Business Intelligence and Applications

ASSIGNMENT 3: DATA MINING WITH WEKA

May 14, 2023

Andrea Cardia
Xiana Carrera Alonso
Shradha Maria

Università della Svizzera italiana

Contents

1 Introduction

This report will document the process of data mining of the Electronic Component Dataset. As tools, we chose to combine Weka and Python, mainly focusing on the former but overcoming some of its limitations with the latter.

2 Attribute analysis

In this section we will do a descriptive analysis of the attributes of the dataset and of the relationships between them.

In Weka, loading the dataset immediately indicates that we have 9999 instances (i.e., electronic components) and 10 attributes (columns in the dataset).

For each of the attributes, we can see their type (numerical or nominal) and the number of missing entries, of distinct entries (how many different values are there for that attribute) and of unique entries (how many instances have a value for that attribute that is not repeated anywhere else in the dataset). The relevant data has been collected into Table 1.

Attribute	Type	Missing values	Distinct values	Unique values
impedance	Numeric	0 (0%)	9999	9999 (100%)
impedance_fluctuation	Numeric	0 (0%)	9999	9999 (100%)
applied_voltage	Numeric	1 (0%)	40	0 (0%)
output_current	Numeric	6 (0%)	9993	9993 (100%)
configured_resistance	Numeric	0 (0%)	105	2 (0%)
saturated	Nominal	0 (0%)	2	0 (0%)
max_force_applicable	Numeric	0 (0%)	9999	9999 (100%)
quality_index	Numeric	0 (0%)	9999	9999 (100%)
connector_type	Nominal	0 (0%)	3	0 (0%)
broke	Nominal	0 (0%)	2	0 (0%)

Table 1: Basic attribute information as shown in Weka.

The identified type of the variables is correct. The unique count is not of much importance to us, but the existence of missing values implies that we should later process them. The number of distinct entries is specially interesting for the nominal attributes, since it indicates the number of categories. For example, *connector_type* has three: type_a, type_b and other.

By default, Weka identifies the last column as the class. In this case, that is correct: we are trying to predict the values of *broke*.

For the numerical variables we also can see their minimum, maximum, mean and standard deviation; for the nominal ones we can see the count of each of their categories. Alternatively, Weka offers histograms (for the numerical attributes) and bar plots (for the nominal ones) of the data, where colors indicate the value of the class. Red corresponds to a component that did not break during testing and blue, to one that broke (figure 1).

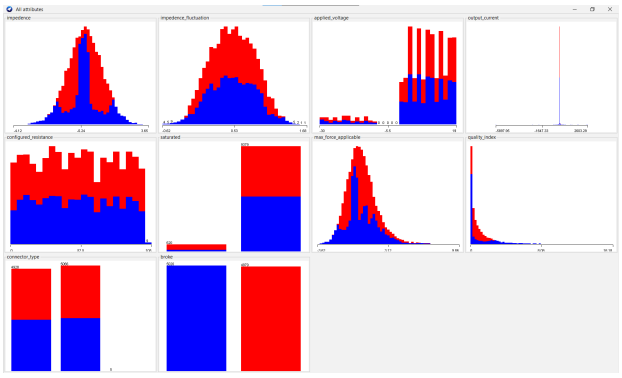

Figure 1: Plots of the distribution of the variables, generated by Weka.

There are some remarks that can be made here:

1. Output current is very concentrated around its mean, with the exception of a few, but very extreme outliers. Once removed (see the section on preprocessing), its distribution appears almost normal (figure 2).
2. Because of their shape, it seems plausible that *impedance* and *impedance_fluctuation* come from normal distributions.
3. Only 0.05% of the components are of the *connector_type* "other". The rest are almost equally distributed between types a and b (see figure 3).
4. The class, *broke*, is quite evenly distributed between 1 and 0, with 5020 and 4979 entries, respectively.
5. Only 6.2% of the components are saturated (figure 4).

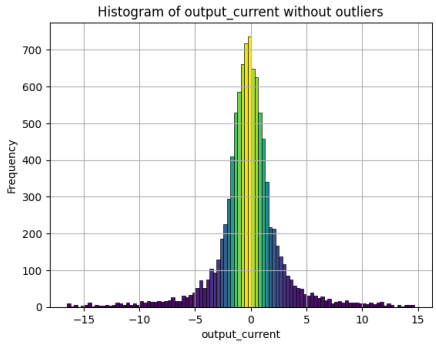

Figure 2: Histogram of *output_current* without outliers. Made in Python.

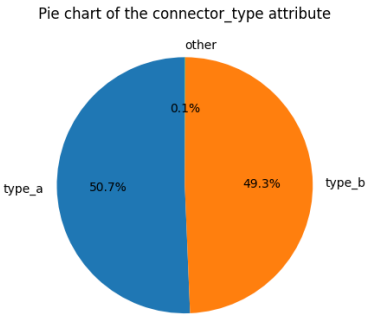

Figure 3: Pie chart with the distribution of *connector_type*. Made in Python.

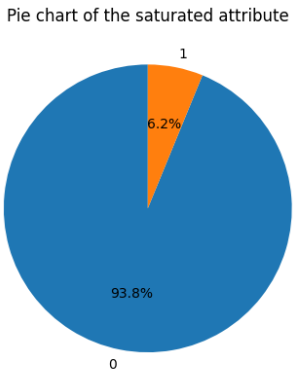

Figure 4: Pie chart with the distribution of *saturated*. Made in Python.

Using Python, we performed a Kolmogorov-Smirnov test to check if *output_current* without outliers, *impedance* and *impedance_fluctuation*, came from normal distributions. With a significance level of 5%, we only had statistical evidence in favour of rejecting this idea for the latter two. For *impedance*, the test gave a p-value of $0.65 > 0.05$, so we can assume that the attribute is indeed normally distributed.

2.1 Correlation between attributes

The "Visualize" tab on Weka generates a scatter plot for each pair of attributes, which can serve to quickly identify any potential strong relationship.

For example, we can see that the *quality_index* attribute appears to form a parabola with respect to *impedance* (figure 5). In fact, by fitting a quadratic polynomial to it in Python, we obtained that the polynomial $x^2 + 0.2x + 0.02$, where x is *impedance*, approximates really well *quality_index*. Indeed, the R^2 score (the proportion of the variance of *quality_index* explained by *impedance*) is very high: 0.999, which supports the validity of the quadratic model.

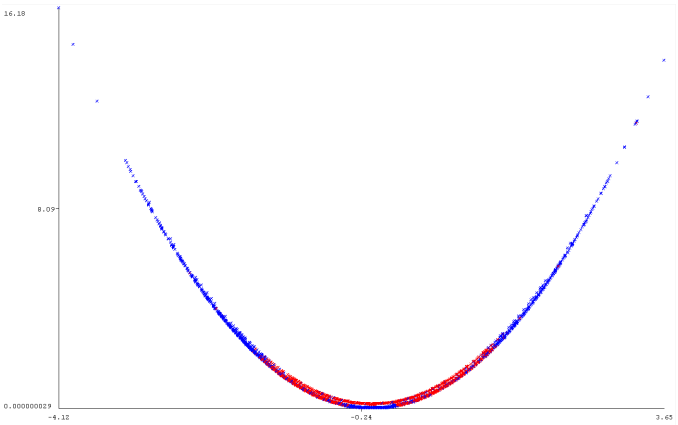

Figure 5: Scatter plot between *impedance* (x axis) and *quality_index* (y axis), generated by Weka. There is some extra jitter to add random noise to the data and better see overlapping points. Colors correspond to *broke*.

Weka also offers the possibility of calculating the correlation coefficients between the class and the rest of the attributes by choosing the method **CorrelationAttributeEval** from the "Select attributes" tab. However, we were able to create a simpler visualization using a heatmap in Python (figure 6).

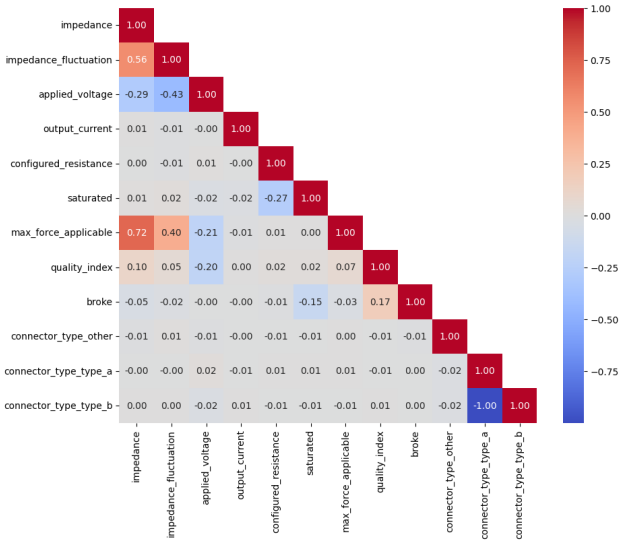

Figure 6: Heatmap of the correlation matrix of the variables. Created using Python.

Observe that in this heatmap, *connector_type* has been divided into 3 boolean variables, each one indicating being/not being of a certain type, so that all of them can be treated numerically. See the next section for more details.

Nonetheless, none of the correlation coefficients is too remarkable. The strongest one is -1, between the types a and b of *connector_type*, the reason being that the number of connectors of type "other" is extremely low, so it is almost always true that if a connector is not of type a, then it is of type b, and vice versa. That is, the variables are inverses of each other, which implies a perfect inverse relationship.

The only other correlation that we can consider significant is that between *impedance* and *max_force_applicable* (examined in a later section). None of the rest surpass the 0.6 mark, and most are really close to 0.

Interestingly, even though *impedance* and *impedance_fluctuation* appear directly related, the points in their scatter plot are too dispersed in a cloud-like manner to bear resemblance to a line (figure 7), and so the numerical result is lower than what could be expected.

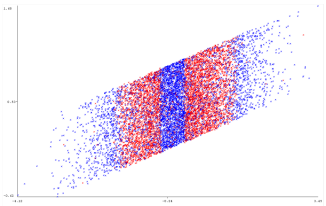

Figure 7: Scatter plot between *impedance* (x) and *impedance_fluctuation* (y), made in Weka.

3 Preprocessing

In this section we will describe the preprocessing and selection of attributes of the dataset.

We first decided to:

1. **Impute the missing values using the "Most Common" heuristic**, i.e., replacing them with the mean value of the attribute, for those that are numeric, or the most frequent value, for those that are nominal. The reason why we chose this instead of simply discarding the entries is that imputing gave slightly better results for accuracy when using classification methods in subsequent exercises. Because of the nature of the classification process, we cannot provide a specific reason for the change, although we have hypothesized that since the size of the dataset is not too big, losing entries has a bigger impact than altering them.
2. **Substitute the *connector_type* categorical variable with three binary variables**, one for each category, indicating whether a given component belongs or not to it. This facilitates working with the attributes, as we can treat them all as being numerical. For example, now it is possible to compute correlations for them.

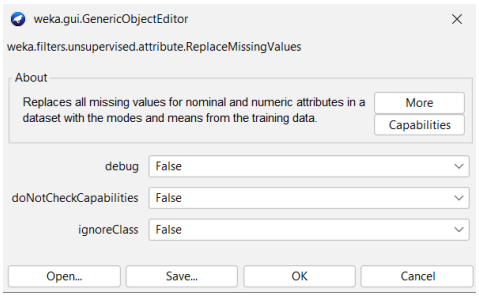

Figure 8: Filter used to impute missing values in Weka.

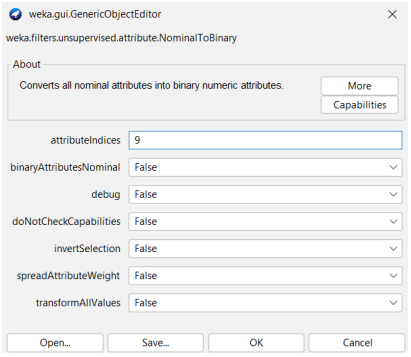

Figure 9: Filter used to replace *connector_type* with three binary variables.

For some time we considered removing the extreme values of *output_current*, or replacing them with the mean value. Since they are quite rare, we had hypothesized that they could be due to errors in the measurements or to faulty components. However, when we compared the statistical properties of these components to the ones of the whole dataset in Python, we noticed that they were very prone to breaking. For example, if we select the components that are below the 0.02 quantile and those above the 0.98 quantile, we can see that 98% have broken. In contrast, only 50.2% of the components of the

whole dataset broke. This huge difference was shown to help the classification process: not deleting nor altering these entries increased the accuracy of the algorithms. Therefore, we decided to maintain them as they are.

Af first, we also considered replacing with the mean or removing the highest values of *quality_index*. Their scarcity and strange distribution was difficult to interpret, and so discarding them seemed natural. However, once we found the quadratic relationship of *quality_index* with *impedance* in the descriptive analysis, the appearance of those values was explained: they were just (approximately) the squared value of the highest *impedance* scores. Furthermore, removing or altering them diminished the accuracy of the classification algorithms (probably because of the strong relationship of this variable with the class). Thus we decided to leave them as they were.

3.1 Attribute selection

Not all of the attributes are significant to the study of components that might break. In general, working with a reduced number of variables helps with explainability (for example, it is more difficult to explain the effect that a variable that has no relationship with the class might have, than one that has a clear correlation with it). Additionally, it reduces the computational effort and amount of the noise in the data, which decreases the probability of overfitting and might improve performance in classification.

Weka has a tab dedicated to the selection of relevant attributes, with different heuristic possibilities. After the previous preprocessing, we used:

1. **CorrelationAttributeEval** with the **Ranker** search method, with default parameters (i.e., not discarding any attribute). The variables are ranked according to the strength of their Pearson correlation with *broke*.

2. **InfoGainAttributeEval** with the **Ranker** search method, with default parameters. The information gain estimates how much each attribute reduces the entropy of *broke*, that is, how much they diminish the uncertainty in the class. The score for a variable is higher the more it reduces *broke*'s entropy.

3. **WrapperSubsetEval** with a **J48** classifier and the **GreedyStepwise** search method. This method trains a learning algorithm (in this case, the decision tree classifier **J48**) and evaluates its performance on different subsets of the attributes. The default parameters were used: **GreedyStepwise** performs a forward search, **J48** has a 0.25 confidence factor, etc. Note that **WrapperSubsetEval** cannot be used with **Ranker**, since it does not work on individual variables, and so we chose **GreedyStepwise** instead of **BestFirst** to obtain more accurate results at the cost of execution time.

All of these methods were evaluated with a seed of 1 and a cross-validation of 10 folds. That is, the dataset was divided into 10 subsets and, for each of the 10 executions done for the method, a different subset was used as the test data, with the other 9 being the training data. The results can be seen in table 2.

Position	By correlation	By information gain	With J48
1	quality_index	quality_index	impedance (100%)
2	saturated	impedance	output_current (100%)
3	impedance	output_current	configured_resistance (100%)
4	max_force_applicable	max_force_applicable	quality_index (100%)
5	impedance_fluctuation	saturated	saturated (50%)
6	connector_type=other	impedance_fluctuation	connector_type=type_a (20%)
7	configured_resistance	configured_resistance	applied_voltage (10%)
8	applied_voltage	connector_type=type_b	max_force_applicable (10%)
9	connector_type=type_b	applied_voltage	connector_type=type_b (10%)
10	connector_type=type_a	connector_type=type_a	impedance_fluctuation (0%)
11	output_current	connector_type=other	connector_type=other (0%)

Table 2: Results of the attribute selection process in Weka. For **J48**, the attributes are ordered according to the percentage of folds in which they were selected (that appears between parenthesis).

In Python, we performed a similar analysis using the `SelectKBest()` feature selection test of *scikit-learn* with an F-test for classification. This method selects the attributes with the best score according to their linear dependency with respect to *broke*. The results indicate that the first variables to eliminate, in order, should be *output_current*, *connector_type=type_b*, *connector_type=type_a*, and *applied_voltage*.

The ranking for each attribute varies depending on the method, but *applied_voltage* and all the three connector types consistently get bad results. Additionally, since "type b" is almost the complete opposite of "type a", and there are very few "other" instances, we consider that it is enough to maintain only "type a" ("type b" could also have been chosen, but "type a" appears to rank higher in general). Consequently, we have decided to remove *applied_voltage*, *connector_type=other* and *connector_type=type_b*, which can be done directly in Weka by using a **Remove** filter.

This choice was later confirmed as adequate: for example, many algorithms' performance is not affected by the inclusion or elimination of these variables.

4 Dependency of impedance with the maximum force applicable to the electric component

In this section we will describe our findings for the relation between *impedance* and *max_force_applicable*.

We started by creating a scatter plot between the two. Using the `regplot()` method of the *seaborn* library in Python, we can create a scatter plot and add a best-fit regression line to it (figure 10).

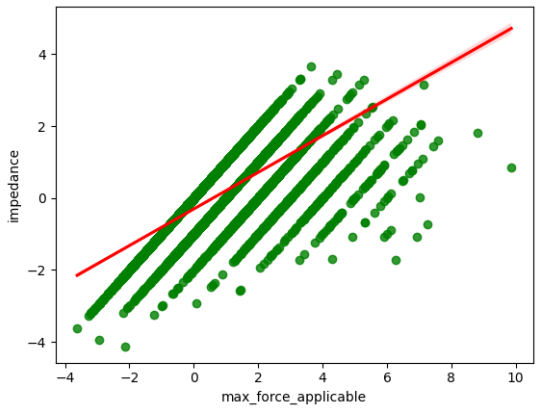

Figure 10: Scatter plot between *impedance* and *max_force_applicable* with a best-fit regression line, made in Python.

Since the scatter plot showed a set of parallel, diagonal lines, we ventured that there could be different subsets in the data, each of them corresponding to a line and showing an almost perfect linear relation between the variables for that subset.

Even if a simple regression line cannot explain this behaviour, we first wanted to check if there was linearity, for which we used the function `linregress()` of *scipy.stats*. The p-value of the model was 0, which implies that the hypothesis that there is no correlation between the variables can be rejected. The slope of the line would be 0.51. Since it is positive, the relationship would be direct: the greater the maximum force applicable, the greater the impedance, and viceversa.

We also had already seen that the correlation coefficient is 0.72. It is relatively high, so the strength of the linear relationship is quite strong.

When trying to fit polynomials of degree 2, 3, 4 and 5, the root mean squared error (RMSE) scores were slightly lower than that of the linear model (the minimum was 0.5858 for a polynomial of degree 4, whereas the linear model had 0.6778), but they clearly do not emulate the parallel lines behaviour either, so none of them is too adequate.

We later examined the scatter plot available in Weka using colours to represent a third attribute

(see figure 11 for an example). We expected to find one whose values were clearly different for each of the diagonal sets of points, since that would indicate that there are subgroups originating from that third attribute. However, all of them were quite uniform across the sets. Even when we isolated the points from some of the lines and compared their statistical properties to those of the whole dataset in Python, no clear difference was apparent.

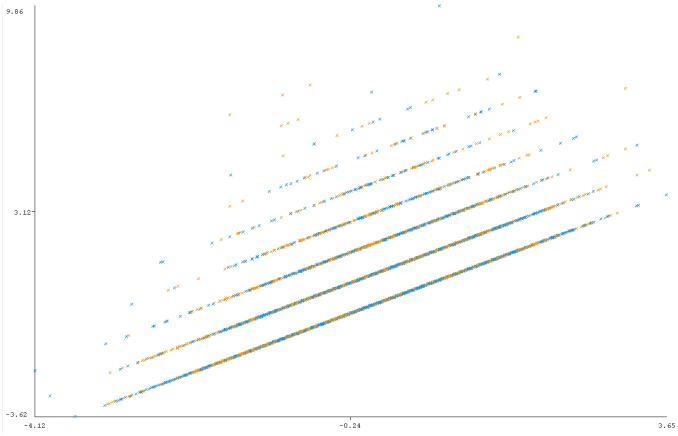

Figure 11: Scatter plot between *impedance* and *max_force_applicable* with colours representing the attribute *connector_type_type_a*. No relationship is apparent.

Although we have found no proof of a cause for the existence of subgroups, we also have found no evidence against this hypothesis, and so we cannot reject it. We conclude that the two variables have a relevant direct correlation and that there exist subgroups of components for which the impedance increases with respect to the maximum force applicable in an almost perfect linear way, but that the reason that causes a particular component to belong to one group or another is complex and may even arise from phenomena not reflected on this dataset.

5 Classification

In this section we will examine the usage of different classification algorithms in order to predict whether a particular component broke during testing or not. The analysis was performed using only Weka, as we have found that it offers sufficiently varied options with a practical interface and a complete result description.

All computations used cross-validation with 10 folds, and default parameters.

To have some lower bounds for the accuracy and error metrics, we started by applying the **ZeroR** and **OneR** algorithms.

ZeroR always predicts the most common value for the class. Since *broke* is a nominal attribute, **ZeroR** predicts the mode: 1. Thus, the accuracy (the percentage of correctly classified instances, 50.205%), is equal to the percentage of values that broke during testing. The RMSE, which can serve to compare algorithms, was 0.5. Recall (the ratio between true positives and the sum of true positives and false negatives) was 1 for the broken instances (because there are no false negatives), and 0 for the non-broken. Their weighted average is 0.502 (the ratio of correctly classified instance). Precision (the ratio between true positives and the sum of true positives and false positives) was 0.502 for the broken instances, too. It is not computable for the non-broken ones because the denominator equals 0.

OneR selects one attribute as the best predictor for the class. First, for each value of all attributes, it computes the most frequent class value, and records both the rule *attribute value* $\implies$ *most frequent class value* and the number of errors obtained using that rule. Then, it computes the total error for each attribute as the sum of the errors of all of its values. Finally, it selects the column with the smallest total error. It will do all the predictions using the rules associated with it. Note that, for this procedure, all attributes are assumed to be nominal (so numerical ones must be discretized).

OneR selected *quality_index* as the predictor, and correctly classified 81.0281% of the instances, with a RMSE of 0.4356, a precision weighted average of 0.821, and a recall weighted average of 0.810.

Afterwards, we tried to select diverse algorithms, with at least one of each type. Table 3 shows a summary of the results of a selection of the algorithms.

Algorithm	Accuracy	Precision weighted average	Recall weighted average	RMSE
ZeroR	50.205%	-	0.502	0.5
OneR	81.0281%	0.821	0.810	0.4356
BayesNet	91.2591%	0.914	0.913	0.2694
MultilayerPerceptron	82.8483%	0.845	0.828	0.3585
VotedPerceptron	55.3355%	0.554	0.553	0.6681
Bagging	95.5996%	0.956	0.956	0.1948
MultiScheme	50.205%	-	0.502	0.5
JRip	94.9195%	0.949	0.949	0.2174
DecisionStump	70.9871%	0.804	0.710	0.4314
HoeffdingTree	87.8488%	0.879	0.878	0.3087
RandomForest	95.9196%	0.959	0.959	0.1924

Table 3: Summary of the results of a selection of classification algorithms.

We started with **BayesNet**, a Bayesian Network classifier ([2]), which is a generalization of Naive Bayes. It is a probabilistic model that represents the variables as nodes in a directed acyclic graph, where an edge represents a causal relationship between variables, which have associated conditional probabilities. The classification is done by computing the joint probabilities while traversing the graph. Its performance greatly improved that of Naive Bayes.

The results showed that a refined algorithm does not necessarily imply a good performance per se. For example, a **MultilayerPerceptron** (a neural network) barely surpassed **OneR** in accuracy, with 82.8483%, and **VotedPerceptron** was even worse. These bad results are probably due to the huge dataset sizes required to adequately train neural networks.

We tried all of the metaheuristic algorithms, and found that, with the default parameters, **Bagging (Bootstrap Aggregation)** offered the best results, with a very high accuracy (95.5996%). It creates multiple random samples from the training data, with replacement ([1]), and trains an independent classification algorithm on each (by default, a **REPTree**, which is a standard decision tree). A prediction for a new instance is made by averaging the votes of the classifiers.

Other meta algorithms offered very poor results because they were wrapped around **ZeroR** by default, and obtained performances similar to it (for example, **MultiScheme** classified correctly 50.205% of the instances). Lazy schemas (which postpone the construction of the model until the prediction phase) and the miscellaneous methods also performed poorly.

All the rules algorithms different than **ZeroR** and **OneR** significantly increase their performance, with all having accuracies of more than 90%. The best one was **JRip (Repeated Incremental Pruning to Produce Error Reduction)**. This method iteratively generates IF-THEN rules over the data using a divide-and-conquer strategy, and pruning the rules that do not satisfy quality requirements.

Among the trees, the performance was varied. For example, **Hoeffding trees** did not get particularly good results, probably because they are designed to analyze large data streams, a different type of data set. They did not reach an accuracy of 90%. A **DecisionStump**, a decision tree with only one internal node and two leafs, is too simple and did not surpass **OneR**. In contrast, **RandomForest** obtained the best metrics out of all the methods tried, with 95.9196% correctly classified instances, and a precision and recall weighted average of 0.959. **RandomForest** generates some samples from the data, with replacements, and trains a decision tree for each one, using a random subset of the attributes. Prediction is done by averaging the votes of the trees.

In general, it appears that rules and decision trees (and metaheuristics wrapping them) have worked particularly well, probably because they need less data to be trained than more advanced options such as neural networks and/or because *broken* is better explained with clear decision boundaries in our features, instead of by complex patterns, for which neural networks and special methods excel. Other algorithms may have failed due to not having a proper hyperparameter tuning, such as in the case of metaheuristics based on **ZeroR** by default.

6 Hyperparameter tuning

In this section we will describe the exploration of the hyperparameters of the best methods used, with the goal of improving the quality of the classification process.

The algorithms that had obtained the highest accuracies in the previous section were **RandomForest**, **Bagging** and **JRip**, so we will focus on those.

Most of the parameter tuning only slightly affected the performance, and sometimes it even decreased it. One possible explanation is that Weka already selects very good parameters for each algorithm by default, and so tweaking them does not ensure a better result.

For example, the default number of trees in **RandomForest** is 100. If we increase it to 300, the accuracy decreases to 95.9096%. If we use 500 trees, the value increments to 95.9296%. In any case, the difference is too small to be considered relevant: the changes only correct the classification of 1 or 2 instances.

Another possibility is to activate "computeAttributeImportance", which causes the method to use Gini Importance, also called Mean Decrease in Impurity (MDI), to select the variable with which to split a node in a tree. Gini Importance takes into account the number of times that the attribute has been chosen, whereas the default option is based on the accuracy decrease. However, maintaining 500 trees and changing this option did not alter the results.

Reducing the size of the subset used to build each tree (bagSizePercent), expressed in a percentage with respect to the size of the dataset, did not show relevant alterations. Very extreme values, such as 10%, slightly reduced the accuracy (94.8795% with 100 trees).

Weka selects the depth of the trees as unlimited by default, but it can be changed to a fixed maximum number of levels. This can severely impact the results: a maximum depth of 4 gives a 91.8892% of accuracy, showing that the complexity of the model demands many more internal nodes.

The best recorded result was actually just a matter of luck: changing the seed to 30 and maintaining the rest of the parameters as the default ones causes accuracy to increase to 96.0096%.

We then tried to tweak **Bagging**. We started by only changing the underlying classifier to **RandomForest** (with the default parameters). The execution was very slow, because of the complexity of the method, but accuracy was incremented to 95.9696%. Using a simpler but very popular decision tree algorithm, **J48**, resulted in 95.6796%, and in 95.8996% when the number of iterations was increased to 50.

JRip also easily showed improvements. By incrementing the number of optimizations and the minimum weight of instances in a rule, we found a peak at 6 and 12, respectively, obtaining a 95.2495% of accuracy. Further increasing the parameters resulted in slightly worse results.

Our conclusion is that some algorithms have potential for improvement with a good choice of hyperparameters, like in the case of **JRip** and **Bagging**. In others, such as **RandomForest**, this process is very delicate and does not ensure an improvement. However, in general Weka has very good choices as the default options, and so if an algorithm already has a good performance, hyperparameter tuning will not alter it significantly.

Bibliography

[1] IBM. *What is bagging?* URL: `https://www.ibm.com/topics/bagging`. (accessed: 14.05.2023).

[2] Parviainen P. *Bayesian networks.* URL: `https://www.uib.no/en/rg/ml/119695/bayesian-networks`. (accessed: 14.05.2023).