188.429 Business Intelligence
Hans-on Assignment #1: Data Mining
"Cardiotocography - Classifying Fetal Heart Rate Patterns"

Daniel Füvesi
Student - 1326576
e1326576@student.tuwien.ac.at

Michael Ion
Student - 1005233
e1005233@student.tuwien.ac.at

ABSTRACT
In this paper, we used the Azure ML Studio to investigate the dataset on fetal heart rate patterns and trained a classifier for finding a corresponding pattern class to a new test sample. We used two classification algorithms, Neural Networks and Multiclass Decision Forest, for which we first found perfect parameters through tuning the hyperparameters and lots of experimentation. Then, we researched if scaling brings a positive effect on our results. Furthermore, we experimented with different splits of training and test data to see how this affects results.

Finally, we wrote a script that inserts missing values in our test data. We looked into how these would affect the best models from our previous steps.

Keywords
Classification; Azure; Machine Learning; Data Science; Algorithm; Medicine; Cardiotocography; Missing Values;

1. INTRODUCTION
First, we will introduce the topic and talk about the characteristics of the dataset. We will discuss what a first preprocessing would entail. Then, we will discuss our chosen algorithms, how we found the fitting parameters and how we evaluated them. There, we will also see how we can improve our values with application of scaling and see how different splits will affect our results.

The next section will be about the introduction of missing values in our dataset and how it influences our classification. Finally, a summary will subsume all our findings and the lessons we learned in this exercise.

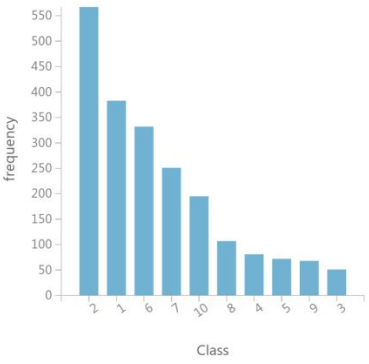
2. DATASET AND PREPROCESSING
2.1 About the Dataset
For this dataset, we used the "Cardiotocography" dataset[1] which comprises data of fetal heart activity[2]. This heart activity can be used to predict the wellbeing of the fetus. There are two versions of this dataset available: The first differentiates the dataset into three classes: Normal, suspect and pathologic. The second one expands this classification by differentiating heart rate pattern, having a total of 10 different "Fetal Heart Rate Pattern" classes. We use this latter set for our experiments.

We inspected the data with both WEKA Explorer and Azure Visualizer. The attributes of the data were labeled V1 to V35, but it didn't tell us anything about it. We found the original source of the data at another location[3], with more labels, so we could allocate what each attribute actually meant. We noticed:

A. Columns V4 and V5 denote the attribute "LB" and are the same, thus one can be omitted.
B. Columns V26-V35 are ten binary columns that indicate if the row belongs to any of the ten classes, in other words: a 1-N coding of the class, which can thus be omitted.
C. The last column "Class" indicates the class of the data row (heart rate pattern).
D. The first attribute V1 is nowhere to be found in the original source. It consists of the few numeric values, where rows of data repeat the same value after one another. It seems that it was used as an index for splitting up the dataset in subsampling, so we decided to omit it as well.

We also clean the data with removing identical rows, as this doesn't give us new insight for adjusting the ML algorithms. After all, we are left with a set of 2115 data samples with 21 attribute and 1 class column.

Figure 1. Frequency per class in dataset



Classes 1 and 2 are overrepresented in the data set (they make up almost 50% of all samples), classes 4, 9, 5 and 3 are underrepresented (3-4% of the set). This is an important insight for the results of later classification. As we don't have an unmanageable amount of data, we decided not to use subsampling. To the contrary, it turned out that more training data could have increased the accuracy of our classifier. In the figure above you can see the frequencies of the different classes.

[1] http://www.openml.org/d/1466
[2] For more reading on the subject: https://en.wikipedia.org/wiki/Cardiotocography

[3] https://archive.ics.uci.edu/ml/datasets/Cardiotocography

In the submission folder, you can find folders for the used datasets, experiment results, and details to the Missing Value part. There are four datasets in the "0_DATASET" Folder:

A. 0cardiotocography_original.arff is the dataset from openml.org
B. 0Original_Data_Source.xls is the original source from archive.ics.uci.edu
C. 1cardiotocography_labeled.arff is the dataset from A with labels obtained from B.
D. 2cardiotocography_processed.csv is dataset C. but cleaned after the changes described above.

In the "1_EXPERIMENT_RESULTS" folder you will find the detailed results for every experiment we did in Azure. We will refer to those in the following sections, but we only inserted tables and figure when we deemed them absolutely necessary for better comprehension. They are ordered as the experiments in the report. You will find both the original Excel file as well as PDF versions for quicker access.

3. TRAINING AND TESTING - METHOD

For the cross validations and generally every process involving randomness, we used random seed and to ensure traceability. We also set shuffle examples to "false" for this reason.

As a performance metric, we used **precision**: As our experiment are in the domain of medicine and healthcare, the correctness of classification is the most important step - we don't care if we don't find all instances of a class, as long as the predictions we do make are generally precise.

For every sub-experiment, we created tables with the results. These include the parameters used in the experiment and other details about the environment, the precision per class, both the average and standard error, as well as a confusion matrix which gives the absolute number of classifications. You find those tables in the different pdfs in the folder. In this report, we will not use every single result or display every single confusion matrix from there, but only the one we deem important to make our case.

4. TRAINING AND TESTING - NEURAL NETWORKS

The Neural Network is a set of interconnected layers, in which the inputs lead to outputs by a series of weighted edges and nodes. The weights on the edges are learned when training the neural network on the input data. The direction of the graph proceeds from the inputs through the hidden layer, with all nodes of the graph connected by the weighted edges to nodes in the next layer.[4] The parameters allow us to adjust this different features in the best way for our use.

4.1 Parameters

In the very first step, we experimented with different values to find the best arguments for the algorithms parameters. For *normalization*, we always used Min-Max for this step. In order for NN (*Neural Networks*) to work properly, they need values between 0 and 1 (we will show an example of what happens if that's not the case).

We started out with trying the default values: HN (*Hidden Nodes*) = 100, LR *(Learning Rate*) = 0.1, LI *(Learning Iterations)* = 100, ILW (*Initial learning weights diameter*) = 0.1, M (*Momentum*) = 0 This setup actually gave an already well working result as seen in Table 1 (refer to the other files in the submission for confusion matrix and more details). This was obtained with *10-fold cross validation* over the whole dataset. We wrote an additional python script to execute in Azure that gives us *macro-averaged precisio*n (the built-in functions were not working with the "Cross Validate" function) as well as the standard error of the precision per class. It has to be added as a "Python Script" Block. You can find it on the path "1_EXPERIMENT_RESUTS/macroAvgSkript.py"

Table 1. Precision per class 1-10, Macro-Averaged Precision and Precision Standard Error for NN Default Values

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.77	0.92	0.62	0.81	0.73	0.92	0.92	0.92	0.92	0.65	**0.81**	0.11

We see that the classifier works really well for classes 2, 4, 6, 7, 8, and 9 and not so well for 3, 5 and 10. One explanation is that we have less values for 3, 4, 5 and 10 than for 2, 6 or 7, so we have more foundation for our models. This is especially important for NN. The other is, that the attributes for certain classes overlap and then the rows get classified as classes from where we have more data. This is the case in the default for Class 10, which gets often classified as Class 1, look at the figures below.

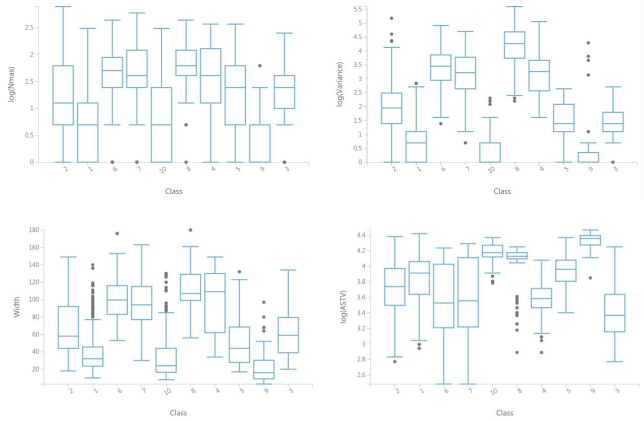


Figure 2. Attributes split by class. You can see that the ranges in the attributes variance, Nmax, Width and ASTV intersect

For finding the best parameters for the NN, we used the possibility in Azure to automatically combine lots of different values with the "*Tune hyper-parameters*" block. We set *"learning rate*" and "*number of iterations*" to the range of "0.002 - 0.36" and "51 - 450" respectively. We split these ranges up in 30 steps. This would create a grid of possible test instances. Our "Tune hyper parameters" would do 100 random runs on this grid, combining both parameters. We repeated this for a specified number of *hidden nodes*: 50, 100, 250 and 400. Run time varied from around half a minute to half an hour. See an example setup in Figure 3 below.

We also tried to tweak the parameters manually, although those results were not that good compared to the automated random grid. We realized, however, that there seemed to be a tradeoff between *learning rate* and *iterations*. An decrease in learning rate meant often more precise results, but only when number of iterations was not changed. The same way, if number of iterations was increased, however it did not hold when both were changed. The same goes for hidden nodes. For just 50 and less, results were

[4] https://msdn.microsoft.com/library/en-us/Dn906030.aspx

worse. However, more than 150 would mean a big increase in computation time, but the results didn't improve by a significant amount. Finally, we chose 125 hidden nodes for computation, as this gave us the best result from the "Tune hyper parameters run". The parameter values were: HN= 125, LR = 0.054, LI = 363, ILW = 0.1, M = 0.5. This gave us the results displayed in Table 2.

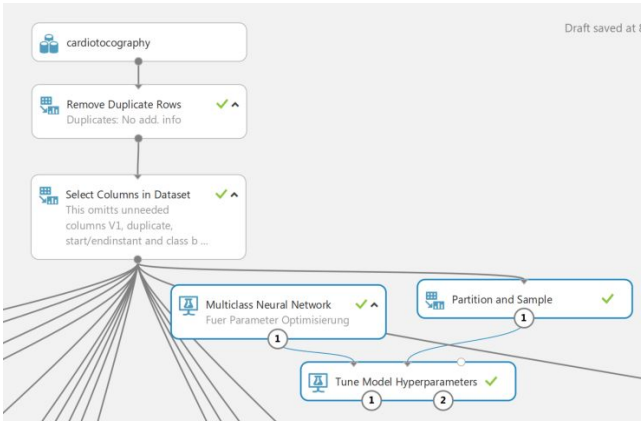


Figure 3. Setup in Azure for tuning Model Hyperparameters. "Partition and Sample" enforces 10-fold cross validation

We see that we could significantly improve precision for Classes 3 and 10. We see that generally, a small learning rate (a small step size taken by each iteration) improves the result. This avoids overshooting local minima. The amount of hidden nodes is highly dependent on the data, and we found out that our model works best with about 125. Subsequently, providing enough learning iterations is important to get enough adjustments for our node weights in the network. Some classes got a bit of a worse precision, but overall we could improve the macro average, which was our goal. We want to perform in each class overall well, as we do not have any knowledge about the topic (i.e. if it is better to classify one class better over the other). We also reduced the variation of the values.

Table 2. Precision per class 1-10, Macro-Averaged Precision and Precision Standard Error for NN Best Parameter model HN= 125, LR = 0.054, LI = 363, ILW = 0.1, M = 0.5

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.78	0.91	0.72	0.84	0.74	0.89	0.92	0.91	0.83	0.81	**0.83**	0.06

We ran also some experiments with obviously wrong settings, discussed in the following. Except the changed parameters, all parameters were the same as in the default described above.

In the first negative example, we set the number of *hidden nodes* very low, on 4. This means that the capability of differentiation of the NN is vastly restricted. Of course, too many means that there is a lot of redundancy. But this example shows it is better to have more than too few. Results are especially bad in the classes with very few instances, as they get classified as the more represented ones. See in Table 3.

Table 3. Precision per class 1-10, Macro-Averaged Precision and Precision Standard Error for Hidden Nodes = 4

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.65	0.95	0	0.76	0.58	0.93	0.88	0.88	0.2	0.55	0.63	0.30

The next is, if we set the learning rate too high. This tends to converge very fast and "stop" at certain values, however it is not very exact if used carelessly - like here. It tends to overshoot local minima and we see that the less represented classes suffer here. The rest of the classes get classified still averagely well - see Table 4.

Table 4. Precision per class 1-10, Macro-Averaged Precision and Precision Standard Error for Learning Rate = 0.9999

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.73	0.89	0.46	0.80	0.67	0.89	0.86	0.91	0.85	0.59	0.77	0.14

If we set too few learning iterations for our NN, our network is not refined very well during the training weights. This means, that the weights of the nodes are not tuned properly and we could have way better results to differentiate better. As a network is kind of like a black box process, we do not see the differences right in the model, but we can definitely see the difference in Table 5.

Table 5. Precision per class 1-10, Macro-Averaged Precision and Precision Standard Error for Learning Iterations = 3

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.71	0.61	0	0	0	0.81	0.68	0.94	0	0.47	0.42	0.36

A combination of all the bad factors above gives us an even worse result, so if we set HN = 3, LR = 0.8, LI=3 then we get the average precision of 0.32 with standard error 0.33

4.2 Scaling

In this step, we experimented with different kinds of scaling to improve our results over the previous step. This can be seen as a step of *preprocessing*, as we process the data before the training phase of the algorithm. It is an iterative process, as we make adjustments, see the result of those and begin again.

For the Neural Network, we always used the regular *min/max* scaling, which gives the lowest value of a dataset the value 0, the highest 1 and assigns the value then in between. We didn't have any serious outliers, so we applied it directly.

Let's start with some bad examples. We mentioned we needed values between 0 and 1 for NN. We tried to feed the non normalized data to our neural network. Obviously, the NN cannot work with this, see table 6. All examples in this section are also evaluated with 10-fold cross validation. The parameters of the Neural Network where the best ones found from the previous step (Table 2).

Table 6. Neural network with Default Values but no Normalization of Data

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.0156	0.244	0	0	0	0	0	0	0	0	0.026	0.073

All values were classified as belonging to either Class 1 or 2 (the most common classes in our data). However a lot of Class 1 samples were classified as Class 2 and vice versa. We see that no normalization is a no-go.

On Azure, we can select the normalization directly in the Neural Network. However, just selecting some normalization and then applying is in most cases a bad idea and doesn't give any benefits. We did this as giving bad examples and applied *logistic, tanh* and *zero mean/unit variance* to all data values separately. See below in table 7 the results of applying *logistic normalization* to all attributes, for example.

Table 7. Application of logistic normalization to all attributes

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.69	0.85	0.50	0.55	0.55	0.87	0.88	0.93	0.7	0.57	0.71	0.15

Some classes do not suffer that much as others, but overall, we do not have a satisfying result - we can do better. So we will not give the same example for the other normalization techniques (the results are similar and can be seen in the appended data sheets).

The right way to do it is to inspect the different attributes one by one and see if we can clarify the situation with the transformation of values. This is a very painstaking process. We did this with the normal Azure visualization tool for datasets, however, this is probably not the best tool for complication Data Analysis using Information Visualization tools. One example attribute that we found out this way is "Variance". We realized that the values are crammed together, and a couple others are very 'out there' after the default min/max transform. When we use the *logarithmic scale* for the visualization, we see that the values are better distributed, so a ML algorithmic could differentiate better between the values, as seen in Figure 4. This is how we ended up applying *logNormal normalization* to the "Variance" attribute. For this, we disabled the integrated normalization parameter in the Neural Network and used "Normalize Data" blocks in Azure.

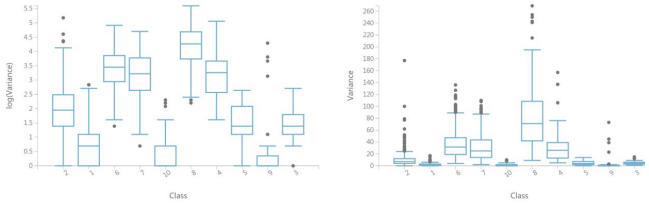


Figure 4. The attribute "Variance" per class. On the left on a logarithmic scale, on the right on a normal numeric scale. Better differentiation on the left, more useful for classification

Additionally, we also went to the effort to test the attributes by a leave-one-out method. This means, application of one normalization technique to one attribute column and see if this has any effect. For this reason, we set up lots of experiments (one for every changed attribute) and executed them at once, like in Figure 5. However, we realized that sometimes, combining two normalizations of two different attributes, which by themselves give an improvement, doesn't result in a better model. Because of this behaviour, it was quite hard to find a lot of normalization combinations that gave a big benefit.

With this technique we found following combination: The attributes "*Variance*" and "*DS_DR*" were normalized with *LogNormal* and *TanH Normalization* was applied to attribute "*FM*". The rest of the attributes were, as usually, normalized with *min/max normalization*.

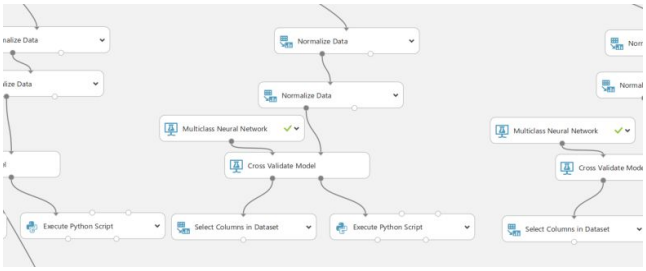


Figure 5. How we tested out different normalizations. The first "Normalize Data" normalizes all attributes to min/max, the second applies another normalization to a single attribute

See Table 8 for the *macro-averaged* results of this approach (after, again, 10-fold cross validation). We could improve the overall result by a bit. Especially precision of Class 5 was raised by a considerate amount. However, we got a bit of a lower precision in Classes 10 and 4. As we are interested in macro averages due to the uneven amount of classes in the set, we decided to go for this result though. The gain in the other classes compensates the loss in a few classes.

Table 8. The best result for different combinations of scaling methods: LogNormal Normalization for Attributes "Variance", "DS_DR" and TanH Normalization for "FM"

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.80	0.91	0.74	0.91	0.80	0.89	0.93	0.93	0.85	0.79	**0.86**	0.065

4.3 Training / Test Set Splits

In the next step, we performed multiple runs with the NN using different sizes of test and training data. For the model, we used the best one from the previous steps - the best parameters from step 4.1 with the normalization used in 4.2. We began with 5% training and 95% test data, then changed this amount by steps of 10%. We used the "Split Data" block to accomplish this step, as seen in Figure 6.

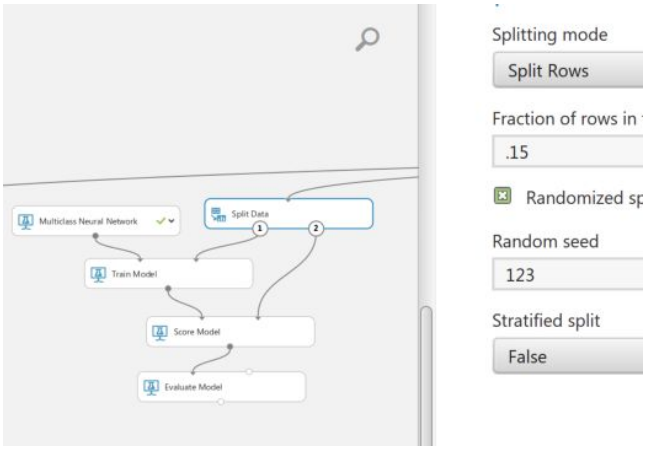


Figure 6. The definition of different split sizes in Azure ML Studio. Here, we use 15% of the dataset as a training set

We didn't use *stratification*, the reason being, that we wanted to have the "full impact" of our data sets. We wanted to see how it affects the different classes by being picked up completely randomly. We set up an experiment with 10 different classifiers for all different blocks and ran all of them three times to see different effects of the different splits.

We see the runs for each separate split summarised in Table 9. Overall, we can see that the results are not very good with a small training set. At 5% training set amount, precision is only around 65%. With a very little training set, it depends on randomness which classes can be found in the training set and thus are better or worse classified. Even though our three runs here show very similar (low) average precision, if we look specifically at the precision of the separate classes, we can see huge differences in the runs. In the first run, as you can see in Figure 7 on the left, we have a far lower Precision for instances of class 1 than in the second run on the right. However, Precision for class 5 on the other hand is a lot better in run 1 (even if quite low, comparably) than run 2. This is both due to the low amount of class 5 instances as well as the tiny amount of training data. A classifier cannot generalise very well with little training. In the test data there are probably many instances that the classifier would be "unprepared" for.

However, the result gets better quickly: We see a big increase in precision when choosing 15%, 25% and 35% training data. The variance of the classes is already at 15% very low: The precision per class is very similar here to the one already depicted in Table 8: The precision for underrepresented classes is lower than the rest, but there are no big jumps between runs, the precision per class remains stable in all runs.

Table 9. Macro-averaged Precision for different Test / Training splits in different runs

Training/ Test split	Macro-avg. precision: Run 1	Macro-avg. precision: Run 2	Macro-avg. precision: Run 3
5% / 95%	0.649982	0.648341	0.643175
15% / 85%	0.767398	0.750672	0.723784
25% / 75%	0.758399	0.738575	0.735617
35% / 65%	0.803835	0.800677	0.795512
45% / 55%	0.806466	0.774976	0.800611
55% / 45%	0.820939	0.846246	0.825259
65% / 35%	0.800648	0.804236	0.84923
75% / 25%	0.827224	0.783249	0.799948
85% / 15%	0.831212	0.850117	0.784354
95% / 5%	0.812289	0.862742	0.843624

As expected, the precision gets better the more training data we use, but it doesn't rise significantly after the 50% split. As you can see in Table 9, we even have cases where the precision is lower the more training and the less test data we have. The problem here is, that if he have too few test data, we cannot make a generalised assumption about the validity. The result is to have always enough data and use techniques like *cross validation* or *bootstrapping* to ensure that we always have fresh combinations of our data, regardless if for training or testing.

Figure 7. Confusion matrices of run 1 (top) and run 2 (bottom) for 5% training and 95% test data

5. TRAINING AND TESTING - MULTICLASS DECISION FOREST

The Multiclass Decision Forest belongs to the set of decision forest algorithms. It works by building multiple decision trees and subsequently voting on the most popular output class. During the aggregation process each label receives a probability. As a consequence the trees with high prediction confidence have greater weight when making the final decision of the ensemble. The advantages of this algorithm are that it can represent non-linear decision boundaries, it can handle noise in features and shows good computational performance and memory usage for both training and prediction. The trade-offs are the sensitivity to the data's stability and the potential of overfitting, for a careful setting of parameters is crucial.

5.1 Parameters

In Azure the Multiclass Decision Forest classifier consists of 6 parameters which we tweaked until the highest precision has been reached. The *Resampling method* parameter can take either *Bagging* or *Replicate***.** In case of bagging (bootstrap aggregating) each tree is generated based on a new sample while replicate takes the same input data for training each tree. The second parameter *Create trainer mode* specifies how the model is trained, i.e. how the rest of the parameters are picked. If single parameter is chosen, specific values need to be provided. Using a *Parameter range* lets the user to put multiple values separated by comma so that the trainer evaluates every combination and returns the best model. Furthermore, the *Number of decision trees* specifies the maximum number of trees that are generated. Similarly, the *Maximum depth of the decision trees* parameter stands for the same except for the fact that the number limits the depth of each generated tree. The *Number of random splits per node* indicates the number of splits (features on the a node are randomly divided) used when building each node of the tree. Obviously, setting these last 3 parameters high can produce a performance overhead. With

the *Minimum number of samples per leaf node* parameter one can influence the creation of terminal nodes in trees. The higher the value, the more cases have to meet the same conditions. The last checkbox parameter, *Allow unknown values for categorical features*, might be an interesting one when working with unknown values. When unknown values are accepted, the model is likely to be less accurate but it could provide better predictions for new values. Since our dataset does not contain any unknown values, changing this parameter does not affect the model.

Per default, the decision forest classifier uses bagging for resampling and the following single parameters: 8 decision trees, 32 as the maximum depth, 128 random splits and 1 min. sample per leaf node. As already mentioned before, the unknown values parameter does not influence the results. Therefore it is set to true and should be always assumed. Every classifier has been trained with the *Cross-validate Model* module using a random-seed which performs a 10 cross fold validation. Table 10 shows the precisions per class and the macro-averaged precision that we used to evaluate a parameter set. Additionally Fig. 8 depicts the confusion matrix of the model.

In the next step the same experiment has been conducted by using the replicate resampling instead of bagging. Although replicate seems to perform slightly better using default parameters, bagging worked out for future experiments with better parameters. A comparison between the scored precisions can be found in Table 10 and Table 11.

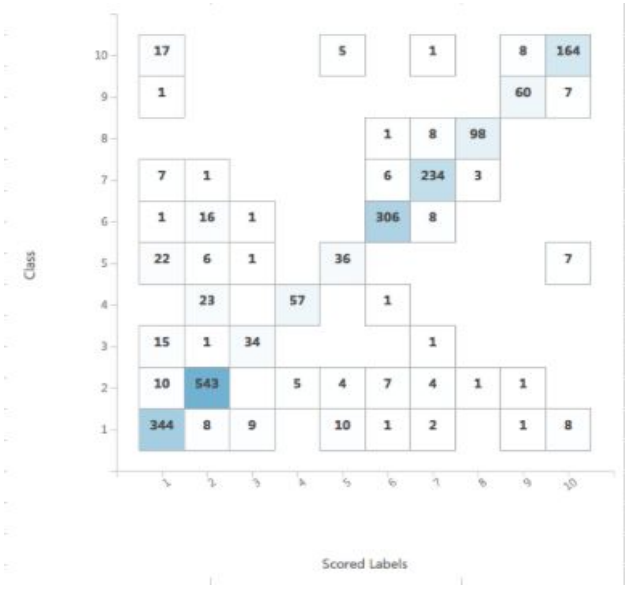


Figure 8: Confusion matrix of MDF w. default parameters

Table 10. Precision per class 1-10, Macro-Averaged Precision and Precision Standard Error for MDF Default Values (bagging)

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.82	0.90	0.71	0.91	0.65	0.95	0.90	0.95	0.82	0.87	**0.85**	0.09

Table 11. Precision per class 1-10, Macro-Averaged Precision and Precision Standard Error for MDF Default Values (replicate)

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.83	0.91	0.75	0.90	0.67	0.93	0.89	0.95	0.88	0.89	**0.86**	0.09

For the next step, the goal was to find the best possible parameters. In order to do that the *Tune Model Hyperparameters* has been applied. The purpose of this module is to test models using different combinations of settings and return the optimal one. As the metric for measuring performance of classification precision was set to ensure consistency. In case of the metric for regression the mean absolute error was selected. The entire grid parameter sweeping mode was selected over random sweep as suggested by Azure's documentation. On the one hand, entire grid tries all possible combinations of the parameter range thus ensures a full sweep. On the other hand, specifying a too broad range can lead to extreme execution times. This is important, especially when using the free Azure account where computation time of a single task is limited by 1 hour. Using the decision forest's range builder, the following ranges have been provided for the hyperparameters module:

- Number of decision trees: 1, 8, 32, 64, 128
- Maximum depth of the decision trees: 1, 32, 16, 64, 128
- Number of random splits per node: 1, 128, 1024, 2048
- Minimum number of samples per leaf node: 1, 4, 8

Additionally the whole dataset has been fed in because of its manageable size. After running the sweep, the tuner lists the best parameters ordered by the precision they induce. Since there were multiple parameters with the same highest precision, we took the lowest ones. The idea is, that if the model can reach the same precision with 32 and 128 decision trees, then the lower one should be preferred because of its performance relevance. A few extra ranges have been tested as well for confirmation. At the end, the following parameters were picked and used for training the tuned model:

- Number of decision trees: 32
- Maximum depth of the decision trees: 64
- Number of random splits per node: 1024
- Minimum number of samples per leaf node: 1

Subsequently the classifier has been tested with this parameter set, just like in the default case. The results are shown in Table 12 where one can see that the model ended up with a 3% raise (2% in comparison to the replicate default) in the precision.

Table 12. Precision per class 1-10, Macro-Averaged Precision and Precision Standard Error for MDF Tuned Values (bagging)

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.86	0.92	0.76	0.91	0.76	0.95	0.89	0.97	0.91	0.87	**0.8845**	0.07

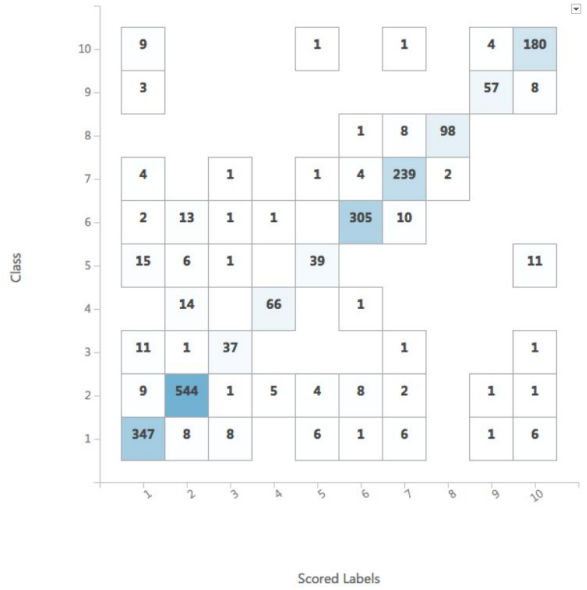


Figure 9: Confusion matrix of MDF w. tuned parameters

In order to validate the above assumptions, 2 more test cases were conducted. In the first one the (256, 32, 2048,1) parameter set has been used (please refer to the names of parameters listed above). In the second one, extreme parameters has been set (512, 256, 4096, 1) to show that overlearning does not yield better results. The precisions are shown in Table 13 and Table 14 for each test case respectively. It can be deduced that it does not matter how much higher parameters are chosen, the 88% will unlikely be exceeded. As we can see differences in the average precision happen at the third or fourth decimals. Consequently the above presented tuned classifier is used to perform the subsequent steps.

Table 13. Precision per class 1-10, Macro-Averaged Precision and Precision Standard Error for MDF 2nd Tuned Values (bagging)

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.88	0.92	0.77	0.93	0.73	0.95	0.89	0.97	0.87	0.87	**0.8833**	0.07

Table 14. Precision per class 1-10, Macro-Averaged Precision and Precision Standard Error for MDF Overlearn Values (bagging)

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.88	0.92	0.77	0.93	0.71	0.96	0.89	0.97	0.87	0.86	**0.8819**	0.07

Detailed listing of the parameters with precise results can be found in the project archive. Additionally it should be pointed out that theoretically there might be better performing parameter sets. However finding those requires big ranges for each of the parameters that would need several hours or even days to analyze.

5.2 Scaling

One advantage of using tree-based algorithms is that they do not generate distance-based models. As an example, the k-nearest neighbour classifier determines the class of a dataset based on the neighbours that are calculated using a distance function. Therefore scaling the dataset is crucial in order to avoid domination of particular features. In case of the multiclass decision forest no distance function is used, consequently no scaling is required.

Nevertheless a few test cases were conducted to verify our assumption and prove the statement empirically. Each of the scaling functions were tested, namely Z-Score, MinMax, Logistic, LogNormal and Tanh. While both Z-Score and MinMax resulted in exactly the same model, the Logistic and Tanh functions produced significantly poorer models (77% and 76% mean precision). Parallel to the neural network scaling, the combination of LogNormal (on DS_DR, Variance), Tanh (on FM) and MinMax (on all the rest) has been tested and surprisingly resulted in a slightly better model than the best until now with a mean precision of 88.490% (current best: 88.4549%). After all, the best model in this scaling task has been produced by the LogNormal function with a mean precision of 88.4963% (detailed results in Table 15). With that being said, the findings are almost consistent with our assumptions, namely that scaling (almost) does not affect tree-based algorithms. While an improvement could be recorded, the 0.0004 difference can not be seen as a theoretical disproof.

Table 15. Precision per class 1-10, Macro-Averaged Precision and Precision Standard Error for MDF Tuned Values with LogNormal Scaling (bagging)

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P avg.	P std.
0.87	0.92	0.77	0.90	0.77	0.95	0.89	0.97	0.91	0.86	**0.8849**	0.06

5.3 Training / Test Set Splits

Similarly to Neural Networks, the next step consisted of evaluating the effects of different training and test set splits. For this task, the current best parameters from previous steps have been chosen (the parameter set from step 5.1 and scaling from 5.2). Figure 10 shows a setup in Azure ML Studio for this step. The nodes are displayed with comments to provide explanation. In the depicted example a training/test set split of 55%/45% is used.

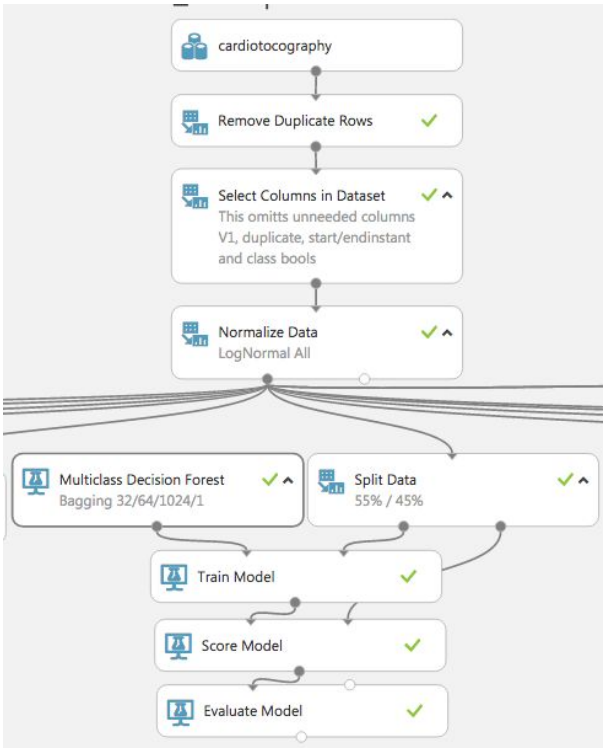


Figure 10: MDF Training/Test Split Setup

A total of 10 such cases or group of nodes were created with each case representing a different data percentage split. Each test case were executed 3 times using the following 3 random seeds: 42, 123 and 666. The use of different random seeds leads to different randomized datasets which have an effect on the evaluation.

Due to the semantical background of the data, precision was consistently used as an evaluation metric. Table 16 shows the macro-averaged precisions (overall performance) for each training/test set distribution and runs.

Table 16. MDF Training/Test Set Splits Macro-averaged precision for 3 runs

Training/ Test split	Macro-avg. precision: Run 1	Macro-avg. precision: Run 2	Macro-avg. precision: Run 3
5% / 95%	0.585973	0.755698	NaN
15% / 85%	0.795854	0.764701	0.765633
25% / 75%	0.795854	0.817507	0.868088
35% / 65%	0.852505	0.846992	0.871431
45% / 55%	0.860949	0.839709	0.873027
55% / 45%	0.859269	0.84265	0.861039
65% / 35%	0.861772	0.859396	0.860551
75% / 25%	0.881176	0.815216	0.86949
85% / 15%	0.940623	0.79465	0.891185
95% / 5%	NaN	0.839772	0.856832

As we can see by the colours, there are 2 main edge cases namely the first one and its complement. This can be explained by the fact that in both cases one part of the splits is too small. 5% of 2115 gives us 106 (105,75) rows. In the first case this amount of data is simply not enough to obtain a sophisticated classifier, thus the majority of evaluations fail. The confusion matrix of the third run with NaN as precision can be seen in Figure 11.

Figure 11: Confusion matrix of evaluation with 5% training and 95% test sets

It is obvious that both class 8 and 10 were completely misclassified. This is because of the random nature of the generated datasets. With this particular random seed these classes are likely to be very underrepresented in the training set. Having only a total of 2115 rows, tight splits like 5-10% might not be enough for proper training especially, when there are already significantly underrepresented classes in the whole set.

The other extreme run is - already mentioned before - when the splits are the other way round. Figure 12 shows the evaluation of the 95%/5% split in a confusion matrix. As one can see almost everything has been classified correctly except for class 5 being totally misclassified. First of all the reason why the average precision is high is that the training set provides enough data (95%) for the classifier to train itself. The remaining 5% is used for testing which theoretically would not be a problem for the model, if those 5% rows weren't the crucial ones. In one of the three runs namely this is the case (see Table 16). Described in the beginning, there are only about 60-80 rows (≈ 3-4%) in the whole dataset representing class 5. Consequently it is highly possible that the 106 rows (5% of the whole) contains nothing/little/everything that represents class 5. If it contains nothing, then this particular class can not be tested. If it contains everything that means the training set does not have anything to help the classifier learn about class 5. Thus each test for class 5 fails here as well. If the 5% contains a good amount of data representing class 5 then the result is again up to fortune that is why 2 of the three runs actually reached more than 80% average precision. How could there NaN be possible? Supposedly the data for class 5 is evenly distributed across training and test sets. The total misclassification can occur, if all of the rows for class 5 that hold the crucial characteristics for the decision making remain in

one of the two sets. Hypothetically, if there are 20 rows in the dataset that would help distinguish between class 5 and class 1, and they all stay in the test set then the classifier has no chance to learn those characteristics thus generates a model, where everything that supposed to belong to class 5 is classified to class 1. This is a reasonable explanation, why in Figure 12 one run produced an evaluation where class 5 was completely misclassified and resulted in a NaN average precision.

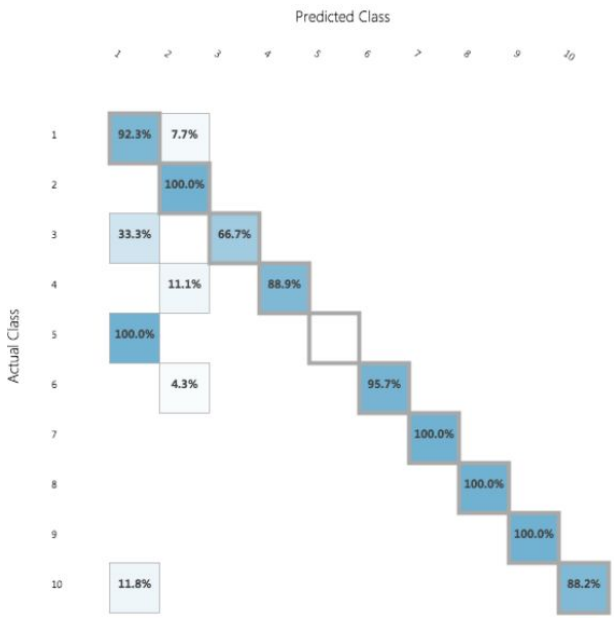


Figure 12: Confusion matrix of evaluation with 95% training and 5% test sets

The above explained behaviour can be verified by looking at the precisions of each run in Table 16. As the two split ratios come closer the variance between runs gets lower. In case of the 65/35 split the runs almost reach the same average precision, because both datasets contain a reasonable amount of samples. After that the variance gets higher again due to the critical small distribution. The best test-run has been produced with 85/15 split that reached a macro-averaged precision of 94.06%, as seen in Figure 13.

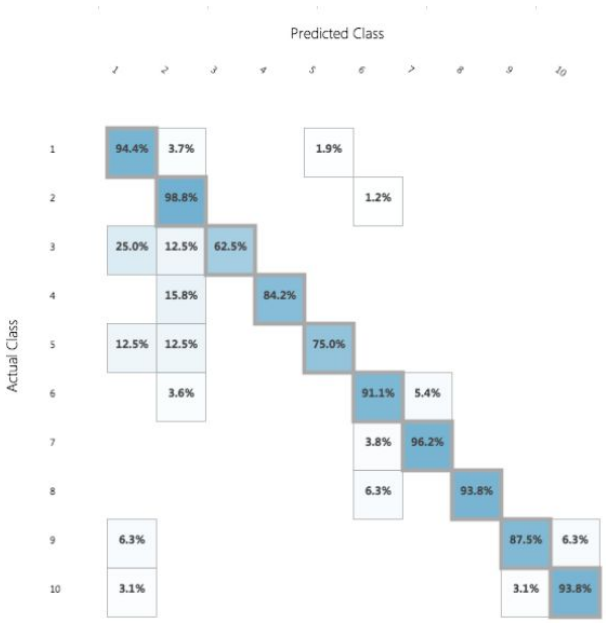


Figure 13: Confusion matrix with 85/15 split (run 1)

Figure 14 sums up the evolution of both micro- and macro-averaged precisions. The values were averaged across the three runs as well. It can be seen, that the explained first and last splits go hand in hand and produce significant worse results for being the edge cases.

Note: for better readability only the training set ratio is displayed on the x-axis. The portion of the test set should always be assumed to be its counterpart.

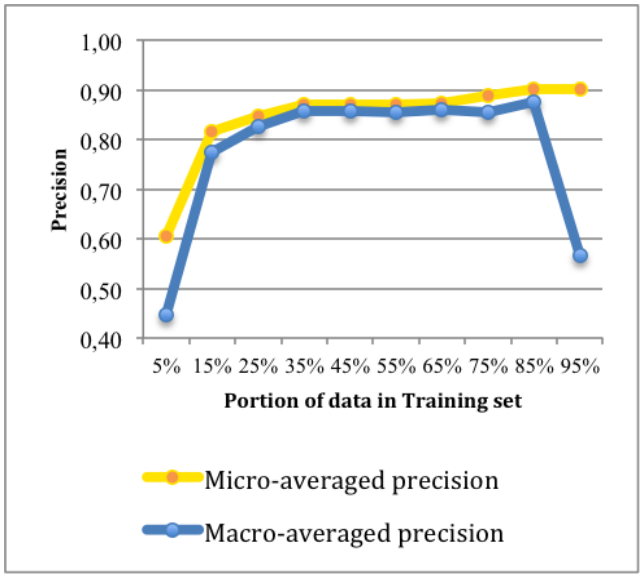


Figure 14: Comparison of Macro- & Micro-avg. precision of MDF Training/Test Set Splits (averaged across 3 runs)

6. MISSING VALUES

6.1 How we implemented it

We chose to use the Decision Forest model for the missing values, for two reasons. First, the Decision Forest gave us overall better

classification results. Secondly, the Decision Forest deals better with Missing values. The Neural Network cannot work with single empty values, it needs a cleaning strategy like removing instances or replacing the missing value with a measure.

We wrote a script in R for us to generate missing values, it is: "2_MISSING_VALUE_SKRIPT/0generateMissingValues.R". This script takes the processed file from the first step, "2cardiotocography_processed.csv" as an input. It consists of four different functions:

I. replaceRandom = function(percentMissing , newfilename)

This replaces randomly an amount of "percentMissing" values in the dataset and saves it under newfilename.

II. replaceAttributes = function(percentMissing, attributeNumber, newfilename)

This replaces the values in attribute "attributeNumber" randomly with an amount of "percentMissing" and saves it under the specified newfilename.

III. replaceWithMeanByClass = function (data , newfilename)

This takes the output from one of the functions from I or II and replaces the missing values with the mean of the attribute column by class. So it calculates the mean of each class for this attribute beforehand.

IV. replaceWithMeanOverall = function(data, newfilename)

This takes the output from one of the functions from I or II and replaces the missing values with the overall mean of the attribute, which it calculates beforehand.

The generated 18 datasets (3 types x 2 amount of missing values x 3 different strategies) are also in the folder "2_MISSING_VALUE_SKRIPT", but you can run the R-script again to generate them again.

In Azure, we set up an experiment for each of these values, as seen in Figure 15.

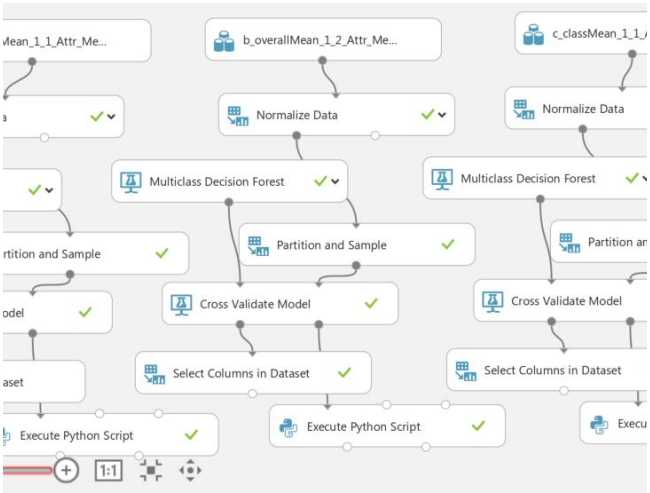


Figure 15. The data with missing values is the input, then the build-up is equal to the one in the previous chapter. We use 10-fold-cross validation again.

As column with high information gain we chose "mean", for low IG we chose "NZeros". We found this after experimenting with a leave-one-out approach of deleting columns, then running the model, then comparing.

6.2 The Results of missing values

When we insert missing values across all attributes, the results were quite expected. At 5%, we suffered a bit worse results than in the previous steps. The precision in classes 3 and 5 (not well represented) suffered especially, with precision of 0.28 and 0.33 each for 33% of missing values. When we just replace one attribute with missing values, the effect is more subtle, but still present. It is again Class 3 where this is the best visible: For 10% missing from "mean", we get a significantly lower precision of 0.75. It is interesting though, that at 50% missing, the precision for Class 3 is slightly higher. We think that is, that other attributes still compensate quite well for this specific class. However, all of our other classes perform far weaker when "mean" is missing more - especially compared to "NZeros". We see this also reflected in the macro Precision in Table 17: When we ignore missing attributes for "NZeros", the results are always better than for column "Mean".

Table 17.

Missing values strategy was applied to...	% of values missing	Strategy	macro mean Precision
ALL VALUES	*5 %*	Ignoring missing attributes	**0.839823**
		Replace with overall mean	**0.851705**
		Replace with mean p. class	**0.879632**
ALL VALUES	*33 %*	Ignoring missing attributes	**0.621689**
		Replace with overall mean	**0.701061**
		Replace with mean p. class	**0.931872**
Column "Mean"	*10 %*	Ignoring missing attributes	**0.878808**
		Replace with overall mean	**0.876335**
		Replace with mean p. class	**0.887897**
Column "Mean"	*50 %*	Ignoring missing attributes	**0.877047**
		Replace with overall mean	**0.881026**
		Replace with mean p. class	**0.912387**
Column "NZeros"	*10 %*	Ignoring missing attributes	**0.87893**
		Replace with overall mean	**0.882851**
		Replace with mean p. class	**0.882987**
Column "NZeros"	*50 %*	Ignoring missing attributes	**0.881451**
		Replace with overall mean	**0.879198**
		Replace with mean p. class	**0.90354**

Significant improvements happen when we choose a proper replacement strategy. We see an improvement in every case - except for "Nzeros" at 50% (interestingly, ignoring is slightly better than replacing with overall mean. However, also here, replacement by class is better than the other two approaches).

What is really stunning, is the replacement by mean of class. Well, stunning at first. We have the best result of all (Precision = 0.93) when we have a lot of missing values in all attributes and replace them with the means. But since this applies to the test data as well, it is obvious why we have a better classification - the test and training data have the same values with an increasing amount of missing values (up to a point). However, the obtained model would be worthless when applied to real world data, as this would probably vary by a big amount.

We should be aware of this "overfitting", because it is probably the reason in some of the vast improvements when using per class average. In general, however, we see that the results improve when using this approach - per class and also macro averaged, as seen in Table 17.

7. SUMMARY AND LESSONS LEARNED

In this experiment we performed data mining on a dataset about fetal heart rate patterns with the help of Microsoft's Azure Machine Learning Studio. During the experiment mainly 2 structurally different classification algorithms, Multiclass Decision Forest and Neural Networks, were examined, tuned, trained, tested and evaluated by taking precision as a deciding factor. Using the best parameters for both algorithms, the Multiclass Decision Forest seemed to perform better than Neural Networks, however by only a slight difference of 2%.

We learned that it takes a lot of time to inspect and clean up the data. It was interesting that finding good scaling methods was very tricky, and the combination of different approaches does not always perform better on the result than done separately. It was interesting that for the test splits, the best results were found somewhere in the middle.

An important lesson is, that if we have surprisingly good results - like above in the missing values with per-class mean replacement - something fishy is going on with our model or our data. Here, it would have probably been better to split the data and then insert missing values only in our training data. However, it was described in the project assignment to do it this way and then do 10-fold cross validation, so we sticked to that.

In the retrospect it turned out that the big difference between representation of classes is an issue. While class 2 takes up almost 27% of the whole dataset, classes 3, 4, 5 and 9 do not even reach 5% individually. This unfavourable distribution caused many rows getting misclassified in case of the above classes especially when training/test set split tests were conducted where the randomization and therefore luck was dominant. A good distribution would have been 10% for 10 classes of course. However, from an alternative point of view, the original class distribution would have been manageable, if the dataset were bigger. For this, we simply duplicated the dataset with the "Add rows" module resulting in a set of 4230 rows. With this amount, 5% of the data equals to about 212 rows which seemed to be sufficient for the classifier since a simple test-case with the Multiclass Decision Forest using the best parameters and set-up from 5.1 resulted in an accuracy of almost 99%. As a result, either the dataset should be bigger, or be stratified in the first place. Since the last part is unlikely to happen in real life, a deep analysis should be conducted to find dominating features so that these can be stratified both manually and during training/test set splits.

All presented tables, figures along with other experiments can be found in the project archive.