Second Assignment

Sergi Carol and Balbina Virgili

April 23, 2018

Introduction

The aim of this second assignment is to improve the knowledge acquired during our first assignment. To be able to do this aim, we have decided to perform a comparison between *KNIME* and *RapidMiner*, which is different tool that also helps us to implement the algorithms that we used during the first assignment. This way, we also expect to extend our knowledge of the implemented algorithms, which are the *Decision tree* and *Naive Bayes*.

We hope to provide meaningful explanations about the difference between the two tools and give more insight into the implementation of the before mentioned algorithms.

This document begins with an small explanation of *Rapidminer* tool and an explanation of our developed workflow. It continues with an explanation of the algorithms implemented with a discussion of the different results obtained with both tools. Finally, this document ends with some conclusions of the results obtained during the development of the assignment.

1 RapidMiner

Rapidminer [1] is a tool similar to *Knime*, which is used to perform various data science, machine learning, and data mining tasks. As with *Knime*, it performs its duties using a *GUI* in which a user can drop various *operators* (similar to *nodes* in *Knime*) and interconnect them in order to produce a workflow for the aim the users want to achieve. Rapidminer also offers an *AutoModel* tool which allows the creation of simple workflows.

It also offers the possibility of creating various processes and interconnect them, unfortunately, *Rapidminer* does not allow to extract the same output to two or multiple different nodes. That is why in our case In our case we have used **2** different processes, one for each algorithm used.

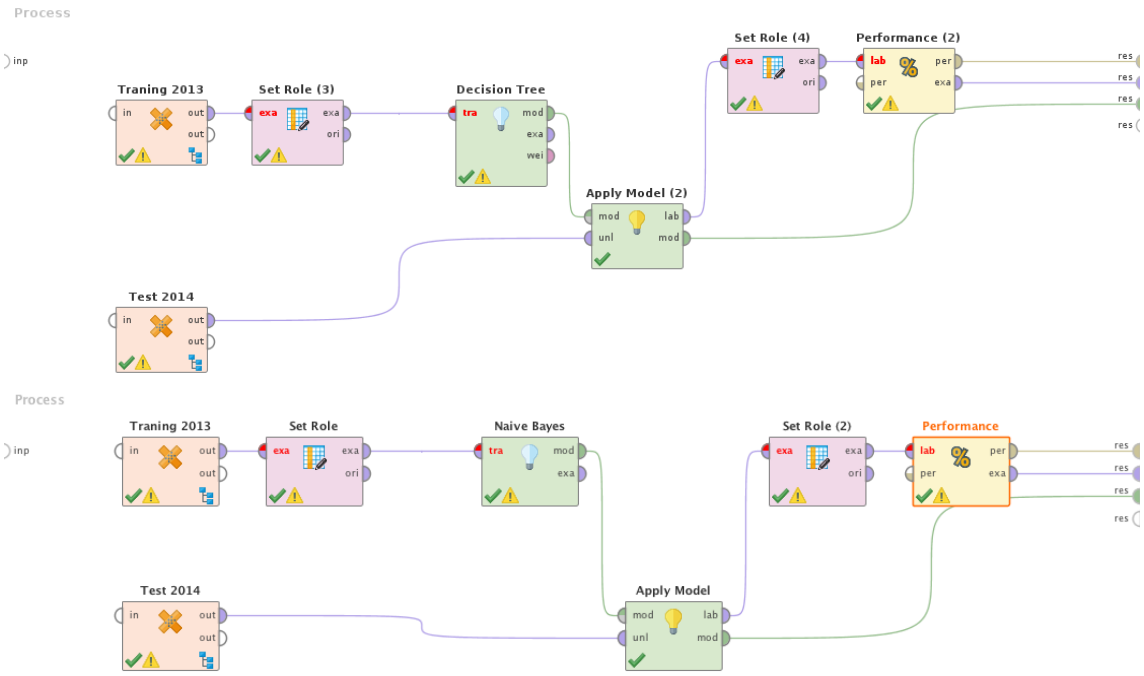

Figure 1: Final workflow implemented with RapidMiner

In our implementation, we begin with two own-costumed operators which are *Traning 2013* and *Test 2014* which are the composition of other operators. We needed to create them, due to the fact that *RapidMiner* does not have any operator to cache data to re-use it in other iterations without doing previous calculations. This became a problem since the pre-processing took quite a bit of time to be executed and we were losing a lot of time between iterations each time that we wanted to re-execute the workflow. As such, the operator consists of two different parts, one that tries to read a file already pre-processed, while the other would be executed in case the pre-processed file does not exists, which then it will run the *R* script and save the new processed file.

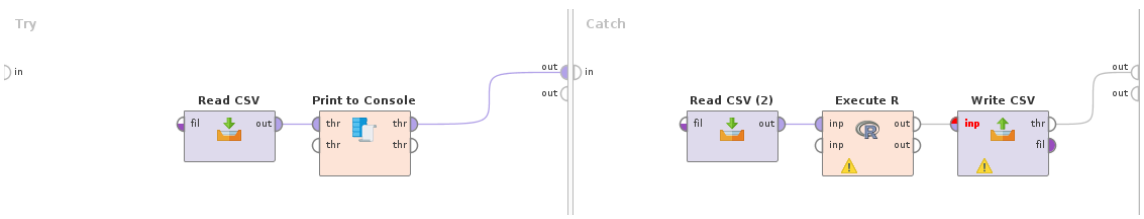

Figure 2: Training and Test costumed modules implementation

With this we were able to overcome the lack of a cache operator.
After reading and pre-processing the file we can start to apply the different algorithms used and check their performance.

Before doing it, first, we would like to make a general comparison between the used tools, *Knime* and *RapidMiner*. It seems that *Knime* is a more intuitive tool to use and that less operators (or nodes) are needed to perform the same task, which helps newcomers a lot when starting with a new program. For example, before an operators performs a task in *RapidMiner* a *Select Role* operator is needed, while in *Knime* it is possible to select the attribute directly into the task node. We also found that the operators are better done in *Knime*, not their implementation per se, but their actual feel and look is much better, and more intuitive in *Knime* rather than *RapidMiner*.

2 Pre-processing

As with the previous assignment, we must do some pre-processing with our data. This pre-processing involves creating more rows, as many times as the *count* value indicates for a given row. To do so, we decided to try and use the *R* [2] extension for *RapidMiner*. Since we are not so proficient with the *R* language, we thought it would be a good exercise to try to improve our knowledge doing the pre-processing script in *R*, instead of directly use the one that we had already developed in *Python* during the first assignment.

RapidMiner, just as with *Knime*, offers an extension browser to find and install complements into *RapidMiner* [3], in order to install an extension, in our case the R scripting extension, into the *Rapidminer* environment we have to click on the *Extensions* menu, go to *Marketplace* and then a new pop-up will appear with all the extension available for the user. In our case we can just search for the *R* extension and install it.

The final code developed for obtaining our desired pre-processing behavior, is showed just below.

```
rm_main = function(data)
{
        end <- nrow(data)
        dataOutput <- data
        for (i in 1:end){
                count <- data[i,]$Count
                for (j in 1:count){
                        dataOutput <- dplyr::bind_rows(
                                        dataOutput, data[i,])
                }
        }

        return(dataOutput)
}
```

In this case, it is worth mentioning a few things, first the main function must be called **rm_main** and the output will be the one returned from this function, there can also be as many inputs as necessary as long as the parameters of the function are updated accordingly.

It is also worth noting that we can use the library *dplyr* because it has already been installed in our system, the extension uses the runtime environment of our local system, and as such any packages already installed can be used and no need any special addition of installation or loading.

3 Decision Tree

The first algorithm that we have implemented is the *Decision tree algorithm*, with the decision tree we wanted to compare the performance between the *Knime* version and the *Rapidminer* version. Our first iteration of the algorithm gave us a really bad performance, **44.5%** accuracy. Naturally we were quite surprised with this, *Knime* was nearly giving us double the performance than *Rapidminer*. After a bit of discussion we came up with two possible reasons for why the performance dropped.

- The pre-processing does something different than the *Knime* version.
- The default values of our decision tree give a bad performance for our dataset.

We decided that the most plausible reason was the tunning of the algorithm, as such we started to tune the parameters of the algorithm. We thought that this is the more plausible because the outputs of both preprocessing (the *Rapidminer* and the *Knime*) appeared to be identical.

3.1 Tunning of the decision tree

Our first tunning was to disable both pre-pruning and post-pruning [4] which are techniques to not perfectly fit (avoid *Underfitting* and *Overfitting*) a model using the training data, in our case this is not necessary since we use the data from a previous year in order to train the model and the data from the next year to test the model. By actually applying the pre and post pruning we were reducing the amount of data for our training data.

The next approach was to change the *Quality measure* in which we switched from *gain ratio* to *gini index*.

- **Information gain:** The entropies of all the Attributes are calculated and the one with least entropy is selected for split. This method has a bias towards selecting Attributes with a large number of values.
- **Gini ratio:** A variant of information gain that adjusts the information gain for each Attribute to allow the breadth and uniformity of the Attribute values.
- **Gini index:** A measure of inequality between the distributions of label characteristics. Splitting on a chosen Attribute results in a reduction in the average gini index of the resulting subsets.
- **Accuracy:** An Attribute is selected for splitting, which maximizes the accuracy of the whole tree.
- **Least Square:** An Attribute is selected for splitting, that minimizes the squared distance between the average of values in the node with regards to the true value.

In our case we can not use the *Leas Square* criteria since it requires that the predicted variable to be nominal, and in our project it is not.

We executed the remaining criteria to see which one would give us a better performance, in our case we expected *Information Gain* and *Gini index* to be the better ones. This is due to the fact that the gini index tries to split the decisions as best as possible regarding the mixture of the different classes in the model, meaning that a perfect separation of the classes would give a *Gini index* of 0 [5]. While *Information Gain* is more based on the decrease of entropy after splitting the dataset, meaning that constructing the decision tree is about finding the attribute that gives more information. [6]

	Information Gain	**Gini Ratio**	**Gini Index**	**Accuracy**
Accuracy	73.59%	59.97%	75.43%	50.11%

We can see how the *Gini Index* criteria gives us more accuracy by an small margin, as such this is the criteria that we have chosen for our decision tree. With this change our accuracy improved

to up to **75.43%** accuracy. We can take a closer look at the confusion matrix resulting from the decision tree in the figure below.

accuracy: 75.43%

	true BLACK NON HISPANIC	true HISPANIC	true WHITE NON HISPANIC	true ASIAN AND PACIFIC ISLA...	class precision
pred. BLACK NON HISPANIC	4704	336	0	1182	75.60%
pred. HISPANIC	0	15251	4146	140	78.06%
pred. WHITE NON HISPANIC	0	7856	24162	0	75.46%
pred. ASIAN AND PACIFIC ISL...	2029	0	150	4503	67.39%
class recall	69.86%	65.06%	84.90%	77.30%	

Figure 3: Confusion matrix decision tree

From the confusion matrix we can see that the Hispanics names usually get confused with the White non Hispanics names, which might mean that they are actually quite similar, still out accuracy in this case is up to 78%. We can see this more clearly by plotting an histogram of the predictions vs the actual ethnicity of the babies.

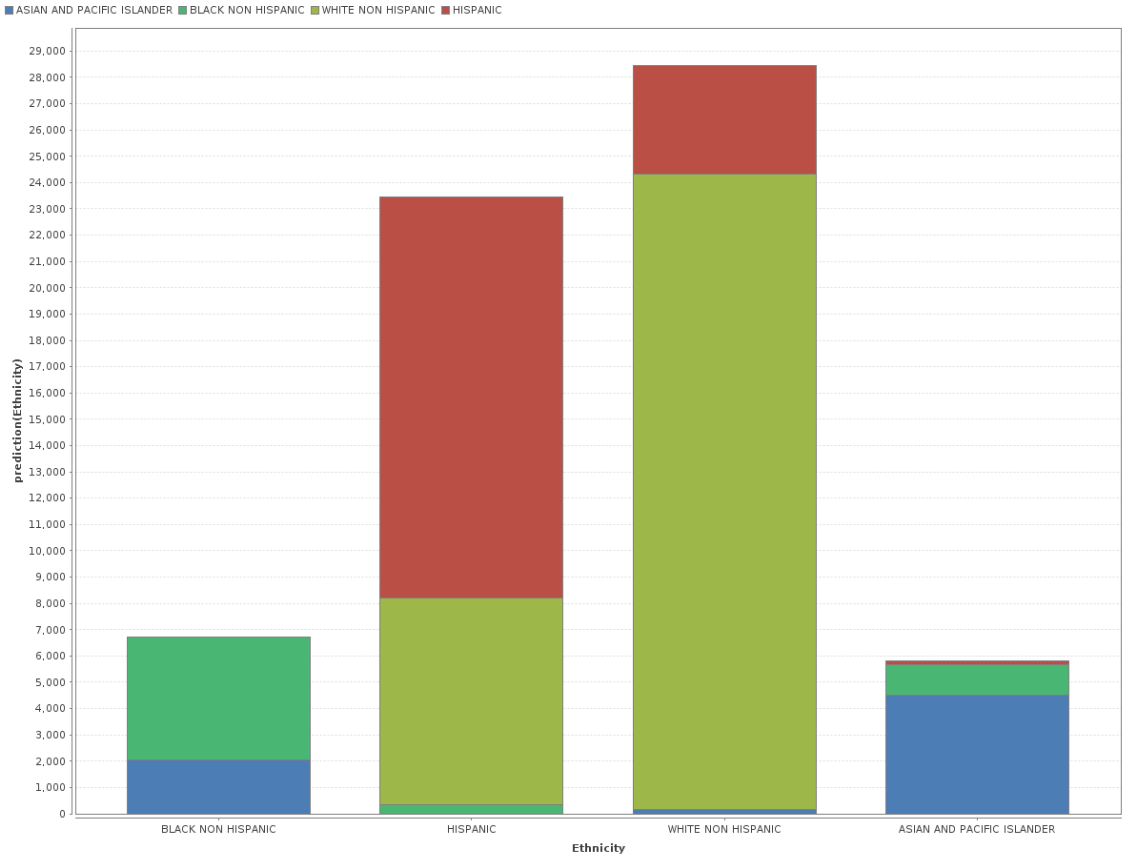

Figure 4: Histogram prediction for decision tree

It is clear how the ethnicity prediction for the *Hispanic* ethnicity gets quite confused with the *White non Hispanic* names.

4 Naive Bayes

The second algorithm that we have implemented is the *Naive Bayes algorithm* because, as we have already done with *Decision tree algorithm* on *Section 3*, we want to see the differences retrieved between the *Knime* version and the *Rapidminer* version.

After executing the first iteration of the algorithm, we have obtained a surprisingly accuracy of **64.76%**. With this value, we realized that *Rapidminer* has obtained a much higher accuracy in relation of that we obtained using *Knime*.

As we already checked during the last section that the pre-processing obtains the same results for both implementations, we assumed that the default values for our *Naive Bayes* implementation were different from the ones that we defined on *Knime* scenario. As a consequence, the new results obtained on this scenario have considerably increased. To be able to see the causes for the result's differences, we decided to start tuning the parameters of the algorithm.

4.1 Tunning of the Naive Bayes

After analyzing the advanced parameters that *Naive Bayes* operator provide, we realized that there is just one, the *Laplace correction*[7].

Laplace correction helps to avoid one of the principal weakness that Naive Bayes includes, which take place when within the training data a given attribute value never occurs in the context of a given class. As a consequence, when this value occurs is multiplied together with other probabilities, the results become misleading. That is why, Laplace correction adds a default probability to each count to avoid the occurrence of zero values.

Then, we decided to compute the implemented scenario with, and without Laplace correction option. The results obtained are showed below. As we can see, there is just nearly insignificant difference between both results.

	With Laplace Correction	**Without Laplace Correction**
Accuracy	64.76%	64.70%

As there was not much else to configure with the given options of Rapidminer ,because it was just a tick of use or not to use it, we decided that it could be a good idea to investigate and play with different values of the Laplace correction of KNIME. This way, we could be able to find out the default value that Rapidminer uses as Laplace correction.

To do it, we re-executed the scenario that we created during the first assignment, changing the value of the parameter *Default probability* of the node *Naive Bayes Learner*.

Default Prob. Knime	**0.0**	**0.0001**	**0.0002**	**0.001**	**0.2**	**1.0**
Accuracy	44.15%	65.48%	65.10%	55.04%	24.71%	24.69%

We executed the test with different values, extreme and middle-ones in order to verify how it effects the final accuracy of the test. With the results retrieved, we realized that the extreme values, 0 or 1, were obviously not good and that the best accuracy result is the one with the lowest value, but always higher than 0.

That is why we believe that *Rapidminer* must use the approximate value of 0.000001 for performing the Laplace correction, when indicated. The default value defined is a good one compared with the 0.01 that *Knime* has.

Once we have been able to verify and understand the differences on the results obtained from each tool, we are going to take a look to the new confusion matrix with the improved accuracy.

accuracy: 64.70%

	true BLACK NON HIS...	true HISPANIC	true WHITE NON HIS...	true ASIAN AND PAC...	class precision
pred. BLACK NON H...	3843	2457	722	1299	46.18%
pred. HISPANIC	1508	14517	5868	1218	62.81%
pred. WHITE NON H...	819	5467	20945	911	74.43%
pred. ASIAN AND P...	563	1002	923	2397	49.07%
class recall	57.08%	61.92%	73.60%	41.15%	

Figure 5: Confusion matrix for Naive Bayes

From the confusion matrix we can see that the Hispanics names usually get confused with the White non Hispanics names, which might mean that they are actually quite similar. Also, all other ethnics also tend to predict Hispanic in many cases that it predicts a wrong result, which might be a sign that Hispanic names are common on all ethnicities.
We can also see this more clearly by plotting an histogram of the predictions vs the actual ethnicity of the babies.

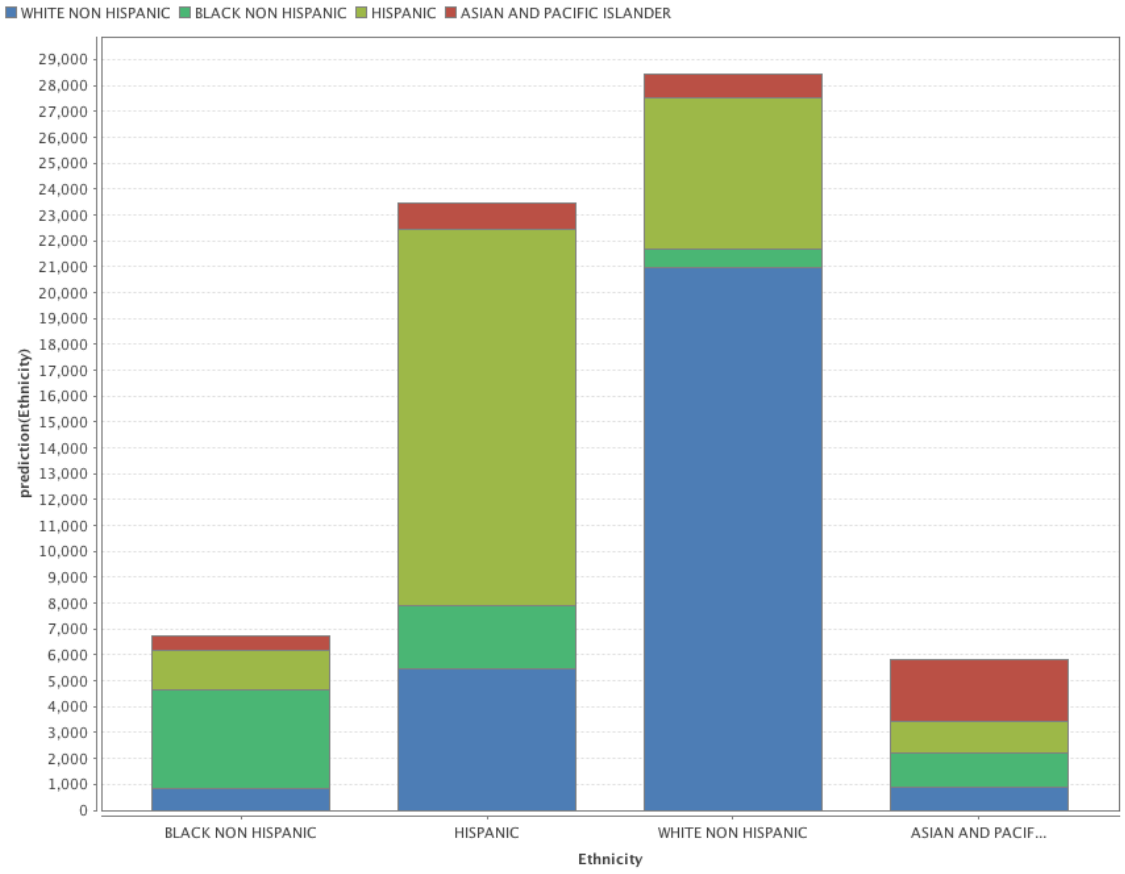

Figure 6: Histogram prediction for Naive Bayes

With the histogram showed above, it is clear to see that Asian and pacific islander is the worst predicted ethnic.

5 Conclusions

After finishing the whole process described on the above developed sections, we can conclude that *KNIME* is a more easy and intuitive tool for newcomers than *Rapidminer*. We can base our assumption on the following reasons:

- Modules on Knime are fully ready-to-use and no additional nodes are needed for implementing them. This means that each time you want to apply a model a node to select the label must be used before the model node.
- Both of them are suitable to create new nodes with different standard languages.
- Knime contains an already-implemented node *cache*, which is very useful to keep data obtained from time-costly operations. This way, they just need to be re-executed when needed and not each time that the scenario is executed.
- Knime allows changing the float value of Laplace correction when Naive Bayes is performed, while Rapidminer just allow a tick to use or not to use it but not further configuration is permitted.
- Rapidminer does not have basic nodes of results visualization. It is true that it has different options on 'Results' section but we think that Knime has more accurate and complete range of nodes of results representation.

Regarding the creation of scripts with R, we think that it has been helpful for us to increase our knowledge on this language, in particular, on adding new rows on already created *dataframes*. Furthermore, we believe that the procedure of adding costumed modules with different languages is very similar but it is needed to well-know the language because there is no test environment before compilation.

Moreover, we have also been able to deeply understand Naive Bayes and Decision Tree predictors, by checking their properties, configuration, different behaviors and accuracy of the predictions made for our chosen dataset. Although being considered as simple classification predictors, their fully-knowledge must be acquired for being able to obtain the most accurate and precise results.

Thanks to the deep knowledge acquired during this second assignment, we have been able to increase the accuracy of the prediction performed with both, Naive Bayes and Decision Tree predictors. And, additionally,our objective to know the reason why the initial results on both data mining tools were different for the a priory same implementation.

To sum up, we have decided that for our dataset the best configuration of each predictor is the one showed below.

- **Decision Tree predictor**

  Disable both pre-pruning and pruning.

  Use gini index as the quality measure.

- **Naive Bayes predictor**

  Set the default value of Laplace correction to 0.002.

References

[1] Admin. *Lightning Fast Data Science Platform*. 2018. URL: https://rapidminer.com/.

[2] *R Extension*. URL: https://marketplace.rapidminer.com/UpdateServer/faces/product_details.xhtml?productId=rmx_r.

[3] RapidMiner GmbH. *Adding Extensions to RapidMiner*. URL: https://docs.rapidminer.com/latest/studio/installation/adding-extensions.html.

[4] *Overfitting in a decision tree*. URL: http://www.saedsayad.com/decision_tree_overfitting.htm.

[5] *How To Implement The Decision Tree Algorithm From Scratch In Python*. 2018. URL: https://machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/.

[6] *Decision Tree classifier*. URL: http://www.saedsayad.com/decision_tree.htm.

[7] *Naive Bayes - RapidMiner Documentation*. 2018. URL: https://docs.rapidminer.com/latest/studio/operators/modeling/predictive/bayesian/naive_bayes.html.