

# 1 INTRODUCTION

This report covers solving the data mining of the water table. It consists of four tasks; each task has its own folder. There are saved training data, testing data, predicted labels and models. The whole program is written in the `main.py` file. Design, implementation and evaluation are further described below.

## Running the code instructions

To run particular task navigate to the `src` folder in the command window and write `python main.py <task_id> <train/test/predict> <other_arguments>` where task ID, action type and other are arguments.

There are four types of task parameter: `task1`, `task2`, `task3` and `task4`. Each task has three action argument `train`, `test` and `predict`.

Folders already contain a trained model, but if the user changes any parameters, the model must be trained before testing or predicting.

Task 4 is about solving imbalanced data set problems; therefore, there is a third argument that specifies what method to use to process imbalanced data. These arguments are *over* (for over-sampling), *under* (for under-sampling), and *smote* (for SMOTE over-sampling technique).

Example of a command for calling task4: `python main.py task4 train smote`

For task 4, there always have to be the third argument; otherwise, it will throw an error. If the smote model is trained, then test and predict also must have smote argument. By default, there is a saved smote model. If the user wants to use a different technique, it must be re-trained

—Sklearn version 0.24.1.

## 2 DESIGN, IMPLEMENTATION AND EVALUATION

The whole program is run from the `main.py` file, and it contains all functions needed for each task. It starts with running the program with arguments corresponding to the particular task. The program accepts the arguments and decides which function to run. There are two main functions `task1()` function for solving the first task and `task234()` function used for task 2-4. Function `task234()` then further decides how to behave according to other arguments. For task 4, the third argument is needed, as mentioned above. The program also contains other functions that are called by these two main functions according to the arguments. These functions will be further described subsequently.

Function `task1()` is used for solving the first task, and it accepts *act* argument, which decides whether the action is training, testing or predicting. It loads and splits the data from the csv document according to the action. Further, it drops unnecessary columns.

When the action is training, it creates a pipeline with a standard scaler and model and passes this pipeline to the `train()` function that trains the model. The train function is used across different tasks, and its purpose is to train models and display accuracy metrics. After this, the trained model is saved.

Test action causes loading the trained model, loading appropriate data and calling `test()` function with five parameters (loaded model, X test data, y test data, label used for encoding y values and task argument). This function is again used across the tasks, and it is used to make predictions on the loaded model and print accuracy metrics. It also saves predicted labels to the text file.

Predict action loads model and the document without labels, prepare data, extracts id column for id predictions, and call `predict()` function with three arguments (X data, loaded model and id column data). The function is used for all tasks, and it serves to make predictions on the dataset without correct labels and print ID of pumps that are predicted to be functional but needs repair.

Function `task234()` works completely the same way for task 2 and 3; for task 4, it performs additional functionality that is handling imbalanced data. Firstly, it loads appropriate data and calls on them `cleanData()` function. This function drops unnecessary columns and replaces unknown values with NaN according to the data analysis described in the task 2 section below.

If the action is train, it finds duplicate data and drops them not to create bias on training. It then creates two pipelines, one for categorical data with simple imputer imputing most frequent values for NaNs and `OneHotEncoder` encoding categorical data, the other one for numerical data with mean values imputing and `StandardScaler` for scaling numeric values. These pipelines are put together to the column transformer that specifies to which columns it should be applied. The NN (neural network) model

(MLPClassifier) is created and added to the pipeline together with the column transformer. Then it calls the train function a saves trained model. Train and test part works the same as in the task1() function.

Task 4 has additional functionality triggered by an imbalance argument entered into the task234() function. This argument decides which approach is chosen to deal with an imbalanced dataset, and it can take three different forms. The first one is 'under', which represents the under-sampling approach. Function underSample() is a manually created function that deletes random rows of majority class to have the same count as the minority class. The second one is 'over', which represents the over-sampling approach. It uses imblearn RandomOverSample function that creates randomly more representations of minority class by copying existing cases. The third option is 'smote', an over-sampling technique that uses 5-nearest neighbours of minority class and creates a new sample according to them. In this part, the implementation was quite challenging because we do not want to apply smote on the test set, and the data also must be pre-processed before applying smote. This was solved by saving the pre-processor as well as the model and then opening and applying it again when testing.

It was described in general how the program works, and next, it will go through each task.

## 2.1 Task 1

Jupyter notebook was used for the data analysis, data cleaning, and finding the most suitable neural network parameters. Its setup is used in html file called A3jupTask1 in the JupyterCodeUsedForAnalysis folder.

First, the whole dataset was inspected. There are no missing values, and the data types were also checked. The histograms of the data can be analysed to find any other interesting context, such as that they do not have a normal distribution and some outliers can be spotted.

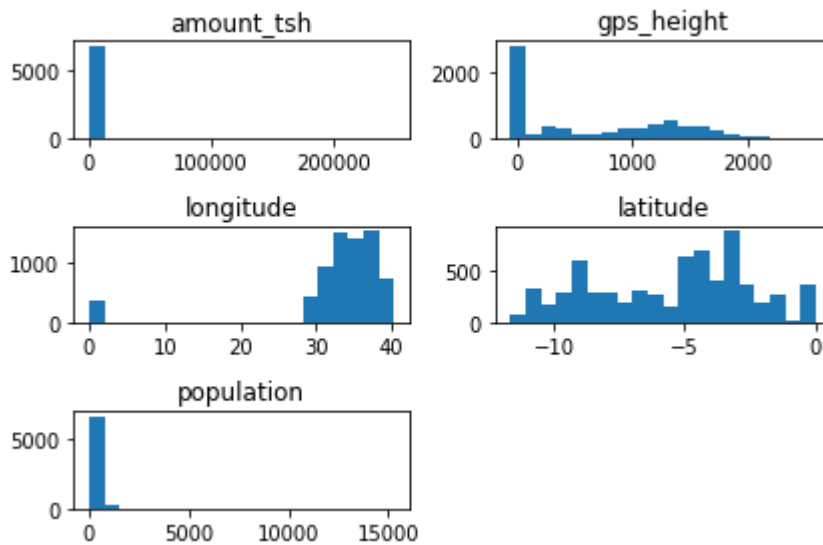


Figure 1: Histograms of the training data set

Id and private\_num columns were dropped because there does not seem to be any reasonable relation between the private number or id and wells that need to be repaired, and thus such data are not relevant for prediction.

After cleaning the data, they were split into X and y and then further divided into training and validation in the 80/20 ratio. This was done in order to select the best parameters for MLPClassifier without peeking into the test set. This was done only for analysis, and the program A3main.py trains on the full dataset.

Task 1 dataset contains only numerical values, and there are no missing values; thus, only the scaling was applied, for data scaling standardisation was chosen. Unlike normalisation, standardisation does not have a bounding range. So, even if there are outliers in your data, they will not be affected by standardisation. After testing, there was no significant difference between normalisation and standardisation.

The pipeline is created with the pre-processor and the model. The pipeline is then fitted with training data, saved and later used for predictions while testing.

## Evaluation

MLPClassifier always uses Softmax activation function for the output layer and Cross-Entropy loss function for the training; it does not need to specify that there. The default weight initialisation method in scikit-learn is Xavier/Glorot method.

To find out how many hidden units to use, the trial-error approach was used with the tactic to start small and increase. Hidden layers have all the same activation function. As an activation function TanH (hyperbolic tangent) function was used. It is more computationally expensive, but it performed well and since the training time was not that high. I have decided to stick with this function. It also has a nice property that scales everything between -1 and 1. Other parameters were batch size-100, for finding minimum stochastic gradient descend was used, learning rate-0.001 with early stopping that stops the training when the model started showing overfitting behaviour and it also stopped when the performance was not changing for 10 epochs.

These parameters were changed and tried with different NN dimensions. The best performance it got on training was 0.61 and 0.7 on the validation set with 2 hidden layers with 10 nodes in the first one and 11 in the second one. Accuracy on the test set was 0.6.

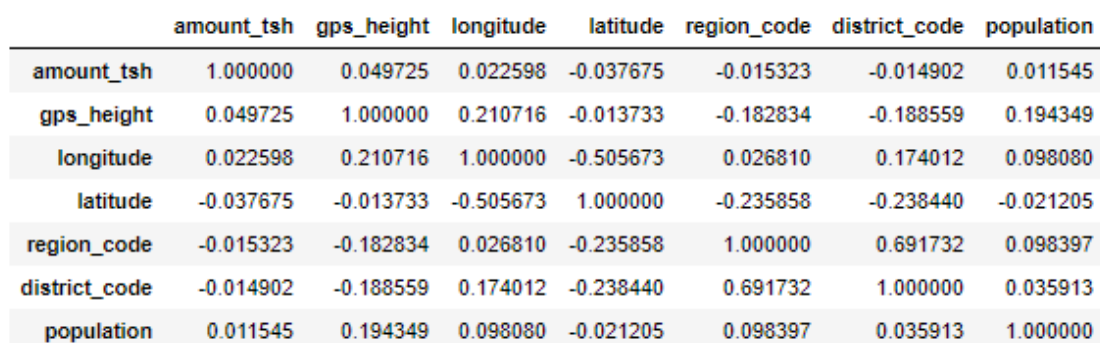
The linear regression model was also tested for comparison, which performed with 0.58 training accuracy.

We are getting a low score on the training set, which indicates that underfitting adding more hidden layers or units or other changes of the neural network model is not helping; thus, there must be insufficient input data and features.

## 2.2 Task 2

Similarly, as in task 1, the Jupyter notebook was used for data analysis saved as A3dataAnalysis.html. It shows what was used for the analysis of the data and how it was done cleaning the data. It was also used to find the best parameters for MLPClassifier.

Feature selection is important because it leads to a better performing model, easier to understand model, and a faster model. It starts with dropping the columns that are obviously not relevant for the prediction, such as id or recorded\_by. The next steps look at correlation and multiplied information.



	amount_tsh	gps_height	longitude	latitude	region_code	district_code	population
amount_tsh	1.000000	0.049725	0.022598	-0.037675	-0.015323	-0.014902	0.011545
gps_height	0.049725	1.000000	0.210716	-0.013733	-0.182834	-0.188559	0.194349
longitude	0.022598	0.210716	1.000000	-0.505673	0.026810	0.174012	0.098080
latitude	-0.037675	-0.013733	-0.505673	1.000000	-0.235858	-0.238440	-0.021205
region_code	-0.015323	-0.182834	0.026810	-0.235858	1.000000	0.691732	0.098397
district_code	-0.014902	-0.188559	0.174012	-0.238440	0.691732	1.000000	0.035913
population	0.011545	0.194349	0.098080	-0.021205	0.098397	0.035913	1.000000

Figure 2: Checking correlation

The features that were highly correlated or were the same information twice were deleted. For example, the quality column and group\_quality column is doubled

information. Further, there is the dropping of columns according to less importance or missing values to make the data set as clean and simple as possible. It drops subvillage because 2/3 of the training data have different subvillage. The construction year was also dropped since half of the data were missing. Data types were checked, and the unknown values in the data were changed to NaN. Throughout the analysis, csv analysis documents were created and are accessible in the JupyterCodeUsedForAnalysis folder.

When the cleaning of the data was done, the data pre-processor that is placed in the pipeline with the model was created. Pre-processor uses a simple imputer to substitute NaN values with the most frequent values. It uses standardisation for numerical data and OneHotEncoder to encode categorical data into numbers.

The same principle of data processing is used in task 3 and 4.

## Evaluation

The functioning of the MLPClassifier was mentioned in task 1. In this task activation function, ReLu was used since it has faster performance, and the data set is larger now and get even larger in the next tasks. Stochastic gradient descendant is used as well as early stopping.

Again it uses a validation set to evaluate the best parameters; many different parameter combinations were tested (some of them in the table below).

Hidden layers	Batch size	Learning rate	Iteration no change	Train accuracy	Validation accuracy	
10, 7	100	0.001	10	78%	72%	
20,20,5	50	0.0009	70	83%	75%	Suggests overfitting
10, 10	100	0.001	50	81%	90%	
15,15,5	100	0.001	50	83%	85%	

It got similar results for different combinations, but to stick with Occam's razor principle, it is better to keep it as simple as possible. Therefore, the parameters with 10,10 hidden layers were chosen.

Training on the full set then achieved 87% accuracy and 76,6% on the test set. These results suggest that the model is not generalising well.

## 2.3 Task 3

### Evaluation

This task is dealing with imbalanced data set using the same algorithm used for task 2. When the algorithm is run on the dataset given for task 3, it achieves 0.936% accuracy, which seems pretty good. However, accuracy on imbalanced data is not necessarily a good performance metric. It means that the functional needs repair class covers only like

10% of the dataset; therefore, if the algorithm only predicts class others, it would achieve 90% accuracy. For these occasions, accuracy is generally not the preferred performance measure, especially when dealing with skewed datasets (when some classes are much more frequent than others).

```

Accuracy on training set: 0.9368775723455527

```

	precision	recall	f1-score	support
functional needs repair	0.72	0.19	0.30	3337
others	0.94	0.99	0.97	43556
accuracy			0.94	46893
macro avg	0.83	0.59	0.63	46893
weighted avg	0.93	0.94	0.92	46893

Figure 3: Task 3 train imbalanced data results

To measure such data sets' performance, the confusion matrix is used with precision and recall score (figure 3). Precision and recall are then put together into an f1-score. As we can see, the f1-score for the functional needs repair class is now very low (30%). Similar results were received for the test set in figure 4. Here is also a displayed confusion matrix showing true positive, true negative, false-positive and false-negative cases.

```

Accuracy on test set: 0.931986531986532
Confussion matrix:
[[ 157  729]
 [   79 10915]]

```

	precision	recall	f1-score	support
functional needs repair	0.67	0.18	0.28	886
others	0.94	0.99	0.96	10994

Figure 4: Test imbalanced data results

It is possible to compare the Confusion matrix from task 2 in figure 5. As you can see, the matrix here is balanced.

```

Confussion matrix:
[[687 189]
 [215 636]]

```

Figure 5: Task 2 confusion matrix

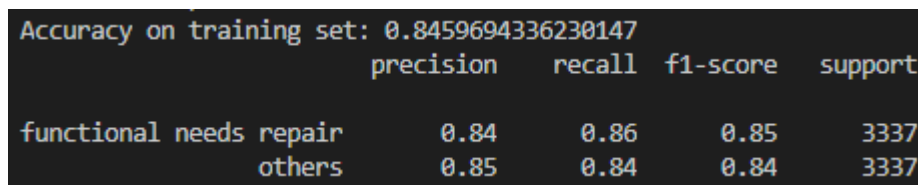
## 2.4 Task 4

### Evaluation

This task aimed to propose a solution to the imbalanced data set problem. The presented algorithm implemented three different solutions, which can be called from the command line with parameters over, under or smote.

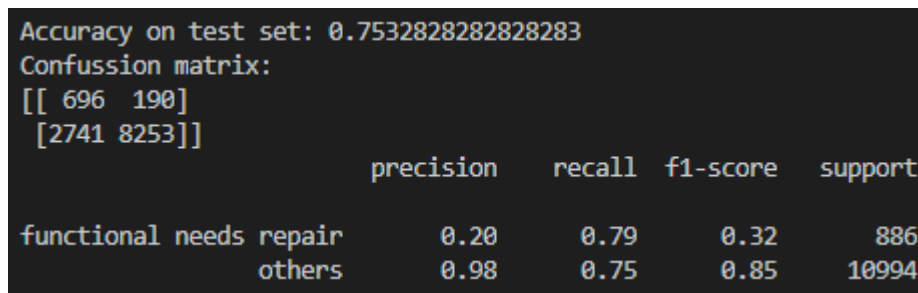
The first possible solution is under-sampling. A function takes the dataset and randomly deletes samples majority class until the classes are balanced. A significant disadvantage of this technique is that we are losing most of the data.

Figure 6 and 7 presents training and testing results. It can be seen that the algorithm is not performing well on the test data; thus, it is not generalising well.



Accuracy on training set: 0.8459694336230147					
	precision	recall	f1-score	support	
functional needs repair	0.84	0.86	0.85	3337	
others	0.85	0.84	0.84	3337	

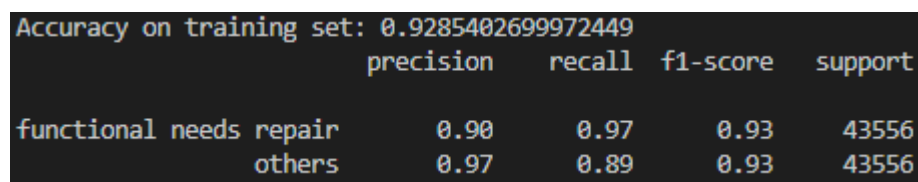
Figure 6: Under-sampling train results



Accuracy on test set: 0.7532828282828283					
Confussion matrix:					
[[ 696 190]					
[2741 8253]]					
	precision	recall	f1-score	support	
functional needs repair	0.20	0.79	0.32	886	
others	0.98	0.75	0.85	10994	

Figure 7: Under-sampling test results

The second solution is over-sampling the data. It randomly copies minority class occurrences until the data are balanced. Figure 8 and 9 present training and testing results. We can see that its performance is high on the train data, and it correctly classifies most of them, but the score on the test data is low again. However, if the f1-score from task 3 (28%) is compared, we can see that the performance was increased by 10%.



Accuracy on training set: 0.9285402699972449					
	precision	recall	f1-score	support	
functional needs repair	0.90	0.97	0.93	43556	
others	0.97	0.89	0.93	43556	

Figure 8: Over-sampling train results



```

Accuracy on test set: 0.8458754208754209
Confussion matrix:
[[ 556  330]
 [1501 9493]]

```

		precision	recall	f1-score	support
functional needs	repair	0.27	0.63	0.38	886
	others	0.97	0.86	0.91	10994

Figure 9: Over-sampling test results

The third option is to use SMOTE over-sampling algorithm. It creates new samples for the minority class according to the original data. It considers 5 neighbouring samples and produces new according to them. Its performance on the train set is a bit better than just a simple over-sampling technique. Nevertheless, the test set's performance remains low; however, compared to the f1-score from task 3; it improved by 10%.

```

Accuracy on training set: 0.9367366149325007

```

		precision	recall	f1-score	support
functional needs	repair	0.90	0.98	0.94	43556
	others	0.97	0.90	0.93	43556

Figure 10: SMOTE train results

```

Accuracy on test set: 0.8512626262626263
Confussion matrix:
[[ 538  348]
 [1419 9575]]

```

		precision	recall	f1-score	support
functional needs	repair	0.27	0.61	0.38	886
	others	0.96	0.87	0.92	10994

Figure 11: SMOTE test results