearity of nearest neighbor regressor (S4)

Computes the mean squared error of a 1-nearest neighbor regressor to the new randomly interpolated points.

Table 2.1: Complexity Metrics from [2] and [3]

Category	*Name*	*Acronym*	*Min*	*Max*	*Asymptotic Cost*
Feature-based	Maximum Fisher's discriminant ratio	F1	≈ 0	1	$O(m \cdot n)$
	Directional vector maximum Fisher's discriminant ratio	F1v	≈ 0	1	$O(m \cdot n \cdot n_c + m^3 \cdot n_c^2)$
	Volume of overlapping region	F2	0	1	$O(m \cdot n \cdot n_c)$
	Maximum individual feature efficiency	F3	0	1	$O(m \cdot n \cdot n_c)$
	Collective feature efficiency	F4	0	1	$O(m^2 \cdot n \cdot n_c)$
Neighborhood	Faction of borderline points	N1	0	1	$O(m \cdot n^2)$
	Ratio of intra/extra class NN distance	N2	0	≈ 1	$O(m \cdot n^2)$
	Error rate of NN classifier	N3	0	1	$O(m \cdot n^2)$
	Non linearity of NN classifier	N4	0	1	$O(m \cdot n^2 + m \cdot l \cdot n)$
	Fraction of hyper-spheres covering data	T1	0	1	$O(m \cdot n^2)$
	Local set average cardinality	LSC	0	$1 - \frac{1}{n}$	$O(m \cdot n^2)$
Linearity	Sum of the error distance by linear programming	L1	0	≈ 1	$O(n^2)$
	Error rate of linear classifier	L2	0	1	$O(n^2)$
	Non linearity of linear classifier	L3	0	1	$O(n^2 + m \cdot n \cdot n_c)$
Dimensionality	Average number of features per dimension	T2	≈ 0	m	$O(m + n)$
	Average number of PCA dimensions per points	T3	≈ 0	m	$O(m^2 \cdot n + m^3)$
	Ratio of the PCA dimension to the original dimension	T4	0	1	$O(m^2 \cdot n + m^3)$
Class Imbalance	Entropy of classes proportions	C1	0	1	$O(n)$
	Imbalance ratio	C2	0	1	$O(n)$
Network	Density	Density	0	1	$O(m \cdot n^2)$
	Clustering Coefficient	ClsCoef	0	1	$O(m \cdot n^2)$
	Hubs	Hubs	0	1	$O(m \cdot n^2)$
Correlation	Maximum Feature Correlation to the Output	C1	0	1	$O(n \cdot m \cdot logm)$
	Average Feature Correlation to the Output	C2	0	1	$O(n \cdot m \cdot logm)$
	Individual Feature Efficiency	C3	0	1	$O(n \cdot m^2)$
	Collective Feature Efficiency	C4	0	1	$O(n \cdot (d + n \cdot logn))$
Smoothness	Output Distribution	S1	0	-	$O(n^2)$
	Input Distribution	S2	0	-	$O(m \cdot (n + logm))$
	Error of a nearest neighbor regressor	S3	0	-	$O(n^2)$

2.2. Supervised Classification

Machine Learning has different methods or styles available:

Supervised Trains itself on labeled data set.

Unsupervised Ingests unlabeled data and uses algorithms to extract meaningful features needed to label or sort data - without human intervention.

Semi-supervised Uses smaller labeled dataset to set up a guide and large amounts of unlabeled data for feature extraction

In this work, the only the first of them is used. Getting into more detail on supervised learning, it means that the machine learning model is built on data that has enough labeled information on how to determine and classify new incoming data.

That could be the example of a model ready to identify dogs. Many images with dogs, that would label the breed, and other characteristics, would be necessary to differentiate an Alaskan Malamute from a Beagle.

Now, it is true that this technique requires less input data to train a model, which make it easier to obtain a decent amount of data to train and test the model. Also, data can easily be tested, thanks to labeled data - which is a bit more expensive to generate than unlabelled data (for obvious reasons). There is also the danger of overfitting the model. It means that the model is too close to the training set. It is translated to a poor performance over slight variations from new data.

In supervised learning, an optimal scenario is considered when the model is able to label correctly unseen data (which might come from a testing set or from new data).

Other applications, that have not been mentioned so far, could be database marketing, pattern recognition, spam detection, etc.

For this project, supervised learning is used more as a mean rather than as a final goal. In this paper, it it more important what affects supervised classification rather than classifying something.

2.3. Imbalance

On any dataset obtained from a real world environment, there will be a class within the target outputs that will be more predominant than the rest of the classes. When the difference between the predominant class and the rest of the classes is not “remarkable” the dataset is called balanced. On the other hand, the classes are imbalanced if the number of samples from one class vastly outnumbers the number of samples from the rest of the classes, this is called the imbalance problem. This means, that not enough data from the minority classes is available to train the algorithm.

Even though it might seem as a trivial matter, having an overwhelming amount of samples of just one of the classes means that the training algorithm will overfit the data, and therefore result on a high classification error.

Most canonical classification algorithms (e.g. SVM, decision tree and neural networks) suffer from the *majority class bias* - they perform nicely on balanced datasets, but very few of them does under an

imbalanced scenario. Since most of these learning processes are oriented to global accuracy, the resulting classifiers tend to have majority class bias. This is translated to an apparent good performance, that acts poorly on terms of accuracy in the minority class.

Typical attempts to eliminate this bias is by data re-sampling and re-weighting through the learning process.

All the aforementioned metrics about datasets (and some recommended for imbalanced datasets) do not say how to mitigate its effects on classification algorithms. The different approaches to achieve this goal - deal with imbalanced data - are sampling techniques, cost-sensitive, ensemble approaches or hybrid approaches. They are going to be briefly described in the incoming sections.

2.3.1. Sampling Techniques

Techniques that are classified as *Sampling Techniques* are oversampling and undersampling. These methods are based on the addition or removal of instances of a given training dataset - as a pre-processing step. The process of replicating or creating instances from a minority class towards a more balanced distribution (number of samples of majority and minority class are more or less similar) is called Random OverSampling (ROS); whereas Random Under-Sampling (RUS) is the procedure of removing instances from the majority class to reduce the difference between the amount of samples of each class-.

Rather than using this early proposals, there can be found more sophisticated approaches generating new artificial samples, rather than the replication of already existing instances. Some of these proposals are going to be explained in the following sections.

2.3.1.1. Undersampling

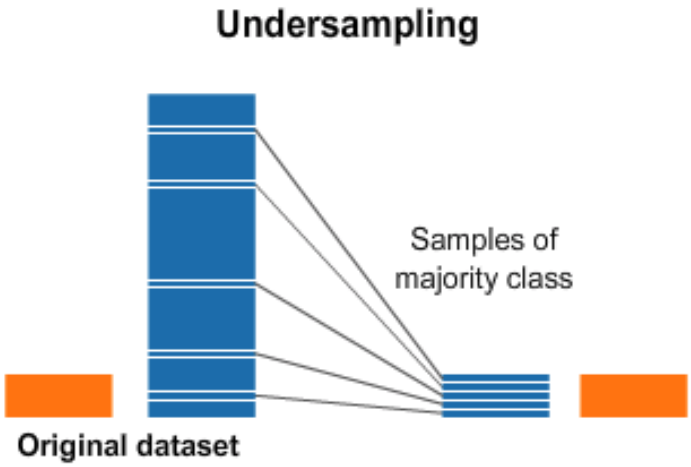


Figure 2.1: Undersampling Technique [1]

It involves removing samples from the dataset until the data is balanced. The reasons to use undersampling are usually related to practical reasons, such as resource costs. The techniques presented are:

2.3.1.1.1. Random Undersampling It involves deleting samples from the majority class, with or without replacement.It is one of the early proposals to deal with imbalanced datasets. Although it may alleviate the imbalance in the dataset, it may also increase the variance of the classifier or discard meaningful samples from the majority class.

2.3.1.1.2. Cluster Centroid It replaces a cluster of samples by the cluster centroid of a K-means algorithm. The number of clusters is set by the level of undersampling.

2.3.1.1.3. Near Miss Refers to a collection of undersampling methods that select samples based on the distance of the majority class samples to the minority class samples [9]. There are three versions of this algorithm: (1) *NearMiss-1* selects majority class samples with minimum average distance to three closest minority class samples; (2) *NearMiss-2* selects majority class examples with minimum average distance to the three furthest minority class examples; (3) *NearMiss-3* selects majority class examples with minimum distance to each minority class example.

Among all the available techniques, *Near Miss* has been the on used for this research.

Other techniques that remove instances intelligently include the Edited Nearest Neighbor (ENN) and Wilson's Editing that remove instances in which close neighbors belong to a different class [10].

2.3.1.2. Oversampling

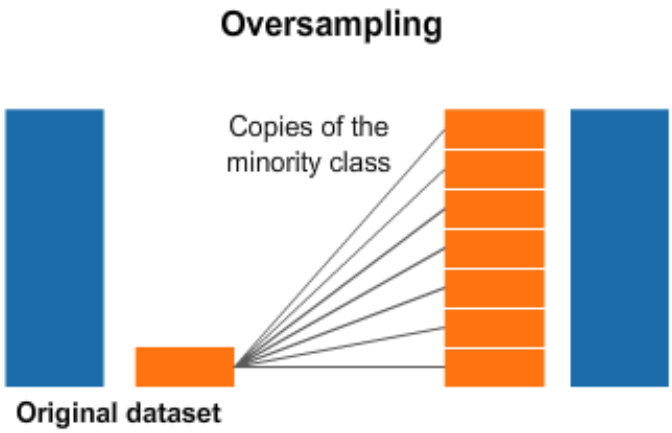


Figure 2.2: Oversampling Technique [1]

Most commonly used. It involves creating new data points from the already existing data. The techniques analyzed are:

2.3.1.2.1. Random Oversampling . Involves copying supplementary data from the minority classes. It can be done more than once (actually, as many times as the developers sees fit). One of the early proposals regarding imbalanced datasets. It is robust, as it may randomly replace some of the samples from the minority class.

2.3.1.2.2. SMOTE Acronym for Synthetic Minority Oversampling Technique [11]. It is one of the most popular techniques used nowadays. It works by selecting samples close in the feature space. That means, that a line is drawn between the samples in the feature space and the a new sample at a point along that line. It is effective because the samples from the minority class created are plausible - relatively close in feature space to existing samples from minority class. A downside of this technique is that samples are created without looking at the majority class, meaning a possible overlapping of classes.

2.3.1.2.3. ADASYN Acronym for ADaptative SYNthetic sampling algorithm. Build on SMOTE methodology. Shifts the importance of the classification boundary to those minority classes which are difficult. Weights the most difficult to learn classes so that those have more importance when creating new data. ADASYN generates more synthetic data in the minority class samples that are harder to learn.

The technique selected for this article is the SMOTE method, implemented by the *imblearn* Python package.

2.3.2. Cost-Sensitive Classifiers

Also known as CSC. These are adapted classifiers that handle imbalanced datasets by either:

1. Adding weights to instances[3].
2. Resampling the training data according to the costs assigned to each class in a predefined cost matrix.
3. Or generating a model that minimizes the expected cost[4]. The idea behind this methodology is to penalize differently each type of error - in the specific case of binary classification, the false positives (FP) and false negatives (FN).

The problem with CSC is defining the cost matrix as there is no systematic approach to do so. However, it is common practice to set the cost to equalize the class distribution.

2.3.3. Ensembles

Also known as meta-learners are a combination of multiple models with the objective of obtaining better predictions. They are typically classified as *Bagging*, *Boosting* and Stacking - Stacked generalization.

2.3.3.1. Bagging

Bagging [12] (also known as Bootstrap aggregating) is a ensemble technique that involves a base learner applied to multiple - equal size - datasets. These datasets are created from the original data using bootstraping. Predictions are made on voting of the individual predictions.

An advantage of this technique is that it does not require to modify any aspect of the learning algorithm, taking advantage of the instability of the base classifier to create diversity among individual ensembles - so that individual members of the ensemble perform well in different regions of the data.

If the output is robust to perturbation of the data (like is the case with nearest-neighbor (NN) classifiers) the performance drops.

[3] Can only be used if the base classifier algorithm allows it
[4] Can be obtained by multiplying the predicted probability distribution with the misclassification costs

2.3.3.2. Boosting

Boosting techniques generate multiple models that complement each other. The final objective is to induce models that improve certain regions of the data, where previous induced models had low performance. This can be achieved by increasing the weights of instances that have been wrongly classified. That way, new learners focus on those regions.

Classification is based on a weighted voted among all members of the ensemble. One of the most popular boosting algorithms is AdaBoost.M1 [13] for classification. The set of training examples is assigned an equal weight at the beginning and the weight of instances can be increased or decreased depending on the classification of the learner. The next iterations focus on those instances with higher weights. AdaBoost.M1 can be applied to any base learner.

Models from ensembles are difficult to interpret (black box behavior), comparing them to decision trees or rules providing explanation of their decision making process.

2.3.4. Hybrid Approaches

There are other hybrid approaches that can be used. Like that ones that are going to be explained in the following section.

2.3.4.1. SMOTEBoost

The SMOTEBoost tries to reduce the bias from the learning procedure due to the class imbalance, and increase the sampling weight for the minority class. SMOTE [14] is introduced in each round of boosting, which enables each learner to be able to sample more of the minority class cases, and learn better and broader decision regions for the minority class.

It also brings the benefit of enhancing the probability of selection for the difficult minority class cases that are dominated by the majority class points [15]. The variation of boosting procedure of the SMOTEBoost process is a variant of the AdaBoost.M2 procedure [16].

2.3.4.2. RUSBoost

The technique RUSBoost [17] uses the AdaBoost.M2. The difference with SMOTEBoost is that RUSBoost applies Random Under Sampling instead of SMOTE. The application of SMOTE at this point has two drawbacks that RUSBoost is designed to overcome:

- Increases the complexity of the algorithm. SMOTE finds the k nearest neighbors of the minority class samples, and then extrapolates them to make new synthetic samples. RUS, on the other hand, simply deletes the majority class examples randomly.
- RUS produces less training datasets (as it is an undersampling technique, not like SMOTE, which is an oversampling technique). This can be translated into shorter model training times [17].

2.3.4.3. MetaCost

MetaCost [18] is a combination of bagging with cost-sensitive classification. The bagging part of the technique is used to relabel training data so that each training example is assigned a prediction that minimizes the expected cost for that instance. Based on the modified training data, MetaCost induces a new classifier which provides information about how a decision was reached.

2.3.5. Defect Prediction

Regarding the history of defect prediction, and final objective of this article, many classification techniques have been proposed - statistics (regression [**?**], and Support Vector Machines [19], etc.), machine learning (classification trees [20]), neural networks [21]), probability (Naïve Bayes [22] and Bayesian networks), ensembles of different techniques and meta-heuristics (ant colonies [23], etc.).

Despite this fact, some discrepancies are found:

- No classifier is consistently better than others.
- There is no optimum metric that allows the evaluation and comparison of classifiers ([22, 24, 25]).
- Data is also affected by quality issues - class imbalance, overlapping, outliers, etc.).

Some authors highlight the problem of imbalanced datasets when dealing with the project at hand, defect prediction (Seiffert et al. [26] and in Khoshgoftaar et al. [27]).

As it has been mentioned before, there is no classifier that behaves consistently better. No technique is able to give a an outstanding performance when evaluating classifiers. Some authors have compared performance of several measures (like Peng [28], Peng et al. [29], this papers actually propose performance metrics to evaluate merit of classification algorithms and ranked classification algorithms, respectively).

Further made on the subject comes from the hand of Lessman et al. [30]. This paper compared several classifiers, discussing performance metrics such like TP_r and FP_r. But finally advocated to use the AUC[5] as the best indicator for classifiers comparison. This results is known as sub-optimal for highly imbalanced datasets.

Arisholm et al. [**?**] compared different classification algorithms (tree algorithm (C4.5), coverage rule algorithm (PART), logistic regression, back-propagation neural networks and Support Vector Machines) over 13 different Java developed systems. They used three metrics to compare results:

- Object-oriented metrics.
- Churn (δ) metrics between successive releases.
- Process management metrics from a configuration management system.

The conclusion was that large differences can be achieved depending on the comparison criteria of the data. To solve this problem, the paper proposed a new AUC based, cost-effectiveness metric. Same approach has been evaluated and explored in Mende and Koschke [31].

[5] Further explained in Section 3.3.7, Area Under the ROC curve

Chapter 3

Empirical Work

In this section is about the experimental work carried out through this research. Firstly, describes the datasets used for the experimentation; then, the supervised classifiers evaluated; the evaluation metrics chosen; and finally, present and discuss the results.

3.1. Datasets

In this work, publicly available datasets are going to be used, in the domain of Software defect prediction: use of Jureczko and Madeyski dataset [1] [32, 33] and the Harman Search Base dataset [34].

From the first cluster of datasets, 15 open source projects are the ones chosen (a total of 8 are used for this project). The number of defects found in each class collected from the Software Management System (SCM) using a regular expression. The datasets are publicly available (can be found in the PROMISE repository [35]). These datasets have been used in previous researches, [36–38], which allows comparing and analyzing the obtained results.

A sample can be considered as *defective* when the number of defects inside the class is more than 0. Similarly, if the number of defects is indeed 0, then the class is *non-defective*. This allows a binary classification, making easier handling results, operations and comparisons.

Table 3.1: Defect Metrics Dataset Variables

Metric	*Description*
WMC	Weighted methods per class [39]
DIT	Depth of Inheritance Tree [39]
NOC	Number of Children [39]
CBO	Coupling between object classes [39]
RFC	Response for a Class [39]
LCOM	Lack of cohesion in methods [39]
Ca	Afferent couplings [40]
Ce	Efferent couplings [40]
NPM	Number of Public Methods [41]
LCOM3	Lack of cohesion in methods [42]

Continued on next page

[1] http://snow.iiar.pwr.wroc.pl:8080/MetricsRepo/

Table 3.1 – *Continued from previous page*

Metric	*Description*
LOC	Lines of Code [41]
DAM	Data Access Metric [41]
MOA	Measure of Aggregation [41]
MFA	Measure of Functional Abstraction [41]
CAM	Cohesion Among Methods of Class [41]
IC	Inheritance Coupling [43]
CBM	Coupling Between Methods [43]
AMC	Average Method Complexity [43]
MAX_CC	Maximum McCabe's cyclomatic complexity [44]
AVG_CC	Average McCabe's cyclomatic complexity [44]

There is another sole dataset obtained from the same source as the previous collection of datasets, with a very similar structure of data, the *Apache* dataset. It has the same origin and objective as its predecessors.

On the second cluster of datasets, the data comes from a total of 8 different Hadoop versions. Their original purpose is to train a search based fault prediction system.

Similarly to Jureczko and Madeyski dataset [32,33], it uses: (1) WMC, (2) DIT, (3), NOC, (4) CBO, (5) RFC, (6) LCOM, (7) NOM, and (8) LOC; metrics to define each set.

Regarding more information on the content of the datasets the next table (see Table 3.2) summarizes the number of samples on each dataset and further valuable information, such as the absolute and relative number of deffects for those datasets.

Table 3.2: Description of the Datasets

Project	*Version*	*#instances*	*#Non-Defective*	*#Defective*	*%Defective*
ant	1.3	125	105	20	16.00
	1.4	178	138	40	22.47
	1.5	293	261	32	10.92
	1.6	351	259	92	26.21
	1.7	745	579	166	22.28
apache	-	191	107	84	43.98
camel	1.0	339	326	13	3.83
	1.2	608	392	216	35.52
	1.4	872	727	145	16.62
	1.6	965	777	188	19.48
hadoop	0.1	141	91	50	35.60
	0.2	191	149	42	21.99
	0.3	211	158	53	25.12
	0.4	201	159	42	20.90
	0.5	217	180	37	17.05
	0.6	234	203	31	13.25

Continued on next page

Table 3.2 – *Continued from previous page*

Project	*Version*	*#instances*	*#Non-Def*	*#Def*	*%Def*
	0.7	250	202	48	19.20
	0.8	240	224	16	6.67
ivy	1.4	241	225	16	6.63
	2.0	352	312	40	11.36
jedit	3.2	272	182	90	33.08
	4.0	306	231	75	24.50
	4.1	312	233	79	25.32
	4.2	367	319	48	13.07
	4.3	492	481	11	2.23
log4j	1.0	135	101	34	25.18
	1.1	109	72	37	33.94
	1.2	205	16	189	92.19
synapse	1.0	157	141	16	10.19
	1.1	222	162	60	27.02
	1.2	256	170	86	33.59
xalan	2.4	723	613	110	15.21
	2.5	803	416	387	48.19
	2.6	885	474	411	46.44
	2.7	909	11	898	98.78
xerces	1.2	440	369	71	16.13
	1.3	453	384	69	15.23
	1.4	588	151	437	74.31

The experiments of this paper use the latest version of the datasets *ant*, *apache*, *camel*, *hadoop*, *ivy*, *jedit*, *log4j*, *xalan* and *xerces*; alongide all the available versions of the *hadoop* dataset.

3.2. Supervised Classifiers

This work makes use of several supervised learning algorithms. The experiments carried out use the following algorithms:

Naive Bayes (NB) [45] is a classifier that works on conditional probabilities, uses the Bayes theorem to predict the class for each data input. Calculates the probability of a certain event, given prior knowledge. This classifier assigns a set of attributes $a_1, \ldots, a_n$ to a given class C so that the probability of the class description value of the attributes instances is maximal: $P(C|a_1, \ldots, a_n)$. The probability of the hypothesis, given that the evidence is true, is the probability of the evidence, given the hypothesis is true, multiplied by the probability of the hypothesis; in relation to the probability of the evidence. See Eq. 3.1.

$$P(H|E) = \frac{(E|H) * P(H)}{P(E)} \tag{3.1}$$

For this project it has been selected the Gaussian Naive Bayes to perform the experimentation. Being *Gaussian* means that the likelihood of the features is assumed to be Gaussian. The formula is given by Eq. 3.2.

$$P(x_i|y) = (\frac{1}{\sqrt{2\pi\sigma_y^2}}) \exp(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}) \tag{3.2}$$

CART (Classification And Regression Trees) [46] is a non-parametric decision tree, similar to **C4.5** [47]. The main difference is that this algorithm supports numerical target variables. In exchange, it cannot compute rule sets. CART constructs a two branch bifurcation of the most discriminating attribute, based on the Gini index. It can generate either classification or regression trees - depends on the variable (categorical or numeric, respectively). *Classification and Regression Trees* algorithm is more complex and time consuming than *C4.5*'s since multiple trees need to be built and pruned, but trees are generally simpler [48].

The implementation available in [49] is an optimised version of this algorithm, although it does not support categorical variables.

Nearest Centroid classifier[50], or **Nearest Prototype** classifier, or **Rocchio classifier** for its similarity to an algorithm with the same name (see Eq.3.3).

$$\overrightarrow{Q_m} = (a \cdot \overrightarrow{Q_o}) + (b \cdot \frac{1}{|D_r|} \cdot \sum_{\overrightarrow{D_j} \in D_r} \overrightarrow{D_j}) - (c \cdot \frac{1}{|D_{nr}|} \cdot \sum_{\overrightarrow{D_k} \in D_{nr}} \overrightarrow{D_k}) \tag{3.3}$$

The labels of a given sample are assigned by evaluating the classes of training samples whose mean is closest to the evaluated point. The training procedure, given a labeled training set $(\overrightarrow{x_1}, y_1), \ldots, (\overrightarrow{x_n}, y_n)$ with class labels $y_i \in Y$, compute the per-class centroid with Eq. 3.4.

$$\overrightarrow{\mu_l} = \frac{1}{|C_l|} \sum_{i \in C_l} \overrightarrow{x_i} \tag{3.4}$$

Where C_l is the set of indices of samples belonging to class $l \in Y$. The prediction functions, takes the class assigned to an observation $\overrightarrow{x}$ is Eq. 3.5.

$$\overrightarrow{y} = argmin_{l \in Y} ||\overrightarrow{\mu_l} - \overrightarrow{x}|| \tag{3.5}$$

3.3. Evaluation Metrics

Part of the process of applying a classification algorithm is to measure the success and the results of the training. In order to do it, some rates can be calculated out of the testing to the trained classifier. Here is where the Confusion Matrix comes at hand.

The Confusion Matrix (see Table 3.3) allows us to summarize the performance of a given classification algorithm, it is also the foundation of many of the performance metrics used in classification by:

True Positive (TP) - Data is correctly classified as positive.

True Negative (TN) - Data is correctly classified as negative.

False Positive (FP) - Data being negative classified as positive.

Table 3.3: Confusion Matrix for Binary Classification

		Actual Class	
		Positive	*Negative*
Predicted Class	*Positive*	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

False Negative (FN) - Data being positive classified as negative.

These specifications indicate that the input data should have a binary target: one value that can be classified as *positive* and a second value that can be classified as negative. From this statistical classification [2], many performance measures can be calculated.

Some of the most widely used metrics, and the ones calculated in this paper are the ones explained next.

3.3.1. Precision

Also know as Positive Predictive Value (PPV). It is the relation between the *true positives* calculated and the overall positives detected by the classification algorithm (see Eq. 3.6).

$$PPV = \frac{TP}{TP + FP} = 1 - FDR \tag{3.6}$$

3.3.2. Recall

Also known as sensitivity, hit rate, or True Positive Rate (TPR). It stands for the relation between the *true positives* calculated and the real number of positives (see Eq. 3.7).

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN} = 1 - FNR \tag{3.7}$$

3.3.3. Fall-out

Also called False Positive Rate (FPR). It is the probability of rejecting (falsely) the null hypothesis[3] for a particular test (see Eq. 3.8).

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN} = 1 - TNR \tag{3.8}$$

3.3.4. Balance Accuracy

It goes by the acronym BA. It is a metric generally used to evaluate how good is a (binary) classifier. It is a measure that comes in specially handy for imbalanced datasets. Its formula is represented by the mean of *sensitivity* and *specificity* (see Eq. 3.9).

$$BA = \frac{TPR + TNR}{2} \tag{3.9}$$

[2] Also know as error matrix.

[3] General statement or default position that there is no relationship between two measured phenomena or no association among groups.

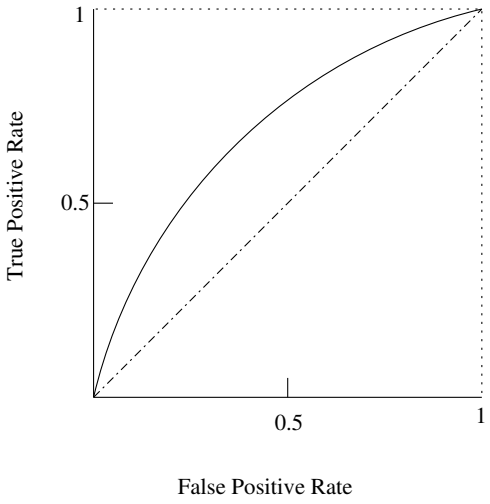


Figure 3.1: ROC Curve

3.3.5. F-Measure

Also known as F_1, or *Sørensen-Dice coefficient* (independently developed by Thorvald Sørensen [51] and Lee Raymond Dice [52]). It is the harmonic mean of precision and sensitivity, the two previous measures, which measures accuracy (see Eq. 3.10). It is twice the relation of the multiplication between *recall* and *precision* and their addition. It is commonly used in highly imbalanced datasets. There are some criticism rewarding this measure, as it does not take into account the *True Negative* (TN) cases.

$$F_1 = 2 \cdot \frac{PPV \cdot TPR}{PPV + TPR} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{3.10}$$

3.3.6. MCC

The Matthews Correlation Coefficient (MCC) [53] or *phi* coefficient, a performance metric (see Eq. 3.11) that measures the quality of a binary classification robust to the imbalance problem. Its range values are between -1 and +1, where -1 represents complete inconsistency (disagreement), 0 indicates that the prediction is no better than a random prediction; and +1 would be a perfect prediction.

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3.11}$$

Other measures cannot be used when data is highly imbalanced, like *accuracy* (Acc) (defined by Eq. 3.12), as it does not take into account the number of labels of different classes.

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.12}$$

3.3.7. Receiver Operating Characteristic Curve

The Receiver Operating Characteristic (ROC)[54] Curve represents graphically the True Positive Rate (TPR) versus the False Positive Rate (FPR) as shown in Figure 3.1.

Once the curve is plotted, the more the curve gets similar to a slope of $TPR = FPR$ the more imbalance that can be found in the dataset. It provides graphical visualization of the results.

The Area Under the ROC Curve (AUC) is a quality measure between positive and negative rates with a single value. This metric allows to compare models.

An approximation to the function can be calculated with Eq 3.13.

$$AUC = \frac{1 + TP_r - FP_r}{2} \tag{3.13}$$

Three unbiased metrics, i.e., AUC, F-score ([51], [52]), Matthews Correlation Coefficient (MCC - [53]), are used to evaluate the performance on imbalanced datasets.

3.4. Results and discussion

3.4.1. Methodology

To answer the RQ in Section 1.2, we run several experiments. First, we analyze the complexity metrics (see Section 2.1) from all selected datasets (see Section 3.1). Then, we retrieve some analytical metrics (see Section 3.3), applying K-fold Cross Validation to those datasets. Finally, we repeat the process applying some under/oversampling techniques to the K-Fold process (see Sections 2.3.1.1 and 2.3.1.2, respectively).

As the final objective is to see how complexity metrics affect classification - and therefore the analytical metrics, a final section is going to summarize the results obtained and the conclusions drawn out of those measures and answer Research Questions 1 to 4.

3.4.2. Metrics Analysis

As it has been mentioned in *Data Complexity Metrics* (see Section 2.1), the software package to calculate them is available in R, ECoL[4]. Therefore, it was necessary to call it from the Python environment. In order to do so, the package RServe, a client server implementation for R workspace to execute functions from other environments like Python, was used. In other words, this is a connector that allow us to execute the ECoL complexity metrics functions.

The code regarding this script can be found in the Appendix B, section B.1.

Regarding the experiment's content and objective, it has been implemented an script that requests the complexity metrics of a dataset and then displays its values on graphs comparing the results.

The structure of the script tries to find and compare the complexity metrics obtained from different datasets, to see if there is a certain relation between metrics.

There has been create a script (see simplification of the algorithm in Algorithm 1) that connects to ECoL library in R and returns the complexity metrics of a certain dataset (see full implementation of the script in Appendix B, Section B.4).

Algorithm 1 Datasets Complexity Metrics Comparison

Require: $\mathcal{D}$ dataset
Ensure: $\mathcal{C}$ complexity metrics
1: con ← connectEcol()
2: **for all** set in $\mathcal{D}$ **do**
3: inputs, targets ← getDataset(set)
4: metrics ← con.getMetrics(inputs, targets)

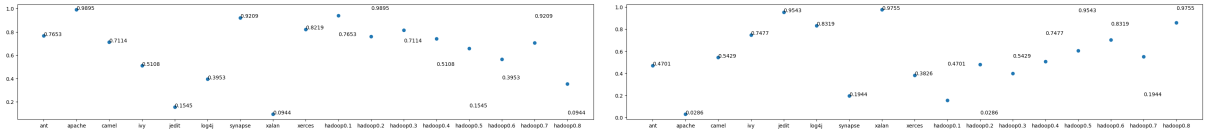
[4] https://github.com/lpfgarcia/ECoL/

5: $\mathcal{C}$.add(metrics)
6: **end for**
7: plotComparison($\mathcal{C}$)

The results are also stored in CSV files, which can be represented as tables C.1 and C.2.

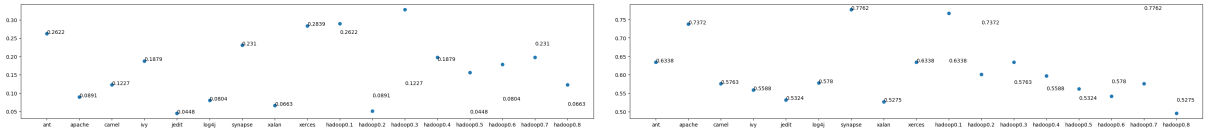
The experiment has been repeated for two independents sets of datasets (see Section 3.1 for more details regarding the datasets and their content). The first experiment's results are summarized in the following table (see table C.1), whereas the second experiment, regarding the Hadoop datasets is represented in table C.2.

The obtained results are showed in the following figures: (3.2) balance, (3.3) correlation, (3.4) dimensionality, (3.5) linearity, (3.6) neighborhood, (3.7) network, (3.8) overlap, and (3.9) smoothness.

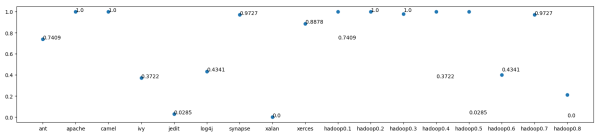


(a) Balance C1 (b) Balance C2

Figure 3.2: Balance Measures

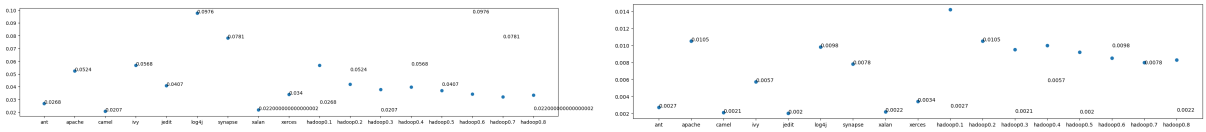


(a) Correlation C2 (b) Correlation C3



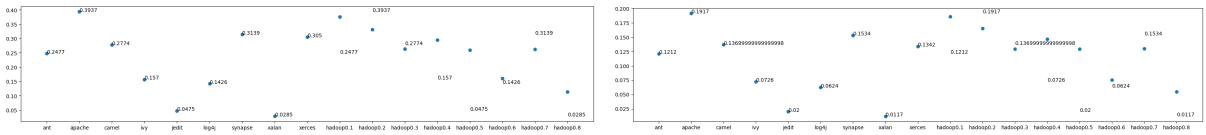
(c) Correlation C4

Figure 3.3: Correlation Measures



(a) Dimensionality T2 (b) Dimensionality T3

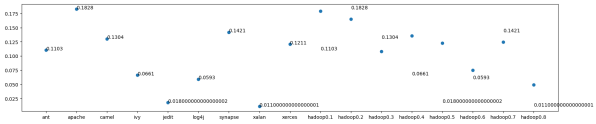
Figure 3.4: Dimensionality Measures



(a) Linearity L1 (b) Linearity L2

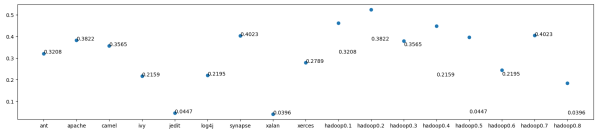
This way it is easier to compare results through datasets and metrics. To see the whole extent of the numerical values, see the tables at Appendix C. The data is summarized in tables C.1 and C.2. To see the images in more detail (bigger) go to Appendix D, Section D.1.

Further results are compared and discussed in Section 3.4.5.

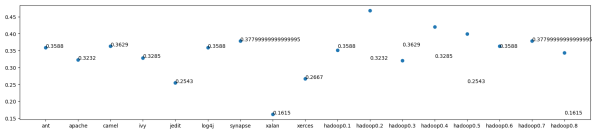


(c) Linearity L3

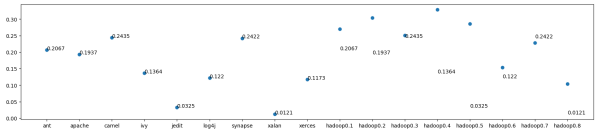
Figure 3.5: Linearity Measures



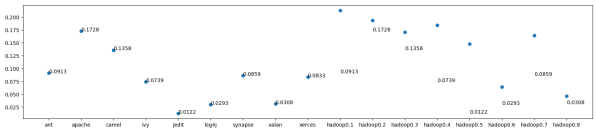
(a) Neighborhood N1



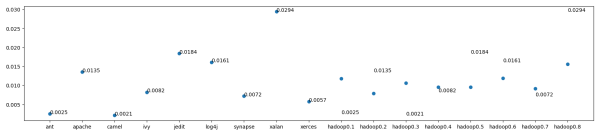
(b) Neighborhood N2



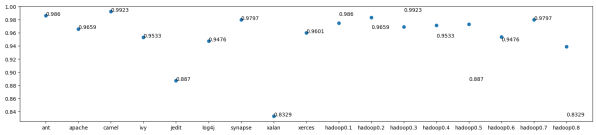
(c) Neighborhood N3



(d) Neighborhood N4

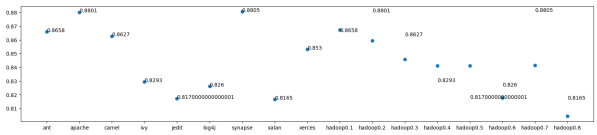


(e) Neighborhood T1

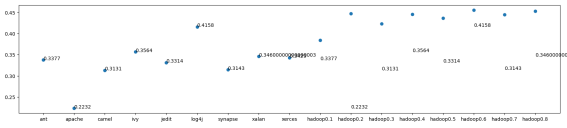


(f) Neighborhood LSC

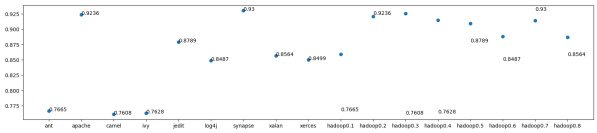
Figure 3.6: Neighborhood Measures



(a) Network Density

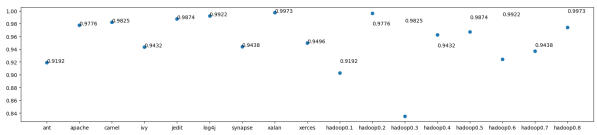


(b) Network ClsCoef

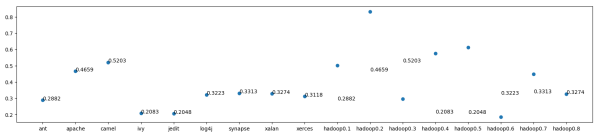


(c) Network Hubs

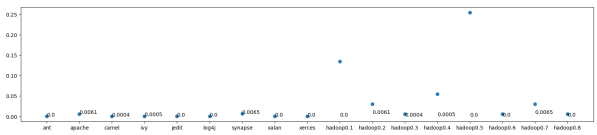
Figure 3.7: Network Measures



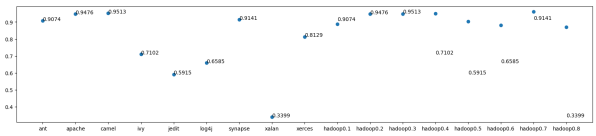
(a) Overlap F1



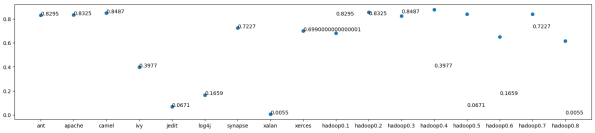
(b) Overlap F1v



(c) Overlap F2

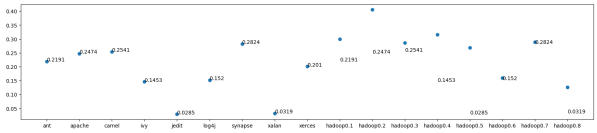


(d) Overlap F3

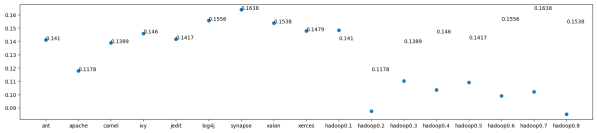


(e) Overlap F4

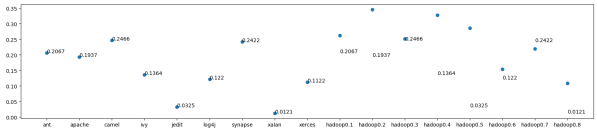
Figure 3.8: Overlap Measures



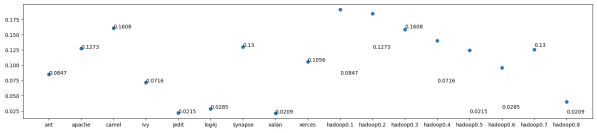
(a) Smoothness S1



(b) Smoothness S2



(c) Smoothness S3



(d) Smoothness S4

Figure 3.9: Smoothness Measures

3.4.3. K-fold on Metrics Analysis

This experiment tries to find the impact of applying the K-fold Cross-Validation. It is a widely extended technique in data science that brings a solution to the problem of performing testing to a data set with no separate training data - a given data set should not be used for both training and testing for the classifier, instead data can be divided so a portion is used for training and another separate portion for testing/validation. Cross validation allows to estimate the performance of the model trained by the whole dataset dividing it into folds of two types:

Training set Known data by the classifier, and used to train it. It represents $k - 1/k$ parts of the original set.

Testing set Also known as validation set. Unknown data for the algorithm. Used to test the performance of the trained classifier. It usually is $1/k$ of the original dataset.

The K-fold algorithm has a total of k iterations, same as the number of folds (parts/divisions) of the dataset. On each iteration, the *testing set* is rotated to another unused fold[5] of the data - no testing set is repeated, which means that each fold is used once for *testing* purposes.

After all iterations, the performance of the classifier that would be obtained by training with the whole dataset is estimated by doing the mean of the performances on each iteration of the algorithm.

The most common values for k (folds) used in data science are 5 and 10. This this experiment, k has been selected as $k = 5$, as a common value of k was needed for all experiments and some of the datasets did not have enough samples for a larger value of k.

The algorithm that analyzes the performance of each fold and the calculates the mean performance can be simplified as the one explained in the Algorithm 2.

Algorithm 2 Metrics Analysis on K-fold

[5]The folds do not change throughout the cross validation process.

Require: $\mathcal{D}$ dataset
1: inputs, targets ← $\mathcal{D}$
2: k ← 5
3: kf ← kfold(k)
4: splits ← kf.split(inputs)
5: **for all** $train_i$, $test_i$ in splits **do**
6: x_{train}, x_{test} ← inputs.get($train_i$), inputs.get($test_i$)
7: y_{train}, y_{test} ← targets.get($train_i$), targets.get($test_i$)
8: clf ← trainNetwork(x_{train}, y_{train})
9: confusionMatrix(x_{test}, y_{test}, clf)
10: **end for**
11: calculateMeanPerformance(clf)

The real script used for this experiment can be found in Appendix B, Section B.4. It basically separates a certain dataset into folds and calculates the analysis metrics for each fold, to later on compute the mean value of those metrics for the entire dataset.

Thanks to the results of the results of the classifiers for each fold, first, the *confusion matrix* is obtained and thereafter, some other metrics are calculated (such as recall, MCC, etc.). Those metrics are then compared using a table like tab 3.4 or 3.5, for each specific dataset, to see the full extent of the comparison refer to Section 3.4.5.

Only two tables of the aforementioned resulting dataset tables have been included in this document, as an example of the results obtained through the experiment.

The first case is the table obtained from Apache dataset (see Table 3.4).

Table 3.4: K-Fold Metrics Tree Apache

Fold	***Function***	*Precision*	*Recall*	*Fall Out*	*Balanced*	*F1*	*MCC*	*AUC*
	Naive Bayes	0.4286	0.1579	0.200	0.4789	0.2308	-0.0548	0.4789
1	**Decision Tree**	0.8235	0.7368	0.1500	0.7934	0.7778	0.5915	0.7934
	Nearest Centroid	0.3333	0.3158	0.6000	0.3579	0.3243	-0.2850	0.3579
	Naive Bayes	0.5000	0.0800	0.1538	0.4631	0.1379	-0.1142	0.4631
2	**Decision Tree**	0.8095	0.6800	0.3077	0.6862	0.7391	0.3552	0.6862
	Nearest Centroid	0.5000	0.1600	0.3077	0.4262	0.2424	-0.1719	0.4262
	Naive Bayes	0.5000	0.0909	0.1250	0.4830	0.1538	-0.0548	0.4830
3	**Decision Tree**	0.8462	0.5000	0.1250	0.6875	0.6286	0.3903	0.6875
	Nearest Centroid	0.4000	0.1818	0.3750	0.4034	0.2500	-0.2166	0.4034
	Naive Bayes	0.3333	0.8182	0.6667	0.5758	0.4737	0.1515	0.5758
4	**Decision Tree**	0.5833	0.6364	0.1852	0.7256	0.6087	0.4402	0.7256
	Nearest Centroid	0.8000	0.7273	0.0741	0.8266	0.7619	0.6727	0.8266
	Naive Bayes	0.3043	1.0000	0.5161	0.7419	0.4667	0.3838	0.7419
5	**Decision Tree**	0.5000	1.0000	0.2258	0.8871	0.6667	0.6222	0.8871
	Nearest Centroid	0.4545	0.7143	0.1935	0.7604	0.5556	0.4451	0.7604
	Naive Bayes	0.4132	0.4294	0.3323	0.5485	0.2926	0.0623	0.5485
Mean	**Decision Tree**	0.7125	0.7106	0.1987	0.7560	0.6842	0.4799	0.7560
	Nearest Centroid	0.4976	0.4198	0.3101	0.5549	0.4268	0.0889	0.5549

The second example is the table obtained on the experiment performed over the Hadoop (v0.8) dataset (see Table 3.5).

Table 3.5: K-Fold Metrics Tree Hadoop 0.8

Fold	***Function***	*Precision*	*Recall*	*Fall Out*	*Balanced*	*F1*	*MCC*	*AUC*
	Naive Bayes	1.0000	0.1250	0.0000	0.5625	0.2222	0.3262	0.5625
1	**Decision Tree**	0.0000	0.0000	0.0250	0.4875	0.0000	-0.0652	0.4875
	Nearest Centroid	0.1667	1.0000	1.0000	0.5000	0.2857	0.0000	0.5000
	Naive Bayes	0.0000	0.0000	0.1064	0.4468	0.0000	-0.0497	0.4468
2	**Decision Tree**	0.0000	0.0000	0.1064	0.4468	0.0000	-0.0497	0.4468
	Nearest Centroid	0.0208	1.0000	1.0000	0.5000	0.0408	0.0000	0.5000
	Naive Bayes	0.0000	0.0000	0.0851	0.4574	0.0000	-0.0440	0.4574
3	**Decision Tree**	0.0000	0.0000	0.0851	0.4574	0.0000	-0.0440	0.4574
	Nearest Centroid	0.0208	1.0000	1.0000	0.5000	0.0408	0.0000	0.5000
	Naive Bayes	0.3333	0.2500	0.0455	0.6023	0.2857	0.2335	0.6023
4	**Decision Tree**	0.0000	0.0000	0.0426	0.4787	0.0000	-0.0304	0.4787
	Nearest Centroid	0.0000	0.0000	0.0227	0.4886	0.0000	-0.0440	0.4886
	Naive Bayes	0.0000	0.0000	0.0000	0.5000	0.0000	0.0000	0.5000
5	**Decision Tree**	0.0000	0.0000	0.0000	0.5000	0.0000	0.0000	0.5000
	Nearest Centroid	0.0000	0.0000	0.0000	0.5000	0.0000	0.0000	0.5000
	Naive Bayes	0.26667	0.0750	0.0474	0.5138	0.1016	0.0932	0.5138
Mean	**Decision Tree**	0.0000	0.0000	0.0484	0.4758	0.0000	-0.0446	0.4758
	Nearest Centroid	0.0417	0.6000	0.6045	0.4977	0.0735	-0.0088	0.4977

The inner results of each fold are not evaluated when analyzing the results of the experiments, but that data could be used to understand how does imbalance exactly affect classification. The objective evaluated in this paper is to see if the metrics obtained for each dataset have any relation with the performance of the classifiers.

3.4.4. Filters affect Classification

After experimenting with K-folding, it is time to see if filters applied to the dataset have any effect on the quality of the trained classifier. In this case it is carried out by applying imbalance filters to the folds in the datasets (see Section 2.3 for more information on imbalance filters).

The main target is, as the avid reader can guess, imbalanced datasets and how filters that affect imbalance also affect the performance and other metrics of classifiers - results are going to be compared on Section 3.4.5.

Out of all the aforementioned techniques (2.3) available for imbalanced datasets, only the two following techniques are going to be used in the experiments:

Undersampling Removing samples from majority class.

Oversampling Creating new samples in the minority class.

These techniques have been selected as they are more than enough to see if classifiers are affected by them. These two techniques have been used in two separate experiments.

3.4.4.1. Undersampling and K-fold on Classification

For this experiment, one undersampling algorithm is applied (2.3.1.1.3) to see how folds are affected by removing majority class samples, and therefore, looking at how it impacts the classification algorithm being used is .

The undersampling technique is not done on the whole dataset. Actually, the folds are the ones affected by this algorithm. That way, instead of removing the imbalance in the whole dataset, it is removed in each fold as necessary, as some folds might have different imbalance level: what might be the majority class in the whole dataset can be the minority class of a certain fold.

The value selected for K-fold Cross Validation is $k = 5$. A simplification of the procedure followed by this experiment can be found in the Algorithm 3.

Algorithm 3 Undersampling and K-fold on classification performance

Require: $\mathcal{D}$ dataset

```
inputs, targets ← 𝒟
k ← 5
kf ← kfold(k)
splits ← kf.split(inputs)
samplingStrategy ← 0:50, 1:50        ▷ Binary class with a post undersampling distribution of 50-50
randomState ← 42
for all train_i, test_i in splits do
    x_train, x_test ← inputs.get(train_i), inputs.get(test_i)
    y_train, y_test ← targets.get(train_i), targets.get(test_i)
    X, Y = makeImbalance(
        x_train, y_train,
        samplingStrategy,
        randomState
    )
    clf ← classifier(NearMiss(2), GaussianNB())
    clf ← trainNetwork(X, Y)
    confusionMatrix(x_test, y_test, clf)
end for
calculateMeanPerformance(clf)
```

Just as before, more than one classification algorithm has been used to measure performance of the alterations done on the dataset - (1) Naive Bayes (Gaussian); (2) Decision Tree; and (3) kNN Nearest Centroid.

The experiment creates the same tables obtained in experiment 3.4.3, that is why sharing all the resulting tables of this experiment would be unnecessary - the results evaluated in this paper can be found in Section 3.4.5. That is why only two of them have been included into this document. The Apache dataset experiment (see Table 3.6).

Table 3.6: K-Fold Metrics with Undersampling Results Apache

Fold	*Function*	*Precision*	*Recall*	*Fall Out*	*Balanced*	*F1*	*MCC*	*AUC*
	Naive Bayes	0.5556	0.8824	0.5455	0.6684	0.6818	0.3620	0.6684
1	**Decision Tree**	0.7000	0.8235	0.2727	0.7754	0.7568	0.5464	0.7754
	Nearest Centroid	0.4167	0.2941	0.3182	0.4880	0.3448	-0.0259	0.4880
	Naive Bayes	0.5500	0.7333	0.3913	0.6710	0.6286	0.3348	0.6710
2	**Decision Tree**	0.5500	0.7333	0.3913	0.6710	0.6286	0.3348	0.6710
	Nearest Centroid	0.3571	0.3333	0.3913	0.4710	0.3448	-0.0587	0.4710
	Naive Bayes	0.7586	0.8800	0.5385	0.6708	0.8148	0.3811	0.6708
3	**Decision Tree**	0.8421	0.6400	0.2308	0.7046	0.7273	0.3883	0.7046
	Nearest Centroid	0.7000	0.2800	0.2308	0.5246	0.4000	0.0530	0.5246
	Naive Bayes	0.6667	0.7273	0.1481	0.7896	0.6957	0.5650	0.7896
4	**Decision Tree**	0.6667	0.7273	0.1481	0.7896	0.6957	0.5650	0.7896
	Nearest Centroid	0.3077	0.3636	0.3333	0.5152	0.3333	0.0290	0.5152
	Naive Bayes	0.5000	0.5625	0.4091	0.5767	0.5294	0.1517	0.5767
5	**Decision Tree**	0.6316	0.7500	0.3182	0.7159	0.6857	0.4264	0.7159
	Nearest Centroid	0.5556	0.3125	0.1818	0.5653	0.4000	0.1518	0.5653
	Naive Bayes	0.5928	0.6662	0.3917	0.6372	0.6059	0.2992	0.6372
Mean	**Decision Tree**	0.6781	0.7348	0.2722	0.7313	0.6988	0.4522	0.7313
	Nearest Centroid	0.4674	0.3167	0.2911	0.5128	0.3646	0.0298	0.5128

And the result obtained on the same experiment for the Hadoop (v0.8) experiment (see Table 3.7).

Table 3.7: K-Fold Metrics with Undersampling Results Hadoop 0.8

Fold	*Function*	*Precision*	*Recall*	*Fall Out*	*Balanced*	*F1*	*MCC*	*AUC*
	Naive Bayes	0.1429	0.5000	0.1304	0.6848	0.2222	0.2092	0.6848
1	**Decision Tree**	0.1053	1.0000	0.3696	0.8152	0.1905	0.2576	0.8152
	Nearest Centroid	0.0345	0.5000	0.6087	0.4457	0.0645	-0.0444	0.4457
	Naive Bayes	0.0000	0.0000	0.2667	0.3667	0.0000	-0.1491	0.3667
2	**Decision Tree**	0.0833	0.3333	0.2444	0.5444	0.1333	0.0497	0.5444
	Nearest Centroid	0.0952	0.6667	0.4222	0.6222	0.1667	0.1193	0.6222
	Naive Bayes	0.0833	0.5000	0.2391	0.6304	0.1429	0.1204	0.6304
3	**Decision Tree**	0.0000	0.0000	0.3261	0.3370	0.0000	-0.1406	0.3370
	Nearest Centroid	0.0000	0.0000	0.4348	0.2826	0.0000	-0.1762	0.2826
	Naive Bayes	0.1333	0.3333	0.3095	0.5119	0.1905	0.0170	0.5119
4	**Decision Tree**	0.1765	0.5000	0.3333	0.5833	0.2609	0.1153	0.5833
	Nearest Centroid	0.0625	0.1667	0.3571	0.4048	0.0909	-0.1336	0.4048
	Naive Bayes	0.0000	0.0000	0.1556	0.4222	0.0000	-0.1067	0.4222
5	**Decision Tree**	0.2000	0.6667	0.1778	0.7444	0.3077	0.2914	0.7444
	Nearest Centroid	0.0000	0.0000	0.2444	0.3778	0.0000	-0.1408	0.3778
	Naive Bayes	0.0719	0.2667	0.2203	0.5232	0.1111	0.0182	0.5232
Mean	**Decision Tree**	0.1130	0.5000	0.2902	0.6049	0.1785	0.1147	0.6049
	Nearest Centroid	0.0384	0.2667	0.4134	0.4266	0.0644	-0.0751	0.4266

As it has been mentioned in Section 2.3.1.1.3, there are several versions of *Near Miss* algorithm. For this experiment it has been used *version 2*. This undersampling algorithm selects the majority class samples with the least average distance to the 3 farthest minority class samples.

The analysis of the metrics obtained are in Section 3.4.5.

3.4.4.2. Oversampling and K-fold on Classification

In this experiment, similarly to Section 3.4.4.1, an imbalance filter is included to the K-fold Cross Validation process. In this case, the oversampling technique is implemented and is later on compared to other experiment results (see Section 3.4.5).

The technique has been applied to the different folds of the dataset, not to the entire data. Once again, the dataset is divided into $k = 5$ folds.

The logic of the experiment is summarized in the next script (see Algorithm 4). The difference with the undersampling Algorithm 3 is that uses SMOTE technique (see Section 2.3.1.2.2) instead of *Near Miss* undersampling technique.

Algorithm 4 Oversampling and K-fold on classification performance

Require: $\mathcal{D}$ dataset
1: inputs, targets ← $\mathcal{D}$
2: k ← 5
3: kf ← kfold(k)
4: splits ← kf.split(inputs)
5: **for all** $train_i$, $test_i$ in splits **do**
6: x_{train}, x_{test} ← inputs.get($train_i$), inputs.get($test_i$)
7: y_{train}, y_{test} ← targets.get($train_i$), targets.get($test_i$)
8: clf ← classifier(SMOTE(), GaussianNB())
9: clf ← trainNetwork(x_{train}, y_{train})
10: confusionMatrix(x_{test}, y_{test}, clf)
11: **end for**
12: calculateMeanPerformance(clf)

The classifiers used for this experiment are - (1) Naive Bayes (Gaussian); (2) Decision Tree; and (3) kNN Nearest Centroid.

Not all the resulting tables can be shown, so only two examples have been included. The first one is the result on the Apache dataset (see Table 3.8).

Table 3.8: K-Fold Metrics with Oversampling Results Apache

Fold	***Function***	*Precision*	*Recall*	*Fall Out*	*Balanced*	*F1*	*MCC*	*AUC*
	Naive Bayes	0.4286	0.1579	0.2000	0.4789	0.2308	-0.0548	0.4789
1	**Decision Tree**	0.7222	0.6842	0.2500	0.7171	0.7027	0.4354	0.7171
	Nearest Centroid	0.3333	0.3158	0.6000	0.3579	0.3243	-0.2850	0.3579
	Naive Bayes	0.5000	0.0800	0.1538	0.4631	0.1379	-0.1142	0.4631
2	**Decision Tree**	0.8889	0.6400	0.1538	0.7431	0.7442	0.4619	0.7431
	Nearest Centroid	0.5000	0.1600	0.3077	0.4262	0.2424	-0.1719	0.4262

Continued on next page

Table 3.8 – *Continued from previous page*

Fold	*Function*	*Precision*	*Recall*	*Fall Out*	*Balanced*	*F1*	*MCC*	*AUC*
	Naive Bayes	0.5000	0.0909	0.1250	0.4830	0.1538	-0.0548	0.4830
3	**Decision Tree**	0.8000	0.5455	0.1875	0.6790	0.6486	0.3616	0.6790
	Nearest Centroid	0.4286	0.1364	0.2500	0.4432	0.2069	-0.1447	0.4432
	Naive Bayes	0.3333	0.8182	0.6667	0.5758	0.4737	0.1515	0.5758
4	**Decision Tree**	0.5385	0.6364	0.2222	0.7071	0.5833	0.3959	0.7071
	Nearest Centroid	0.6000	0.5455	0.1481	0.6987	0.5714	0.4092	0.6987
	Naive Bayes	0.3182	1.0000	0.4839	0.7581	0.4828	0.4052	0.7581
5	**Decision Tree**	0.5000	1.0000	0.2258	0.8871	0.6667	0.6222	0.8871
	Nearest Centroid	0.4545	0.7143	0.1935	0.7604	0.5556	0.4451	0.7604
	Naive Bayes	0.4160	0.4294	0.3259	0.5518	0.2958	0.0666	0.5518
Mean	**Decision Tree**	0.6899	0.7012	0.2079	0.7467	0.6691	0.4554	0.7467
	Nearest Centroid	0.4633	0.3744	0.2999	0.5373	0.3801	0.0505	0.5373

The second example is the one from the Hadoop (v0.8) experiment (see Table 3.9).

Table 3.9: K-Fold Metrics with Oversampling Results Hadoop 0.8

Fold	*Function*	*Precision*	*Recall*	*Fall Out*	*Balanced*	*F1*	*MCC*	*AUC*
	Naive Bayes	0.1176	0.2500	0.3750	0.4375	0.1600	-0.0974	0.4375
1	**Decision Tree**	0.2500	0.1250	0.0750	0.5250	0.1667	0.0674	0.5250
	Nearest Centroid	0.1667	1.0000	1.0000	0.5000	0.2857	0.0000	0.5000
	Naive Bayes	0.1000	1.0000	0.1915	0.9043	0.1818	0.2843	0.9043
2	**Decision Tree**	0.0000	0.0000	0.4468	0.2766	0.0000	-0.1286	0.2766
	Nearest Centroid	0.0208	1.0000	1.0000	0.5000	0.0408	0.0000	0.5000
	Naive Bayes	0.0000	0.0000	0.1915	0.4043	0.0000	-0.0701	0.4043
3	**Decision Tree**	0.0000	0.0000	0.1064	0.4468	0.0000	-0.0497	0.4468
	Nearest Centroid	0.0208	1.0000	1.0000	0.5000	0.0408	0.0000	0.5000
	Naive Bayes	0.0667	0.5000	0.6364	0.4318	0.1176	-0.0778	0.4318
4	**Decision Tree**	0.2500	0.2500	0.0682	0.5909	0.2500	0.1818	0.5909
	Nearest Centroid	0.0000	0.0000	0.0000	0.5000	0.0000	0.0000	0.5000
	Naive Bayes	0.0000	0.0000	0.0000	0.5000	0.0000	0.0000	0.5000
5	**Decision Tree**	0.0000	0.0000	0.0435	0.4783	0.0000	-0.0435	0.4783
	Nearest Centroid	0.0000	0.0000	0.0000	0.5000	0.0000	0.0000	0.5000
	Naive Bayes	0.0569	0.3500	0.2789	0.5356	0.0919	0.0078	0.5356
Mean	**Decision Tree**	0.1000	0.0750	0.1480	0.4635	0.0833	0.0055	0.4635
	Nearest Centroid	0.0417	0.6000	0.6000	0.5000	0.0735	0.0000	0.5000

The analysis of the metrics obtained are in Section 3.4.5.

3.4.5. Compare Results

In this part of the paper, we are going to analyze the results coming from the previous experiments. These are displayed in tables 3.10, 3.11 and 3.12.

To answer RQ1 and RQ2 (1.2 and 1.2, respectively) we need to observe, for example, the values from *overlapping*, we assume that there is a certain relation with the classification performance. For most cases in our experimentation, the higher the overlapping rate, the less efficient the classifier becomes. Of course, this assumption is not backed up by any statistical experiment on the datasets and/or the classifiers, but it looking at the results it looks right at first sight.

For example, looking at the *overlapping* measures, when *overlapping F1* is higher than 0.90, almost none of the classifiers is able to surpass 0.5 in *precision* score, something similar happens with the *recall* an *fall-out* scores (all three measures indicate on a relative value how many input samples are correctly predicted). It makes sense once the reader realizes that the more interlaced the samples are (the more *overlapping*), it should be harder for a classifier to identify what data belongs to each class (RQ2 - see 1.2).

In a similar way, it *linearity* measures (RQ1 - see 1.2) are directly related to the performance of the classification algorithms. Having high *overlapping* measures means that the *linearity* should be smaller (it is harder to separate data using a linear function if data is interlaced) - RQ2 (see 1.2). Therefore, the higher the *linearity* measures obtained, the higher the performance should be. No examples of this behavior can be shown, as all the selected datasets seem to have high *overlapping* values.

A third complexity metric that should be taken into account, regarding classifiers performance, is *balance* - (RQ1 and RQ3, 1.2 and 1.2, respectively). Its measures try to approximate the level of imbalance of a certain class. Therefore, the higher the measures' values, the greater should be the gap between the number of samples from the majority and minority class (all the datasets work with binary classes).

It can be observed that the greater the imbalances, not only the results have poor performance (less than 0.7 of *precision*, and similarly with *recall* and *fall-out* measures), but it also means that the different classifiers have very different results on the same dataset. This can be translated as, the performance on certain classifiers might depend on the imbalance of a certain dataset (RQ2 - see 1.2).

Taking a look at *Balanced Accuracy* measure (useful for imbalanced datasets, like the ones used for the experiments on this paper), it can be observed that most of the classifiers do not behave as they should in either the positive or negative prediction. In example, all of the hadoop datasets, where the imbalance ratio (*balance* measures) is not ideal, and the overlapping is too high.

Therefore, *MCC*, as a measure to see the quality of a binary classifier should also be low - just like in hadoop datasets.

As for this last observation, it could be assumed that filters that would reduce imbalance, should also be able to increase the performance of the classifiers.

After some experimentation regarding this matter, no clear improvements have been noticed in any of the analytical measures. Furthermore, undersampling seems not to be a good technique when the number of samples is too low, as the number of samples in the resulting training dataset is not enough to obtain a good classifier. In some cases, it even reduces the performance.

After using SMOTE oversampling technique, the *Balanced Accuracy* metric seems to indicate that the classifier identified slightly better some of the testing data samples, but the resulting classifier still does not meet the desired quality (*MCC*).

But, overall, the techniques applied do not seem to fix perfectly the problems arose from either imbalance or overlapped datasets. This topic is going to be further discussed in Section 4, as well as some other remarks about the selected classifiers and possible further experimentation on this topic.

On trying to answer the final RQ, *Do complexity metrics tell us something about the quality of the dataset?* (see 1.2), we can assume (looking at the results), they do. A dataset with bad quality could be considered one that has high overlapping, imbalanced classes, etc. As the results seem to indicate a relation between those values and the performance of the obtained classifier - as the analytical metrics tell us.

Table 3.10: All Metrics

Measure	*Metr.*	*ant*			*apache*			*camel*			*ivy*			*jedit*			*log4j*		
Balance	*C1*	0.7653			0.9895			0.7114			0.5108			0.1545			0.3953		
	C2	0.4701			0.0286			0.5429			0.7477			0.9543			0.8319		
Correlation	*C2*	0.2622			0.0891			0.1227			0.1879			0.0448			0.0804		
	C3	0.6338			0.7372			0.5763			0.5588			0.5324			0.5780		
	C4	0.7409			1.0000			1.0000			0.3722			0.0285			0.4341		
Dimension.	*T2*	0.0268			0.0524			0.0207			0.0568			0.0407			0.0976		
	T3	0.0027			0.0105			0.3000			0.0057			0.0020			0.0098		
Linearity	*L1*	0.2477			0.3937			0.2774			0.1570			0.0475			0.1426		
	L2	0.1212			0.1917			0.1370			0.0726			0.0200			0.0624		
	L3	0.1103			0.1828			0.1304			0.0661			0.0180			0.0593		
Neighbor.	*N1*	0.3208			0.3822			0.3565			0.2159			0.0447			0.2195		
	N2	0.3588			0.3232			0.3629			0.3285			0.2543			0.3588		
	N3	0.2067			0.1937			0.2435			0.1364			0.0325			0.1220		
	N4	0.0913			0.1728			0.1358			0.0739			0.0122			0.0293		
	T1	0.0025			0.0135			0.0021			0.0082			0.0184			0.0161		
	LSC	0.9860			0.9659			0.9923			0.9533			0.8870			0.9476		
Network	*Density*	0.8658			0.8801			0.8627			0.8293			0.8170			0.8260		
	ClsCoef	0.3377			0.2232			0.3131			0.3564			0.3314			0.4158		
	Hubs	0.7665			0.9236			0.7608			0.7628			0.8789			0.8487		
Overlap	*F1*	0.9192			0.9776			0.9825			0.9432			0.9874			0.9922		
	F1v	0.2882			0.4659			0.5203			0.2083			0.2048			0.3223		
	F2	0.0000			0.0061			0.0004			0.0005			0.0000			0.0000		
	F3	0.9074			0.9476			0.9513			0.7102			0.5915			0.6585		
	F4	0.8295			0.8325			0.8487			0.3977			0.0671			0.1659		
Smooth.	S1	0.2191			0.2474			0.2541			0.1453			0.0285			0.1520		
	S2	0.1410			0.1178			0.1389			0.1460			0.1417			0.1556		
	S3	0.2067			0.1937			0.2466			0.1364			0.0325			0.1220		
	S4	0.0847			0.1273			0.1608			0.0716			0.0215			0.0285		
		NB	*DT*	*NC*	*NB*	*DT*	*NC*	*NB*	*DT*	*NC*	*NB*	*DT*	*NC*	*NB*	*DT*	*NC*	*NB*	*DT*	*NC*
Precision	1	0.5600	0.4598	0.6263	0.4132	0.7125	0.4976	0.4478	0.3275	0.3304	0.2687	0.2533	0.3401	0.0645	0.1000	0.0496	0.9734	0.9350	0.9093
	2	0.5093	0.3699	0.6367	0.5928	0.6781	0.4674	0.3937	0.2548	0.3099	0.2901	0.1836	0.3451	0.0179	0.0402	0.1127	0.9665	0.9630	0.9173
	3	0.5152	0.4200	0.6166	0.4160	0.6899	0.4633	0.4372	0.2891	0.3243	0.2422	0.2478	0.3220	0.0427	0.1686	0.0491	0.9734	0.9510	0.9149
Recall	1	0.5497	0.4677	0.5105	0.4294	0.7106	0.4198	0.2600	0.3732	0.2550	0.3122	0.2389	0.4372	0.1300	0.0500	0.1900	0.4912	0.9151	0.3216
	2	0.6068	0.6416	0.4788	0.6662	0.7348	0.3167	0.2990	0.6038	0.2850	0.5508	0.6765	0.5000	0.3667	0.8500	0.4167	0.4786	0.5498	0.2804
	3	0.5913	0.5115	0.5164	0.4294	0.7012	0.3744	0.3212	0.3528	0.3203	0.4011	0.3122	0.4372	0.1800	0.3800	0.1900	0.5017	0.9261	0.3430
Fall Out	1	0.1231	0.1552	0.0866	0.3323	0.1987	0.3101	0.0763	0.1838	0.1266	0.1104	0.0734	0.1008	0.1027	0.0226	0.1136	0.2167	0.7333	0.4000
	2	0.1703	0.3126	0.0790	0.3917	0.2722	0.2911	0.1171	0.4313	0.1399	0.1603	0.3546	0.0996	0.3132	0.4359	0.0624	0.3200	0.2133	0.3533
	3	0.1609	0.2051	0.0919	0.3259	0.2079	0.2999	0.0995	0.2072	0.1600	0.1520	0.1248	0.1101	0.2876	0.0643	0.1300	0.2167	0.5333	0.4000
Balanced	1	0.7133	0.6563	0.7120	0.5485	0.7560	0.5549	0.5919	0.5947	0.5642	0.6009	0.5827	0.6682	0.6046	0.6097	0.6341	0.6373	0.5909	0.4608
	2	0.7182	0.6645	0.6998	0.6372	0.7313	0.5128	0.5909	0.5862	0.5726	0.6952	0.6609	0.7002	0.5267	0.7071	0.6771	0.5793	0.6682	0.4635
	3	0.7152	0.6532	0.7122	0.5518	0.7467	0.5373	0.6109	0.5728	0.5801	0.6245	0.5937	0.6635	0.5098	0.7508	0.6229	0.6425	0.6964	0.4715
F1	1	0.5547	0.4598	0.5610	0.2926	0.6842	0.4268	0.3285	0.3453	0.2850	0.2844	0.2428	0.3795	0.0796	0.0667	0.0656	0.6448	0.9248	0.4713
	2	0.5445	0.4672	0.5423	0.6059	0.6988	0.3646	0.3333	0.3573	0.2845	0.3703	0.2812	0.3956	0.0334	0.0754	0.1664	0.6301	0.6914	0.4216
	3	0.5489	0.4606	0.5605	0.2958	0.6691	0.3801	0.3679	0.3158	0.3203	0.3009	0.2719	0.3703	0.0619	0.2033	0.0649	0.6548	0.9372	0.4946
MCC	1	0.4295	0.3096	0.4577	0.0623	0.4799	0.0889	0.2285	0.1789	0.1425	0.1929	0.1647	0.3027	0.0481	0.0576	0.0497	0.1564	0.1543	-0.0402
	2	0.4117	0.2809	0.4446	0.2992	0.4522	0.0298	0.2050	0.1376	0.1454	0.2943	0.2032	0.3449	0.0036	0.1113	0.1781	0.1162	0.1648	-0.0409
	3	0.4114	0.2872	0.4538	0.0666	0.4554	0.0505	0.2497	0.1342	0.1601	0.2056	0.1726	0.2886	0.0122	0.2086	0.0464	0.1612	0.4101	-0.0274
AUC	1	0.7133	0.6563	0.7120	0.5485	0.7560	0.5549	0.5919	0.5947	0.5642	0.6009	0.5827	0.6682	0.5285	0.5222	0.5653	0.6373	0.5909	0.4608
	2	0.7182	0.6645	0.6998	0.6372	0.7313	0.5128	0.5909	0.5862	0.5726	0.6952	0.6609	0.7002	0.5267	0.7071	0.6771	0.5793	0.6682	0.4635
	3	0.7152	0.6532	0.7122	0.5518	0.7467	0.5373	0.6109	0.5728	0.5801	0.6245	0.5937	0.6635	0.4782*	0.7062*	0.5463*	0.6425	0.6964	0.4715

Table 3.11: All Metrics

Measure	*Metr.*	*synapse*			*xalan*			*xerces*			*hadoop 0.1*			*hadoop 0.2*			*hadoop 0.3*		
Balance	*C1*	0.9209			0.0944			0.8219			0.9381			0.7600			0.8132		
	C2	0.1944			0.9755			0.3826			0.1559			0.4777			0.3970		
Correlation	*C2*	0.2310			0.0663			0.2839			0.2868			0.0520			0.3171		
	C3	0.7762			0.5275			0.6338			0.7741			0.6006			0.6330		
	C4	0.9727			0.0000			0.8878			1.0000			1.0000			0.9810		
Dimension.	*T2*	0.0781			0.0220			0.0340			0.0496			0.0366			0.0332		
	T3	0.0078			0.0022			0.0034			0.0142			0.0052			0.0095		
Linearity	*L1*	0.3139			0.0285			0.3050			0.3770			0.3327			0.2658		
	L2	0.1534			0.0117			0.1342			0.1870			0.1664			0.1305		
	L3	0.1421			0.0110			0.1211			0.1832			0.1644			0.1055		
Neighbor.	*N1*	0.4023			0.0396			0.2789			0.4965			0.4712			0.4028		
	N2	0.3780			0.1615			0.2667			0.3474			0.4611			0.3151		
	N3	0.2422			0.0121			0.1173			0.2553			0.3037			0.2417		
	N4	0.0859			0.0308			0.0833			0.2411			0.2408			0.1374		
	T1	0.0072			0.0294			0.0057			0.0127			0.0079			0.0118		
	LSC	0.9797			0.8329			0.9601			0.9755			0.9838			0.9686		
Network	*Density*	0.8805			0.8165			0.8530			0.8678			0.8610			0.8478		
	ClsCoef	0.3143			0.3460			0.3429			0.3965			0.4411			0.4281		
	Hubs	0.9300			0.8564			0.8499			0.8595			0.9252			0.9305		
Overlap	*F1*	0.9438			0.9973			0.9496			0.9129			0.9958			0.8365		
	F1v	0.3313			0.3274			0.3118			0.5101			0.8483			0.3019		
	F2	0.0065			0.0000			0.0000			0.1818			0.0508			0.0135		
	F3	0.9141			0.3399			0.8129			0.8865			0.9791			0.9479		
	F4	0.7227			0.0055			0.6990			0.6879			0.9215			0.8294		
Smooth.	S1	0.2824			0.0319			0.2010			0.3643			0.3421			0.2952		
	S2	0.1638			0.1538			0.1479			0.1465			0.0908			0.1134		
	S3	0.2422			0.0121			0.1122			0.2411			0.3246			0.2417		
	S4	0.1300			0.0209			0.1056			0.2003			0.1863			0.1552		
		NB	*DT*	*NC*	*NB*	*DT*	*NC*	*NB*	*DT*	*NC*	*NB*	*DT*	*NC*	*NB*	*DT*	*NC*	*NB*	*DT*	*NC*
Precision	1	0.6317	0.5581	0.6431	0.9963	0.9944	0.9971	0.9339	0.9374	0.9283	0.6933	0.4782	0.5091	0.1357	0.0286	0.1097	0.6467	0.3671	0.5821
	2	0.5499	0.4756	0.6472	0.9942	0.9973	1.0000	0.9200	0.9395	0.9385	0.6515	0.5028	0.5933	0.1335	0.1998	0.1852	0.6384	0.3965	0.6784
	3	0.6325	0.5645	0.6444	0.9952	0.9955	0.9971	0.9322	0.9347	0.9283	0.6933	0.5246	0.5091	0.1324	0.1059	0.1403	0.6133	0.3666	0.5738
Recall	1	0.4927	0.5641	0.4950	0.8721	0.9911	0.3930	0.5768	0.9250	0.3007	0.3086	0.6962	0.3276	0.1736	0.1000	0.3305	0.7038	0.8410	0.7295
	2	0.5353	0.5741	0.5309	0.9511	0.7158	0.4923	0.6346	0.8836	0.2577	0.4814	0.6905	0.4447	0.2634	0.3915	0.4613	0.5322	0.8189	0.4900
	3	0.5618	0.6636	0.5342	0.9032	0.9933	0.4019	0.5931	0.9218	0.3007	0.3276	0.7795	0.3276	0.2471	0.1638	0.3442	0.7133	0.8457	0.7391
Fall Out	1	0.1613	0.2309	0.1467	0.2167	0.3667	0.0500	0.1200	0.1832	0.0669	0.0835	0.3650	0.1437	0.2528	0.3141	0.5494	0.0464	0.1936	0.0820
	2	0.2225	0.3016	0.1399	0.3867	0.0800	0.0000	0.1576	0.1607	0.0498	0.1468	0.3506	0.1628	0.3558	0.4480	0.5418	0.0996	0.4086	0.1129
	3	0.1767	0.2634	0.1579	0.3167	0.2667	0.0500	0.1271	0.1880	0.0669	0.0835	0.3553	0.1437	0.5278	0.3680	0.5611	0.0566	0.1841	0.0872
Balanced	1	0.6657	0.6666	0.6742	0.8277	0.8122	0.6715	0.7284	0.8710	0.6169	0.6126	0.6656	0.5920	0.4604	0.3929	0.3906	0.7739	0.7642	0.7619
	2	0.6564	0.6362	0.6955	0.7822	0.8179	0.7461	0.7385	0.8614	0.6039	0.6673	0.6699	0.6410	0.4538	0.4717	0.4598	0.7163	0.7051	0.6885
	3	0.6926	0.7001	0.6881	0.7933	0.8633	0.6759	0.7330	0.8669	0.6169	0.6221	0.7121	0.5920	0.3597	0.3979	0.3916	0.7784	0.7737	0.7688
F1	1	0.5450	0.5495	0.5506	0.9295	0.9927	0.5623	0.7126	0.9311	0.4534	0.4071	0.5033	0.3942	0.0702	0.0444	0.0969	0.6307	0.3834	0.5626
	2	0.5286	0.5124	0.5754	0.9717	0.8268	0.6487	0.7500	0.9103	0.3993	0.5389	0.5776	0.5015	0.1642	0.2637	0.2389	0.5738	0.5184	0.5567
	3	0.5823	0.5955	0.5721	0.9467	0.9944	0.5712	0.7244	0.9282	0.4534	0.4235	0.5523	0.3942	0.0805	0.0812	0.1134	0.6092	0.3915	0.5657
MCC	1	0.3601	0.3322	0.3780	0.1999	0.4730	0.0693	0.3998	0.7314	0.2375	0.2969	0.2480	0.2093	-0.0751	-0.1679	-0.1940	0.4862	0.2945	0.4438
	2	0.3131	0.2548	0.4089	0.3450	0.1551	0.0962	0.4167	0.6855	0.2254	0.3677	0.3349	0.3051	-0.0992	-0.0367	-0.0672	0.4569	0.3571	0.4334
	3	0.3971	0.3855	0.3983	0.1871	0.6163	0.0708	0.4073	0.7251	0.2375	0.3108	0.3369	0.2093	-0.2479	-0.1659	-0.1773	0.4592	0.2994	0.4363
AUC	1	0.6657	0.6666	0.6742	0.8277	0.8122	0.6715	0.7284	0.8710	0.6169	0.6126	0.6656	0.5920	0.4604	0.3929	0.3906	0.8543*	0.8540*	0.8571*
	2	0.6564	0.6362	0.6955	0.7822	0.8179	0.7461	0.7385	0.8614	0.6039	0.6673	0.6699	0.6410	0.4538	0.4717	0.4598	0.7163	0.7051	0.6885
	3	0.6926	0.7001	0.6881	0.7933	0.8633	0.6759	0.7330	0.8669	0.6169	0.6221	0.7121	0.5920	0.3597	0.3979	0.3916	0.8480*	0.8599*	0.8538*

Table 3.12: All Metrics

Measure	*Metr.*	*hadoop 0.4*			*hadoop 0.5*			*hadoop 0.6*			*hadoop 0.7*			*hadoop 0.8*		
Balance	*C1*	0.7395			0.6589			0.5642			0.7056			0.3534		
	C2	0.5062			0.6056			0.7015			0.5501			0.8579		
Correlation	*C2*	0.1899			0.1601			0.1705			0.1997			0.1241		
	C3	0.5956			0.5589			0.5397			0.5737			0.4917		
	C4	1.0000			1.0000			0.7692			0.9720			0.2125		
Dimensional.	*T2*	0.0348			0.0322			0.0299			0.0280			0.0292		
	T3	0.0050			0.0046			0.0042			0.0040			0.0042		
Linearity	*L1*	0.2950			0.2620			0.1996			0.2654			0.1134		
	L2	0.1473			0.1310			0.0981			0.1327			0.0550		
	L3	0.1408			0.1274			0.0875			0.1254			0.0489		
Neighborh.	*N1*	0.4378			0.4009			0.3333			0.3840			0.1708		
	N2	0.4207			0.3960			0.3457			0.3682			0.3151		
	N3	0.3234			0.2857			0.1838			0.2200			0.1208		
	N4	0.1990			0.1429			0.1282			0.0960			0.0417		
	T1	0.0101			0.0096			0.0100			0.0087			0.0143		
	LSC	0.9708			0.9725			0.9686			0.9786			0.9428		
Network	*Density*	0.8433			0.8430			0.8232			0.8426			0.8030		
	ClsCoef	0.4453			0.4398			0.4383			0.4435			0.4465		
	Hubs	0.9121			0.9152			0.8929			0.9074			0.8912		
Overlap	*F1*	0.9641			0.9643			0.9519			0.9370			0.9711		
	F1v	0.5797			0.6427			0.4057			0.4816			0.3269		
	F2	0.1107			0.3023			0.0257			0.0552			0.0329		
	F3	0.9900			0.9032			0.9530			0.9600			0.8750		
	F4	0.9453			0.8479			0.8846			0.9000			0.6875		
Smooth.	S1	0.3100			0.2778			0.2446			0.2731			0.1255		
	S2	0.1075			0.1108			0.1122			0.1043			0.0961		
	S3	0.3284			0.2857			0.1709			0.2080			0.1208		
	S4	0.1516			0.1646			0.1175			0.1631			0.0625		
		NB	*DT*	*NC*	*NB*	*DT*	*NC*	*NB*	*DT*	*NC*	*NB*	*DT*	*NC*	*NB*	*DT*	*NC*
Precision	1	0.4150	0.4632	0.3438	0.3619	0.1294	0.2726	0.3700	0.1600	0.0000	0.3149	0.3302	0.2639	0.2667	0.0000	0.0417
	2	0.4267	0.2003	0.3316	0.2332	0.1710	0.2274	0.3267	0.2323	0.5909	0.3433	0.2340	0.3688	0.0719	0.1130	0.0384
	3	0.3778	0.3571	0.3395	0.2267	0.1775	0.2155	0.1567	0.2318	0.0000	0.3067	0.2968	0.2693	0.0569	0.1000	0.0417
Recall	1	0.2478	0.4868	0.3240	0.2174	0.1687	0.3062	0.2192	0.0500	0.0000	0.2554	0.5484	0.3218	0.0750	0.0000	0.6000
	2	0.4165	0.4451	0.3321	0.2713	0.5327	0.2982	0.5695	0.7195	0.4889	0.3500	0.5833	0.3643	0.2667	0.5000	0.2667
	3	0.3164	0.4344	0.3565	0.1841	0.2228	0.3723	0.2067	0.1525	0.0000	0.2954	0.5537	0.4270	0.3500	0.0750	0.6000
Fall Out	1	0.0990	0.1244	0.1629	0.0885	0.2025	0.1959	0.0683	0.0524	0.0000	0.0928	0.1716	0.1774	0.0474	0.0272	0.6045
	2	0.1498	0.4898	0.1637	0.2145	0.4981	0.1926	0.1765	0.3742	0.0792	0.1585	0.4321	0.1740	0.2203	0.2902	0.4134
	3	0.1522	0.1915	0.1757	0.1387	0.2685	0.3335	0.0820	0.1207	0.0000	0.1181	0.2058	0.1989	0.2789	0.1480	0.6000
Balanced	1	0.5744	0.6812	0.5805	0.5645	0.4831	0.5551	0.5754	0.4988	0.5000	0.6753	0.7644	0.6602	0.5138	0.4864	0.4977
	2	0.6334	0.4777	0.5842	0.5284	0.5173	0.5528	0.6965	0.6726	0.7048	0.5958	0.5756	0.5951	0.5232	0.6049	0.4266
	3	0.5821	0.6214	0.5904	0.5227	0.4772	0.5194	0.5623	0.5159	0.5000	0.6827	0.7519	0.7021	0.5356	0.4635	0.5000
F1	1	0.2896	0.4405	0.3090	0.2507	0.1397	0.2680	0.1813	0.0762	0.0000	0.2347	0.2974	0.2314	0.1016	0.0000	0.0735
	2	0.3952	0.2738	0.3268	0.2268	0.2554	0.2453	0.4051	0.3372	0.4850	0.3403	0.3224	0.3525	0.1111	0.1785	0.0644
	3	0.3165	0.3699	0.3324	0.1834	0.1789	0.2679	0.1486	0.0750	0.0000	0.2524	0.2784	0.2502	0.0919	0.0833	0.0735
MCC	1	0.1832	0.3324	0.1645	0.1589	-0.0435	0.1108	0.1595	0.0220	0.0000	0.1572	0.2647	0.1267	0.0932	-0.0347	-0.0088
	2	0.2686	-0.0322	0.1628	0.0600	0.0153	0.0882	0.3136	0.2382	0.4386	0.1930	0.1153	0.2090	0.0182	0.1147	-0.0751
	3	0.1767	0.2116	0.1781	0.0489	-0.0172	0.0347	0.1053	0.0395	0.0000	0.1599	0.2277	0.1595	0.0078	0.0055	0.0000
AUC	1	0.5744	0.6812	0.5805	0.5645	0.4831	0.5551	0.5754	0.4988	0.5000	0.6091*	0.7656*	0.6052*	0.5138	0.4864	0.4977
	2	0.6334	0.4777	0.5842	0.5284	0.5173	0.5528	0.6965	0.6726	0.7048	0.5958	0.5756	0.5951	0.5232	0.6049	0.4266
	3	0.5821	0.6214	0.5904	0.5227	0.4772	0.5194	0.5623	0.5159	0.5000	0.6173*	0.7449*	0.6576*	0.5356	0.4635	0.5000

Chapter 4

Conclusions and Future Work

4.1. Conclusions

In this work, we analyzed complexity metrics, analytical metrics obtained from the confusion matrix and some basic classifiers.

To recapitulate, the following work was conduced through this dissertation:

1. An introduction to Data complexity metrics, class imbalance and techniques to mitigate its effects, datasets used for the experimentation, and analytical metrics (see Chapter 2, Sections 3.1 and 3.3).
2. Python implementation of the experimentation (connecting to R's ECol to obtain the complexity metrics) - see Section **??**:
   - Obtain the complexity metrics of the datasets.
   - Calculate the analytical metrics of the datasets using K-fold.
   - Compute the analytical metrics of the datasets after applying Under/Oversampling to each fold.
3. Analysis of the obtained results (see Section 3.4.5).

The experiments carried out show that the datasets selected have many *problems* that go with them: *imbalanced classes*, or *data overlapping* are some of them. This could be one of the reasons why the classifiers do not give the expected results.

It has also been tried out a few way to improve the classification in those datasets, such as K-folding Cross Validation combined with Undersampling and Oversampling techniques. The result of those experiments led to the intuition that either the applied techniques are not fit to the datasets (do not manage to increase the performance); or that the classification algorithms selected do not perform as they were expected.

To solve this, better datasets could be selected/generated, trying to reduce the value of metrics such as overlapping, and therefore maximize linearity; or reduce the imbalance of the class (see 4.2 to know more proposals).Other option would be to use other Under/Oversampling techniques than the ones used for this paper.

That means that the nature of the data might affect the results.

4.2. Future Work

There are several paths that can follow this work. First we need to run all experiments with more datasets - trying to have a wider variety of the values obtained in the complexity metrics - and classification algorithms, that could lead to better results than the ones used. Other way would be to apply other imbalance techniques could be applied to the datasets.

The final objective is to try minimize/maximize the aforementioned complexity metrics, which should lead to better classification results.

There are also other paths to follow, for example, analyze the data/quality in the context of feature selection. There is also another path to to do so with interpretable machine learning [55].

Bibliography

[1] I. Corporation, "Removing unfair bias in machine learning," pp. 1–27, 2019.

[2] A. C. Lorena, L. P. Garcia, J. Lehmann, M. C. Souto, and T. K. Ho, "How complex is your classification problem?: A survey on measuring classification complexity," *ACM Computing Surveys*, vol. 52, no. 5, aug 2019. [Online]. Available: https://arxiv.org/abs/1808.03591

[3] A. Lorena, A. Maciel, P. Miranda, I. Costa, and R. Prudêncio, "Data complexity meta-features for regression problems," *Machine Learning*, 12 2017.

[4] T. K. Ho and M. Basu, "Complexity measures of supervised classification problems," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 3, pp. 289–300, March 2002.

[5] M. Basu and T. K. Ho, Eds., *Data Complexity in Pattern Recognition*. Springer London, 2006. [Online]. Available: https://doi.org/10.1007/978-1-84628-172-3

[6] A. C. Lorena, L. P. F. Garcia, J. Lehmann, M. C. P. Souto, and T. K. Ho, "How complex is your classification problem? a survey on measuring classification complexity," 2018.

[7] J. Friedman and L. Rafsky, "Multivariate generalizations of the wald-wolfowitz and smirnov two-sample tests," *The Annals of Statistics*, vol. 7, no. 4, pp. 697–717, 1979.

[8] F. Smith, "Pattern classifier design by linear programming," *IEEE Trans. Computers*, vol. 17, no. 4, pp. 367–372.

[9] J. Zhang and I. Mani, "Knn approach to unbalanced data distributions: A case study involving information extraction," 2003. [Online]. Available: https://www.site.uottawa.ca/~nat/Workshop2003/jzhang.pdf

[10] D. Wilson, "Asymptotic properties of nearest neighbor rules using edited data," *IEEE Transactions on Systems, Man and Cybernetics*, no. 3, pp. 408–421, 1972.

[11] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Intell. Res. (JAIR)*, vol. 16, pp. 321–357, 2002.

[12] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, pp. 123–140, 1996. [Online]. Available: http://dx.doi.org/10.1007/BF00058655

[13] Y. Freund and R. E. Schapire, "Experiments with a new boosting algorithm," in *Thirteenth International Conference on Machine Learning*. San Francisco: Morgan Kaufmann, 1996, pp. 148–156.

[14] N. Chawla, K. Bowyer, L. Hall, and W. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[15] N. Chawla, A. Lazarevic, L. Hall, and K. Bowyer, "Smoteboost: Improving prediction of the minority class in boosting," in *7th European Conference on Principles and Practice of Knowledge Discovery in Databases(PKDD 2003)*, 2003, pp. 107–119.

[16] Y. Freund and R. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[17] C. Seiffert, T. Khoshgoftaar, J. Van Hulse, and A. Napolitano, "RUSBoost: A hybrid approach to alleviating class imbalance," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 40, no. 1, pp. 185–197, 2010.

[18] P. Domingos, "Metacost: a general method for making classifiers cost-sensitive," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '99. New York, NY, USA: ACM, 1999, pp. 155–164. [Online]. Available: http://doi.acm.org/10.1145/312129.312220

[19] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016412120700235X

[20] T. M. Khoshgoftaar, E. Allen, and J. Deng, "Using regression trees to classify fault-prone software modules," *IEEE Transactions on Reliability*, vol. 51, no. 4, pp. 455–462, 2002.

[21] T. M. Khoshgoftaar, E. Allen, J. Hudepohl, and S. Aud, "Application of neural networks to software quality modeling of a very large telecommunications system," *IEEE Transactions on Neural Networks*, vol. 8, no. 4, pp. 902–909, 1997.

[22] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with precision: A response to comments on data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 637–640, 2007.

[23] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen, "Mining software repositories for comprehensible software fault prediction models," *Journal of Systems and Software*, vol. 81, no. 5, pp. 823–839, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121207001902

[24] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE'09)*. New York, NY, USA: ACM, 2009, pp. 1–10.

[25] H. Zhang and X. Zhang, "Comments on "data mining static code attributes to learn defect predictors"," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 635–637, 2007.

[26] C. Seiffert, T. Khoshgoftaar, and J. Van Hulse, "Improving software-quality predictions with data sampling and boosting," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 39, no. 6, pp. 1283–1294, 2009.

[27] T. M. Khoshgoftaar and N. Seliya, "Analogy-based practical classification rules for software quality estimation," *Empirical Software Engineering*, vol. 8, no. 4, pp. 325–350, 2003.

[28] Y. Peng, G. Kou, G. Wang, H. Wang, and F. Ko, "Empirical evaluation of classifiers for software risk management," *International Journal of Information Technology & Decision Making (IJITDM)*, vol. 08, no. 04, pp. 749–767, 2009.

[29] Y. Peng, G. Wang, and H. Wang, "User preferences based software defect detection algorithms selection using MCDM," *Information Sciences*, vol. In Press., pp. –, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0C-4YXK4KM-1/2/0841843c022cfd6b78886eb45bc8ccf0

[30] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, July-Aug. 2008.

[31] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CSMR'10)*, ser. CSMR'10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 107–116. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2010.18

[32] 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 - PROMISE '10*. ACM Press, 2010.

[33] L. Madeyski and J. Hrysko, "Bottlenecks in software defect prediction implementation in industrial projects," vol. 40, no. 1, 2015.

[34] M. Harman, S. Islam, Y. Jia, L. L. Minku, F. Sarro, and K. Srivisut, "Less is more: Temporal fault predictive performance over multiple hadoop releases," in *Search-Based Software Engineering*, C. Le Goues and S. Yoo, Eds. Cham: Springer International Publishing, 2014, pp. 240–246.

[35] "The promise repository of empirical software engineering data," 2015.

[36] Z. Xu, J. Liu, X. Luo, and T. Zhang, "Cross-version defect prediction via hybrid active learning with kernel principal component analysis," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, mar 2018.

[37] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, 2016.

[38] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "HYDRA: Massively compositional model for cross-project defect prediction," *IEEE Transactions on Software Engineering*, vol. 42, no. 10, pp. 977–998, oct 2016.

[39] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, jun 1994.

[40] R. Martin, "OO Design Quality Metrics - An Analysis of Dependencies," in *Proceedings Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*, 1994, pp. 151–170.

[41] J. Bansiya and C. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[42] Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*. ADDISON WESLEY PUB CO INC, 1995. [Online]. Available: https://www.ebook.de/de/product/6141164/henderson_sellers_object_oriented_metrics_measures_of_complexity.html

[43] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics," in *Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403)*. IEEE Comput. Soc.

[44] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, dec 1976.

[45] T. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[46] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Belmont, California: Wadsworth International Group, 1984, https://books.google.es/books/about/Classification_and_Regression_Trees.html?id=JwQx-WOmSyQC&redir_esc=y.

[47] J. Quinlan, *C4.5: Programs for machine learning*. San Mateo, California: Morgan Kaufmann, 1993.

[48] T. Oates and D. Jensen, "The effects of training set size on decision tree complexity," in *Proceedings of the Fourteenth International Conference on Machine Learning (ICML)*. Citeseer, 1997, pp. 254–262.

[49] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[50] "Conformal prediction for reliable machine learning," in *Conformal Prediction for Reliable Machine Learning*, V. N. Balasubramanian, S.-S. Ho, and V. Vovk, Eds. Boston: Morgan Kaufmann, 2014, p. i. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780123985378000146

[51] T. Sørensen, "A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on danish commons," vol. 5(4), pp. 1–34, 1948.

[52] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26(3), pp. 297–302, 1945.

[53] B. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA) - Protein Structure*, vol. 405, no. 2, pp. 442–451, Oct. 1975. [Online]. Available: https://doi.org/10.1016/0005-2795(75)90109-9

[54] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, Jun. 2006. [Online]. Available: https://doi.org/10.1016/j.patrec.2005.10.010

[55] C. Molnar, *Interpretable Machine Learning*, 2019, https://christophm.github.io/interpretable-ml-book/.

Appendix A

Prerequisites

Some relevant code sections have been included. The whole project is publicly available on GitHub: https://github.com/PabloAceG/ComputingProject.

To be able to execute the experiments within the repository, Python3 is needed. Anaconda or the official Python located in the official repositories can be used as long as version 3 or posterior is used. Trying to replicate the experiments on some operative systems might terminate in error. If this is the case, python can be changed (*Linux*) with:

```
    sudo update-alternatives --config python
```

R (programming language) is needed before trying to execute the project. Also, the following packages are mandatory in order to replicate the experiments:

- ECoL - Dataset Complexity Metrics Package.
  - https://github.com/lpfgarcia/ECoL
  - https://cran.r-project.org/web/packages/ECoL/
- Rserve - server, responds requests made to R: https://rforge.net/Rserve/doc.html.

Once the previous requisites are fulfilled, the R server can be started by executing the following commands:

```
library(Rserve) # import the library
run.Rserve() # start the server. Or simply Rserve()
```

Now, it is time to download the project to install the remaining Python packages. The project can be downloaded from https://github.com/PabloAceG/ComputingProject/.

Same as before, some Python packages are mandatory to execute the project. These packages are available in requirements.txt [1] file. To automatically install those packages, run[2]:

```
pip install -r .\code\requirements.txt
```

It might happen that `pip install -r` might not install all packages. To solve this, the failing packages must be installed manually:

```
pip install <package_name>
```

[1] https://github.com/PabloAceG/ComputingProject/blob/master/code/requirements.txt
[2] All commands are executed from the parent folder of the repository.

Now, the experiments should be replicable. The experiment's code is under the folder https://github.com/PabloAceG/ComputingProject/tree/master/code To run them, execute:

```
python code/metrics_comparison.py
python code/metrics_kfold.py
python code/metrics_kfold_undersampling.py
python code/metrics_kfold_oversampling.py
```

Each of the previous commands execute one experiment.

As final remarks, the class `r_connect.py` [3] is the client connection the server in R (Rserve). It makes the requests to the `ECoL` package to obtain the complexity metrics.

The class `data.py` [4] standardizes the datasets input (parsing data) and some other metrics from the package sklearn [5].

[3] https://github.com/PabloAceG/ComputingProject/blob/master/code/r_connect.py
[4] https://github.com/PabloAceG/ComputingProject/blob/master/code/data.py
[5] https://scikit-learn.org/stable/index.html

Appendix B

Python Relevant Code

B.1. Rserve Python Client

The class `r_connect.py` is a client for R package Rserve. Makes requests to ECoL R package and parses data.

The import statements have been made redundant in this snippet and the rest of the snippets in this appendix.

```
class r_connect:

    __metrics = None

    def __init__ (self):
        self.__connection = self.__connect()

    def __connect (self):
        return pyRserve.connect()

    def get_metrics (self, X=None, Y=None):
        if X is None and Y is None :
            if self.__metrics is not None:
                return self.__metrics
            else :
                # No data and no parameters: finish execution
                error_message = '...'
                raise Exception(error_message)
                sys.exit (400)
        else :
            # Stores connection to R's RPC.
            connect = self.__connection

            # Sends the input matrix and the output vector to R.
            connect.r.X = X
            connect.r.y = Y

            # Library to use in R.
            connect.r('df_X <- as.data.frame(X)')
            connect.r('df_y <- as.data.frame(y)')
            connect.r('library("ECoL")')

            ## Metrics, uses a dictionary to provide a faster access to its
```

```
            # contents.
            metrics = {}

            # Balance: C1, C2
            balance = self.safe_connect('balance(df_X, df_y)')
            balance_dic_entry = { 'balance' :  balance }
            metrics.update (balance_dic_entry)

            # Correlation: C1, C2, C3, C4
            correlation = self.safe_connect('correlation(df_X, df_y, summary=c("mean"))')
            correlation_dic_entry = { 'correlation' : correlation }
            metrics.update (correlation_dic_entry)

            # Dimensionality: T2, T3, T4
            dimensionality = self.safe_connect('dimensionality(df_X, df_y, summary=c("mean"))'
    )
            dimensionality_dic_entry = { 'dimensionality' : dimensionality }
            metrics.update (dimensionality_dic_entry)

            # Linearity: L1, L2, L3
            linearity = self.safe_connect('linearity(df_X, df_y, summary=c("mean"))')
            linearity_dic_entry = { 'linearity' : linearity }
            metrics.update (linearity_dic_entry)

            # Neighborhood: N1, N2, N3, N4, T1, LSC
            neighborhood = self.safe_connect('neighborhood(df_X, df_y, summary=c("mean"))')
            neighborhood_dic_entry = { 'neighborhood' : neighborhood }
            metrics.update (neighborhood_dic_entry)

            # Network: Density, ClsCoef, Hubs
            network = self.safe_connect('network(df_X, df_y, summary=c("mean"))')
            network_dic_entry = { 'network' : network }
            metrics.update (network_dic_entry)

            # Overlap: F1, F1v, F2, F3, F4
            overlap = self.safe_connect('overlapping(df_X, df_y, summary=c("mean"))')
            overlap_dic_entry = { 'overlap' : overlap }
            metrics.update (overlap_dic_entry)

            # Smoothness: S1, S2, S3, S4
            smoothness = self.safe_connect('smoothness(df_X, df_y, summary=c("mean"))')
            smoothness_dic_entry = { 'smoothness' : smoothness }
            metrics.update (smoothness_dic_entry)

            self.__metrics = metrics

            return metrics

    def print_metrics (self, metrics=None) :
        print ('\n\n=== Printing metrics ===', end='\n\n')

        ...
        ...

    def get_print_metrics(self, X, Y):
        self.get_metrics (X, Y)
        self.print_metrics (self.__metrics)

        return self.__metrics
```

```
92
93      def safe_connect(self, operation) :
94          connection = self.__connection
95
96          return connection.r(operation)
```

Listing B.1: Connection code to requests R ECoL functions

B.2. Datasets and Operations on Data

The following code contains the logic to read and parse datasets requested through the function `getDataset(...)`. The datasets can be shuffled if specified through parameter, but in this situation no experiments used that option (as the results should be replicated, the input that should always be the same).

The class also uses *sklearn* library to calculate some metrics. The functions in this code simply parse the results so that they are easier to read afterwards (no need to have more than 4 digits of precision in float numbers).

Not all the code has been copied, as some parts repeat.

```
1  ...
2  ...
3
4  def __load_arff(path):
5      '''
6          Loads the dataset content of an .arff file into the workspace.
7          Input:
8              - path: Location of the file.
9          Output:
10             - dataset: Data from the file.
11     '''
12
13     dataset = []
14
15     with open(path, 'r') as data:
16         dataset = arff.load(data)['data']
17
18     return dataset
19
20 def __load_csv(path):
21     '''
22         Loads the data content of an .csv file into the workspace.
23         Input:
24             - path: Location of the file.
25         Output:
26             - dataset: Data from the file.
27     '''
28
29     dataset = []
30
31     with open (path, 'r') as csv_file:
32         dataset = pd.read_csv(
33             csv_file,
34             sep=','
35         )
36
```

```
37      return dataset
38
39  def __parte_dataset(dataset, input_select, target_select):
40      '''
41          Takes a dataset and parses it to only take what is necessary.
42          Inputs:
43              - dataset: Input raw data as a Pandas Data Framework.
44              - start: Where columns start to be useful.
45          Output:
46              - input: Input arrays of the dataset.
47              - target: Target column of the dataset.
48      '''
49
50      num_rows    = len(dataset)
51      last_column = target_select if (target_select < 0) else None
52
53      # Input
54      input_columns = dataset.columns[input_select:last_column]
55      input = dataset[input_columns].head(num_rows).astype(float).to_numpy()
56
57      # Target
58      target_column = dataset.columns[target_select]
59      target = dataset[target_column].head(num_rows)
60
61      # Parse output
62      sample = target[0]
63      if not isinstance(sample, str):
64          target = numpy.where(target > 0, 1, 0)
65      else:
66          has_faults = target == 'yes'
67          target = numpy.where(has_faults , 1, 0)
68
69      return input, target
70
71  def __data_preparation(path, input_select, target_select, type='arff', shuffle=False):
72      '''
73          Loads an .arff/.csv file and parses its content to input/output valid
74          for a Machine Learning application.
75          Input:
76              - path: Location of the file.
77              - start: Where columns start to be useful (string columns are not
78                  necessary).
79              - type: It can be:
80                  - arff
81                  - csv
82          Output:
83              - input: Input arrays of the dataset.
84              - target: Target column of the dataset.
85      '''
86
87      # Load data
88      dataset = []
89      if   type == 'arff':
90          dataset = __load_arff(path)
91      elif type == 'csv':
92          dataset = __load_csv(path)
93      else:
94          raise Exception('Not a valid file type. Try with arff or csv!')
95          sys.exit(404)
```

```
    dataset = pd.DataFrame(dataset)

    if shuffle:
        dataset = dataset.sample(frac=1)

    # Parse data
    data = (dataset, input_select, target_select)
    input, target = __parte_dataset(*data)

    return input, target

def __data_preparation_iris_dataset(path):
    # Load data
    dataset = __load_csv(path)
    dataset = pd.DataFrame(dataset)

    # Parse data
    start    = 0
    last     = -1
    num_rows = len(dataset)

    # Input
    input_columns = dataset.columns[start : last]
    input = dataset[input_columns].head(num_rows).astype(float).to_numpy()

    # Target
    target_column = dataset.columns[last]
    target = dataset[target_column].head(num_rows)


    # Parse target
    values = {'setosa': 1, 'versicolor': 2, 'virginica': 3}
    target = numpy.array( [
        values[type] for type in target
    ] )

    return input, target

def get_dataset(name, shuffle=False):
    '''
        Retrieves the data of a given dataset, separated into input information
        and target output - .arff or .csv files only.
        Input:
            - name: Dataset name.
                - ant
                - apache
                - camel
                - iris
                - ivy
                - jedit
                - log4j
                - poi
                - synapse
                - xalan
                - xerces
        Output:
            - input: Input arrays of the dataset.
            - target: Target column of the dataset.
    '''
```

```
155
156     path       = ''     # Location of file
157     start      = 0      # From which columns to use
158     output_col = -2     # Position of output column
159     type       = 'arff' # Type of file to be read. Default .arff
160
161     if name == 'ant':
162         # Retrieve data
163         path = './dataset/ant-1.7.arff'
164         start = 3
165
166     elif name == 'apache':
167         # Retrieve data
168         path = './dataset/Apache.csv'
169         start = 3
170         output_col = -1
171         type = 'csv'
172
173     elif name == 'camel':
174         # Retrieve data
175         path = './dataset/camel-1.6.arff'
176         start = 3
177
178     elif name == 'iris': # Special case. Non-binary output. Needs different
179                          # treatment.
180         # Retrieve data
181         path = './dataset/iris.csv'
182         input, target = __data_preparation_iris_dataset(path, shuffle)
183
184         return input, target
185
186     elif name == 'ivy':
187         # Retrieve data
188         path = './dataset/ivy-2.0.arff'
189         start = 3
190
191     ...
192     ...
193     elif name == 'xerces':
194         # Retrieve data
195         path = './dataset/xerces-1.4.arff'
196         start = 3
197
198     elif name == 'hadoop-1':
199         # Retrieve data
200         path = './dataset/hadoop-proc-0.1.csv'
201         start = 2
202         output_col = 1
203         type = 'csv'
204
205     ...
206     ...
207     elif name == 'hadoop-8':
208         # Retrieve data
209         path = './dataset/hadoop-proc-0.8.csv'
210         start = 2
211         output_col = 1
212         type = 'csv'
213
```

```
214
215     else:
216         raise Exception('The given dataset name is not valid.')
217         sys.exit(404)
218
219     # Return results
220     input, target = __data_preparation(path, start, output_col, type, shuffle)
221
222     return input, target
223
224 def confusion_matrix(x_test: list, y_test: list, classifier):
225     '''
226         Obtains the confusion matrix for a given testing dataset, with binary
227         output.
228         Input:
229             - x_test: paremeters for testing dataset.
230             - y_test: target/desired output for testing dataset.
231             - classifier: trained classifier.
232         Output:
233             (
234                 true_positive: successfully predicted positives
235                 true_negative: successfully predicted negatives
236                 false_positive: unsuccessfully predicted positives
237                 false_negative: unsuccessfully predicted positives
238             )
239     '''
240     # Confusion Matrix Cells
241     true_positive:float  = 0
242     true_negative:float  = 0
243     false_positive:float = 0
244     false_negative:float = 0
245
246     # Calculate number of repetitions of each classification.
247     for (input, target) in zip(x_test, y_test):
248         prediction:int = classifier.predict([input])[0]
249
250         if prediction == target:    # Success
251             if prediction: true_positive += 1
252             else:          true_negative += 1
253         else:                       # Wrong
254             if prediction: false_positive += 1
255             else:          false_negative += 1
256
257     # Transform absolute to relative values
258     num_samples = len(y_test)
259     true_positive  = true_positive  / num_samples
260     true_negative  = true_negative  / num_samples
261     false_positive = false_positive / num_samples
262     false_negative = false_negative / num_samples
263
264     return (true_positive, true_negative, false_positive, false_negative)
265
266 def recall(targets, predictions) -> float:
267     '''
268         Sensitivity, recall, hit rate or True Positive Rate (RPR)
269         https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html
270                TP       TP
271         TPR = ---- = --------- = 1 - FNR
272                P      TP + FN
```

```
    '''
    return round(metrics.recall_score(targets, predictions), 4)

def fall_out(targets, predictions) -> float:
    '''
        Fallout, or False Positive Rate (FPR)
               FP       FP
        FPR = ---- = --------- = 1 - TNR
               N      FP + TN
    '''
    # Calculate TN and FP rates
    num_items:int = len(targets)
    false_positive:float = 0
    true_negative: float = 0
    # Count
    for (t, o) in zip(targets, predictions):
        if (o == t and o == 0):
            true_negative  += 1
        if (o != t and o == 1):
            false_positive += 1
    try:
        # Rates
        true_negative  /= num_items
        false_positive /= num_items

        # False Positive Rate
        fdr:float = false_positive / (false_positive + true_negative)
        return round(fdr, 4)
    except:
        return 0

def precision(targets, predictions) -> float:
    '''
        Precision or positive predictive value (PPV).
        https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html
                  TP
        PPV = --------- = 1 - FDR
               TP + FP
    '''
    return round(metrics.precision_score(targets, predictions), 4)

def balanced(targets, predictions) -> float:
    '''
        Balanced Accuracy (BA)
        https://scikit-learn.org/stable/modules/generated/sklearn.metrics.
    balanced_accuracy_score.html#sklearn.metrics.balanced_accuracy_score
              TPR + TNR
        BA = -----------
                  2
    '''

    return round(metrics.balanced_accuracy_score(targets, predictions), 4)

def f1(targets, predictions) -> float:
    '''
        F1 Score. Is the harmonic mean of precision and sensitivity.
        https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html
                PPV x TPR         2 TP
        F1 = 2 ----------- = ----------------
```

```
                PPV + TPR     2 TP + FP + FN
    '''
    return round(metrics.f1_score(targets, predictions), 4)

def mcc(targets, predictions) -> float:
    '''
        Matthews Correlation Coefficient (MCC).
        https://scikit-learn.org/stable/modules/generated/sklearn.metrics.matthews_corrcoef.
    html
                          TP x TN - FP x FN
        MCC = -------------------------------------------
              sqrt((TP + FP)(TP + FN)(TN + FP)(TN + FN))
    '''
    return round(metrics.matthews_corrcoef(targets, predictions), 4)

def auc(targets, predictions) -> float:
    '''
        Area Under Receiver Operating Characteristic Curve,
        Area Under ROC Curve or AUC.
        https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html
        https://scikit-learn.org/stable/modules/generated/sklearn.metrics.auc.html
              1 + TPR - FPR
        AUC = ---------------
                    2
    '''
    fpr, tpr, thresholds = metrics.roc_curve(targets, predictions)

    return round(metrics.auc(fpr, tpr), 4)

def store_results(filename:str, metrics:list):
    with open('./code/results/' + filename + '.csv', mode='a', newline='') as csvfile:
        writer = csv.writer(csvfile, delimiter=',', quoting=csv.QUOTE_MINIMAL)
        writer.writerow(metrics)

def calculate_results(targets:list, predictions:list) -> list:
    # Metrics
    ppv:float = precision(targets, predictions)
    tpr:float = recall   (targets, predictions)
    fpr:float = fall_out (targets, predictions)
    ba :float = balanced (targets, predictions)
    fm :float = f1       (targets, predictions)
    m  :float = mcc      (targets, predictions)
    a  :float = auc      (targets, predictions)
    metrics:list  = [ppv, tpr, fpr, ba, fm, m, a]

    return metrics

def train_predict(clf, input_train, target_train, test):
    # Train
    clf.fit(input_train, target_train)
    # Test
    return clf.predict(test)
```

Listing B.2: Load datasets, calculate metrics, export results to CSV files, etc.

B.3. Experiment 1. Complexity Metrics Comparison

The objective is to see if there is any dependency between some complexity metrics (balance, dimensionality, overlapping, etc.).

This programs obtains the complexity metrics (more in section 2.1) from different datasets, and stores the results on a CSV file for their comparison. The dataset is loaded, metrics are retrieved for every dataset and those metrics are plotted and stored.

```
...
...
from r_connect import r_connect
import data as DATASETS

def add_metrics(storage, dataset_metrics):
    '''
        Includes a new set of calculated metrics into a variable.
        It creates an array for each metric's measure.
        Input:
            - storage(dict) : Data structure to store the list of measures from
                the metrics of datasets.
            - dataset_metrics (dict): Metrics computed from a dataset.
        Output:
            - storage (dict): Metrics added to the already stored ones.
    '''
    # Iterate through the metrics
    for m in storage:
        # Metric might not have been calculated for dataset
        if m in dataset_metrics:
            metric = dataset_metrics[m]

            # Iterate through the measures of the metric
            for measure in storage[m]:
                try:
                    # Measure might not have been calculated for dataset
                    if measure in storage[m]:
                        storage[m][measure].append(float(metric[measure]))
                except Exception:
                    pass
        else:
            print('Something went wrong')

    return storage

def plot_metrics_comparison(metrics):
    '''
        Input:
            - metrics:
        Output:
    '''
    i = 0
    j = 0
    for metric in metrics:
        rows = 2
        cols = int(math.ceil(len(metrics[metric]) / rows))
        fig, axs = plt.subplots(rows, cols)
        for measure in metrics[metric]:
            DATASETS.store_results('metrics-hadoop', [metric, measure] + [round(i, 4) for i in
     metrics[metric][measure]])
```

```
            if cols == 1:
                if metrics[metric][measure]:
                    axs[i].bar(datasets, metrics[metric][measure])

                else:
                    axs[i].text(0.5, 0.5, 'No data')

                title = metric + ': ' + measure
                axs[i].set_title(title)
            else:
                if metrics[metric][measure]:
                    axs[i, j].bar(datasets, metrics[metric][measure])

                else:
                    axs[i, j].text(0.5, 0.5, 'No data')

                title = metric + ': ' + measure
                axs[i, j].set_title(title)

            if j == cols - 1:
                i += 1
                j =  0
            else:
                j += 1

        i = 0
        j = 0
        plt.show()


if __name__ == '__main__':
    '''
        Main code - to be executed
    '''

    # Datasets to analyze
    '''
    datasets = [
        'ant',
        'apache',
        'camel',
        'ivy',
        'jedit',
        'log4j',
        'synapse',
        'xalan',
        'xerces'
    ]
    '''
    datasets = [
        'hadoop-1',
        'hadoop-2',
        'hadoop-3',
        'hadoop-4',
        'hadoop-5',
        'hadoop-6',
        'hadoop-7',
        'hadoop-8',
    ]
```

```
109
110
111     # Store metrics computed from datasets (datatype: dict)
112     results = {
113         'balance': {
114             'C1': [],
115             'C2': []
116         },
117         'correlation': {
118             'C1': [],
119             'C2': [],
120             'C3': [],
121             'C4': []
122         },
123         'dimensionality': {
124             'T1': [],
125             'T2': [],
126             'T3': []
127         },
128         'linearity': {
129             'L1': [],
130             'L2': [],
131             'L3': []
132         },
133         'neighborhood': {
134             'N1':  [],
135             'N2':  [],
136             'N3':  [],
137             'N4':  [],
138             'T1':  [],
139             'LSC': [],
140         },
141         'network': {
142             'Density': [],
143             'ClsCoef': [],
144             'Hubs':    []
145         },
146         'overlap': {
147             'F1':  [],
148             'F1v': [],
149             'F2':  [],
150             'F3':  [],
151             'F4':  []
152         },
153         'smoothness': {
154             'S1': [],
155             'S2': [],
156             'S3': [],
157             'S4': []
158         }
159     }
160
161     # Connect to R
162     connector = r_connect()
163
164     # Get Metrics
165     for set in datasets:
166         print(set)
167         inputs, targets = DATASETS.get_dataset(set)
```

```
168         metrics = connector.get_metrics(inputs, targets)
169
170         results = add_metrics(results, metrics)
171
172     # Print Metrics
173     print(results)
174     plot_metrics_comparison(results)
```

Listing B.3: Compare complexity metrics of different datasets

B.4. Experiment 2. Compare Metrics on K-fold Cross Validation

The function calls to the *data* (referenced as *DATASETS* in the code) object are unfolded in the snippet B.2.

The objective is to look for a relation between confusion matrix metrics (recall, fallout, etc.) between the folds generated out of a K-fold Cross Validation. See if there is a certain linearity between folds and what might cause certain behaviors.

In order do create this experiment, a dataset is loaded, and then it is folded into k-equally-sized parts. For each part, three different classification algorithms are trained and tested to compare the aforementioned metrics (more about the metrics in 3.3). This same process is repeated for every dataset.

```
1  ...
2  ...
3  import data as DATASETS
4
5  if __name__ == '__main__':
6      # Data
7      datasets = [
8          'ant',
9          'apache',
10         'camel',
11         'ivy',
12         'jedit',
13         'log4j',
14         'poi',
15         'synapse',
16         'xalan',
17         'xerces',
18         'hadoop-1',
19         'hadoop-2',
20         'hadoop-3',
21         'hadoop-4',
22         'hadoop-5',
23         'hadoop-6',
24         'hadoop-7',
25         'hadoop-8'
26     ]
27
28     for d in datasets:
29         print('----------' + d + '----------')
30         inputs, target = DATASETS.get_dataset(d)
31
32         # K-fold Parameters
```

```
        k = 5
        kf = KFold(n_splits=k)
        splits = kf.split(inputs)
        #                   Precision Recall Fallout Balanced F1 MCC AUC
        mean_naive = np.array([0,       0,      0,      0,    0,  0,  0])
        mean_tree  = np.array([0,       0,      0,      0,    0,  0,  0])
        mean_knn   = np.array([0,       0,      0,      0,    0,  0,  0])

        # Get measurements using K-fold
        for train_index, test_index in splits:
            # Data partition
            x_train, x_test = inputs[train_index], inputs[test_index]
            y_train, y_test = target[train_index], target[test_index]

            filename = d + '-k' + str(k)

            # Train NN
            print('---> Naive Bayes')
            clf   = GaussianNB()
            predictions   = DATASETS.train_predict(clf, x_train, y_train, x_test)
            naive_metrics = DATASETS.calculate_results(y_test, predictions)
            mean_naive = np.add(mean_naive, naive_metrics)
            naive_metrics = ['Naive Bayes'] + naive_metrics
            DATASETS.store_results(filename, naive_metrics)

            print('---> Decision Tree')
            clf   = DecisionTreeClassifier()
            predictions  = DATASETS.train_predict(clf, x_train, y_train, x_test)
            tree_metrics = DATASETS.calculate_results(y_test, predictions)
            mean_tree = np.add(mean_tree, tree_metrics)
            tree_metrics = ['Decision Tree'] + tree_metrics
            DATASETS.store_results(filename, tree_metrics)

            print('---> Nearest Centroid')
            clf   = NearestCentroid()
            predictions = DATASETS.train_predict(clf, x_train, y_train, x_test)
            knn_metrics = DATASETS.calculate_results(y_test, predictions)
            mean_knn = np.add(mean_knn, knn_metrics)
            knn_metrics = ['Nearest Centroid'] + knn_metrics
            DATASETS.store_results(filename, knn_metrics)

            print('------------------------------')

        # K-fold Mean
        mean_naive = [round(i, 4) for i in (mean_naive / k)]
        DATASETS.store_results(filename, ['Naive Bayes Mean'] + mean_naive)
        mean_tree = [round(i, 4) for i in (mean_tree / k)]
        DATASETS.store_results(filename, ['Decision Tree Mean'] + mean_tree)
        mean_knn = [round(i, 4) for i in (mean_knn / k)]
        DATASETS.store_results(filename, ['Nearest Centroid Mean'] + mean_knn)

        print([mean_naive, mean_tree, mean_knn])
```

Listing B.4: Calculate Metrics from Confusion Matrix to compare in-fold results

B.5. Experiment 3. Compare Metrics with Under-sampling and K-fold Cross Validation

The function calls to the *data* (referenced as *DATASETS* in the code) object, can be observed in the snippet B.2. Also, the experiment is the same as in B.4, although an under-sampling filter is applied to each fold of the dataset before using any classification algorithm.

This experiment has to be executed as many times as datasets want to be analyzed, as a bulk execution of all the datasets would probably end in error. The reason for this is that if the sampling strategy is too low, then there should not be enough data to properly train the classification. Also, it depends on the amount of samples of the minority class, that is why not a low enough value can be set for all datasets, and a single-dataset execution must be performed.

The values commented at the right of each dataset indicate the recommended sampling strategy for that dataset.

```
...
...
import data as DATASETS

if __name__ == '__main__':
    # Data
    # In this case a loop cannot be used to make all experiments, as the value
    # of the undersampling may vary on each dataset.
    dataset ='ant'          # 50
    #dataset = 'apache'     # 50
    #dataset = 'camel'      # 50
    #dataset = 'ivy'        # 30
    #dataset = 'jedit'      # 7
    #dataset = 'log4j'      # 11
    #dataset = 'synapse'    # 50
    #dataset = 'xalan'      # 6
    #dataset = 'xerces'     # 50
    #dataset = 'hadoop-1'   # 38
    #dataset = 'hadoop-2'   # 31
    #dataset = 'hadoop-3'   # 35
    #dataset = 'hadoop-4'   # 32
    #dataset = 'hadoop-5'   # 26
    #dataset = 'hadoop-6'   # 22
    #dataset = 'hadoop-7'   # 34
    #dataset = 'hadoop-8'   # 10

    inputs, target = DATASETS.get_dataset(dataset)

    # K-fold Parameters
    k = 5
    kf = KFold(n_splits=k, shuffle=True, random_state=1)
    splits = kf.split(inputs)
    #                   Precision  Recall Fallout Balanced F1 MCC AUC
    mean_naive = np.array([0,       0,      0,      0,    0,  0,  0])
    mean_tree  = np.array([0,       0,      0,      0,    0,  0,  0])
    mean_knn   = np.array([0,       0,      0,      0,    0,  0,  0])

    RANDOM_STATE = 42

    # Get measurements using K-fold
    for train_index, test_index in splits:
```

```
        # Data partition
        x_train, x_test = inputs[train_index], inputs[test_index]
        y_train, y_test = target[train_index], target[test_index]

        X, Y = make_imbalance(
            x_train, y_train,
            sampling_strategy={0:50, 1:50},
            random_state=RANDOM_STATE)

        filename = dataset + '-k' + str(k) + '-under'

        # Train NN
        print('---> Naive Bayes')
        clf   = GaussianNB()
        pipeline = make_pipeline(
            NearMiss(version=2),
            clf)
        predictions   = DATASETS.train_predict(pipeline, X, Y, x_test)
        naive_metrics = DATASETS.calculate_results(y_test, predictions)
        mean_naive = np.add(mean_naive, naive_metrics)
        naive_metrics = ['Naive Bayes'] + naive_metrics
        DATASETS.store_results(filename, naive_metrics)

        print('---> Decision Tree')
        clf   = DecisionTreeClassifier()
        pipeline = make_pipeline(
            NearMiss(version=2),
            clf)
        predictions  = DATASETS.train_predict(pipeline, X, Y, x_test)
        tree_metrics = DATASETS.calculate_results(y_test, predictions)
        mean_tree = np.add(mean_tree, tree_metrics)
        tree_metrics = ['Decision Tree'] + tree_metrics
        DATASETS.store_results(filename, tree_metrics)

        print('---> Nearest Centroid')
        clf   = NearestCentroid()
        pipeline = make_pipeline(
            NearMiss(version=2),
            clf)
        predictions = DATASETS.train_predict(pipeline, X, Y, x_test)
        knn_metrics = DATASETS.calculate_results(y_test, predictions)
        mean_knn = np.add(mean_knn, knn_metrics)
        knn_metrics = ['Nearest Centroid'] + knn_metrics
        DATASETS.store_results(filename, knn_metrics)

        print('------------------------------')

    # K-fold Mean
    mean_naive = [round(i, 4) for i in (mean_naive / k)]
    DATASETS.store_results(filename, ['Naive Bayes Mean'] + mean_naive)
    mean_tree = [round(i, 4) for i in (mean_tree / k)]
    DATASETS.store_results(filename, ['Decision Tree Mean'] + mean_tree)
    mean_knn = [round(i, 4) for i in (mean_knn / k)]
    DATASETS.store_results(filename, ['Nearest Centroid Mean'] + mean_knn)

    print([mean_naive, mean_tree, mean_knn])
```

Listing B.5: Calculate Metrics from Confusion Matrix to compare in-fold results after applying under-sampling filter

B.6. Experiment 4. Compare Metrics with Over-sampling and K-fold Cross Validation

The function uses *data* (referenced as *DATASETS* in the code) object - which is unfolded in the snippet B.2. Also, the experiment is the same as in B.4 and B.5, but in this situation an over-sampling filter is applied to each fold of the dataset before starting the classification process.

Once again, a bulk execution with all the datasets is performed - unlike in Section B.5 there is minimum number to meet, so all the datasets can be treated generically.

The oversampling algorithm selected for this experiment has been SMOTE (see Section 2.3.1.2.2 for more information). Also, the same three classifiers used for the previous experiments are the ones used to compare results: (1) Naive Bayes - Gaussian; (2) Decission Tree; and (3) kNN Nearest Centroid.

```
...
...
import data as DATASETS

if __name__ == '__main__':
    # Data
    datasets = [
        'ant',
        'apache',
        'camel',
        'ivy',
        'jedit',
        'log4j',
        'poi',
        'synapse',
        'xalan',
        'xerces',
        'hadoop-1',
        'hadoop-2',
        'hadoop-3',
        'hadoop-4',
        'hadoop-5',
        'hadoop-6',
        'hadoop-7',
        'hadoop-8'
    ]

    for d in datasets:
        print('----------' + d + '----------')
        inputs, target = DATASETS.get_dataset(d)

        # K-fold Parameters
        k = 5
        kf = KFold(n_splits=k)
        splits = kf.split(inputs)
        #                   Precision Recall Fallout Balanced F1 MCC AUC
        mean_naive = np.array([0,       0,      0,      0,    0,  0,  0])
        mean_tree  = np.array([0,       0,      0,      0,    0,  0,  0])
        mean_knn   = np.array([0,       0,      0,      0,    0,  0,  0])

        # Get measurements using K-fold
        for train_index, test_index in splits:
            # Data partition
            x_train, x_test = inputs[train_index], inputs[test_index]
```

```
            y_train, y_test = target[train_index], target[test_index]

            filename = d + '-k' + str(k) + '-over'

            # Train NN
            # Pipeline to NN
            print('---> Naive Bayes')
            clf = make_pipeline(
                SMOTE(),
                GaussianNB())
            predictions   = DATASETS.train_predict(clf, x_train, y_train, x_test)
            naive_metrics = DATASETS.calculate_results(y_test, predictions)
            mean_naive = np.add(mean_naive, naive_metrics)
            naive_metrics = ['Naive Bayes'] + naive_metrics
            DATASETS.store_results(filename, naive_metrics)

            print('---> Decision Tree')
            clf = make_pipeline(
                SMOTE(),
                DecisionTreeClassifier())
            predictions   = DATASETS.train_predict(clf, x_train, y_train, x_test)
            tree_metrics = DATASETS.calculate_results(y_test, predictions)
            mean_tree = np.add(mean_tree, tree_metrics)
            tree_metrics = ['Decision Tree'] + tree_metrics
            DATASETS.store_results(filename, tree_metrics)

            print('---> Nearest Centroid')
            clf = make_pipeline(
                SMOTE(),
                NearestCentroid())
            predictions   = DATASETS.train_predict(clf, x_train, y_train, x_test)
            knn_metrics = DATASETS.calculate_results(y_test, predictions)
            mean_knn = np.add(mean_knn, knn_metrics)
            knn_metrics = ['Nearest Centroid'] + knn_metrics
            DATASETS.store_results(filename, knn_metrics)

            print('------------------------------')

        # K-fold Mean
        mean_naive = [round(i, 4) for i in (mean_naive / k)]
        DATASETS.store_results(filename, ['Naive Bayes Mean'] + mean_naive)
        mean_tree = [round(i, 4) for i in (mean_tree / k)]
        DATASETS.store_results(filename, ['Decision Tree Mean'] + mean_tree)
        mean_knn = [round(i, 4) for i in (mean_knn / k)]
        DATASETS.store_results(filename, ['Nearest Centroid Mean'] + mean_knn)

        print([mean_naive, mean_tree, mean_knn])
```

Listing B.6: Calculate Metrics from Confusion Matrix to compare in-fold results after applying over-sampling filter

Appendix C

Tables

C.1. Complexity Metrics OO datasets

Table C.1: Complexity Metrics Analysis

Measure	*Metric*	*ant*	*apache*	*camel*	*ivy*	*jedit*	*log4j*	*synapse*	*xalan*	*xerces*
Balance	*C1*	0.7653	0.9895	0.7114	0.5108	0.1545	0.3953	0.9209	0.0944	0.8219
	C2	0.4701	0.0286	0.5429	0.7477	0.9543	0.8319	0.1944	0.9755	0.3826
Correlation	*C2*	0.2622	0.0891	0.1227	0.1879	0.0448	0.0804	0.2310	0.0663	0.2839
	C3	0.6338	0.7372	0.5763	0.5588	0.5324	0.5780	0.7762	0.5275	0.6338
	C4	0.7409	1.0000	1.0000	0.3722	0.0285	0.4341	0.9727	0.0000	0.8878
Dimensionality	*T2*	0.0268	0.0524	0.0207	0.0568	0.0407	0.0976	0.0781	0.0220	0.0340
	T3	0.0027	0.0105	0.3000	0.0057	0.0020	0.0098	0.0078	0.0022	0.0034
Linearity	*L1*	0.2477	0.3937	0.2774	0.1570	0.0475	0.1426	0.3139	0.0285	0.3050
	L2	0.1212	0.1917	0.1370	0.0726	0.0200	0.0624	0.1534	0.0117	0.1342
	L3	0.1103	0.1828	0.1304	0.0661	0.0180	0.0593	0.1421	0.0110	0.1211
Neighborhood	*N1*	0.3208	0.3822	0.3565	0.2159	0.0447	0.2195	0.4023	0.0396	0.2789
	N2	0.3588	0.3232	0.3629	0.3285	0.2543	0.3588	0.3780	0.1615	0.2667
	N3	0.2067	0.1937	0.2435	0.1364	0.0325	0.1220	0.2422	0.0121	0.1173
	N4	0.0913	0.1728	0.1358	0.0739	0.0122	0.0293	0.0859	0.0308	0.0833
	T1	0.0025	0.0135	0.0021	0.0082	0.0184	0.0161	0.0072	0.0294	0.0057
	LSC	0.9860	0.9659	0.9923	0.9533	0.8870	0.9476	0.9797	0.8329	0.9601
Network	*Density*	0.8658	0.8801	0.8627	0.8293	0.8170	0.8260	0.8805	0.8165	0.8530
	ClsCoef	0.3377	0.2232	0.3131	0.3564	0.3314	0.4158	0.3143	0.3460	0.3429
	Hubs	0.7665	0.9236	0.7608	0.7628	0.8789	0.8487	0.9300	0.8564	0.8499
Overlap	*F1*	0.9192	0.9776	0.9825	0.9432	0.9874	0.9922	0.9438	0.9973	0.9496
	F1v	0.2882	0.4659	0.5203	0.2083	0.2048	0.3223	0.3313	0.3274	0.3118
	F2	0.0000	0.0061	0.0004	0.0005	0.0000	0.0000	0.0065	0.0000	0.0000
	F3	0.9074	0.9476	0.9513	0.7102	0.5915	0.6585	0.9141	0.3399	0.8129
	F4	0.8295	0.8325	0.8487	0.3977	0.0671	0.1659	0.7227	0.0055	0.6990
Smoothness	*S1*	0.2191	0.2474	0.2541	0.1453	0.0285	0.1520	0.2824	0.0319	0.2010
	S2	0.1410	0.1178	0.1389	0.1460	0.1417	0.1556	0.1638	0.1538	0.1479
	S3	0.2067	0.1937	0.2466	0.1364	0.0325	0.1220	0.2422	0.0121	0.1122

Continued on next page

Table C.1 – *Continued from previous page*

Measure	*Metric*	*ant*	*apache*	*camel*	*ivy*	*jedit*	*log4j*	*synapse*	*xalan*	*xerces*
	S4	0.0847	0.1273	0.1608	0.0716	0.0215	0.0285	0.1300	0.0209	0.1056

C.2. Complexity Metrics Hadoop datasets

Table C.2: Complexity Metrics Analysis on Hadoop datasets

Measure	*Metric*	*Hadoop 0.1*	*H 0.2*	*H 0.3*	*H 0.4*	*H 0.5*	*H 0.6*	*H 0.7*	*H 0.8*
Balance	*C1*	0.9381	0.7600	0.8132	0.7395	0.6589	0.5642	0.7056	0.3534
	C2	0.1559	0.4777	0.3970	0.5062	0.6056	0.7015	0.5501	0.8579
Correlation	*C2*	0.2868	0.0520	0.3171	0.1899	0.1601	0.1705	0.1997	0.1241
	C3	0.7741	0.6006	0.6330	0.5956	0.5589	0.5397	0.5737	0.4917
	C4	1.0000	1.0000	0.9810	1.0000	1.0000	0.7692	0.9720	0.2125
Dimensionality	*T2*	0.0496	0.0366	0.0332	0.0348	0.0322	0.0299	0.0280	0.0292
	T3	0.0142	0.0052	0.0095	0.0050	0.0046	0.0042	0.0040	0.0042
Linearity	*L1*	0.3770	0.3327	0.2658	0.2950	0.2620	0.1996	0.2654	0.1134
	L2	0.1870	0.1664	0.1305	0.1473	0.1310	0.0981	0.1327	0.0550
	L3	0.1832	0.1644	0.1055	0.1408	0.1274	0.0875	0.1254	0.0489
Neighborhood	*N1*	0.4965	0.4712	0.4028	0.4378	0.4009	0.3333	0.3840	0.1708
	N2	0.3474	0.4611	0.3151	0.4207	0.3960	0.3457	0.3682	0.3151
	N3	0.2553	0.3037	0.2417	0.3234	0.2857	0.1838	0.2200	0.1208
	N4	0.2411	0.2408	0.1374	0.1990	0.1429	0.1282	0.0960	0.0417
	T1	0.0127	0.0079	0.0118	0.0101	0.0096	0.0100	0.0087	0.0143
	LSC	0.9755	0.9838	0.9686	0.9708	0.9725	0.9686	0.9786	0.9428
Network	*Density*	0.8678	0.8610	0.8478	0.8433	0.8430	0.8232	0.8426	0.8030
	ClsCoef	0.3965	0.4411	0.4281	0.4453	0.4398	0.4383	0.4435	0.4465
	Hubs	0.8595	0.9252	0.9305	0.9121	0.9152	0.8929	0.9074	0.8912
Overlap	*F1*	0.9129	0.9958	0.8365	0.9641	0.9643	0.9519	0.9370	0.9711
	F1v	0.5101	0.8483	0.3019	0.5797	0.6427	0.4057	0.4816	0.3269
	F2	0.1818	0.0508	0.0135	0.1107	0.3023	0.0257	0.0552	0.0329
	F3	0.8865	0.9791	0.9479	0.9900	0.9032	0.9530	0.9600	0.8750
	F4	0.6879	0.9215	0.8294	0.9453	0.8479	0.8846	0.9000	0.6875
Smoothness	S1	0.3643	0.3421	0.2952	0.3100	0.2778	0.2446	0.2731	0.1255
	S2	0.1465	0.0908	0.1134	0.1075	0.1108	0.1122	0.1043	0.0961
	S3	0.2411	0.3246	0.2417	0.3284	0.2857	0.1709	0.2080	0.1208
	S4	0.2003	0.1863	0.1552	0.1516	0.1646	0.1175	0.1631	0.0625

Appendix D

Figures

D.1. Complexity Metrics

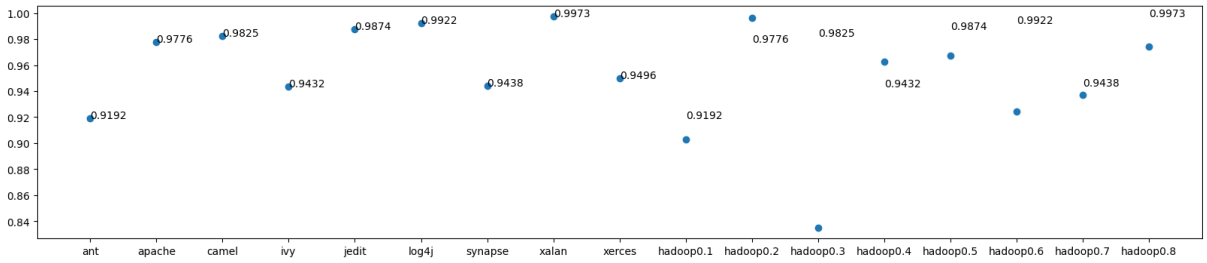


Figure D.1: Overlap F1

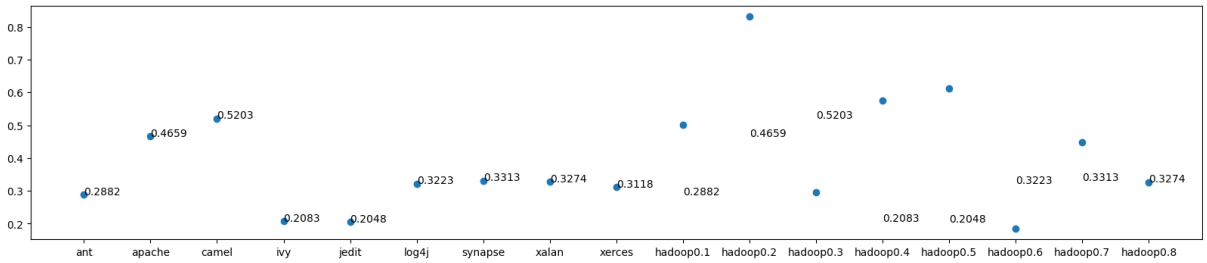


Figure D.2: Overlap F1v

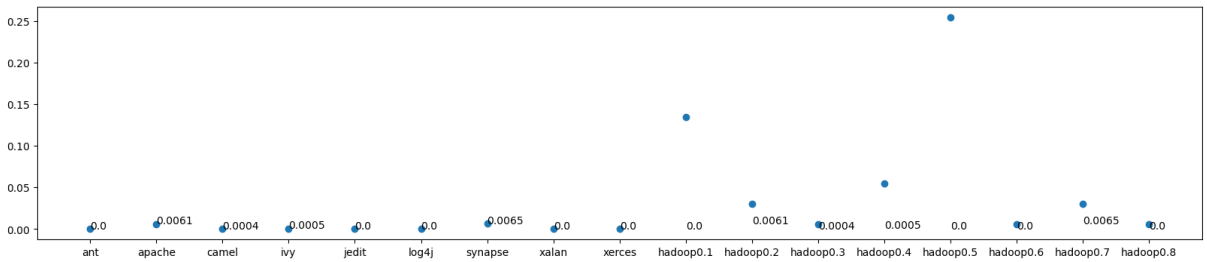


Figure D.3: Overlap F2

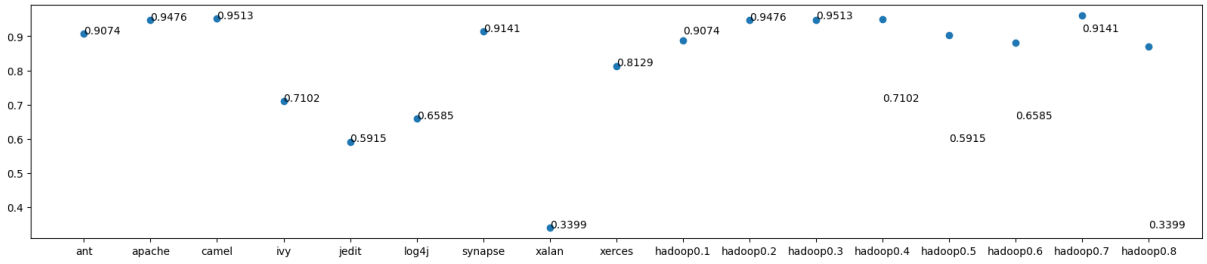


Figure D.4: Overlap F3

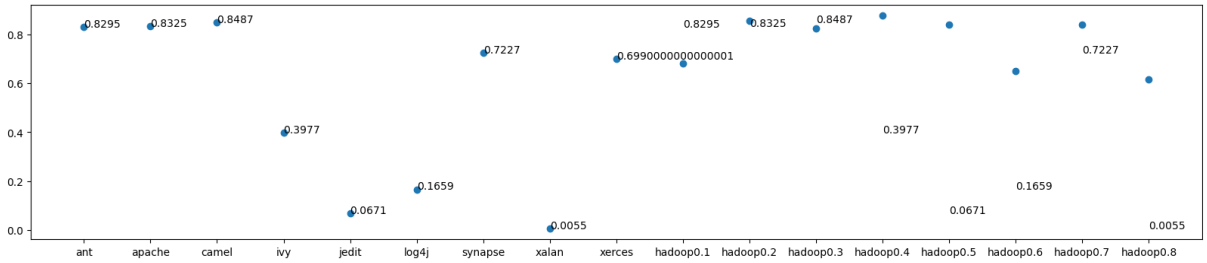


Figure D.5: Overlap F4

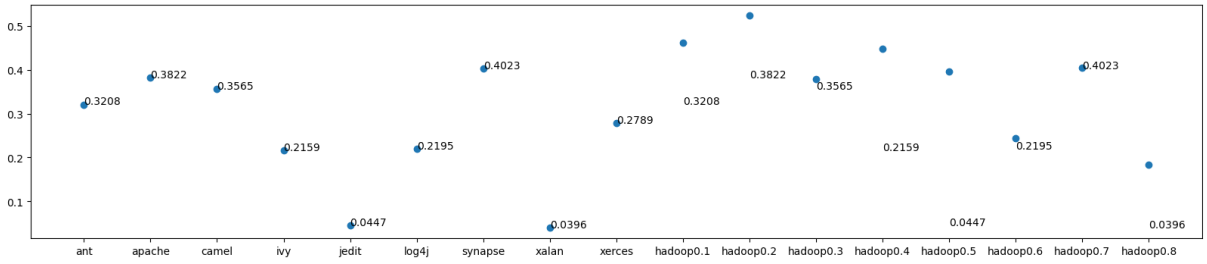


Figure D.6: Neighborhood N1

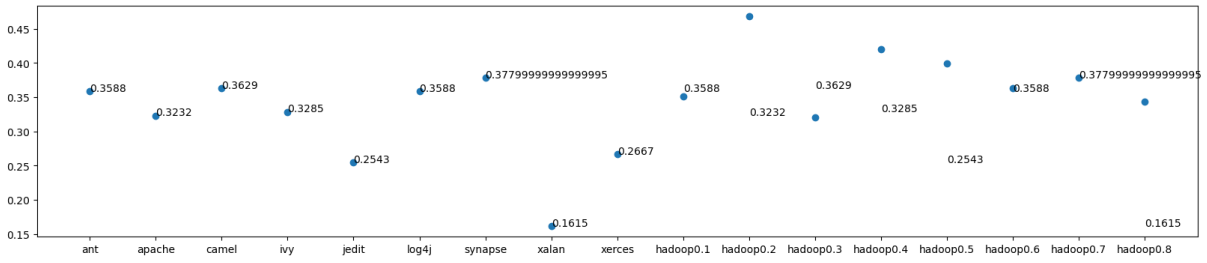


Figure D.7: Neighborhood N2

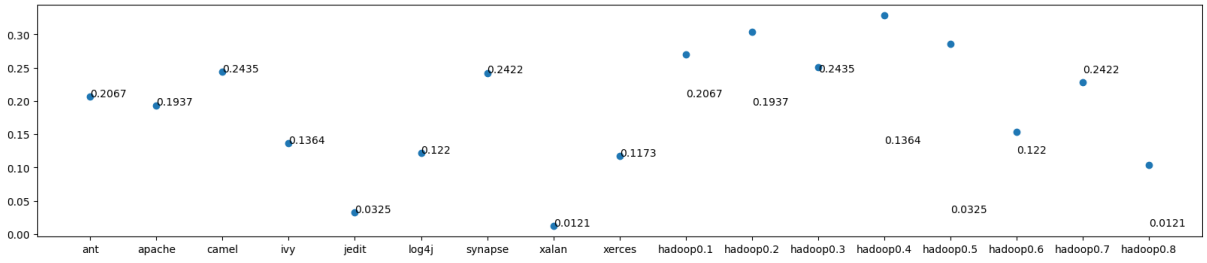


Figure D.8: Neighborhood N3

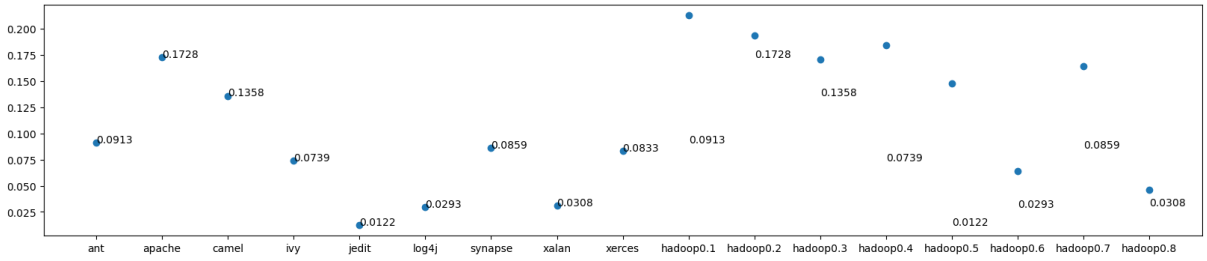


Figure D.9: Neighborhood N4

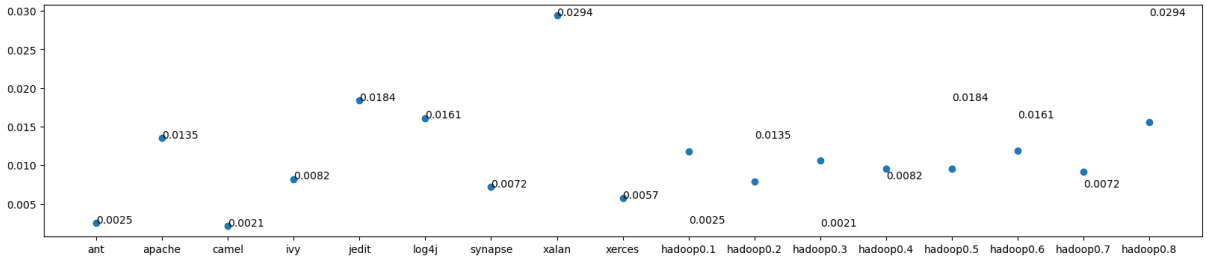


Figure D.10: Neighborhood T1

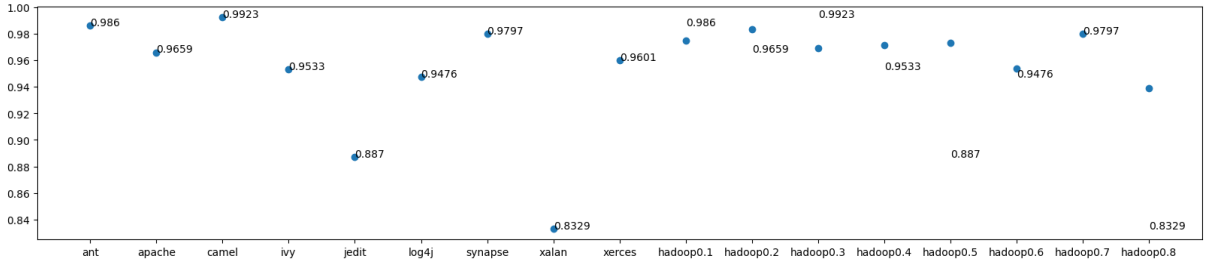


Figure D.11: Neighborhood LSC

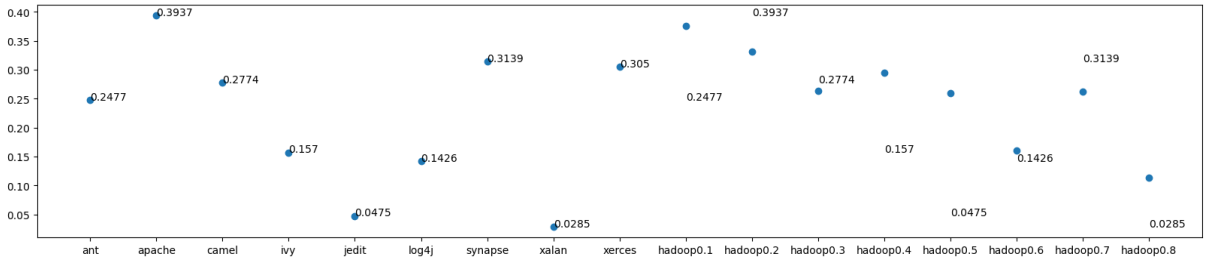


Figure D.12: Linearity L1

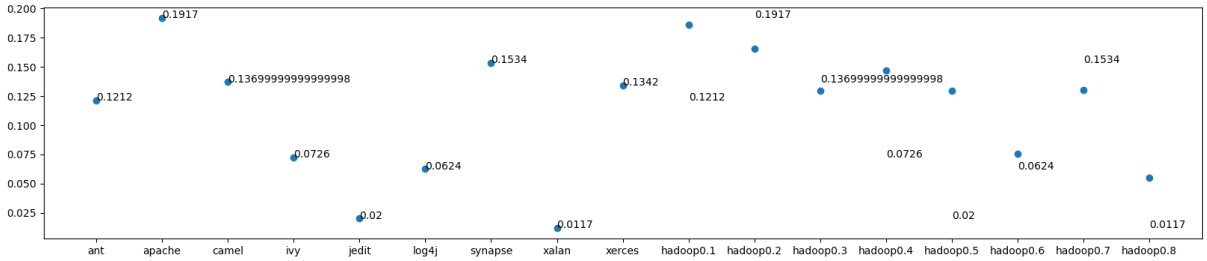


Figure D.13: Linearity L2

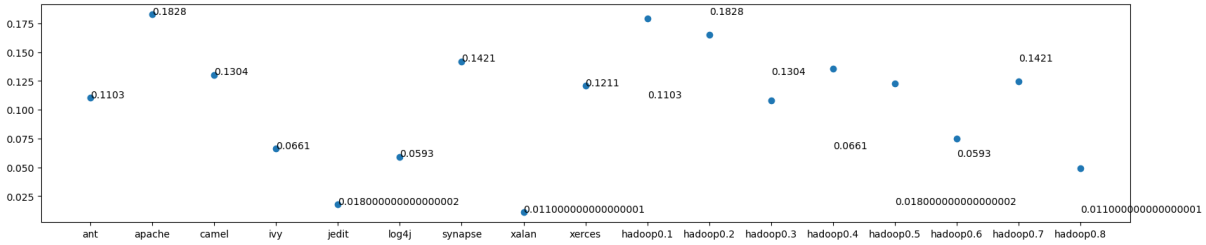


Figure D.14: Linearity L3

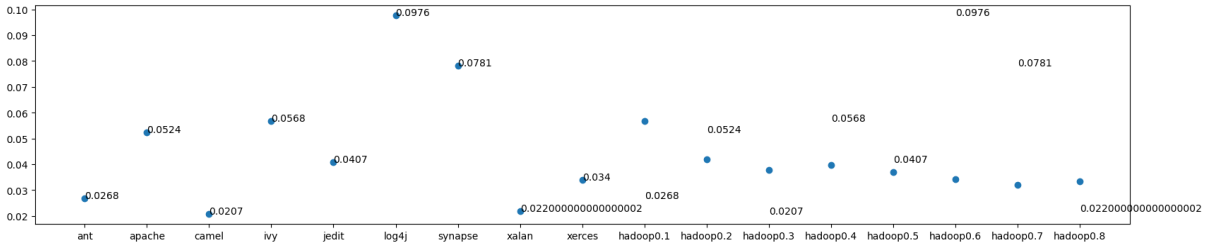


Figure D.15: Dimensionality T2

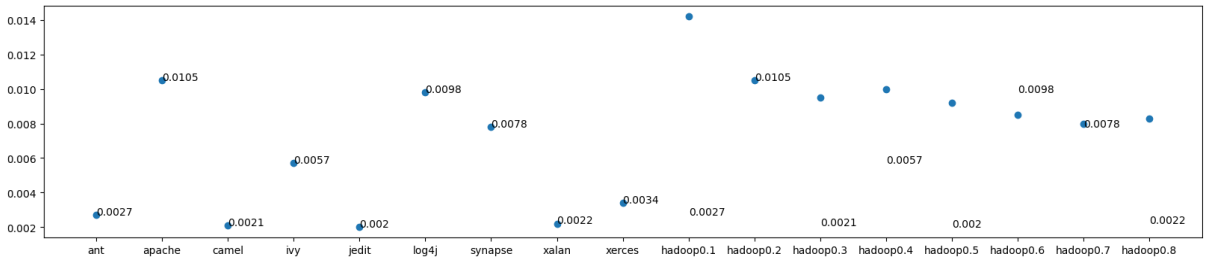


Figure D.16: Dimensionality T3

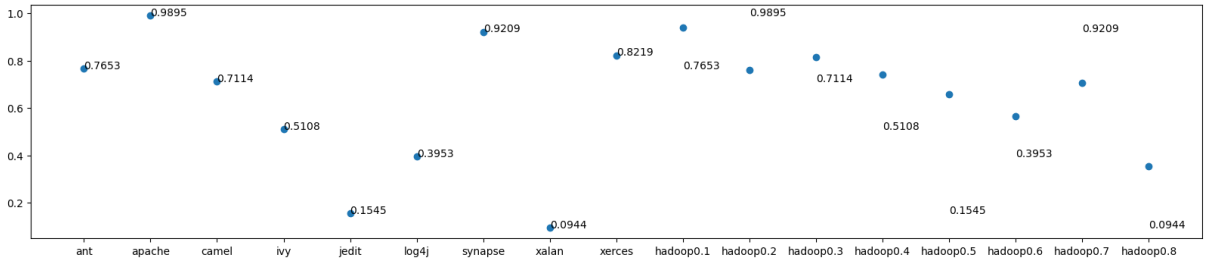


Figure D.17: Balance C1

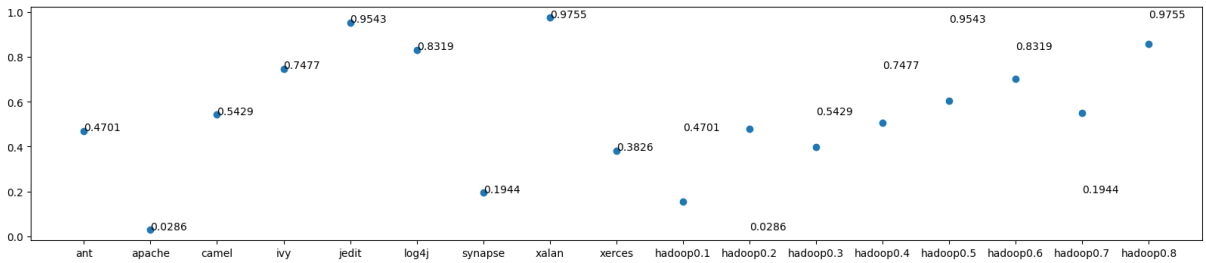


Figure D.18: Balance C2

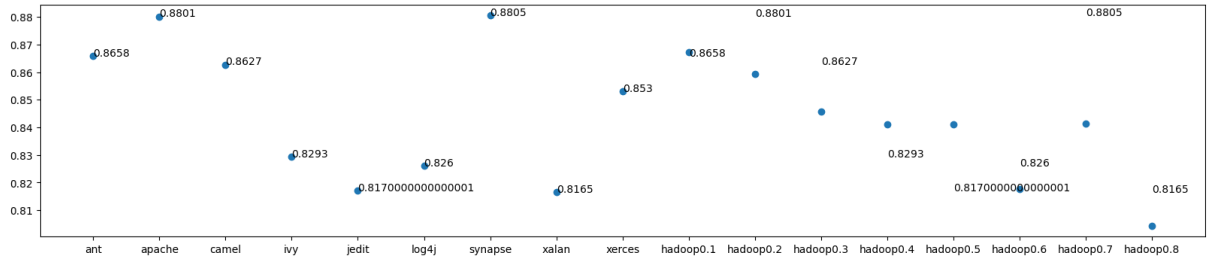


Figure D.19: Network Density

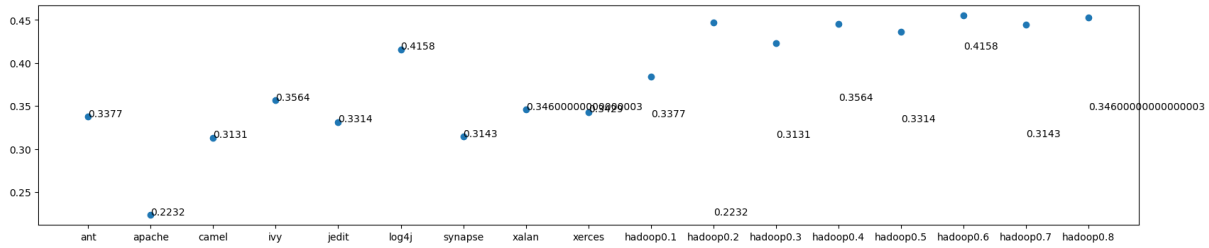


Figure D.20: Network ClsCoef

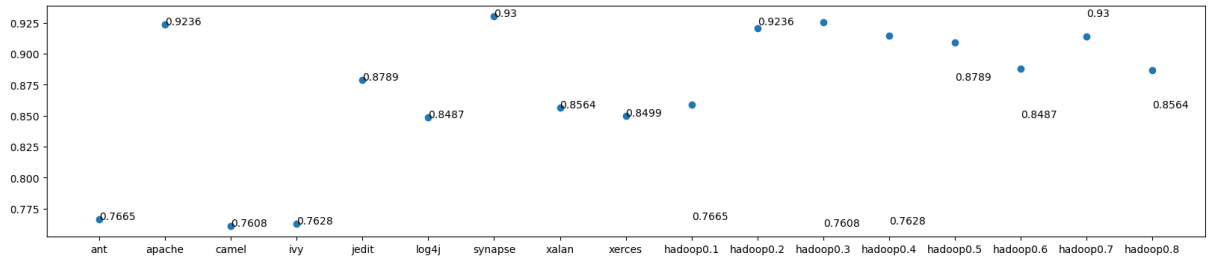


Figure D.21: Network Hubs

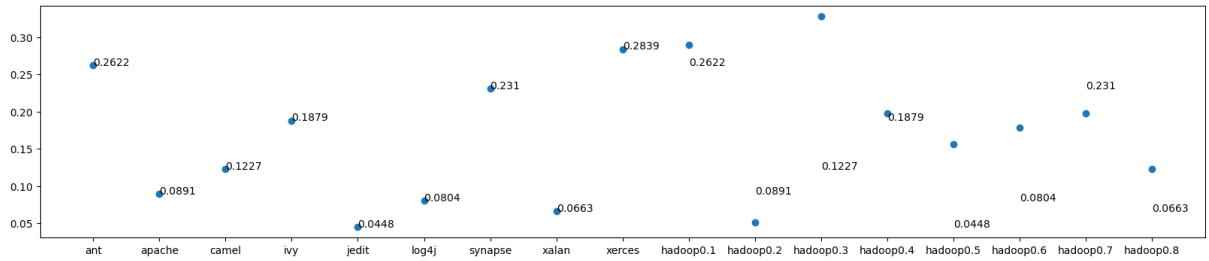


Figure D.22: Correlation C2

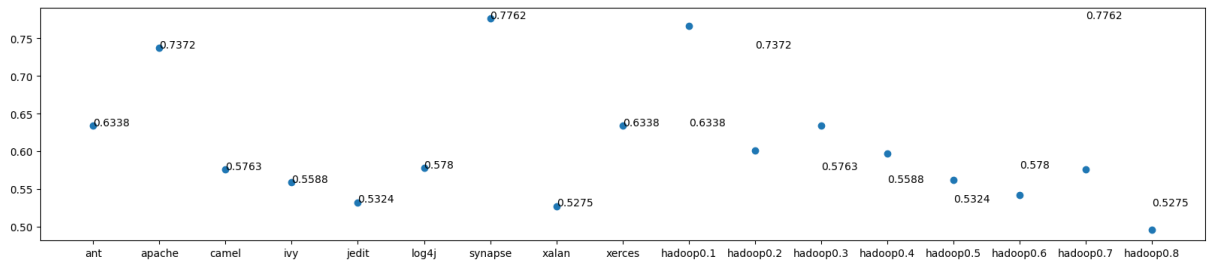


Figure D.23: Correlation C3

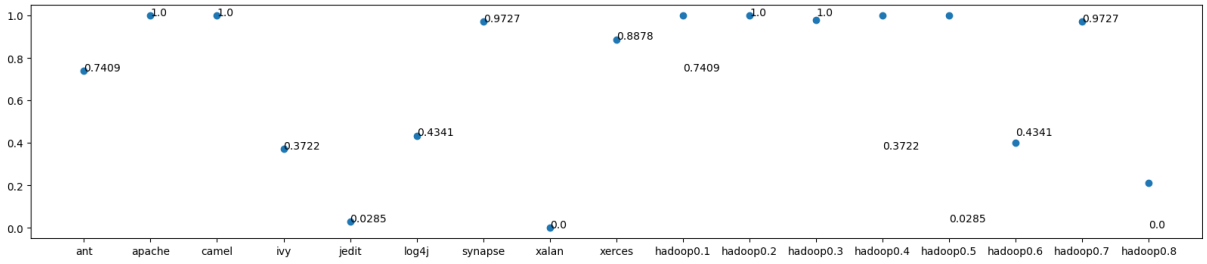


Figure D.24: Correlation C4

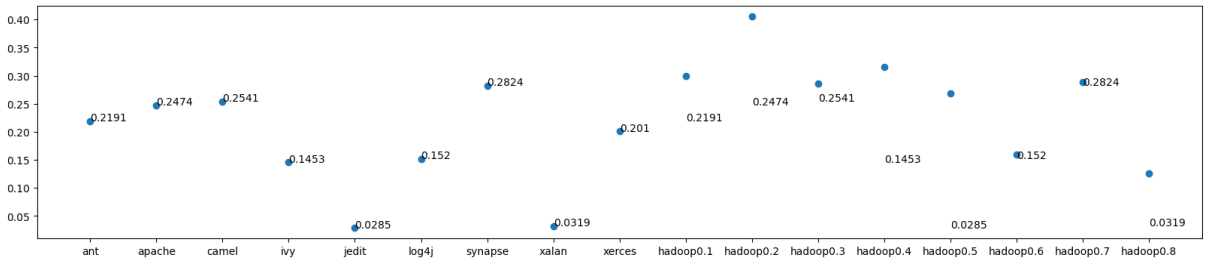


Figure D.25: Smoothness S1

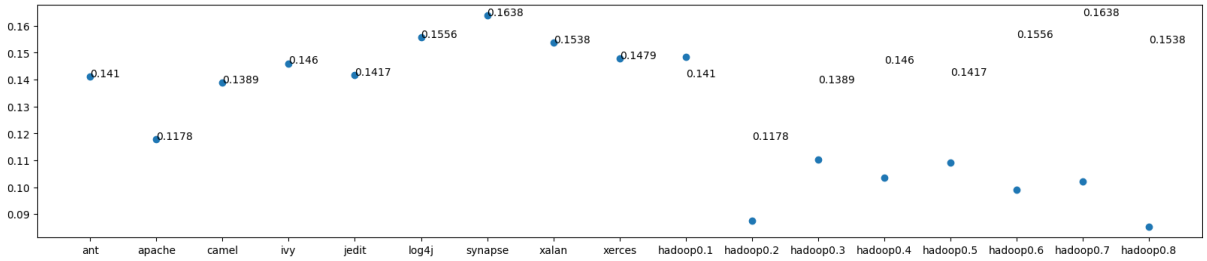


Figure D.26: Smoothness S2

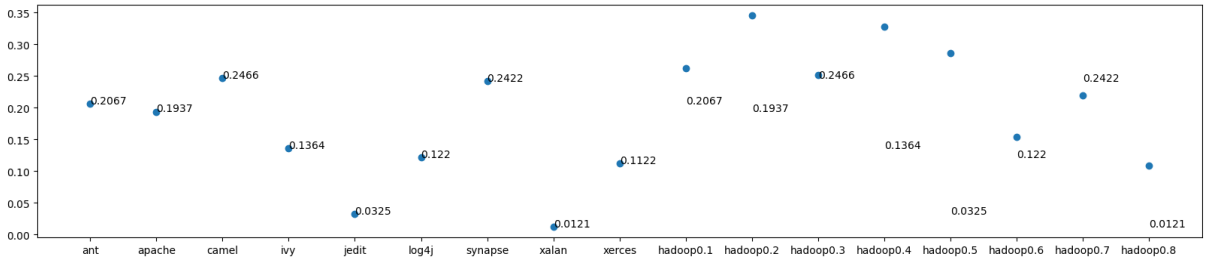


Figure D.27: Smoothness S3

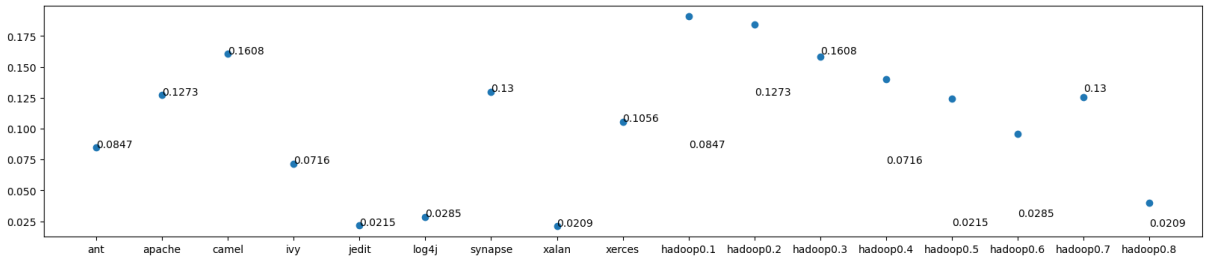


Figure D.28: Smoothness S4

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá