Assignment 2: Data Analytics

Group 50

Aleksa Stanojlovic
01428932
e1428932@student.tuwien.ac.at

Mensur Besirovic
01529872
e1529872@student.tuwien.ac.at

ABSTRACT

In this exercise report it will be shown the evaluation of the two classification algorithms: random forests and multilayer perceptron (MLP) on the example of an analysis of thyroid disease record dataset. Dataset consist of 3772 instances and 30 attributes. The data is preprocessed, and the mentioned algorithms are executed, in Weka Toolkit (Explorer).

1 Dataset

1.1 Select a dataset

The source of the dataset is OpenML[1] and the chosen dataset is Thyroid disease records supplied by the Garavan Institute and J. Ross Quinlan.

1.2 Analyze the characteristics of the dataset

It contains 3772 instances and 30 attributes, which are divided into 4 classes. Out of 30 attributes 23 are nominal, that are to be preprocessed and 7 are numerical. Although most of the nominal values are actually boolean but stated as 'f' or 't' character, which creates a demand for preprocessing them into numerical. There are 6064 missing values, which is around 5.4% of the total number of values. The attribute TBG (thyroxine-binding globulin) has no values, meaning, that 3772 values for this particular attribute is missing.

The algorithms should predict the potential development of primary hypothyroidism, secondary hypothyroidism or compensated hypothyroidism, based on the attributes like age, sex, pregnancy, thyroid surgery, sickness, thyroxine treatment, thyroid hormones (TSH, T4U, T3, TT4, FTI, TBG) and etc.

The dataset contains 4 classes: negative, primary hypothyroid, compensated hypothyroid and secondary hypothyroid. The instances are unequally distributed over classes, where 3481 are classified as negative, 194 as compensated hypothyroid, 95 primary hypothyroid and only 2 as secondary hypothyroid. This might cause certain issues on testing part, since it might be hard to detect clear predictions due to the lack of for example secondary hypothyroid instances (see Figure 1).

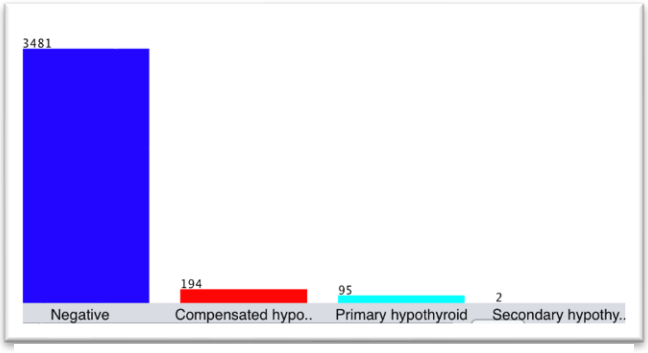

Figure 1: Distribution of the instances over the four classes

2 Classification: Analysis of Train/Test Set Splits, Performance and Parameters

2.1 Select two algorithms or two platforms

2.1.1 Multilayer Perceptron (MLP)

A perceptron is very simple classification algorithm that performs binary classification, that predicts the particular category of interest of an input (e.g. blue or not_blue), by separating two categories with a straight line.[2] Input is consisted of a feature vector x multiplied by its weight w and added to a bias b.

$$y = w * x + b$$

A single output is produced by the perceptron based on several input values, that form linear combination using the weights. This represents a single-layer perceptron, uncapable of performing non-linear classification (e.g. XOR function).

Multilayer perceptron is deep neural network, which consists of two or more perceptrons. It is made up of an input and output layer, as well as hidden layer (see Figure 2). The input layer receives a signal, the output layer makes prediction about the input, and the hidden layer acts as a computational engine of the algorithm. Supervised learning problems can be solved by MLP.

[1] https://www.openml.org/d/57 (last visited on 16.01.2020)

[2] https://pathmind.com/wiki/multilayer-perceptron (last visited on 16.01.2020)

The training is done by building a correlation model based on a set of input-output pairs. Values of weights and biases of the model are adjusted to minimize the error. Adjusted values are passed to MLP in the backward pass. In the forward pass, new values are pulled through from the input, over hidden to the output layer again. This back and forth propagation are done by the network until the error cannot go any lower (convergence state).

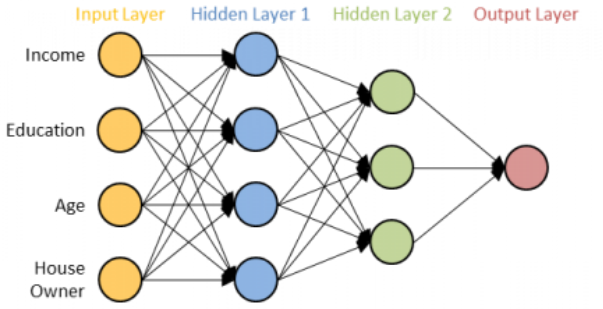

Figure 2: Multilayer perceptron architecture

The algorithm has potential to produce high quality classification model, which is one of the goals of this assignment. Furthermore, most of our data was numerical, or at least easily transformed into numerical values, due to the 'boolean type' attributes. MLP is known to work the best with low number of missing values and when the values are equally distributed over each attribute. Our dataset has some missing values and the range of the features are not equally scaled, which makes it interesting to work and experiment with. MLP should in theory get worse results than Random forest, because of the mentioned facts, but we will prove or disprove it later.

2.1.2 Random Forest (MLP)

Random Forest is one of the most commonly used supervised machine learning algorithm for tasks related to regression and classification.

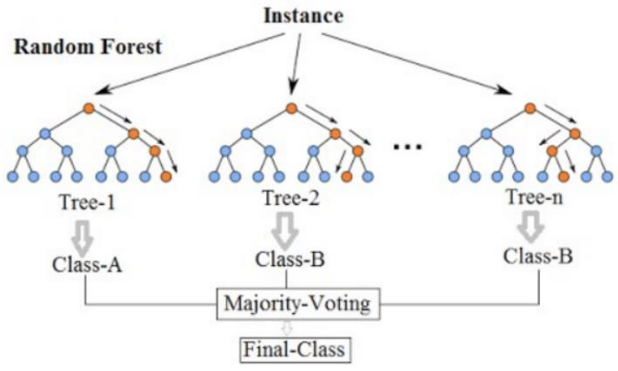

Figure 3: Random Forest architecture

Random Forest creates and joins multiple decision trees to get more accurate and stable results. A Decision Tree can process both categorical and numerical data which are essential for us. It is called 'Random', because Random Forest adds randomness to the model as the trees grow. Instead of looking for the most important feature while splitting a node, it looks for the best feature among a random subset of features. This leads to a great variety, which generally leads to a better model. The predicted class probabilities of an input sample are computed as the mean predicted class probabilities of the trees in the forest.(Figure 3) One of the biggest problems with machine learning is overfitting, but most of the time, this won't be so easy for a random structurer. This is because if there are enough trees in the forest, the classifier will not overfill the model. The main limitation of Random Forest is that a large number of trees can make the real-time prediction algorithm slow and ineffective. In general, these algorithms are quick to train, but they are slow to predict once they are trained. A more accurate forecast requires more trees, which leads to a slower model. The random forest algorithm is fast enough in most real-world applications, but there may be situations where runtime performance is important and other approaches are preferred. Random Forest is a great algorithm to train early in the model development process to see how it works. This algorithm is also a good choice if you need to develop a model in a short time. Random Forests are also very hard to beat in terms of performance. Of course, you can probably always find a model that performs better, like a neural network, but this development usually takes much longer. In addition, they can deal with many different types of features such as binary, categorical and numerical. Overall, Random Forest is a (mostly) quick, easy, and flexible tool, though it has its limitations.

2.2 Multilayer Perceptron

2.2.1 Preprocessing

First of all, for this algorithm we need to deal with missing values, since they are key for the quality of the resulting model. Throughout analysis of the features, we have found out that the values of TBG feature are all missing. It is the case because attribute TBG_measured is false for every instance, which means that TBG is not measured for any of the instances. These two attributes might cause a lot of errors, so the best way is to remove both of the attribute in order not to badly influence the quality of the model. In this step of preprocessing, we have decided to simply replace leftover missing values by applying 'ReplaceMissingValues' from Weka Explorer in the preprocessing tab. Moreover, the algorithm should work with numerical values, which is not the case for our dataset, because most of the values are of 'boolean'/nominal type. The way for turning them into the numeric value is by using 'NominalToBinary' filter. It converts all nominal attributes into binary numeric attributes.

Weirdly, on one of the computers Weka could start MLP right after turning nominal values into numeric ones, but on the other

computer, it was prohibited to start MLP before removing the TBG value. The value of TBG attribute stayed nominal even after applying 'NominalToBinary' filter, but we could not find any reason for that.

2.2.2 Subsampling

The chosen dataset has only 3772 instances, so subsampling was not necessary in this case.

2.2.3 Training & Testing

2.2.3.1 Parameters

1. ***batchSize*** *– In the case of performing the batch prediction, we can define this parameter as preferred number of instances to process. Lowering batchSize can lead to better results, because more adjustments can be made, as opposed to a higher batchSize.Higher runtime and high fluctuations can occur if the training data is not mixed, which is a result of varying input value ranges. Default: 100*
2. ***decay*** *– enabling decay causes learningRate to decrease after each step. It is directly proportional to learning rate and inversely proportional to number of epochs. This means that the real advantage of the decay can be experienced when the number of epochs is high. Default: false*
3. ***hiddenLayers*** *– This attribute defines hidden layers of a neural network. The value 'a' represents wildcard and it is equal to (# of attributes + # of classes) / 2. The values can be defined either using wildcards or using natural numbers. Default: a*
4. ***learningRate*** *– Defines learning rate for each weight. As the learning rate grows, so does the change of each adjustment. Anyhow, if the rate is too high, fluctiations of the training loss can occur, due to the model trying very different weights. On the other hand, if the rate is too low, model could never reach an optimal state. The range of learningRate is from 0 to 1. Default: 0.3*
5. ***momentum*** *- Depending on the previous weight adjustments, momentum is applied. The range is same as for the learningRate, which is between 0 and 1. It should improve accuracy and speed of the training. Default: 0.2*
6. ***seed*** *– It reproduces randomness. Default: 0*
7. ***trainingTime*** *– It is used for defining number of iterations (epochs) that are used for training. If every group of instances has gone through the network, that means that one iteration has been finished. Default: 500*

Ten different variations of the mentioned parameters will be experimented with. trainingTime, learningRate, hiddenLayers and momentum parameters are changed in order to observe the quality of the resulting model and its performance. Only mentioned values below the figures, define the changed parameters. All the other parameters are set to default. (CCI – correctly classified instances; MAE – mean absolute error)

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3428   46    7    0 |    a = negative
  121   71    2    0 |    b = compensated_hypothyroid
   17    8   70    0 |    c = primary_hypothyroid
    2    0    0    0 |    d = secondary_hypothyroid
```

Figure 4: Confusion matrix with default values

For the multilayer perceptron algorithm with default values the resulting model has CCI = 94.6182 % and MAE = 0.0353.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3467    8    6    0 |    a = negative
  183    7    4    0 |    b = compensated_hypothyroid
   15    6   74    0 |    c = primary_hypothyroid
    0    1    1    0 |    d = secondary_hypothyroid
```

Figure 5: trainingTime = 100 epoch

With the parameter trainingTime adjusted to 100, the values of CCI and MAE have changed as 94.0615 % and 0.0429, respectively. We can see that the model is slightly worse overall. But the precision of the classification of compensated hypothyroid is significantly worse than when using default values.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3459   16    6    0 |    a = negative
  155   34    5    0 |    b = compensated_hypothyroid
   13    6   76    0 |    c = primary_hypothyroid
    2    0    0    0 |    d = secondary_hypothyroid
```

Figure 6: trainingTime = 250

CCI = 94.6182 % and MAE = 0.0383 for the parameters as in figure 6. The results are negligibly different from the default ones, but the time taken for build model and run time is as twice as good as default.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3481    0    0    0 |    a = negative
  194    0    0    0 |    b = compensated_hypothyroid
   95    0    0    0 |    c = primary_hypothyroid
    2    0    0    0 |    d = secondary_hypothyroid
```

Figure 7: trainingTime = 250; hiddenLayers = a, 34, 8, 1

For the parameter configuration as shown in figure 7, the CCI was 92.2853 % and MAE was 0.0733. Because of size of our dataset and the distribution of the values across them, this can be

seen as significant setback in values. Furthermore, time to create a model increased to almost 20 second, which is three time worse than for the configuration shown in figure 5. The precision of classification, for any class other than negative, does not exist.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3456   20    5    0 |    a = negative
  170   17    7    0 |    b = compensated_hypothyroid
   13   18   64    0 |    c = primary_hypothyroid
    1    1    0    0 |    d = secondary_hypothyroid
```

Figure 8: trainingTime = 250; hiddenLayers = a, 30, 15

If parameters are set as in figure 8, CCI = 93.7699 % and MAE = 0.0432.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3481    0    0    0 |    a = negative
  194    0    0    0 |    b = compensated_hypothyroid
   95    0    0    0 |    c = primary_hypothyroid
    2    0    0    0 |    d = secondary_hypothyroid
```

Figure 9: trainingTime = 250; hiddenLayers = a, 10, 10, 10, 10, 10

In the case when parameters are set as in figure 9, CCI is 92.2853 % and MAE is 0.0732.

From the figures 7, 8 and 9, we can summarize that, number of hidden layers and the nodes inside the layers, can remarkably influence the results, in our case the results were worse than with default parameters for hiddenLayers. We did not manage to find any combination which performed better.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3444   29    8    0 |    a = negative
  144   47    3    0 |    b = compensated_hypothyroid
   13    8   74    0 |    c = primary_hypothyroid
    1    0    1    0 |    d = secondary_hypothyroid
```

Figure 10: trainingTime = 250; learningRate = 0.1

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3474    0    7    0 |    a = negative
  190    0    4    0 |    b = compensated_hypothyroid
   23    0   72    0 |    c = primary_hypothyroid
    2    0    0    0 |    d = secondary_hypothyroid
```

Figure 11: trainingTime = 250; learningRate = 0.01

For the figure 10 CCI = 94.5122 % and MAE = 0.0449, and for the figure 11 CCI = 94.0085 % and MAE 0.0618. The values of learningRate as defined in figures 10 and 11 are assumed to perform the best for neural networks, which was not the case for our dataset. Mean absolute error and classification of the values is much worse than for default parameter setting.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3481    0    0    0 |    a = negative
  194    0    0    0 |    b = compensated_hypothyroid
   90    0    5    0 |    c = primary_hypothyroid
    2    0    0    0 |    d = secondary_hypothyroid
```

Figure 12: trainingTime = 250; momentum = 0.9

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3481    0    0    0 |    a = negative
  194    0    0    0 |    b = compensated_hypothyroid
   95    0    0    0 |    c = primary_hypothyroid
    2    0    0    0 |    d = secondary_hypothyroid
```

Figure 13: trainingTime = 250; momentum = 0.99

The common values of momentum are assumed to be 0.5, 0.9 and 0.99 according to 'Machine learning mastery'[3]. We experimented with 0.9 and 0.99 values of momentum and we got as the results of CCI values 92.4178 % and 92.2853 %, and MAE values 0.0395 and 0.0386. The precision of classification is really bad for both cases. When momentum had value was 0.99, it strongly influenced time to build a model, which was ~32 seconds (ten times worse than for default parameters).

From this experiment we can conclude that all of the changed parameters really influence the results and the model quality. For trainingTime, we have found optimal number of epochs, in order to improve the quality of the model but also decrease time to build a model and run time of MLP.

2.2.3.2 Scaling

In this part we used 'Normalize' filter function of the preprocessing tab in Weka. Default values of the filter are 1.0 of scale and 0.0 of translation, meaning that the attribute values are about to be scaled in between 0 and 1. Since our dataset has mentioned 'boolean' types, which are transformed to numeric values 0 or 1, upon preprocessing, this scaling part should not influence the resulting model very much. Actually, only attribute that should be influenced by normalization is 'age'.
After running MLP algorithm with default values, mean absolute error was 0.0353 and correctly classified instances 94.6182%. In

[3] https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/ (last visited on 19.01.2020)

the second run, the for normalization are set to 0.0 and 2.0, which yield a bit worse results, with absolute error 0.041 and 93.3192% of the instances are correctly classified. In the third run, the values of translation and scale were set to -10.0 and 1.0, respectively. Mean absolute error was 0.0329 and correctly classified instances 95.1485%.
After comparing our statements and the results of the three runs of the MLP, we can see that they match. Scaling the attributes did not meaningfully influenced the resulting model.

2.2.3.3 Training/test set split

We have started with splitting values of training and test sets 5%/95% and in each step we increased training set split for 10% and for the same percentage the test set split was decreased. Derived from the results of the experiments from sections 2.2.3.1 and 2.2.3.2, we set all parameter values to default, except trainingTime, which was set to 250.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3205   98    2    0 |    a = negative
  135   47    0    0 |    b = compensated_hypothyroid
   11   73   10    0 |    c = primary_hypothyroid
    2    0    0    0 |    d = secondary_hypothyroid
```

Figure 14: Training/test set split 5%/95%

Starting split configuration resulted with poor values of CCI and MAE, which were 91.041 % and 0.0492. Significant improvement was made when the split was set to 35%/65% (see figure 15). The mean absolute error was 0.0254 and percentage of correctly classified instances 96.7781 %, which are best results than ever before, considering all previous parameter, scale and split variations. This split (actually 66%/33%) is theoretically considered the best, which is confirmed in our case.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 2242   16    4    0 |    a = negative
   31   87    4    0 |    b = compensated_hypothyroid
    8   14   44    0 |    c = primary_hypothyroid
    2    0    0    0 |    d = secondary_hypothyroid
```

Figure 15: Training/test set split 65%/35%

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 1524   40    5    0 |    a = negative
   56   21    6    0 |    b = compensated_hypothyroid
    7    3   33    0 |    c = primary_hypothyroid
    0    2    0    0 |    d = secondary_hypothyroid
```

Figure 16: training/test set split 45%/55%

Oddly, the results got worse, the moment the training/test set split was set to 55%/45% (see figure 16). CCI dropped to 92.9876 % and MAE had increased to 0.0482. Which is almost as double as bad as the split from figure 15.

```
=== Confusion Matrix ===

   a   b   c   d   <-- classified as
 170   1   0   0 |   a = negative
   8   2   0   0 |   b = compensated_hypothyroid
   1   0   7   0 |   c = primary_hypothyroid
   0   0   0   0 |   d = secondary_hypothyroid
```

Figure 17: Training/test set split 95%/5%

The results from the split shown in figures 14 and 17 cannot really be taken into account. They show the classification results, but they are not really accurate, considering that on training set split of 5%, model cannot really be well trained on such small amount of data. Also, having training set split configured too high, as for example 95%, the model have insufficient amount of data to test its quality on test set.

2.2.4 Describe and summarize the findings

2.2.4.1 What trends do you observe in each set of experiments?

Due to the fact that our set is very small, with just 3772 instances and that the values are pretty unequally distributed over the classes, it was hard to detect changes throughout the experiments. Firstly, as the classification algorithm is used, the percentage of (in)correctly classified instances is observed. Furthermore, the true positive rate per class is observed, and mean absolute error as well.
In most of the cases the percentage of correctly classified instances is between ~92% and ~96%. In all of the experiment variations the true positive rate of secondary hypothyroid class was 0 and its precision was undefined. The issue was only 2 instances were classified as secondary hypothyroid in the original dataset, so it was impossible for an algorithm to make any prediction. Although, compensated hypothyroid has more than a twice instances than primary hypothyroid, the precision of the true positives was always higher for primary then for compensated hypothyroid.

2.2.4.2 How easy was to interpret the algorithm and its performance?

The internet is great source of resources and learning materials. Machine learning is raging and developing lately, so do the learning platforms and the number of available tutorials. Mentioned fact have helped finding easy various and useful tutorials for this algorithm. One of the sources that we find most useful was website: https://machinelearningmastery.com/

2.2.4.3 Which classes are most frequently mixed-up? (and why?)

Some of the fasts have already been mentioned in section 2.2.4.1. Most frequently mixed-up classes are compensated hypothyroid and primary hypothyroid. The main reason for that is probably uneven distribution of the values over the classes. Compensated hypothyroid and primary hypothyroid classes have least number of different values, than any other pair of classes.

2.2.4.4 What parameter settings cause performance changes?

As the performance changes we consider observed trends in the experiments from 2.2.4.1 and the time for building a model. The changed parameters were trainingTime, learningRate, momentum and hiddenLayers. Fat the worse performance has the number of hidden layers and the number of nodes inside. For the chosen dataset and the MLP, we have not noticed any improvements by increasing the number of hidden nodes or layers. It has got worse. Changing the value of momentum to 0.99 (which is considered as one of the most common values by https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/), remarkably ruined the model building time as well as the percentage of correctly classified instances. Also, the test option for either using cross validation and percentage split. 10-fold cross validation performs mostly better than percentage split, but it has quite long run time. Significant improvements are made by setting number of epochs to 250, which is achieved by trying different values of this parameter.

2.2.4.5 Do both algorithms show the same behavior in performance, performance degradation/robustness against: smaller and larger set sizes; variations in parameter settings?

See Section **2.3.4.5**

2.2.4.6 Did you observe or can you force and document characteristics such as overlearning?

Overlearning can be achieved in couple of different ways, which we derived from the results of the experimental training. One is splitting the training and test data unequally. Second one is number of epochs (trainingTime). If its value is too high, the algorithm will lose performance on test data, because it is conformed the best for the training data, the same as the unequally splitting. Third one is number of nodes per hidden layer.

2.2.4.7 How does the performance change with different amount of training data being available? What are the best scalings (per attribute/value) and why?

In the section 2.2.3.3, the performance splitting was observed going from 5%/95% of training/test data and increasing in- or decreasing it by 10% to document the changes. In our case the best performing split was 35%/55% and the worst one was, as expected, 5%/95%. In the theory, the best split is 66%/33%, because an algorithm trains on the twice as much data than the training is performed. The training phase requires more data in order to create highly accurate model, but also, not too high percentage of the data, in order not to be overfitted by training phase, which causes 'adoption' of the model to the train data, as discussed in the previous section.

2.3 Random Forest

2.3.1 Preprocessing

Our data set has 3772 instances, 30 features and 3 classes. Most of the features are not numerical but nominal. However, one advantage of decision trees is that ordinary input data need no significant preprocessing. So random forests can work directly on both categorical and continuous variables. So, there is no need for one-hot encoding categorical values. Because there are some missing values in our dataset we have chosen to impute them using WEKAs *ReplaceMissingValues* filter. It is under *unsupervised -> Attribute -> ReplaceMissingValues*.

The *ReplaceMissingValues* filter replaces values with the mean or mode for real and nominal values respectively. As we can see in *Fig. 1* that in comparison to the other three classes, class 'negative' is overrepresented. Difference between the number of class instances of the dataset is always above 50%, so this could have effects on the accuracy of the prediction and have an influence how well one class is differentiated. Therefore, we don't expect well categorization for the compensated hypothyroid, primary hypothyroid and secondary hypothyroid instances. On the other hand, negative instances should be better classified. Therefore, we have to test extreme wrong settings for our classifier in order to analyze the sensitivity of classification outcomes.

In the next section we will describe all parameters which can be set in WEKA for random forest algorithm and show how those parameters affect results of the classifier.

2.3.2 Subsampling

The chosen dataset has only 3772 instances, so subsampling was not necessary in this case.

2.3.3 Training & Testing

In this section we are going to explore effects with different settings of parameter using 10-fold cross-validation. First, we will describe all important parameter, which we can set in WEKA for our classifier:

1. ***bagSizePercent*** *- This parameter specifies the size of the training set's random samples that are used to create the decision trees. With Bagging model is being improved and overfitting is prevented. If the 100 % option is used, then the sample will be as large as the training set.*
2. ***breakTiesRandomly*** *-If set to true, split point is*

determined randomly

3. ***calcOutOfBag*** - *It says if the out of bag error should be calculated.*
4. *computeAttributeImportance - If set to true the* importance of the features is computed which may increase the model building time
5. ***maxDepth*** - *This parameter specifies maximum depth of a tree where the default value 0 means that maximum depth is unlimited.*
6. ***numExecutionSlots*** - *This parameter specifies the number of threads used to construct the model which has a high influence on the model building time. Default value is 1.*
7. ***numFeatures*** - *This parameter specifies the amount of features which one tree is using in order to make a decision at a split. The default value is 0.*
8. ***numIterations*** - *This parameter specifies the number of trees which are build.*
9. ***printClassifiers*** - *If set to true, the decision trees are displayed what could be useful when analyzing them.*
10. ***seed*** - *This parameter specifies seed for random data shuffling.*

Because difference between the number of class instances is big and only one class has a big amount of instances, difference in precision is not going to be that visible. But the time to build a model is more influenced by the parameters.

First, we did experiment for *numIterations* parameter. As expected, when increasing number of iterations, we get better precision but the model building time and run time are also higher. In order to analyze better other parameters, we changed number of iterations from 100.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3471    6    4    0 |    a = negative
    2  192    0    0 |    b = compensated_hypothyroid
    4    3   88    0 |    c = primary_hypothyroid
    2    0    0    0 |    d = secondary_hypothyroid
```

Figure 18: Changed parameters: numIterations: 1000

In first run (Fig. 18) we got best results regarding precision, but because of unlimited depth of a tree and default value for number of iterations, the run time was longest. There were 3751 correctly classified instances (99.4433 %) and runtime was around 31 seconds. In order to improve performance and shorten runtime we set parameter

numExecutionSlots to 10 in order to increase number of threads which are used for model construction. The runtime with new parameter setting was around 10 seconds, which is a big difference.

Next, we tried to adjust maximum depth of a tree in order to see if precision will decrease, as we assumed.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3480    0    1    0 |    a = negative
  194    0    0    0 |    b = compensated_hypothyroid
   81    0   14    0 |    c = primary_hypothyroid
    2    0    0    0 |    d = secondary_hypothyroid
```

Figure 19: Changed parameters: numIterations: 1000, maxDepth: 2

As assumed in second run (Fig. 19) both precision and runtime have decreased. There were 3494 correctly classified instances (92.6299 %) and runtime was around 9 seconds. This difference would be of course even greater if distribution of instances into classes is different in our dataset.

```
=== Confusion Matrix ===

    a    b    c    d   <-- classified as
 3477    1    3    0 |    a = negative
   64  130    0    0 |    b = compensated_hypothyroid
   12    5   78    0 |    c = primary_hypothyroid
    2    0    0    0 |    d = secondary_hypothyroid
```

Figure 20: Changed parameters: numIterations: 1000, numFeatures: 1

The last run (Fig. 20) was with parameter set so that tree is using one feature in order to make a decision at a split. Here was the precision slightly lower -97.6935 %.

After this experiment we have used the best parameter setting for the evaluation of the effect of different training and test set splits. We set *numIterations* to 1000, *maxDepth* and *numFeatures* to default values (0) and *numExecutionSlots* to 10.

Because of the characteristics of our dataset, the runtime didn't vary too much by different splits. For example: For 5% 95% split runtime was 2s. For 50% 50% split runtime was around 3.5s and for 95% 5% runtime was 4s. Like in same experiment with Multilayer Perceptron, the precision results from the split are not really accurate, because 5 % of 3772 instances is small amount of data for both training the model and testing its quality.

2.3.4 Interpretation

2.3.4.1 What trends do you observe in each set of experiments?

Behavior of different parameter settings was quite predictable when it comes to analyzing performance and precision. But it was really hard to find right balance in order to be able to analyze all parameters, because differences in results were significantly small. Almost anything that we have assumed when tuning

specific parameters was indeed the result. For example, increasing the number of iterations will increase precision and runtime or decreasing maximum depth of threes results in decreasing precision and runtime. As expected, TP Rate (true positives) was high for '*negative' class* and of course lower for other classes.

2.3.4.2 How easy was to interpret the algorithm and its performance?

WEKA provides good evaluation metrics, which make analysis and interpretation easy and self-explained. For learning purposes, we have used various internet pages and tutorials as a source.

2.3.4.3 Which classes are most frequently mixed-up? (and why?)

All *compensated hypothyroid, primary hypothyroid* and *secondary hypothyroid* are often mixed up with *negative* class, what is reasonable, because the distribution of instances in training set isn't good. Almost 93% of instances are of the class *negative* and that makes difficult for the classifier to precisely classify them in correct class. Because there is difference over 178 % between other classes and *negative* class, most of the time they will be all classified as *negative.*

2.3.4.4 What parameter settings cause performance changes?

Runtime of the algorithm was mostly affected by number of iterations, number of threads as well as maximum depth of a tree. Where changing of number of features strongly affects the size of the decision trees.

2.3.4.5 Do both algorithms show the same behavior in performance, performance degradation/robustness against: smaller and larger set sizes; variations in parameter settings?

When we compare Random Forest with Multilayer Perceptron, it seems that Random Forest acts more stable when we perform parameter changes. Even though the difference in performance and precision was small (in range 1-10 %), Random Forest had better results, but performance of the MLP has oscillated. When we compare runtime for example, random forest defeats MLP by far, if the tree pruning is set up correctly.

2.3.4.6 Did you observe or can you force and document characteristics such as overlearning?

As said in section 2.2.4.6, overlearning can be achieved when splitting the training and test data unequally. But because small amount of data instances we have seen only small precision and performance changes when the amount of training data is reduced or increased. Therefore, it is difficult to observe modeling error like overfitting.

3 Missing Values

See python script.

4 Summary

4.1 Overall findings

This paper presents how data mining algorithms like: Random Forest and Multilayer Perceptron, perform by different parameter settings, scaling and training/test set splits using WEKA.

Random Forest algorithm gives more accuracy and it is more resistible when it comes to parameter setting. For Random Forest algorithm we don't have to do too much preprocessing, because the algorithm itself can deal with dataset with missing values or similar. But we can't take those results into account, after performing only this experiment. Our data set wasn't that big or challenging in order to definitely say that random forest performs better than MLP. For future experiments and papers we will consider using bigger dataset where instances are more equally distributed into classes.

4.2 Comparison with State of Art

We have used paper [4] where the author had similar approach on the same data set, which we have chosen for analysis. The platform, which was used is WEKA. The author performed K-fold validation using C4.5 on data set where K has varied between 2 and 10. [Table 1]

K=n	Accuracy	TP-rate	FP-rate	Precision	Recall	ROC area
K=2	99.469	0.995	0.01	0.994	0.995	0.996
K=4	99.57	0.996	0.013	0.995	0.996	0.994
K=6	99.60	0.996	0.019	0.995	0.996	0.994
K=8	99.54	0.995	0.019	0.995	0.995	0.993
K=10	99.575	0.996	0.019	0.995	0.996	0.993

Table 1: Detailed Accuracy for different k-folds for C4.5 algorithm

After performing detailed analysis for different k-folds, the author used then the best parameter (K = 6) for further experiment.
The next experiment was to perform 7 different classification algorithms using 6-fold cross-validation.

[4] https://www.researchgate.net/publication/321533689_Diagnosis_and_classification_of_hypothyroid_disease_using_data_mining_techniques (last visited on 19.01.2020)

Algorithm	Accuracy	Precision	Recall	F-Measure	ROC-Area	TP-rate	FP-rate
Mutlilayer perceptron	94.035	0.937	0.94	0.938	0.891	0.94	0.398
RBF Network	95.228	0.945	0.952	0.946	0.898	0.952	0.407
Bayes net	98.59	0.986	0.986	0.986	0.997	0.986	0.08
C4.5	99.57	0.995	0.996	0.995	0.993	0.996	0.019
CART	99.54	0.995	0.995	0.995	0.993	0.995	0.007
Decision stump	95.38	0.95	0.954	0.948	0.981	0.954	0.009
REP tree	99.57	0.995	0.996	0.996	0.993	0.996	0.007

Table 2: Detailed Accuracy by class for various classifiers

From the table 2 we can see that REP tree, C4.5 and CART are performing the best, where Multilayer perceptron has the worst results. Compared to our results Multilayer perceptron had a lower percentage of precision by 0,6 %, but we have used 10-fold cross-validation which can cause this difference.