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 Rodriguez García

2014

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 Rodriguez 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 nuestros alumnos pasados, presentes y futuros. . .

"Empieza haciendo lo necesario, luego haz lo posible y de pronto empezarás a hacer lo imposible."
Francisco de Asís

Acknowledgements

A todos los que la presente vieren y entendieren.

Inicio de las Leyes Orgánicas. Juan Carlos I

Este trabajo es el fruto de muchas horas de trabajo, tanto de los autores últimos de los ficheros de la distribución como de todos los que en mayor o menor medida han participado en él a lo largo de su proceso de gestación.

Mención especial merece Manuel Ocaña, el autor de la primera versión de las plantillas de proyectos fin de carrera y tesis doctorales usadas en el Departamento de Electrónica de la Universidad de Alcalá, con contribuciones de Jesús Nuevo, Pedro Revenga, Fernando Herránz y Noelia Hernández.

En la versión actual, la mayor parte de las definiciones de estilos de partida proceden de la tesis doctoral de Roberto Barra-Chicote, con lo que gracias muy especiales para él.

También damos las gracias a Manuel Villaverde, David Casillas, Jesús Pablo Martínez, José Francisco Velasco Cerpa que nos han proporcionado secciones completas y ejemplos puntuales de sus proyectos fin de carrera.

Finalmente, hay incontables contribuyentes a esta plantilla, la mayoría encontrados gracias a la magia del buscador de Google. Hemos intentado referenciar los más importantes en los fuentes de la plantilla, aunque seguro que hemos omitido alguno. Desde aquí les damos las gracias a todos ellos por compartir su saber con el mundo.

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

Desocupado lector, sin juramento me podrás creer que quisiera que este libro [...] fuera el más hermoso, el más gallardo y más discreto que pudiera imaginarse[1].

Miguel de Cervantes, Don Quijote de la Mancha

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]Tomado de ejemplos del proyecto TEXIS.

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:

- RQ Which complexity metrics are related to miss-classification?
- RQ How complexity metrics are correlated to the outcome of supervised algorithms?
- RQ How complexity metrics and imbalance are related?
- RQ 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

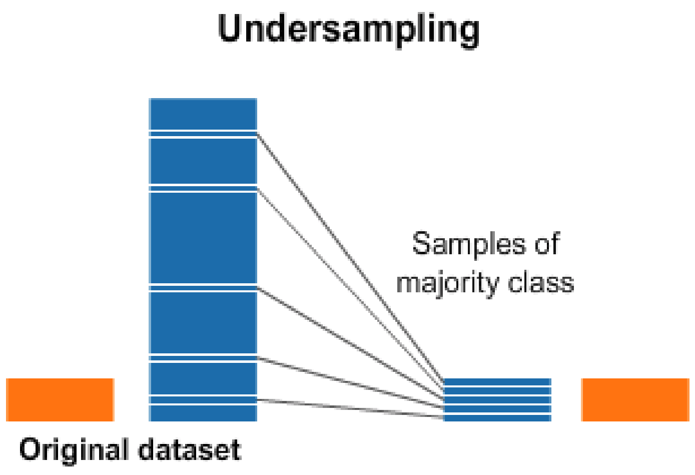

Figure 2.1: Undersampling Technique [1]

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

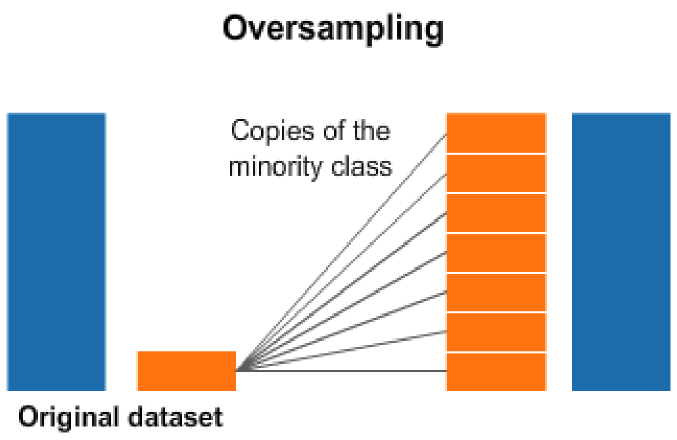

Figure 2.2: Oversampling Technique [1]

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 optimim metric to evaluate and compare classifiers ([22, 24, 25]).
- There are quality issues regarding the data (imbalanced class overlaps, outliers, transformation issues, etc.).

In Seiffert et al. [26] and in Khoshgoftaar et al. [27] it was highlighted the problem of imbalanced datasets when dealing with defect prediction.

Regarding performance and evaluation of the classifiers, no technique seems to consistently perform better than the others. Many papers have compared performance of multiple measures (Peng [28], Peng et al. [29] - that propose performance metrics to evaluate merit of classification algorithms and ranked classification algorithms, respectively).

More research made on the subject comes from the hand of Lessman et al. [30], that 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 ??, Area Under the ROC curve

Chapter 3

Conclusiones y líneas futuras

En este apartado se resumen las conclusiones obtenidas y se proponen futuras líneas de investigación que se deriven del trabajo.

La estructura del capítulo es...

3.1. Conclusiones

Para añadir una referencia a un autor, se puede utilizar el paquete `cite`. En el trabajo [**?**], se muestra un trabajo...

Y podemos usar de nuevo algún acrónimo, como por ejemplo Time Domain Pitch Synchronous Over-Lap Add (TD-PSOLA), o uno ya referenciado como Artificial Neural Network (ANN).

3.2. Líneas futuras

Pues eso.

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

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:

[1] https://github.com/PabloAceG/ComputingProject/blob/master/code/requirements.txt

[2] All commands are executed from the parent folder of the repository.

```
pip install <package_name>
```

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.

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

```
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("me
            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

    def safe_connect(self, operation) :
        connection = self.__connection

        return connection.r(operation)
```

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.

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

```
...
...

def __load_arff(path):
    '''
        Loads the dataset content of an .arff file into the workspace.
        Input:
            - path: Location of the file.
        Output:
            - dataset: Data from the file.
    '''

    dataset = []

    with open(path, 'r') as data:
        dataset = arff.load(data)['data']

    return dataset

def __load_csv(path):
    '''
        Loads the data content of an .csv file into the workspace.
        Input:
            - path: Location of the file.
        Output:
            - dataset: Data from the file.
    '''

    dataset = []

    with open (path, 'r') as csv_file:
        dataset = pd.read_csv(
            csv_file,
            sep=','
        )

    return dataset

def __parte_dataset(dataset, input_select, target_select):
    '''
        Takes a dataset and parses it to only take what is necessary.
        Inputs:
            - dataset: Input raw data as a Pandas Data Framework.
            - start: Where columns start to be useful.
        Output:
            - input: Input arrays of the dataset.
            - target: Target column of the dataset.
    '''

    num_rows    = len(dataset)
    last_column = target_select if (target_select < 0) else None

    # Input
    input_columns = dataset.columns[input_select:last_column]
    input = dataset[input_columns].head(num_rows).astype(float).to_numpy()
    # Target
    target_column = dataset.columns[target_select]
    target = dataset[target_column].head(num_rows)

    # Parse output
    sample = target[0]
    if not isinstance(sample, str):
        target = numpy.where(target > 0, 1, 0)
    else:
        has_faults = target == 'yes'
        target = numpy.where(has_faults , 1, 0)

    return input, target

def __data_preparation(path, input_select, target_select, type='arff', shuffle=False):
    '''
        Loads an .arff/.csv file and parses its content to input/output valid
        for a Machine Learning application.
        Input:
            - path: Location of the file.
            - start: Where columns start to be useful (string columns are not
              necessary).
            - type: It can be:
                - arff
                - csv
        Output:
            - input: Input arrays of the dataset.
            - target: Target column of the dataset.
    '''

    # Load data
    dataset = []
    if  type == 'arff':
        dataset = __load_arff(path)
    elif type == 'csv':
        dataset = __load_csv(path)
    else:
        raise Exception('Not a valid file type. Try with arff or csv!')
        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.
    '''

    path       = ''     # Location of file
    start      = 0      # From which columns to use
    output_col = -2     # Position of output column
    type       = 'arff' # Type of file to be read. Default .arff

    if name == 'ant':
        # Retrieve data
        path = './dataset/ant-1.7.arff'
        start = 3

    elif name == 'apache':
        # Retrieve data
        path = './dataset/Apache.csv'
        start = 3
        output_col = -1
        type = 'csv'

    elif name == 'camel':
        # Retrieve data
        path = './dataset/camel-1.6.arff'
        start = 3

    elif name == 'iris': # Special case. Non-binary output. Needs different
                         # treatment.
        # Retrieve data
        path = './dataset/iris.csv'
        input, target = __data_preparation_iris_dataset(path, shuffle)

        return input, target

    elif name == 'ivy':
        # Retrieve data
        path = './dataset/ivy-2.0.arff'
        start = 3

    ...
    ...
    elif name == 'xerces':
        # Retrieve data
        path = './dataset/xerces-1.4.arff'
        start = 3
    elif name == 'hadoop-1':
        # Retrieve data
        path = './dataset/hadoop-proc-0.1.csv'
        start = 2
        output_col = 1
        type = 'csv'

    ...
    ...
    elif name == 'hadoop-8':
        # Retrieve data
        path = './dataset/hadoop-proc-0.8.csv'
        start = 2
        output_col = 1
        type = 'csv'



    else:
        raise Exception('The␣given␣dataset␣name␣is␣not␣valid.')
        sys.exit(404)

    # Return results
    input, target = __data_preparation(path, start, output_col, type, shuffle)

    return input, target

def confusion_matrix(x_test: list, y_test: list, classifier):
    '''
        Obtains the confusion matrix for a given testing dataset, with binary
        output.
        Input:
            - x_test: paremeters for testing dataset.
            - y_test: target/desired output for testing dataset.
            - classifier: trained classifier.
        Output:
            (
                true_positive: successfully predicted positives
                true_negative: successfully predicted negatives
                false_positive: unsuccessfully predicted positives
                false_negative: unsuccessfully predicted positives
            )
    '''
    # Confusion Matrix Cells
    true_positive:float  = 0
    true_negative:float  = 0
    false_positive:float = 0
```

```python
    false_negative:float = 0

    # Calculate number of repetitions of each classification.
    for (input, target) in zip(x_test, y_test):
        prediction:int = classifier.predict([input])[0]

        if prediction == target:    # Success
            if prediction: true_positive += 1
            else:          true_negative += 1
        else:                       # Wrong
            if prediction: false_positive += 1
            else:          false_negative += 1

    # Transform absolute to relative values
    num_samples = len(y_test)
    true_positive  = true_positive  / num_samples
    true_negative  = true_negative  / num_samples
    false_positive = false_positive / num_samples
    false_negative = false_negative / num_samples

    return (true_positive, true_negative, false_positive, false_negative)

def recall(targets, predictions) -> float:
    '''
        Sensitivity, recall, hit rate or True Positive Rate (RPR)
        https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.h
                TP       TP
        TPR = ---- = --------- = 1 - FNR
                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_scor
                    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_accur
                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_corr
                            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)
```

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.

Listing B.3: Compare complexity metrics of different datasets

```
...
...
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) fo
            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',
]


# Store metrics computed from datasets (datatype: dict)
results = {
    'balance': {
        'C1': [],
        'C2': []
    },
    'correlation': {
        'C1': [],
        'C2': [],
        'C3': [],
        'C4': []
    },
    'dimensionality': {
        'T1': [],
        'T2': [],
        'T3': []
    },
    'linearity': {
        'L1': [],
        'L2': [],
        'L3': []
    },
    'neighborhood': {
        'N1':   [],
        'N2':   [],
        'N3':   [],
        'N4':   [],
        'T1':   [],
        'LSC':  [],
    },
    'network': {
        'Density':  [],
        'ClsCoef':  [],
        'Hubs':     []
    },
    'overlap': {
        'F1':   [],
        'F1v':  [],
        'F2':   [],
        'F3':   [],
        'F4':   []
    },
    'smoothness': {
        'S1': [],
        'S2': [],
        'S3': [],
        'S4': []
    }
}

# Connect to R
connector = r_connect()

# Get Metrics
for set in datasets:
    print(set)
    inputs, targets = DATASETS.get_dataset(set)
    metrics = connector.get_metrics(inputs, targets)

    results = add_metrics(results, metrics)

# Print Metrics
print(results)
plot_metrics_comparison(results)
```

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 **??**). This same process is repeated for every dataset.

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

```
...
...
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)

    # 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])
```

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.

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

```
...
...
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])
```

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.

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

```
...
...
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])
```

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

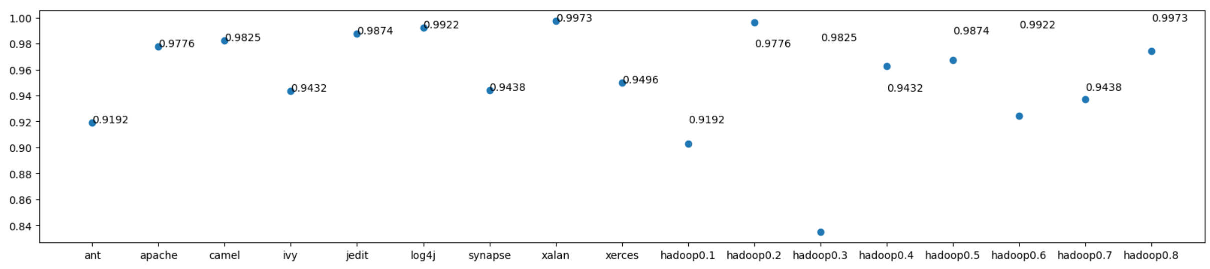

Figure D.1: Overlap F1

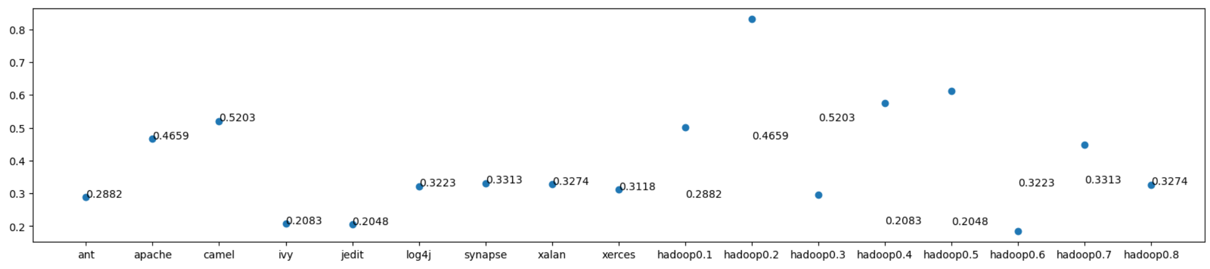

Figure D.2: Overlap F1v

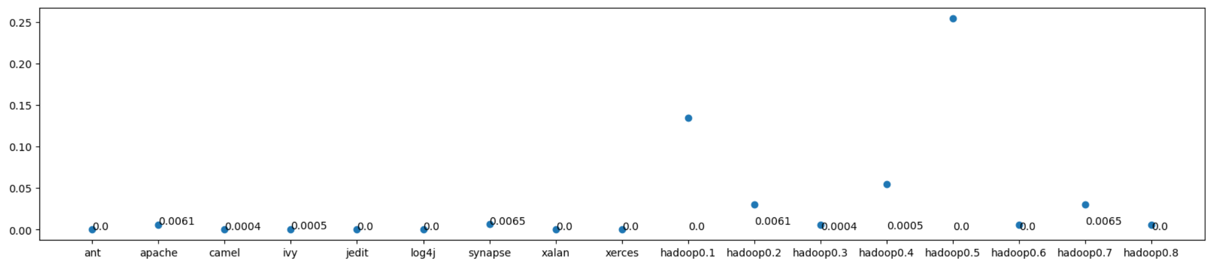

Figure D.3: Overlap F2

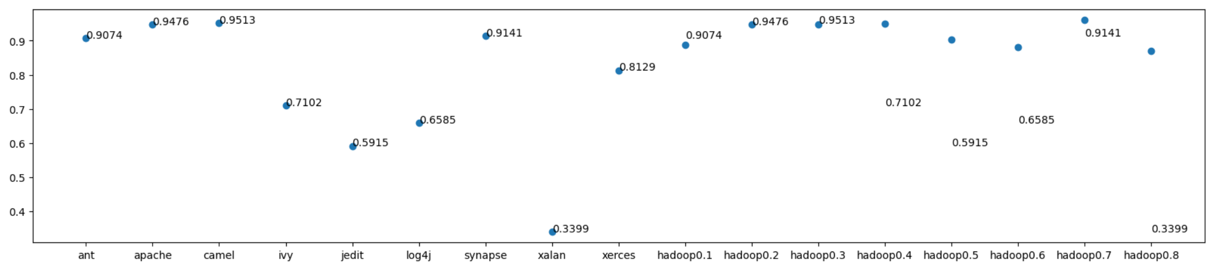

Figure D.4: Overlap F3

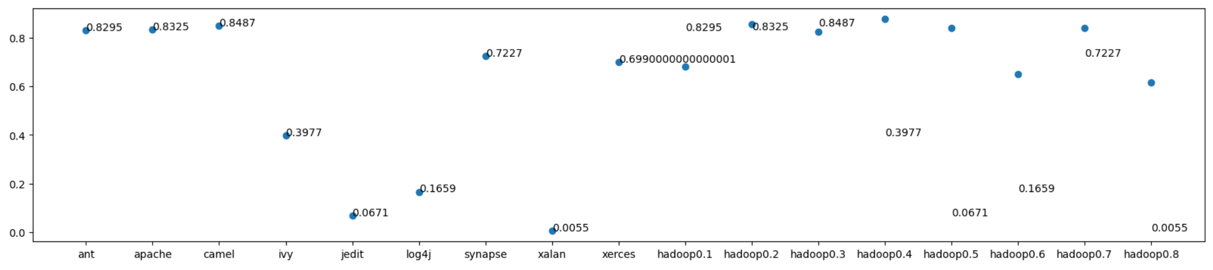

Figure D.5: Overlap F4

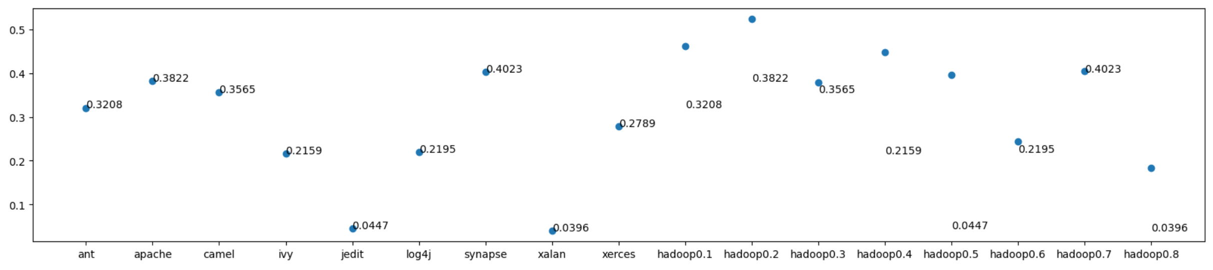

Figure D.6: Neighborhood N1

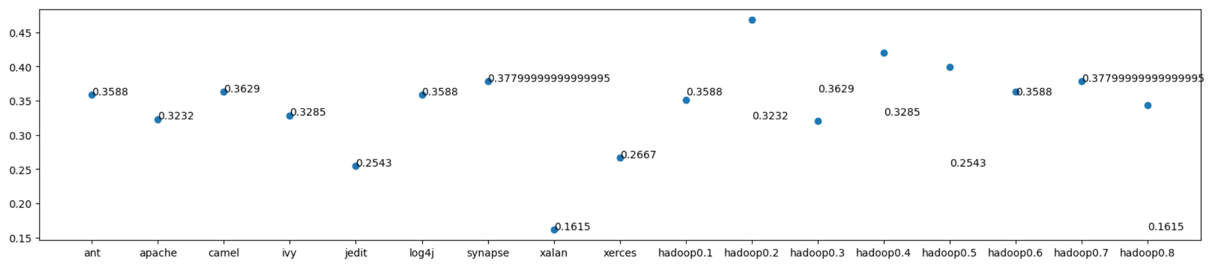

Figure D.7: Neighborhood N2

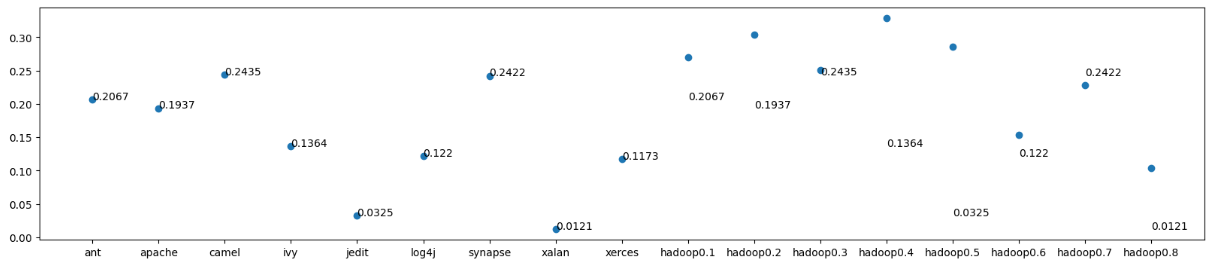

Figure D.8: Neighborhood N3

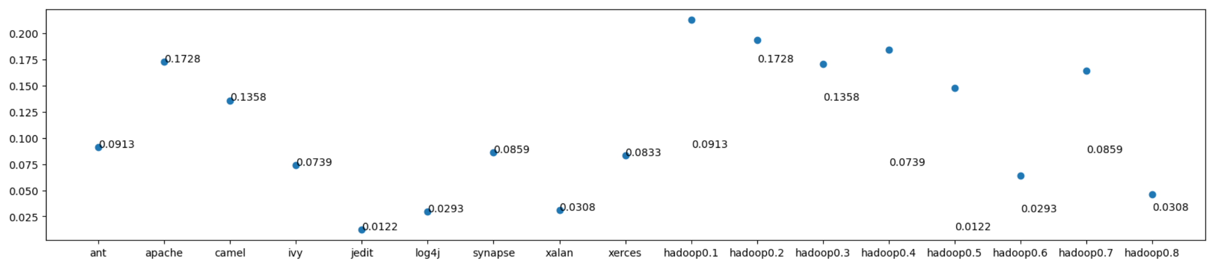

Figure D.9: Neighborhood N4

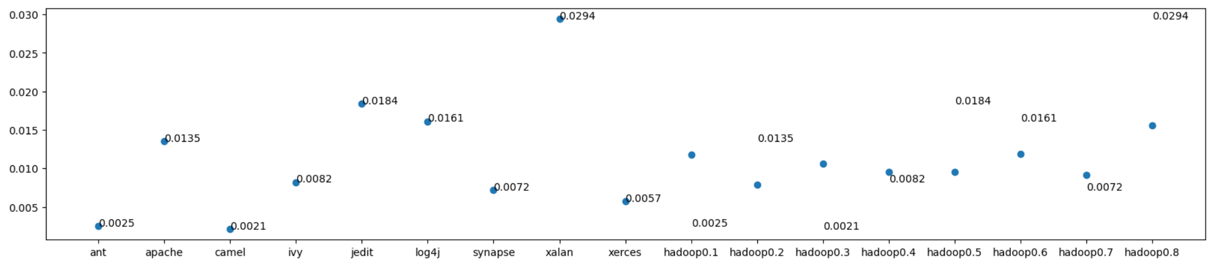

Figure D.10: Neighborhood T1

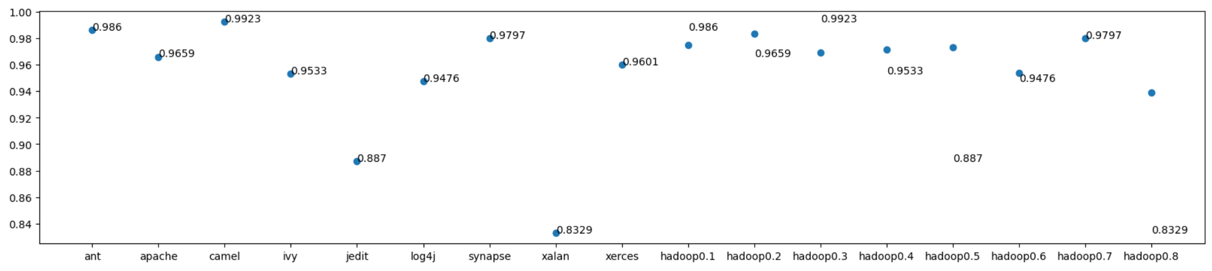

Figure D.11: Neighborhood LSC

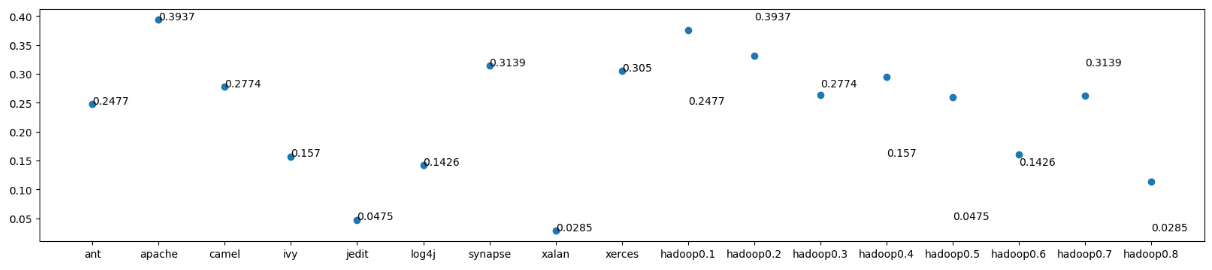

Figure D.12: Linearity L1

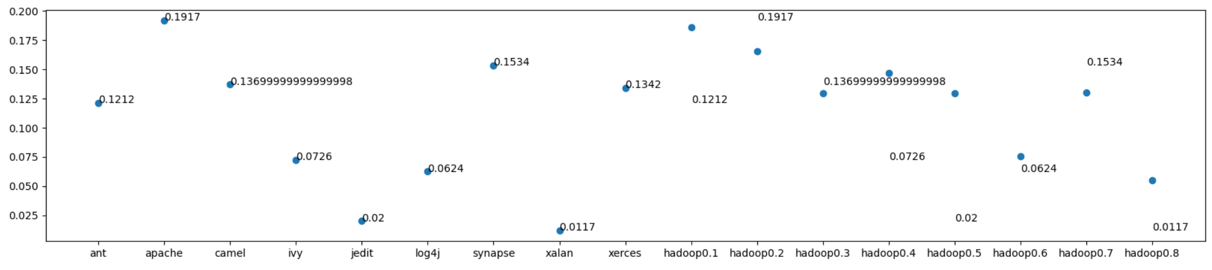

Figure D.13: Linearity L2

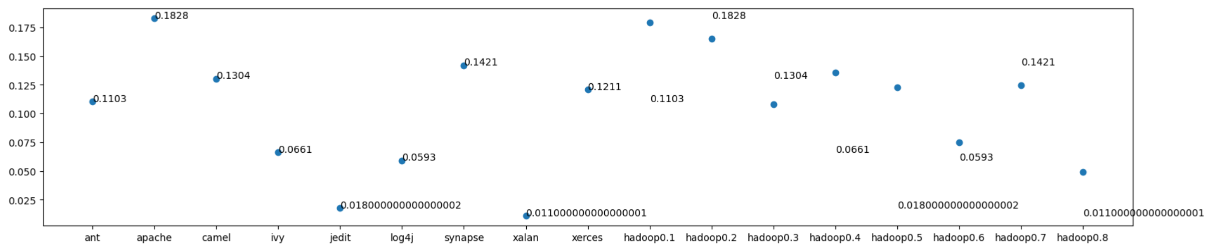

Figure D.14: Linearity L3

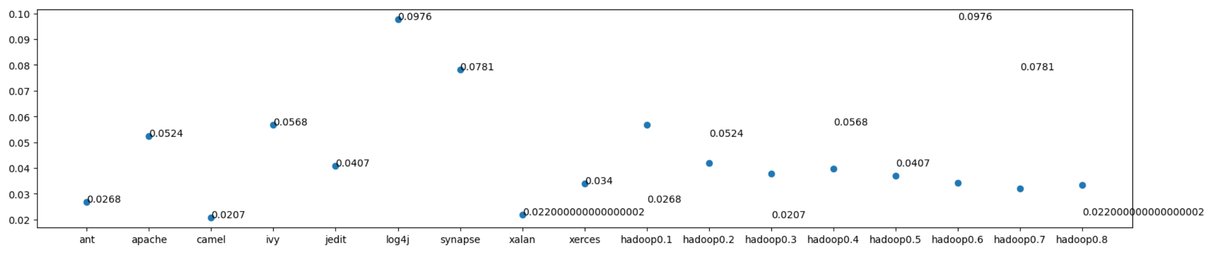

Figure D.15: Dimensionality T2

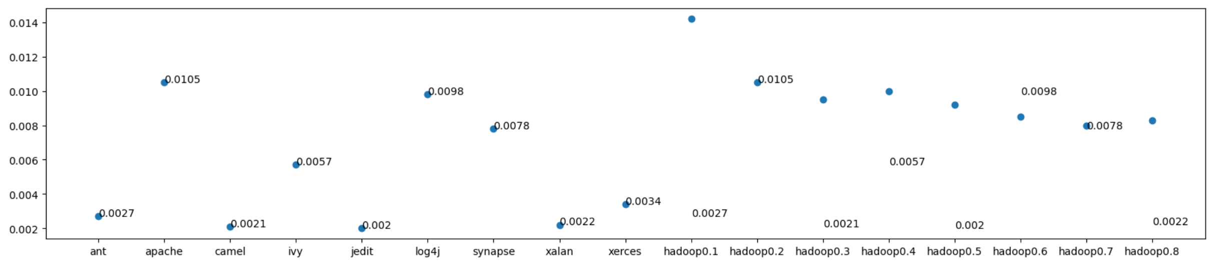

Figure D.16: Dimensionality T3

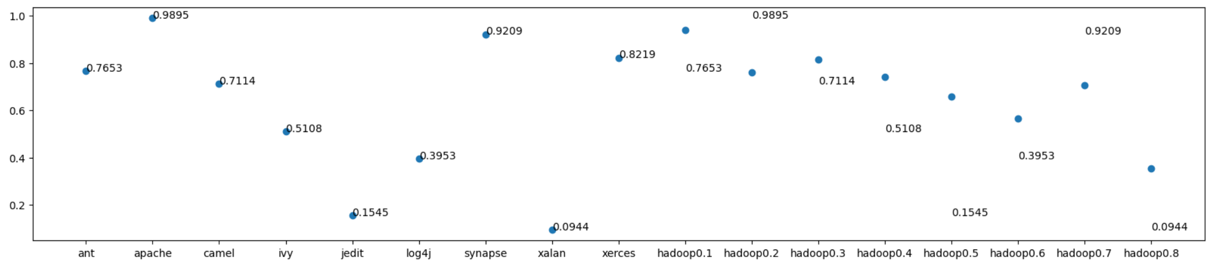

Figure D.17: Balance C1

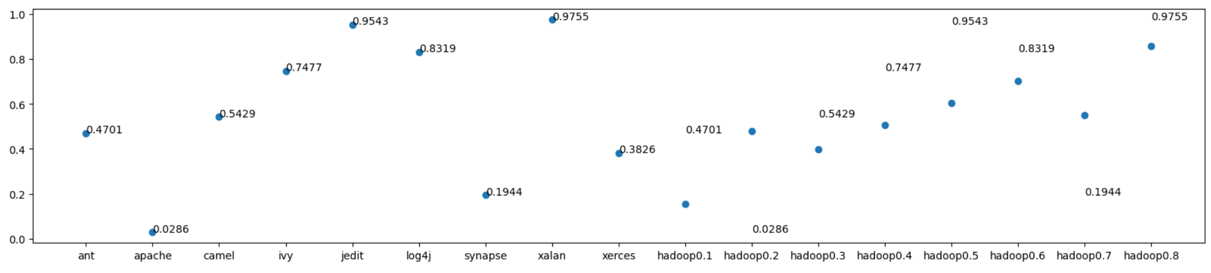

Figure D.18: Balance C2

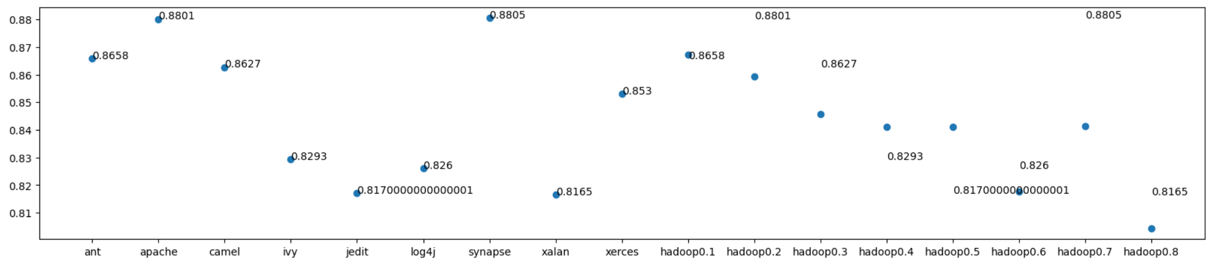
Figure D.19: Network Density

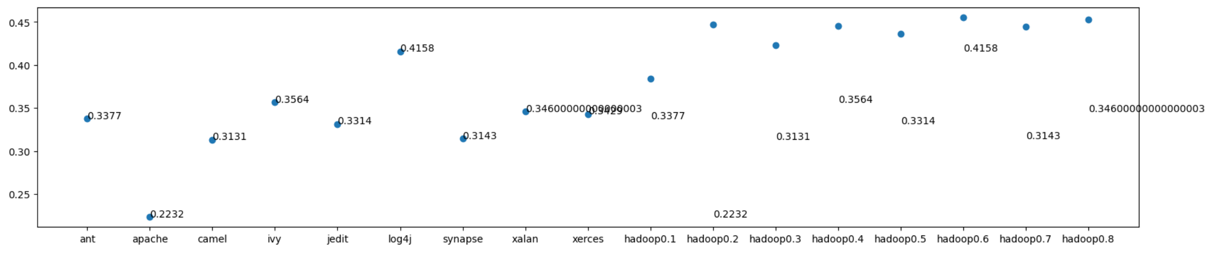
Figure D.20: Network ClsCoef

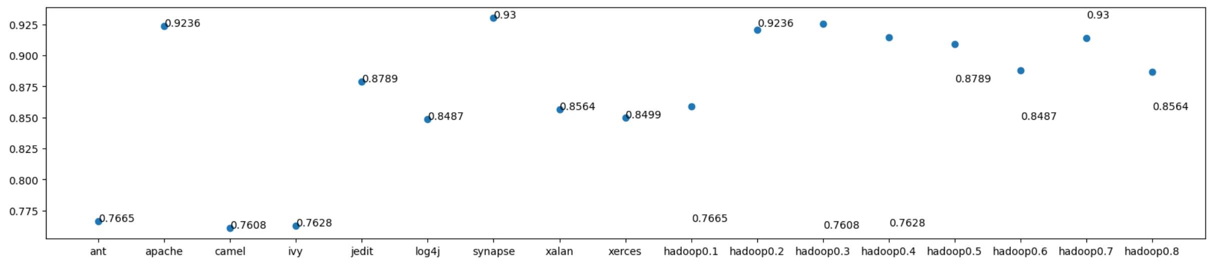
Figure D.21: Network Hubs

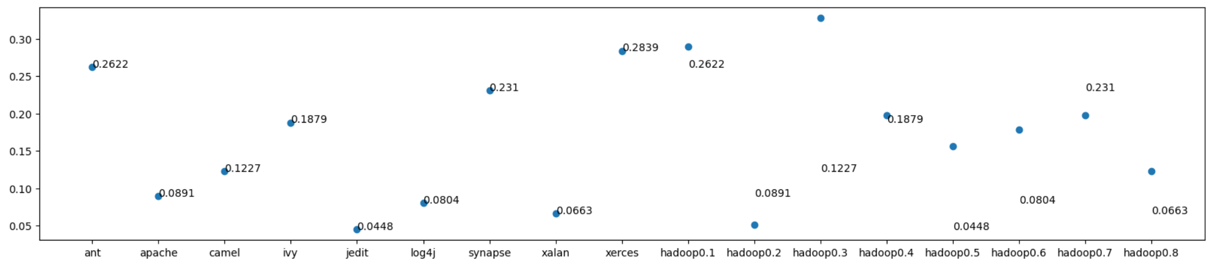
Figure D.22: Correlation C2

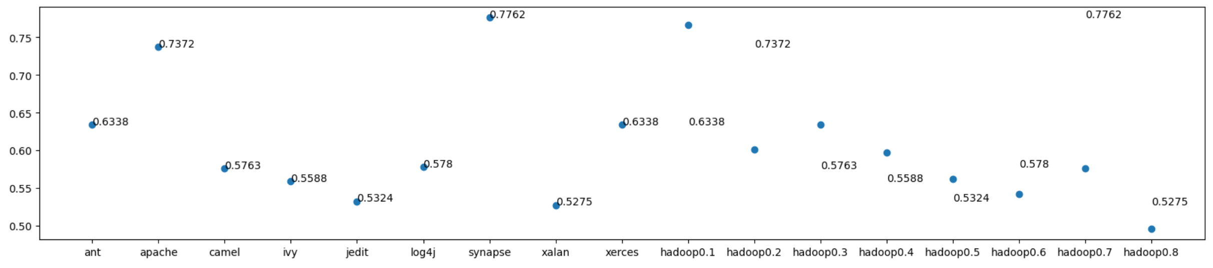
Figure D.23: Correlation C3

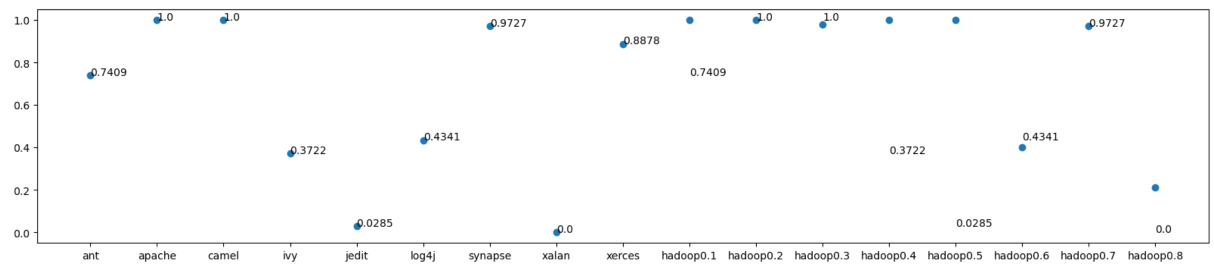

Figure D.24: Correlation C4

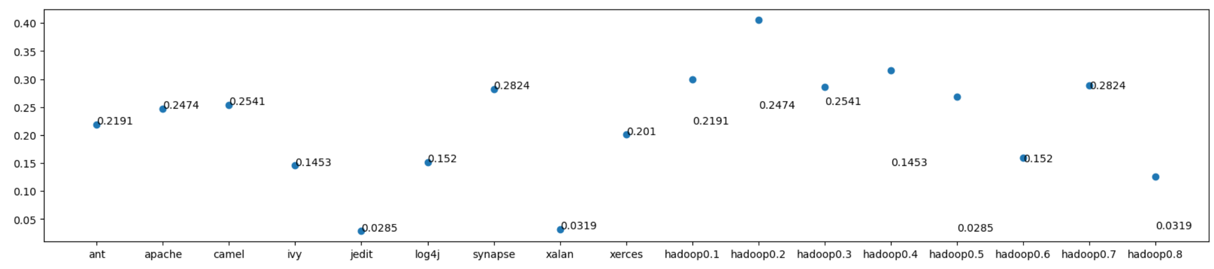

Figure D.25: Smoothness S1

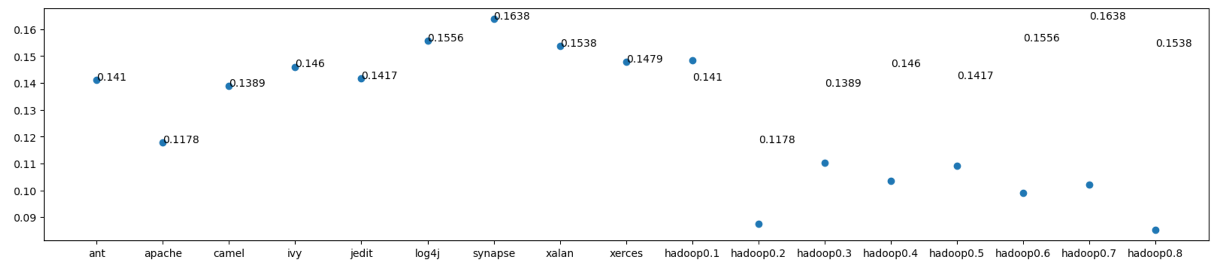

Figure D.26: Smoothness S2

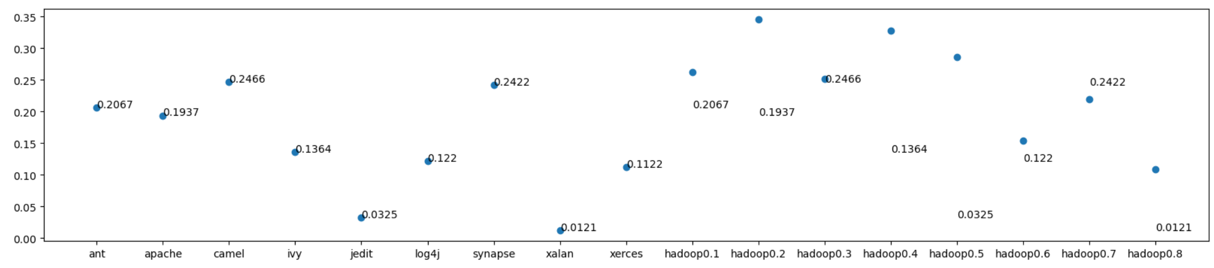

Figure D.27: Smoothness S3

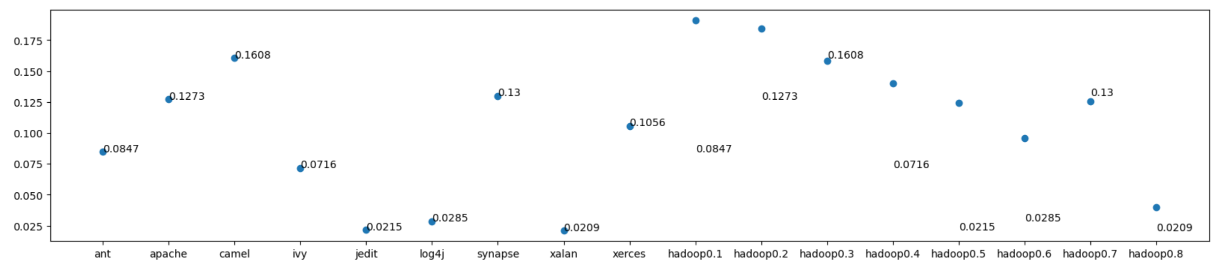

Figure D.28: Smoothness S4