[**IoT Based Noise Pollution Monitoring**](https://www.researchgate.net/publication/353287494_IoT_Based_Air_and_Noise_Pollution_Monitoring_System?enrichId=rgreq-0943d00ed5f2c2440752132ec621cc0b-XXX&enrichSource=Y292ZXJQYWdlOzM1MzI4NzQ5NDtBUzoxMDQ2MTQzMDM5NDY3NTIwQDE2MjY0MzEzNDUxODQ%3D&el=1_x_3&_esc=publicationCoverPdf)

**TEAM MEMBER**

**712221119002 – VIKRAM K**

**Phase 3 Submission Document**

**Project: Noise Pollution Monitoring**

# Introduction

In the last years, the large quantity of data of many different kinds and from different sources has created numerous challenges in the Data Mining area. Not only their size, but their imperfections and varied formats are providing the researchers with plenty of new scenarios to be addressed. Consequently, Data Preprocessing ([García et al.](#_bookmark15), [2015](#_bookmark15)) has become an important part of the *KDD (Knowledge Discovery from Databases)* process, and related software development is also essential to provide practitioners with the adequate tools.

Data Preprocessing intends to process the collected data appropriately so that subsequent learning algorithms can not only extract meaningful and relevant knowledge from the data, but also induce models with high predictive or descriptive performance. Data preprocessing is known as one of the most time-consuming steps in the whole KDD process. There exist several aspects involved in data preprocessing, like *feature selection*, dealing with *missing values* and detecting *noisy data*. Feature selection aims at extracting the most relevant attributes for the learning step, thus reducing the complexity of models and the computing time taken for their induction. The treatment of missing values is also essential to keep as much information as possible in the preprocessed dataset. Finally, noisy data refers to values that are either incorrect or clearly far from the general underlying data distribution.

All these tasks have associated software available. For instance, the KEEL tool ([Alcalá et al.](#_bookmark8), [2010](#_bookmark8)) contains a broad collection of data preprocessing algorithms, which covers all the aforementioned topics. There exist many other general-purpose Data Mining software with data preprocessing functionalities, like WEKA ([Witten and Frank](#_bookmark38), [2005](#_bookmark38)), KNIME ([Berthold et al.](#_bookmark9), [2009](#_bookmark9)), RapidMiner ([Hofmann and Klinkenberg](#_bookmark18), [2013](#_bookmark18)) or R.

Regarding the R statistical software, there are plenty of packages available in the *Comprehensive R Archive Network (CRAN)* repository to address preprocessing tasks. For example, [**MICE**](https://CRAN.R-project.org/package%3DMICE) ([van Buuren](#_bookmark35) [and Groothuis-Oudshoorn](#_bookmark35), [2011](#_bookmark35)) and [**Amelia**](https://CRAN.R-project.org/package%3DAmelia) ([Honaker et al.](#_bookmark19), [2011](#_bookmark19)) are very popular packages for handling missing values, whereas [**caret**](https://CRAN.R-project.org/package%3Dcaret) ([Kuhn](#_bookmark22), [2008](#_bookmark22)) or [**FSelector**](https://CRAN.R-project.org/package%3DFSelector) ([Romanski and Kotthoff](#_bookmark29), [2014](#_bookmark29)) provide a wide range of techniques for feature selection. There are also general-purpose packages for decting outliers and anomalies, like [**mvoutlier**](https://CRAN.R-project.org/package%3Dmvoutlier) ([Filzmoser and Gschwandtner](#_bookmark12), [2015](#_bookmark12)). If we examine software in CRAN developed to tackle label noise, there already exist *non-preprocessing* packages that provide label noise robust classifiers. For instance, [**robustDA**](https://CRAN.R-project.org/package%3DrobustDA) implements a robust mixture discriminant analysis [Bouveyron and Girard](#_bookmark10) ([2009](#_bookmark10)), while [**probFDA**](https://CRAN.R-project.org/package%3DprobFDA) package provides a probabilistic Fisher discriminant analysis related to the seminal work in [Lawrence and Schölkopf](#_bookmark23) ([2001](#_bookmark23)).

However, to the best of our knowledge, CRAN lacks an extensive collection of label noise prepro- cessing algorithms for classification ([Sáez et al.](#_bookmark31), [2016](#_bookmark31); [Garcia et al.](#_bookmark14), [2015](#_bookmark14)), some of which are among the most influential preprocessing techniques ([García et al.](#_bookmark16), [2016](#_bookmark16)). This is the gap we intend to fill with the release of the [**NoiseFiltersR**](https://CRAN.R-project.org/package%3DNoiseFiltersR) package, whose taxonomy is inspired on the recent survey on label noise by B. Frénay and M. Verleysen ([Frénay and Verleysen](#_bookmark13), [2014](#_bookmark13)). Yet, it should be noted that there are other packages that include some isolated implementations of label noise filters, since they are sometimes needed as auxiliary functions. This is the case of the [**unbalanced**](https://CRAN.R-project.org/package%3Dunbalanced) ([Pozzolo et al.](#_bookmark27), [2015](#_bookmark27)) package, which deals with imbalanced classification. It contains basic versions of classical filters, such as *Tomek-Links* ([Tomek](#_bookmark34), [1976](#_bookmark34)) or *ENN* ([Wilson](#_bookmark37), [1972](#_bookmark37)), which are tipically applied after oversampling an imbalanced dataset (which is the main purpose of the **unbalanced** package)

In the following section we briefly introduce the problem of classification with label noise, as well as the most popular techniques to overcome this problem. Then, we show how to use the **NoiseFiltersR** package to apply these techniques in a unified and R-user-friendly manner. Finally, we present a general overview of this work and potential extensions.

Loading and preprocessing a dataset for noise pollution analysis typically involves several steps. In this example, I'll outline a general process for loading and preprocessing a noise pollution dataset using Python and popular data science libraries like Pandas and NumPy. Keep in mind that the specific steps may vary depending on your dataset's format and the analysis you intend to perform. Here's a step-by-step guide:

1. **Import Necessary Libraries**: Start by importing the necessary Python libraries.

pythonCopy code

import pandas as pd import numpy as np

1. **Load the Dataset**: You can load the dataset from a file (e.g., CSV, Excel, or other formats) using Pandas. Use the **read\_csv** function for CSV files, adjusting the filename accordingly.

pythonCopy code

data = pd.read\_csv('noise\_pollution\_data.csv')

1. **Explore the Dataset**: Explore the dataset to get an understanding of its structure. You can use the following methods to inspect the data:
   * **data.head()** to view the first few rows of the dataset.
   * **data.info()** to get information about data types and missing values.
   * **data.describe()** for summary statistics.

pythonCopy code

data.head() data.info() data.describe()

1. **Data Preprocessing**: Depending on your dataset, you may need to perform various preprocessing tasks. Some common preprocessing steps include:
   * Handling missing values: You can use methods like **data.dropna()** or imputation techniques.
   * Data cleaning: Remove duplicates, outliers, or irrelevant columns.
   * Feature engineering: Create new features or transform existing ones.
2. **Feature Selection**: Select the relevant features (columns) for your analysis. For noise pollution, this might include time of day, location, noise level, etc.

pythonCopy code

selected\_features = data[['time\_of\_day', 'location', 'noise\_level']]

1. **Data Transformation**: Depending on your analysis, you might need to scale or normalize your data. For example, you can use the **StandardScaler** from Scikit-Learn to standardize numeric features.

pythonCopy code

from sklearn.preprocessing import StandardScaler scaler = StandardScaler() scaled\_data = scaler.fit\_transform(selected\_features)

1. **Split the Data**: If you plan to build a predictive model, split the dataset into training and testing sets. The common split ratio is 80% for training and 20% for testing, but you can adjust this based on your needs.

pythonCopy code

from sklearn.model\_selection import train\_test\_split X\_train, X\_test, y\_train, y\_test = train\_test\_split(scaled\_data, target, test\_size=0.2, random\_state=42)

1. **Analysis**: Finally, you can use the preprocessed data for various analyses, such as building predictive models or conducting statistical analysis.

Remember that the specific steps may vary based on your dataset and analysis goals. Noise pollution datasets can be quite diverse, so adapt the preprocessing steps accordingly.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | **Noise Identification** | | |
| *Ensemble* | *Similarity* | *Data Complexity* |
| **Noise Handling** | *Remove* | C45robustFilter  C45votingFilter C45iteratedVotingFilter CVCF  dynamicCF edgeBoostFilter EF  HARF INFFC IPF  ORBoostFilter  PF | AENN  BBNR CNN DROP1 DROP2 DROP3 ENG ENN PRISM RNN  TomekLinks | saturationFilter consensusSF classifSF |
| *Repair/ Hybrid* | hybridRepairFilter | EWF GE  ModeFilter |  |

## Calling the filters

When one wants to use a label noise filter in Data Mining applications, all we need to know is the dataset to be filtered and its *class* variable (i.e. the one that contains the label for each available instance). The **NoiseFiltersR** package provides two standard ways for tagging the class variable when calling the implemented filters (see also Figure [2](#_bookmark6) and the example below):

* The *default* method receives the dataset to be filtered in the x argument, and the number for the class column through the classColumn argument. If the latter is not provided, the last column of the dataset is assumed to contain the labels.
* The *formula* method is intended for regular R users, who are used to this approach when fitting regression or classification models. It allows for indicating the class variable (along with the attributes to be used) by means of an expression like Class~Attr1+...+AttrN (recall that Class~. makes use of all attributes).

Next, we provide an example on how to use these two methods for filtering out the iris dataset with

edgeBoostFilter (we did not change the default parameters of the filter):

# Checking the structure of the dataset (last variable is the class one)

* data(iris)
* str(iris)

|  |  |
| --- | --- |
| 'data.frame': | 150 obs. of 5 variables: |
| $ Sepal.Length: num | 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ... |
| $ Sepal.Width : num | 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ... |
| $ Petal.Length: num | 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ... |
| $ Petal.Width : num | 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ... |

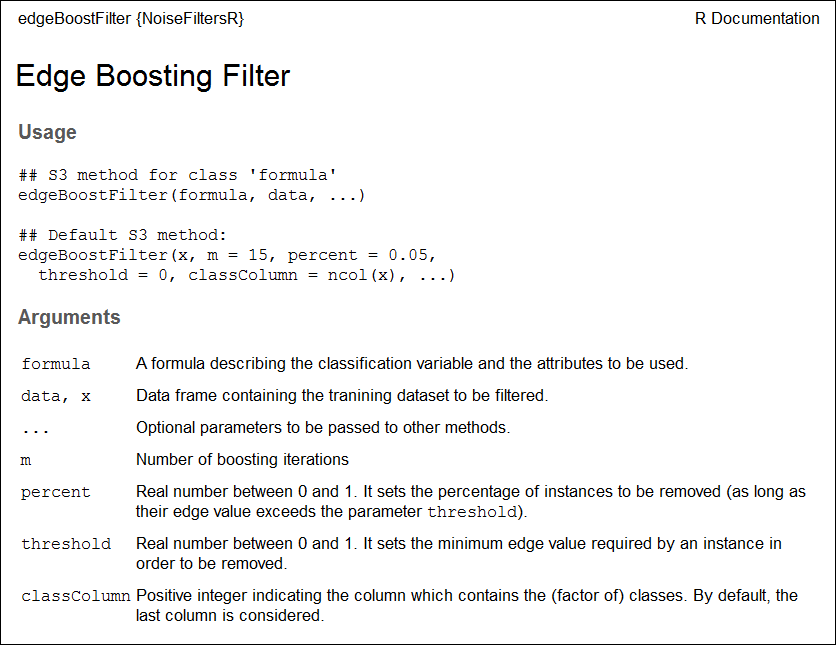
$ Species : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 ... # Using the default method:

* out\_Def <- edgeBoostFilter(iris, classColumn = 5) # Using the formula method:
* out\_For <- edgeBoostFilter(Species~., iris)

# Checking that the filtered datasets are identical:

* identical(out\_Def$cleanData, out\_For$cleanData)

[1] TRUE



**Figure 2:** Extract from edgeBoostFilter’s documentation page, which shows the two methods for calling filters in **NoiseFiltersR** package. In both cases, the parameters of the filter can be tunned through additional arguments.

Notice that, in the last command of the example, we used the $ operator to access the objects returned from the filter. In next section we explore the structure and contents of these objects.

**The** "filter" **class**

The S3 class "filter" is designed to unify the return value of the filters inside the **NoiseFiltersR**

package. It is a list that encapsulates seven elements with the most relevant information of the process:

Top of Form