CS5011 Artificial Intelligence Practice: Assignment 3 Learning

170004680

University of St. Andrews

April 2021

Contents

A Introduction

In this assessment, artificial neural networks (NNs) have been constructed and used to solve a water-pump classification problem with real-world data. The problem and data present various challenges, which have been addressed individually through incremental tasks.

Each task can be executed according to the instructions provided in section A.2. The design and implementation of the system is reported in section B.1. An evaluation of the developed NN solutions is carried out in section B.2. Robust correctness testing of the system is evidenced in section C.

A.1 Checklist

Table 1: Table indicating the completeness of this submission with respect to the specification requirements.

Task	**Section**	**Status**
1 (Balanced, Numeric Data Model)	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.
2 (Balanced, Categorical Data Model)	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.
3 (Unbalanced Data Comparison)	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.
4 (Unbalanced Data Model)	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.
5 (Imbalanced Multi-Class Classification)	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.
6 (Hyper-parameter Optimisation)	Code	Attempted.
	Report	Attempted.

A.2 Compilation & Execution Instructions

From within the `A3src/` directory, the program resources can be acquired with `maven` using the following command:

```
./maven-build.sh
```

From within the `A3src/` directory, the `A3main` program can be compiled using (see figure 1):

```
export CLASSPATH=.:target/dependency/*:$CLASSPATH

javac *.java
```

After compiling the program in the `A3src/` directory, `A3main` can be executed using (see figure 2):

```
java A3main <task_id> <train|test|predict> <other_arguments> [-wl|-us|-os] [-o]
```

, where the program parameters are as described in the practical specification. The '[-o]' parameter is an optional argument indicating for the program to provide a full/test output. The '[-wl|-us|-os]' arguments are required for all tasks from task 4 as it specifies which class balancing technique to use ('-wl' is weighted loss function, '-us' is under-sampling, and '-os' is over-sampling). For example;

```
java A3main task4 train input/task3_train.csv task4_NN -wl -o

java A3main task4 test input/task3_test.csv task4_NN test4.txt -o
```

Figure 1: The following screenshot shows the program can be compiled within the St. Andrews School of Computer Science lab environment without error. All of the `.java` files have been compiled into `.class` files usable for program execution.

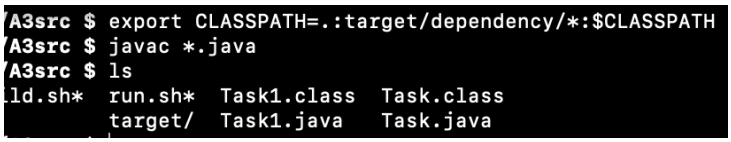
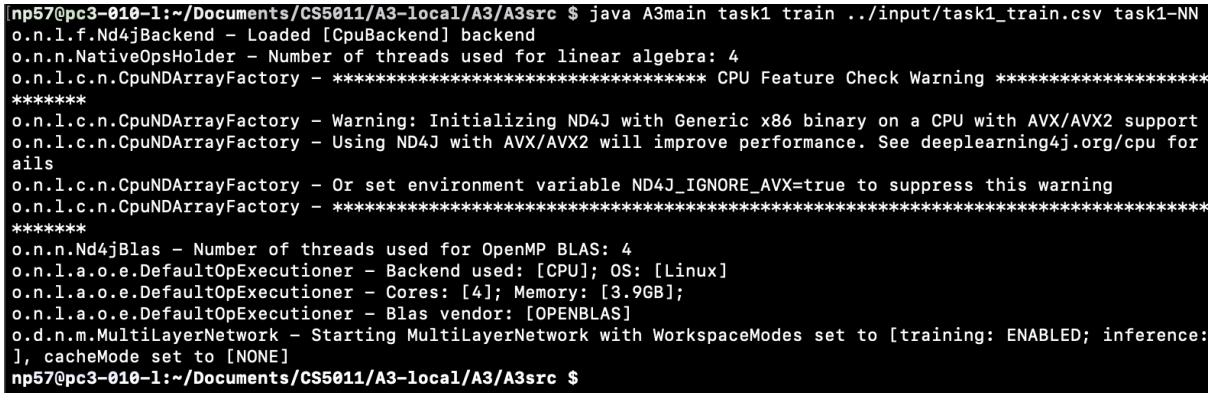


Figure 2: The following screenshot shows the program can be executed within the St. Andrews School of Computer Science lab environment without error.



B Design, Implementation and Evaluation

B.1 Design and Implementation (Word Count: ~ 1100)

B.1.1 Common

Core functionality has been implemented in a common `Task` class. This maintains code modularity and re-usability by recognising that mainly the schema (structure of the input data set), transform process (pre-processing steps), and NN configuration (parameters that define the NN) changes between the individual tasks.

Pre-processing has been split into two functions: one for training data sets, another for testing/prediction data sets. Pre-processing a training data set requires an analysis to be calculated on the features so that steps, such as data scaling, can occur. For pre-processing non-training data sets, the analysis used when training the model should be used instead, which is read from a file where the corresponding NN is saved. As a result, a data set iterator is created which applies the specified pre-processing steps to data batches. This is beneficial as the whole data set is not loaded into memory, though, this introduces the 'batch size' as a parameter to tune.

The NN is instantiated with the same seed for reproducibility. Early-stopping has been implemented for training, which allows the number of training epochs to be optimised to prevent over-fitting. A stopping condition is specified; maximising the classification accuracy until no improvements are made for several epochs. However, minimising the loss function is more beneficial in general as accuracy is not the only important metric to consider.

The 'test' functionality produces evaluation statistics other than just classification accuracy. Other useful metrics are given so the effectiveness of the model can be better analysed: precision, recall, and F1 score.

The 'predict' functionality does not assume the 'ID' feature will be present in the pre-processed data. Thus, predicted water-point 'IDs' for positively classified samples can be output even if 'ID' is dropped as a feature via feature selection.

B.1.2 Task 1

Through data inspection, the data types of the numeric input features was discerned: *id*, *gps_height*, *num_private*, and *population* are integers, whilst *longitude*, *latitude*, and *amount_tsh* are doubles.

Several pre-processing steps are required. The output, *status_group*, is converted ("functional needs repair" as 1, and "others" as 0). This is required as the NN output is numeric and must represent the output classes. All water-point identifiers are unique, therefore, the *id* feature is dropped as it will have negligible correlation with the output. The numeric features are also scaled to assist with training. The type of scaling performed on a feature is determined by its data distribution. However, the scaling used was also investigated empirically to get the best results (figure 3). The *longitude* feature contained zero values, which must represent missing data entries (zero longitude is not Tanzania). As a result, zero values are replaced with the feature mean of ~ 32.888, as *longitude* is standardised.

The NN uses stochastic gradient descent (recommended by FAQ). Cross entropy (multi-class in its binary form in anticipation of future tasks) has been used as the loss function because it minimises negative log-likelihood of the observed data (NN can be optimised) and often leads to better results with gradient-based methods. The logarithmic cross-entropy function also assists training by undoing exponential components introduced by soft-max output activation. Two hidden layers have been used as this appears to be assumed by the specification. The input layer has as many input neurons as there are input features (6). Two output neurons are in the output layer; one for each of the output classes (required for multi-class cross entropy). *Xavier* weight initialisation has been used as it initialises the NN based on its scale, which avoids initial weights being too small or too large. *Relu* non-output activation is used as default with the `DL4J` library. The implementation could be improved if the activation function used by the hidden layers was empirically investigated, as is the case with the rest of the NN hyper-parameters in figure 4.

Figure 3: The following screenshot is a summary of results when empirically investigating what type of data scaling to use on each numerical feature. Each input feature was investigated in isolation as the sole feature used by the model. The model configuration (2 hidden layers of 20 hidden neurons, 2 output neurons, stochastic gradient descent, etc.) was controlled in this investigation. The type of scaling was varied for each feature to see how it would affect model performance on the test data set. Multiple runs were performed to calculate an average performance (by varying the seed used by the model). As a result, standardisation is used for *gps_height*, *longitude*, *latitude*, and *population*. `Log2Mean` scaling has been used for *amount_tsh* and *num_private* (*num_private* has a large value range). Note that using empirical results to inform on the data scaling best for each feature is dependant on the given training and testing sets being representative of the general data distribution within each feature. Without domain knowledge, an empirical investigation was suitable.

	Average Test Set Accuracy:		
Features:	Standardise	MinMax	Log2Mean
amount_tsh	0.509	0.507	0.513
gps_height	0.546	0.536	0.537
longitude	0.589	0.558	0.557
latitude	0.585	0.557	0.552
num_private	0.508	0.507	0.508
population	0.512	0.508	0.508

Figure 4: The following screenshot is a summary of results when empirically investigating the hyper-parameter values of the NN configuration. The learning rate, number of hidden neurons per layer, and batch size have been investigated. Parameter values have been varied in isolation (pre-processing steps and NN parameters not being investigated were controlled for all experiments), along with the configuration seed, to generate an average model performance for each independent variable value. Values that maximise the test set accuracy were chosen (or the value that creates the simplest model when results are comparable).

Learning Rate	0.1	0.01	0.001			
Average Test Set Accuracy	0.659	0.613	0.588			
Num Hidden Neurons	5	10	20	40	80	160
Average Test Set Accuracy	0.611	0.629	0.659	0.664	0.662	0.666
Batch Size	100	250	500	1000	3000	6098
Average Test Set Accuracy	0.606	0.607	0.615	0.613	0.597	0.594

B.1.3 Task 2

Numerous categorical features have been added to the schema. The NN configuration is identical to task 1, so an evaluation of how the additional categorical features improves prediction can occur.

The categorical features are pre-processed by taking the top ten most frequent category values for each feature. All other values are less frequent and are encoded into an "*Other*" category. This implementation could be improved by taking a dynamic amount of most frequent categorical values for each feature, such as the top 10% most common. However, this was not easily achievable with the `DL4J` library. The categorical features are then one-hot encoded (converted to numerical form without introducing artificial relationships between feature values). Missing values that exist within the categorical features are simply encoded into the less frequent category, though future work should investigate more sophisticated means of data imputation.

B.1.4 Task 4

Three different techniques for handling class imbalance are investigated: weighted loss, under-sampling, and over-sampling.

The implementation of the weighted loss function was trivial with the `DL4J` library; a set of weights for each class is specified with the cross-entropy loss function. The majority negative class ("others") was assigned a weight of 0.07, whilst the minority positive class ("functional needs repair") was assigned a weight of 1. The weight of the majority class is ~ 14 times smaller as it is ~ 14 times more frequent in the training data set. The task 4 model may still be improved with optimisation of the assigned class weights.

For re-sampling, the training data is first converted into a new re-sampled CSV file, which is used as input for training. Ideally, re-sampling should be implemented in the transform process of the pre-processing stage; however, this is difficult with the `DL4J` library. Despite this, over- and under-sampling is achieved by reading in batches of the original CSV file and duplicating/removing (respectively) random samples as required to balance the classes.

B.1.5 Task 5

Minor changes were required to extend the task 4 program to be compatible for multi-class imbalanced classification. The loss function used was already multi-class cross-entropy in its binary form, which has been increased to three output classes. The schema and pre-processing steps were updated; the output column ("*status_group*") is pre-processed to map "non functional" to 0, "functional needs repair" to 1 (as before), and "functional" to 2.

The weights used in the weighted loss function were adjusted to account for the additional output class. The minority class ("functional needs repair") has a high weight relative to the other classes for handling imbalance. The weights are assigned using the same logic as discussed for task 4. The re-sampling implementation is identical for task 5. The re-sampling does not guarantee that all output classes will be equal in their share of the output distribution; it guarantees the minority positive class will not be severely imbalanced.

B.1.6 Task 6

The `arbiter` library was investigated for implementing hyper-parameter optimisation. Hyper-parameter spaces were defined for all NN hyper-parameters (learning rate, loss function weights, etc). Random search can be performed to search through the hyper-parameter spaces to optimise for some condition (e.g., maximise F1 score). However, the task 6 implementation was not completed due to difficulties with the `DL4J` API. The lack of documentation for `DL4J` was a progress hindrance with this submission in general.

B.2 Evaluation (Word Counts: < 300 Per Task)

This section seeks to evaluate and compare the effectiveness of the trained NNs for each task. It is assumed the issue of data leakage has been addressed in the provided training and testing data sets for each task. Preventing data leakage is important as ensuring the training data sets have no 'knowledge' of the testing data sets will provide more accurate and realistic analyses for how the models perform on unseen data.

B.2.1 Experimental Setup

The experimental setup was constant for all tasks to allow for valid comparisons between the results of each task.

For every task evaluation, the NN configuration and training methodology established in task 1 was used (controlled variables). Thus, comparisons of results between the tasks demonstrate how the performance of a given model configuration and training technique can be affected by addressing the various issues between the different data sets.

For every task, performance metrics (dependant variables) were attained by executing multiple training runs with the NN using different seeds (independent variable). The seed used by the NN affects weight initialisation, which can provide the NN with a 'head-start'. Thus, an average performance of the NN was calculated across a set of seeds for each task. More accurate results could be attained by executing more runs; however, this was trading off for evaluation time. Performance metrics are calculated on the testing set of each task so the performance of the NN for unseen data can be analysed (issue here raised in section B.2.8).

B.2.2 Evaluation Summaries

The task 1 model, using only the numerical input features, has a better than chance performance at predicting water-points requiring repair. However, the model exhibits relatively low performance.

In task 2, the addition of categorical features resulted in an improvement to model performance; the accuracy, precision, recall, and F1 score all significantly increased.

In task 3, the training set is severely class imbalanced and the resulting model has a very high accuracy. However, this is misleading; the model cannot be said to perform sufficiently well on account of its low precision, recall, and F1 score.

Task 4 implements three separate techniques for handling the severe class imbalance: weighted loss, under-sampling, and over-sampling. The use of all three techniques results in models that improve on the weaknesses of the task 3 model, though performance is still not sufficiently high. The results for task 4 show no clear favourable technique for handling the class imbalance and further investigations are required.

In task 5, the multi-class classification models perform better on average than the binary-class alternatives in task 4 (both sets of models handle severe class imbalance). However, this comparison is not as well-defined, and cost-sensitive multi-class classification could not be evaluated as desired for more realistic model performance analysis.

B.2.3 Task 1

The main performance metric considered is accuracy, which measures the proportion of all predictions that are correct. The model has an average accuracy of just 0.617 (figure 5), which is better than making a random prediction, but not significantly so. The NN and pre-processing steps have basic tuning to maximise accuracy. Therefore, the low accuracy arises from the task 1 numerical features not being complete enough to allow the NN to confidently predict the output; the numerical features in combination do not sufficiently correlate with the status of a water-point.

Further important metrics are also provided for comparison with later tasks:

- Precision - The ratio of correct positive predictions to the total number of positive predictions (i.e., of all the water-points predicted as needing repairs, how many actually require repairs?). In the results, only $\sim 63\%$ of water-points predicted as needing repairs, actually require repairs.
- Recall - The ratio of correct positive predictions to the total number of positive observations (i.e., of all the water-points that actually need repairs, how many were predicted as needing repairs?). In the results, only $\sim 59\%$ of water-points that need repairing were predicted as such.
- F1 score - The weighted average of precision and recall. In context, this depends on the weighting of importance given to the cost of missing a broken water-point compared to the cost of sending engineers to fix a perfectly functional water-point.

Figure 5: Table of summary results for the evaluation of Task 1.

Task 1:	Average
Accuracy	0.617
Precision	0.632
Recall	0.585
F1 Score	0.607
True Negatives	553
True Positives	512
False Negatives	384
False Positives	298

B.2.4 Task 2

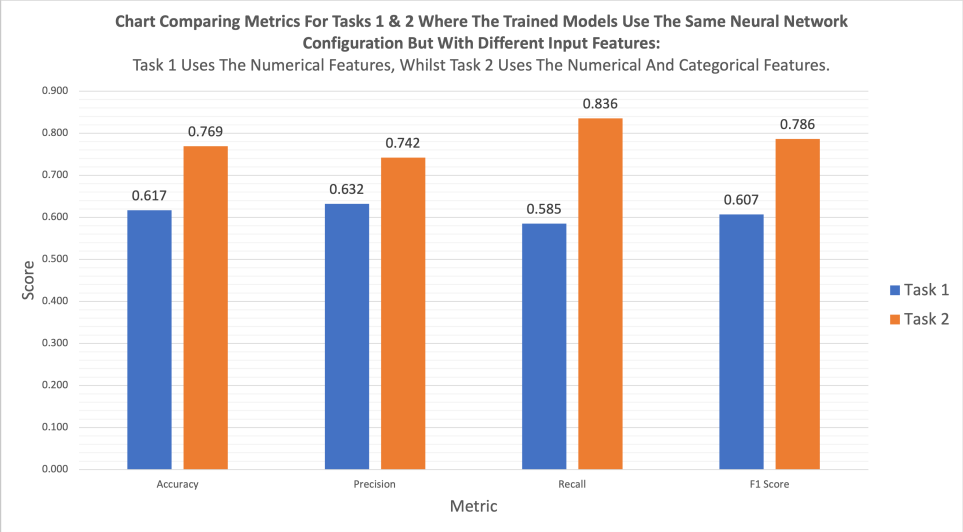
As a result of adding the categorical features, the NN generally performs well (figure 6); the main metric being considered, accuracy, has increased by $\sim 25\%$ from 0.617 to 0.769. The addition of the categorical features allowed for the NN to discover sufficient patterns amongst all of the features that are better at predicting water-points needing repairs.

The chart shown in figure 7 compares all of the summarised metrics between tasks 1 and 2. The task 2 model better predicts the output classes compared to task 1, so the number of false positives and false negatives are reduced. This means that less water-points requiring repairs are being missed by the model, and less of the predicted water-points requiring repairs are incorrect. Thus, precision, recall, and the F1 score are increased: precision has increased by $\sim 17\%$ (to 0.742 from 0.632), recall has increased by $\sim 43\%$ (to 0.836 from 0.585), and the F1 score has increased by $\sim 29\%$ (to 0.786 from 0.607).

Figure 6: Table of summary results for the evaluation of Task 2.

Task 2:	Average
Accuracy	0.769
Precision	0.742
Recall	0.836
F1 Score	0.786
True Negatives	596
True Positives	732
False Negatives	144
False Positives	255

Figure 7: Chart comparison of the summary results between tasks 1 and 2. As seen, the addition of the categorical features in task 2 results in better NN performance across all metrics (accuracy is focused for these tasks).



B.2.5 Task 3

Task 3 uses all of the input features like task 2, but the task 3 data sets are heavily imbalanced. The task 3 training data has a majority negative output class ("others") with 92.8% of the distribution. This is expected of the real-world data as a water-point needing repairs is a rare event.

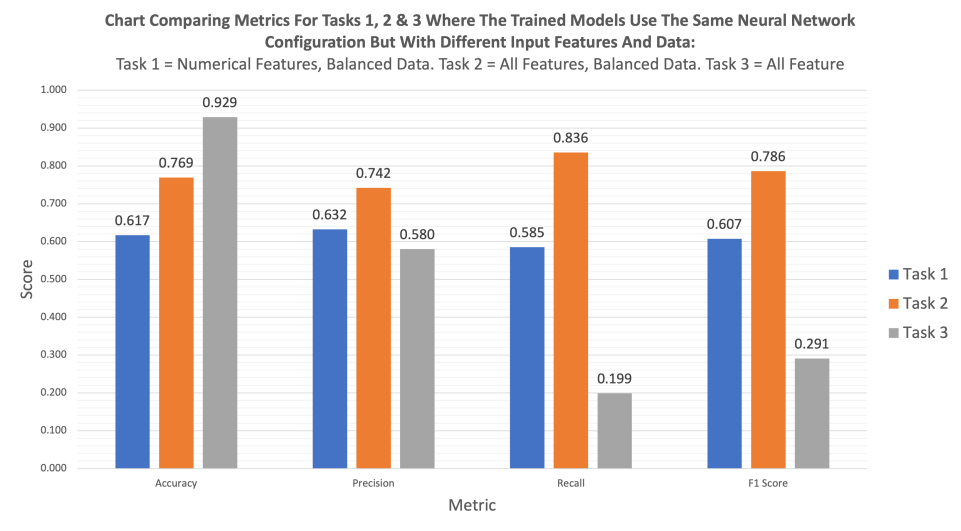
For severely imbalanced data, the minority class ("functional needs repair") is harder to predict because there are few examples of this class; the NN becomes bias to the majority class. When updating the NN using stochastic gradient descent to minimise the loss function, most of the epochs change the parameter values in the direction which allows for correct classification of the majority class. Thus, there is a frequency bias where more emphasis is placed on learning from data observations that occur more commonly.

The effects of the severe class imbalance are seen in figures 8 and 9. The NN predicts the negative output class a vast majority ($\sim 93\%$) of the time, which gives a significantly high accuracy of 0.929. The task 3 model appears to be better than tasks 1 and 2 with accuracy scores of 0.617 and 0.769, respectively. However, this statement is misleading because the positive class is only predicted correctly 58% of the time, and is frequently missed as only $\sim 20\%$ of positive cases are predicted. This demonstrates the inadequacy of the accuracy metric when there is severe class imbalance. Precision and recall are more suitable as they inform on the proportion of positive predictions that are correct and the proportion of positive samples that are predicted. Also, the F1 score is a useful metric to analyse as it summarises both precision and recall (can be optimised by the model to balance both metrics). All three of these metrics decreased with task 3, dramatically so in terms of recall and F1 score.

Figure 8: Table of summary results for the evaluation of Task 3.

Task 3:	Average
Accuracy	0.929
Precision	0.580
Recall	0.199
F1 Score	0.291
True Negatives	10864
True Positives	176
False Negatives	710
False Positives	130

Figure 9: Chart comparison of the summary results between tasks 1, 2 and 3. As seen, task 3 achieves a better accuracy than previous tasks, but at the expense of all other metrics. In fact, this illustrates the issue with using just accuracy as a metric; the majority negative class in the imbalanced data set is favoured by training, so a high accuracy is achieved by guessing the negative class most of the time. However, this results in poor precision and recall.



B.2.6 Task 4

The task 3 model (trained on the imbalanced data set) gave better precision compared to any task 4 model using class balancing techniques (figure 11). However, the task 3 model has significantly worse recall compared to the task 4 models; the task 3 model misses much more of the positive samples. This is expected as the task 3 model is insufficiently trained on the positive class due to frequency bias introduced by the majority negative class. As a result, the task 3 model has the worse F1 score compared to the task 4 models. Notably, none of the models have particularly high performance.

Of the task 4 models (figure 10), under-sampling is most precise, weighted loss has the best recall, and over-sampling gave the best F1 score (though marginally). Whilst either of the re-sampling techniques appear favourable to weighted loss (they have higher F1 scores), the results are not definitive. More robust investigations are required to determine which technique, if any, is preferable. More advanced techniques, such as SMOTE, should also be investigated to this end. Furthermore, a greater domain knowledge is required to determine the best technique to handle the imbalanced data; it is unknown which of incorrectly predicting water-points as requiring repairs and missing faulty water-points is more costly. In this case, the health risks associated with missing faulty water-points is likely more costly than the financial disincentive of sending engineers to fix perfectly functional water-points. Therefore, recall is favoured, and thus the weighted loss technique by these results.

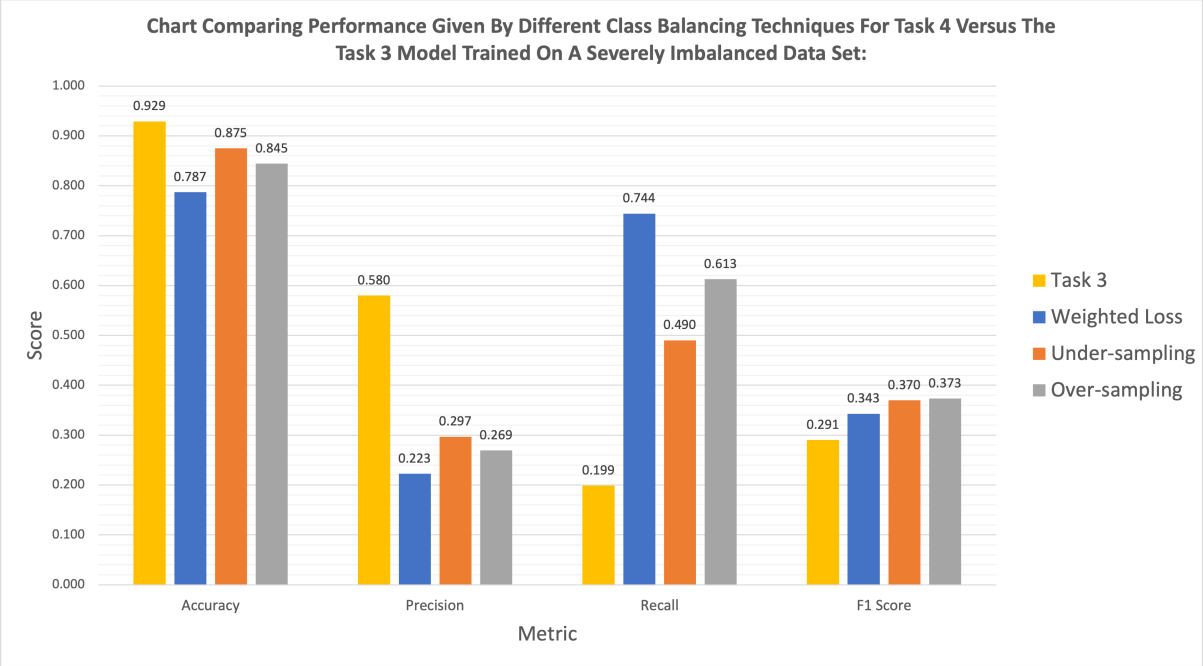
Figure 10: Table of summary results for the evaluation of Task 4. The average metric scores when using a weighted loss function (left), under-sampling (middle), and over-sampling (right) to handle class imbalance are shown.

Weighted Loss:	Average
Accuracy	0.787
Precision	0.223
Recall	0.744
F1 Score	0.343
True Negatives	8691
True Positives	659
False Negatives	227
False Positives	2303

Under-sampling:	Average
Accuracy	0.875
Precision	0.297
Recall	0.490
F1 Score	0.370
True Negatives	9962
True Positives	434
False Negatives	452
False Positives	1096

Over-sampling:	Average
Accuracy	0.845
Precision	0.269
Recall	0.613
F1 Score	0.373
True Negatives	9510
True Positives	543
False Negatives	343
False Positives	1484

Figure 11: Chart comparison of the summarised results attained by applying the three different class imbalance techniques investigated versus the task 3 results. In task 3, the NN was trained on the severely imbalanced data set without any imbalance handling techniques being applied. Applying any of the three techniques improves on the task 3 model. However, none of the three techniques are definitively shown to be the best from the results; more robust investigations are required.



B.2.7 Task 5

Unfortunately, cost-sensitivity with multi-class classification could not be implemented correctly with the `DL4J` library, so cost-sensitive multi-class classification has been omitted. Instead, average metric scores across the multiple output classes (with uniform cost) is evaluated (figures 12 and 13). This evaluation gives an indication of model performance for multi-class classification. Admittedly, a better investigation should consider the relative cost weightings associated with the different classes (e.g. the cost of misidentifying a faulty water-point as functional is higher than predicting a faulty water-point as non functional).

For multi-class, imbalanced classification, re-sampling appears to be the stronger class balancing technique; all metrics have a higher average for under- and over-sampling than weighted loss. However, this may be because the weighted loss technique introduces parameters (the weights) that require further optimisation to achieve comparable (or better) results.

Interestingly, the models in task 5 with ternary output yield better performance than the same models in task 4 with binary output. However, this comparison is not as well defined since task 5 model performance considers the average metric scores across all three equally weighted classes, whilst task 4 model performance considers solely the positive output class.

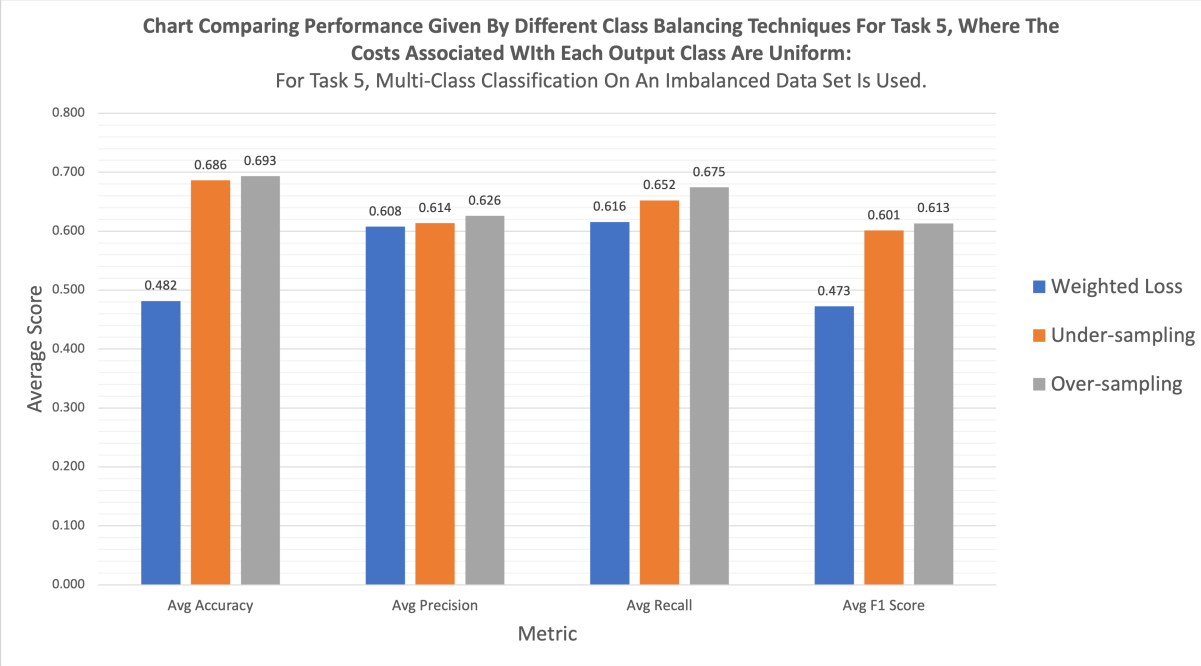
Figure 12: Tables of summary results for the evaluation of task 5. The cost of each class is considered equal (non class-sensitive) and the average of the average metric scores is presented. The tables indicate the performance of the trained task 5 models for multi-class classification.

Weighted Loss:	Average
Avg Accuracy	0.482
Avg Precision	0.608
Avg Recall	0.616
Avg F1 Score	0.473

Under-sampling:	Average
Avg Accuracy	0.686
Avg Precision	0.614
Avg Recall	0.652
Avg F1 Score	0.601

Over-sampling:	Average
Avg Accuracy	0.693
Avg Precision	0.626
Avg Recall	0.675
Avg F1 Score	0.613

Figure 13: The following screenshot shows a comparison of different class balancing techniques for task 5. In task 5, multi-class classification has been used (3 output classes) in conjunction with a severely imbalanced data set. In this comparison, the relative costs of each output class are considered equal. Thus, the chart acts as a baseline for performance that can be expected for multi-class classification before applying cost sensitivity.



B.2.8 Evaluation Criticisms

For all evaluations, early-stopping training occurs by training the NN using the training data, whilst maximising the accuracy score this produces for the unseen testing data. However, this produces the side-effect that maximal models found by the early-stopping implementation are likely to be somewhat tailored to the unseen testing sets. Whilst this minorly affects the comparative evaluations carried out in the evaluation section (every task is evaluated in the same way), the results reported can be made to better reflect real-world performance. Namely, the early-stopping training should split given training data sets into training and validation partitions. Then, early-stopping training can train using the training partition to maximise metrics on the validation partition. As a result, evaluations carried out on the testing set will be unpolluted and will better represent general model performance. This is a limitation of the evaluation that will be addressed in future work.

C Testing Summary

This section outlines core testing carried out on the implementations of all tasks to ensure correctness and conformity with the specification. As most of the work is carried out by external libraries, the testing mostly seeks to ensure that the API of the libraries has been used correctly.

C.1 Common

Table 2: Table outlining the high-level, core correctness testing carried out for the `Task` class, which contains functionality used by all individual tasks.

Test Description	Passed?	Proof
Test that specifying 'train' starts the correct process.	Yes	Figure 14
Test that specifying 'test' starts the correct process.	Yes	Figure 15
Test that specifying 'predict' starts the correct process.	Yes	Figure 16
Test an analysis is calculated and saved to a view-able HTML file when training.	Yes	Figure 17
Test that a training data set iterator is correctly created using the API.	Yes	Figure 18

Table 2: Table outlining the high-level, core correctness testing carried out for the `Task` class, which contains functionality used by all individual tasks.

Test Description	**Passed?**	**Proof**
Test that non-training data set iterators are correctly created using the API.	Yes	Figure 19
Test that a NN is correctly created when given a configuration.	Yes	Figure 20
Test that a trained model with corresponding analysis and schema is saved to and read from a file.	Yes	Figure 21
Test that a configured NN can be trained.	Yes	Figure 22
Test that a testing schema is successfully created for a model.	Yes	Figure 23
Test that a testing transform process is successfully created for a model.	Yes	Figure 23
Test that evaluation statistics for a tested model are correctly shown.	Yes	Figure 24
Test that predictions are correctly written to file and that the i-th line represents the prediction for the i-th input sample from the testing data.	Yes	Figure 25
Test that a predicting schema is successfully created for a model.	Yes	Figure 26
Test that a predicting transform process is successfully created for a model.	Yes	Figure 26
Test that water-point IDs for positively classified predictions are written to the screen.	Yes	Figure 27
Test that training using early stopping is carried out and that training stops early when no improvements are made upon a number of previous iterations.	Yes	Figure 28

Figure 14: The following screenshots show that when 'train' is given as a parameter (top), then the training functionality is correctly started (bottom).

```
task1 train input/task1_train.csv task1_NN -o
```

```
Started Training On: 'input/task1_train.csv', Saving NN To: 'task1_NN' File(s)...
```

Figure 15: The following screenshots show that when 'test' is given as a parameter (top), then the testing functionality is correctly started (bottom).

```
task1 test input/task1_test.csv task1_NN test.txt -o
```

```
Started Testing On: 'input/task1_test.csv', Using NN: 'task1_NN', Saving Predictions To: 'test.txt'...
```

Figure 16: The following screenshots show that when 'predict' is given as a parameter (top), then the predicting functionality is correctly started (bottom).

```
task1 predict input/task1_test_nolabels.csv task1_NN -o
```

```
Started Predicting On: 'input/task1_test_nolabels.csv', Using NN: 'task1_NN'...
```

Figure 17: The following screenshots illustrate that, for a training data set (e.g., `task1_train.csv`), an analysis of the data is calculated and saved to a HTML file. The top screenshot shows that an analysis is calculated and shown by the program. The bottom screenshot shows that this analysis is view-able as an HTML file for improved inspection of the distribution of the input features.

```
Calculated Analysis For Training Data In 'input/task1_train.csv'. Saved To: task1_NN-Analysis.html:
idx   name              type          analysis
0     "id"              Integer       IntegerAnalysis(min=15,max=74233,mean=37158.693499348396,sa
1     "amount_tsh"      Double        DoubleAnalysis(min=0.0,max=250000.0,mean=326.2930360503844,
2     "gps_height"      Integer       IntegerAnalysis(min=-54,max=2585,mean=649.1269726364543,sam
3     "longitude"       Double        DoubleAnalysis(min=0.0,max=40.32280453,mean=32.887934491095
4     "latitude"        Double        DoubleAnalysis(min=-11.64944018,max=-2.0E-8,mean=-5.4669010
5     "num_private"     Integer       IntegerAnalysis(min=0,max=160,mean=0.32039959461415896,samp
6     "population"      Integer       IntegerAnalysis(min=0,max=15300,mean=178.67656001158258,sam
7     "status_group"    Categorical   CategoricalAnalysis(CategoryCounts={others=3466, functional
```

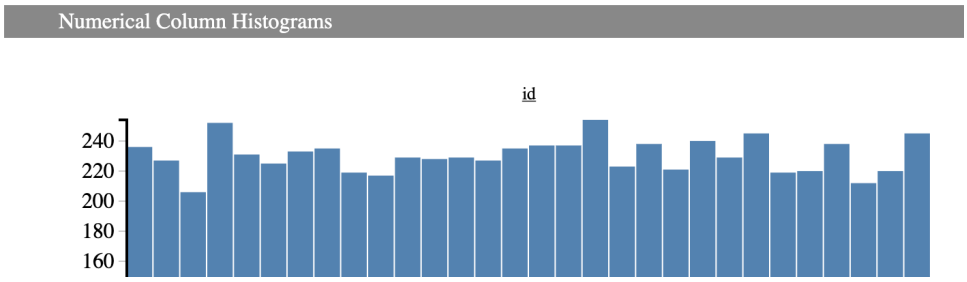


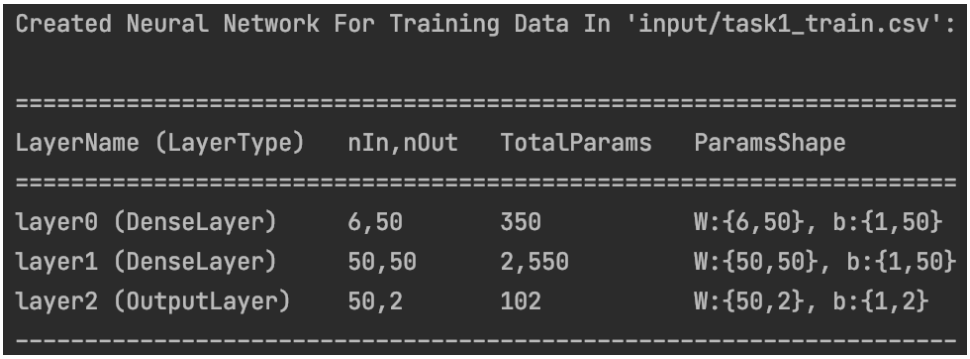
Figure 18: Test showing that a data set iterator for a training data set (e.g., `task1_train.csv`) is successfully created. The data set iterator uses a specified schema for the training set along with a transform process for pre-processing the data before feeding the input samples to the model for training.

```
Created Training Set Iterator (With Pre-processing) For 'input/task1_train.csv'. Batch Size: 80
o.n.l.f.Nd4jBackend - Loaded [CpuBackend] backend
o.n.n.NativeOpsHolder - Number of threads used for linear algebra: 1
o.n.l.c.n.CpuNDArrayFactory - ********************************** CPU Feature Check Warning *****
```

Figure 19: Test showing that a data set iterator for a non-training data set (e.g., `task1_test.csv`) is successfully created. The top screenshot shows that an iterator for the example data set is successfully created and indicated by the program. The bottom screenshot shows a portion of the first batch retrieved from this data set iterator, which is seen to have successfully pre-processed the input features (e.g., features are scaled as specified).

```
Created Data Set Iterator For 'input/task1_test.csv'. Batch Size: 80
[[         0,    0.0534,    0.1358,    0.6046,         0,         0],
 [         0,    0.7164,    0.4799,    0.6119,         0,    0.7960],
 [         0,    0.7516,    0.2338,    0.4636,         0,    0.7109],
 [    0.4703,    0.7838,    0.5157,    0.6200,         0,    0.1290],
 [         0,    0.0534,   -0.1639,    0.6962,         0,         0],
 [         0,    0.0534,    0.4696,    0.4493,         0,         0],
 [         0,    0.0534,    0.1185,    0.2856,         0,         0]
```

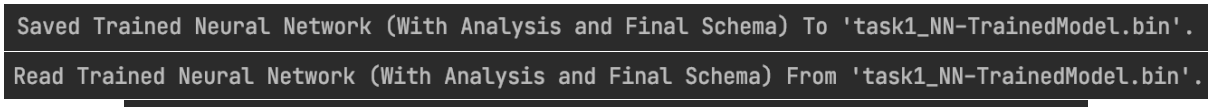
Figure 20: The following screenshot shows that a NN is successfully created for a task when given a specified configuration. In this example for task 1, the NN configuration is set to have 1 input layer (of 6 input neurons), 2 hidden layers (of 50 neurons), and 1 output layer (of 2 output neurons). This is shown in the following NN summary output by the program.



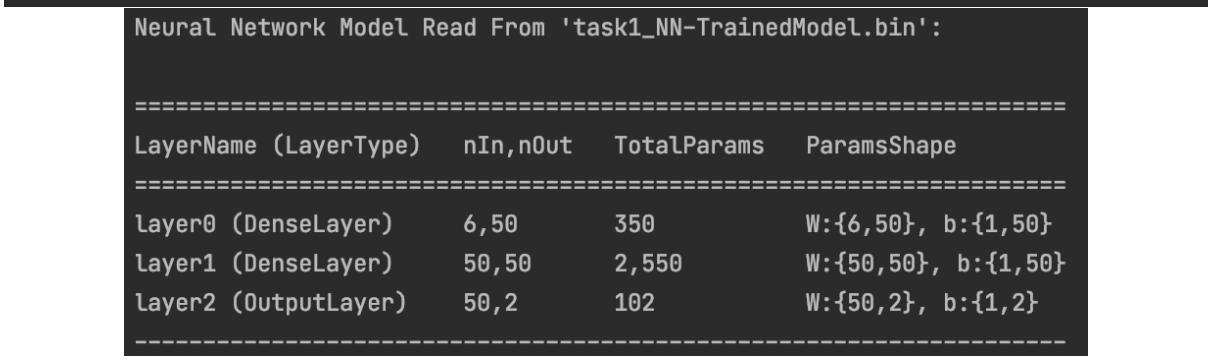
```
Created Neural Network For Training Data In 'input/task1_train.csv':

=================================================================
LayerName (LayerType)   nIn,nOut   TotalParams   ParamsShape
=================================================================
layer0 (DenseLayer)     6,50       350           W:{6,50}, b:{1,50}
layer1 (DenseLayer)     50,50      2,550         W:{50,50}, b:{1,50}
layer2 (OutputLayer)    50,2       102           W:{50,2}, b:{1,2}
-----------------------------------------------------------------
```

Figure 21: The following screenshots show that a model can be successfully saved to and read from a file. The top screenshot shows the output of the program when saving the NN given in figure 20 (training for task 1). The middle screenshot shows the output of the program when testing for task 1, which requires the trained task 1 model. It is indicated that the model has been successfully read from the file. The bottom screenshot gives the summary for the read NN, which is shown to be identical to figure 20, as expected.



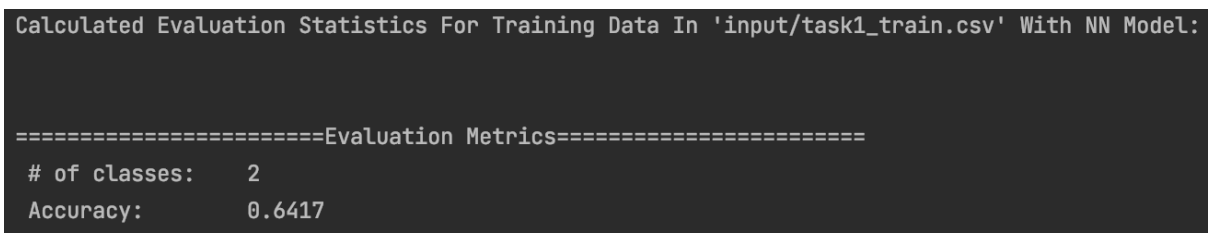
```
Saved Trained Neural Network (With Analysis and Final Schema) To 'task1_NN-TrainedModel.bin'.
```



```
Read Trained Neural Network (With Analysis and Final Schema) From 'task1_NN-TrainedModel.bin'.
Neural Network Model Read From 'task1_NN-TrainedModel.bin':

=================================================================
LayerName (LayerType)   nIn,nOut   TotalParams   ParamsShape
=================================================================
layer0 (DenseLayer)     6,50       350           W:{6,50}, b:{1,50}
layer1 (DenseLayer)     50,50      2,550         W:{50,50}, b:{1,50}
layer2 (OutputLayer)    50,2       102           W:{50,2}, b:{1,2}
-----------------------------------------------------------------
```

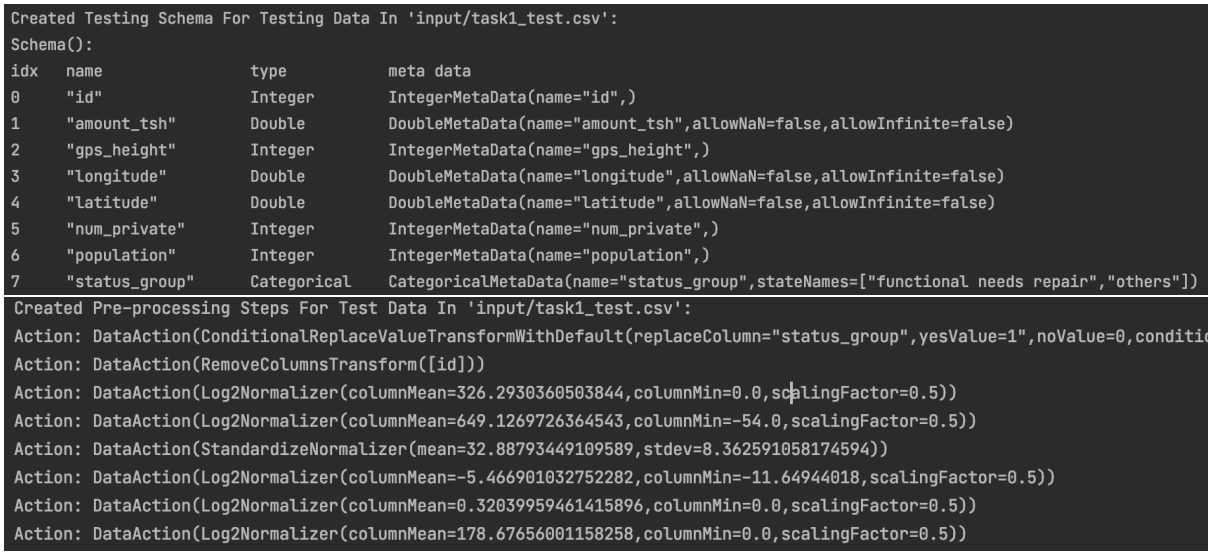
Figure 22: The following screenshot shows that a configured NN can be successfully trained. In the example given, the training data set from task 1 has been trained and an accuracy is given to reflect how successful the training was.



```
Calculated Evaluation Statistics For Training Data In 'input/task1_train.csv' With NN Model:


========================Evaluation Metrics========================
 # of classes:    2
 Accuracy:        0.6417
```

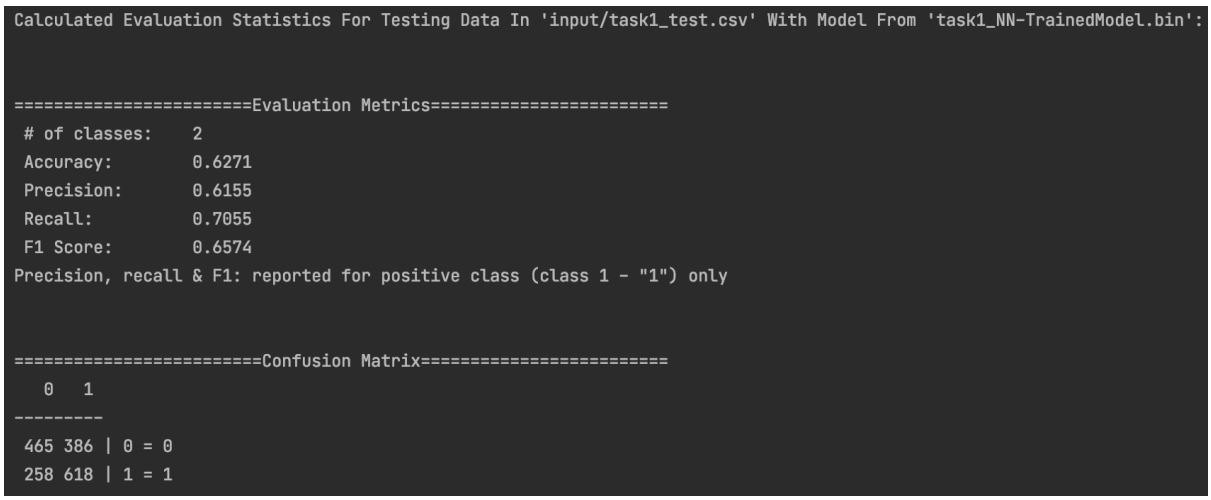
Figure 23: The following screenshot shows that a testing schema (top) and transform process (bottom) are successfully set up for a test data set. In this example, the `task1_test.csv` data set is used, which has an identical schema and transform process to the corresponding task 1 training set, as expected.



```
Created Testing Schema For Testing Data In 'input/task1_test.csv':
Schema():
idx   name               type          meta data
0     "id"               Integer       IntegerMetaData(name="id",)
1     "amount_tsh"       Double        DoubleMetaData(name="amount_tsh",allowNaN=false,allowInfinite=false)
2     "gps_height"       Integer       IntegerMetaData(name="gps_height",)
3     "longitude"        Double        DoubleMetaData(name="longitude",allowNaN=false,allowInfinite=false)
4     "latitude"         Double        DoubleMetaData(name="latitude",allowNaN=false,allowInfinite=false)
5     "num_private"      Integer       IntegerMetaData(name="num_private",)
6     "population"       Integer       IntegerMetaData(name="population",)
7     "status_group"     Categorical   CategoricalMetaData(name="status_group",stateNames=["functional needs repair","others"])
```

```
Created Pre-processing Steps For Test Data In 'input/task1_test.csv':
Action: DataAction(ConditionalReplaceValueTransformWithDefault(replaceColumn="status_group",yesValue=1",noValue=0,conditi
Action: DataAction(RemoveColumnsTransform([id]))
Action: DataAction(Log2Normalizer(columnMean=326.2930360503844,columnMin=0.0,scalingFactor=0.5))
Action: DataAction(Log2Normalizer(columnMean=649.1269726364543,columnMin=-54.0,scalingFactor=0.5))
Action: DataAction(StandardizeNormalizer(mean=32.88793449109589,stdev=8.362591058174594))
Action: DataAction(Log2Normalizer(columnMean=-5.466901032752282,columnMin=-11.64944018,scalingFactor=0.5))
Action: DataAction(Log2Normalizer(columnMean=0.32039959461415896,columnMin=0.0,scalingFactor=0.5))
Action: DataAction(Log2Normalizer(columnMean=178.67656001158258,columnMin=0.0,scalingFactor=0.5))
```

Figure 24: The following screenshot shows that the evaluation metrics for a test data set are correctly calculated and shown by the program, as requested by the specification. The confusion matrix for the model is also shown.

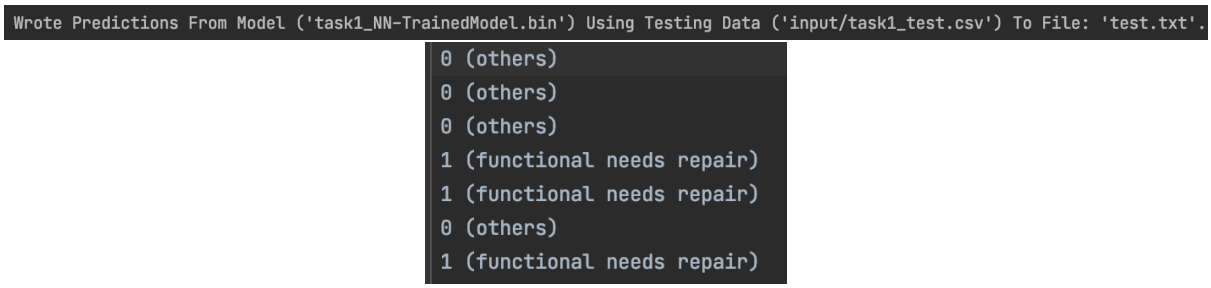


```
Calculated Evaluation Statistics For Testing Data In 'input/task1_test.csv' With Model From 'task1_NN-TrainedModel.bin':


=======================Evaluation Metrics=======================
 # of classes:    2
 Accuracy:        0.6271
 Precision:       0.6155
 Recall:          0.7055
 F1 Score:        0.6574
Precision, recall & F1: reported for positive class (class 1 - "1") only


========================Confusion Matrix========================
   0   1
---------
 465 386 | 0 = 0
 258 618 | 1 = 1
```

Figure 25: The following screenshots show that predictions given for a test data set (e.g., `task1_test.csv`) are successfully wrote to a specified file. The top screenshot shows the program outputting that predictions have been successfully wrote to the user-specified file. The bottom screenshot shows the contents of this file, which gives the predicted integer classes (and corresponding categorical labels) for each sample. Each prediction is given on an individual line with the i^{th} line representing the prediction for the i^{th} input sample from the test data set, as requested by the specification.



```
Wrote Predictions From Model ('task1_NN-TrainedModel.bin') Using Testing Data ('input/task1_test.csv') To File: 'test.txt'.
```

```
0 (others)
0 (others)
0 (others)
1 (functional needs repair)
1 (functional needs repair)
0 (others)
1 (functional needs repair)
```

Figure 26: The following screenshot shows that a prediction schema (top) and transform process (bottom) are successfully set up for a prediction data set. In this example, the `task1_test_nolabels.csv` data set is used. This example has an identical schema and transform process to those given in figure 23, except that the output column has been removed from the schema (and therefore is not transformed). This is correctly shown in the following screenshots.

```
Created Prediction Schema For Prediction Data In 'input/task1_test_nolabels.csv':
Schema():
idx   name              type          meta data
0     "id"              Integer       IntegerMetaData(name="id",)
1     "amount_tsh"      Double        DoubleMetaData(name="amount_tsh",allowNaN=false,allowInfinite=false)
2     "gps_height"      Integer       IntegerMetaData(name="gps_height",)
3     "longitude"       Double        DoubleMetaData(name="longitude",allowNaN=false,allowInfinite=false)
4     "latitude"        Double        DoubleMetaData(name="latitude",allowNaN=false,allowInfinite=false)
5     "num_private"     Integer       IntegerMetaData(name="num_private",)
6     "population"      Integer       IntegerMetaData(name="population",)
```

```
Created Pre-processing Steps For Prediction Data In 'input/task1_test_nolabels.csv':
Action: DataAction(RemoveColumnsTransform([id]))
Action: DataAction(Log2Normalizer(columnMean=326.2930360503844,columnMin=0.0,scalingFactor=0.5))
Action: DataAction(Log2Normalizer(columnMean=649.1269726364543,columnMin=-54.0,scalingFactor=0.5))
Action: DataAction(StandardizeNormalizer(mean=32.88793449109589,stdev=8.362591058174594))
Action: DataAction(Log2Normalizer(columnMean=-5.466901032752282,columnMin=-11.64944018,scalingFactor=0.5))
Action: DataAction(Log2Normalizer(columnMean=0.32039959461415896,columnMin=0.0,scalingFactor=0.5))
Action: DataAction(Log2Normalizer(columnMean=178.67656001158258,columnMin=0.0,scalingFactor=0.5))
```

Figure 27: The following screenshot shows that the water-point IDs for water-points positively classified (i.e., '*functional needs repair*') are output to the screen. The correctness of these IDs as being those positively classified by the model was ensured by checking that every i^{th} line positively classified in the 'test' output file corresponded to an ID shown in the following screenshot (when 'predict' was executed). This is valid since the `task1_test.csv` and `task1_test_nolabels.csv` data sets are equivalent, except that output labels have been removed in the latter data set.

```
Water-Point IDs Identified As 'functional needs repair':
10708
60497
44991
13396
47145
35454
39416
```

Figure 28: The following screenshot shows that early stopping can be used to dynamically change the number of epochs required to train a model. Here, the task 1 training set has been used to train a model to maximise accuracy on the unseen testing data set. The program can run up to 1000 epochs, but stops as desired at epoch 377 as the accuracy has not improved in 100 epochs.

```
Early Stopping Training On 'input/task1_train.csv' Completed:
Termination Reason: EpochTerminationCondition
Termination Details: ScoreImprovementEpochTerminationCondition(maxEpochsWithNoImprovement=100, minImprovement=0.0)
Total Epochs: 377
Best Epoch Number: 276
Score At Best Epoch: 0.6531557614360162
```

C.2 Task 1

Table 3: Table outlining the high-level, core correctness testing carried out for task 1.

Test Description	Passed?	Proof
Test that the specified initial schema for the training data is successfully created.	Yes	Figure 29
Test that a transform process to pre-process the Task 1 training data is successfully created.	Yes	Figure 30
Test that the pre-processing steps specified are correctly carried out on the training data set.	Yes	Figure 31
Test that the NN configuration produces the expected NN.	Yes	Figure 20

Figure 29: The following screenshot shows that a schema for the task 1 training data (`task1_train.csv`) has been successfully created as specified. Here, the '*id*', '*gps_height*', '*num_private*', and '*population*' fields are taken as the integer data type. Also, the '*amount_tsh*', '*longitude*', and '*latitude*' fields are taken as doubles. The '*status_group*' field is the only categorical column (output class) and only accepts "functional needs repair" and "others" as values. This is correct and as expected.

```
Created Initial Schema For Training Data In 'input/task1_train.csv':
Schema():
idx   name               type          meta data
0     "id"               Integer       IntegerMetaData(name="id",)
1     "amount_tsh"       Double        DoubleMetaData(name="amount_tsh",allowNaN=false,allowInfinite=false)
2     "gps_height"       Integer       IntegerMetaData(name="gps_height",)
3     "longitude"        Double        DoubleMetaData(name="longitude",allowNaN=false,allowInfinite=false)
4     "latitude"         Double        DoubleMetaData(name="latitude",allowNaN=false,allowInfinite=false)
5     "num_private"      Integer       IntegerMetaData(name="num_private",)
6     "population"       Integer       IntegerMetaData(name="population",)
7     "status_group"     Categorical   CategoricalMetaData(name="status_group",stateNames=["functional needs repair","others"])
```

Figure 30: The following screenshot shows that a transform process to carry out pre-processing on the training data set (`task1_train.csv`) has been successfully created as specified. The '*status_group*' column will have its values converted: "functional needs repair" as 1, and "others" as 0. The '*id*' column is specified to be dropped (ID is unlikely to correlate to whether a water-point requires repairs). All other columns in this data set are numerical and are pre-processed via scaling according to the observed distribution of data for each feature.

```
Created Pre-processing Steps For Training Data In 'input/task1_train.csv':
Action: DataAction(ConditionalReplaceValueTransformWithDefault(replaceColumn="status_group",yesValue=1",noValue=0,
Action: DataAction(RemoveColumnsTransform([id]))
Action: DataAction(Log2Normalizer(columnMean=326.2930360503844,columnMin=0.0,scalingFactor=0.5))
Action: DataAction(Log2Normalizer(columnMean=649.1269726364543,columnMin=-54.0,scalingFactor=0.5))
Action: DataAction(StandardizeNormalizer(mean=32.88793449109589,stdev=8.362591058174594))
Action: DataAction(Log2Normalizer(columnMean=-5.466901032752282,columnMin=-11.64944018,scalingFactor=0.5))
Action: DataAction(Log2Normalizer(columnMean=0.32039959461415896,columnMin=0.0,scalingFactor=0.5))
Action: DataAction(Log2Normalizer(columnMean=178.67656001158258,columnMin=0.0,scalingFactor=0.5))
```

Figure 31: The following screenshot illustrates that the transform process correctly pre-processes the training set data (`task1_train.csv`). The '*id*' column has been dropped, as specified, and the '*status_group*' column is not an input feature, so it is not included. Thus, the 6 columns in the screenshot are the 6 remaining numerical input features, which have been scaled. Note that the screenshot merely shows that the transform process correctly executes the specified transforms to pre-process the data; the steps to do so are subject to change.

```
[[    0.4103,    0.8116,   -0.3530,    0.5585,         0,    0.8093],
 [         0,    0.3814,    0.2056,    0.1181,         0,    0.4613],
 [         0,    0.0534,   -3.9327,    0.7641,         0,         0],
 [         0,    0.6570,   -0.2981,    0.5207,         0,    1.0140],
  [         0,    0.0534,    0.0138,    0.2520,         0,         0],
  [         0,    0.9914,    0.3987,    0.6347,         0,    0.4397],
  [         0,    0.9023,    0.3224,    0.5909,         0,    0.7258],
  [         0,    0.0534,    0.0340,    0.5463,         0,         0],
  [    0.0218,    0.8584,    0.3524,    0.6178,         0,    0.5418],
  [         0,    0.6765,    0.6114,    0.5597,         0,    0.5968]
```

C.3 Task 2

Table 4: Table outlining the high-level, core correctness testing carried out for task 2.

Test Description	Passed?	Proof
Test that the specified initial schema for the training data is successfully created.	Yes	Figure 32
Test that a transform process to pre-process the Task 2 training data is successfully created.	Yes	Figure 33
Test that the pre-processing steps specified are correctly carried out on the training data set.	Yes	Figure 34
Test that the NN configuration produces the expected NN.	Yes	Figure 35

Figure 32: The following screenshot shows that a schema for the task 2 training data (`task2_train.csv`) has been successfully created as specified. The schema is identical to Task 1, except that additional categorical columns have been included. For each categorical column, a set of the most frequent category values are permitted, as well as an '*Other*' category value for all other possibilities.

```
Started Training On: 'input/task2_train.csv', Saving NN To: 'task2_NN' File(s)...
Created Initial Schema For Training Data In 'input/task2_train.csv':
Schema():
idx   name                    type          meta data
0     "id"                    Integer       IntegerMetaData(name="id",)
1     "amount_tsh"            Double        DoubleMetaData(name="amount_tsh",allowNaN=false,allowInfinite=fals
2     "date_recorded"         Categorical   CategoricalMetaData(name="date_recorded",stateNames=[])
3     "funder"                Categorical   CategoricalMetaData(name="funder",stateNames=["Government Of Tanz
4     "gps_height"            Integer       IntegerMetaData(name="gps_height",)
5     "installer"             Categorical   CategoricalMetaData(name="installer",stateNames=["RWE","DWE","Gov
6     "longitude"             Double        DoubleMetaData(name="longitude",allowNaN=false,allowInfinite=fals
```

Figure 33: The following screenshot shows that a transform process to carry out pre-processing on the training data set (`task2_train.csv`) has been successfully created as specified. This transform process is identical to Task 1, except that additional pre-processing steps has been added for all categorical columns. Categorical columns are converted so that they contain only frequent category values. Then, each categorical column is one-hot encoded.

```
Created Pre-processing Steps For Training Data In 'input/task2_train.csv':
Action: DataAction(ConditionalReplaceValueTransformWithDefault(replaceColumn="status_group",yesValue=1",noValue=0,condition=CategoricalColumnCondition(columnName="status_gr
Action: DataAction(RemoveColumnsTransform([id]))
Action: DataAction(Log2Normalizer(columnMean=326.2930360503844,columnMin=0.0,scalingFactor=0.5))
Action: DataAction(RemoveColumnsTransform([date_recorded]))
Action: DataAction(RemoveColumnsTransform([recorded_by]))
Action: DataAction(ConditionalReplaceValueTransform(replaceColumn="funder",newValue=Other,condition=CategoricalColumnCondition(columnName="funder",NotInSet,[Government Of T
Action: DataAction(CategoricalToOneHotTransform(columnName="funder"))
Action: DataAction(Log2Normalizer(columnMean=649.1269726364543,columnMin=-54.0,scalingFactor=0.5))
Action: DataAction(ConditionalReplaceValueTransform(replaceColumn="installer",newValue=Other,condition=CategoricalColumnCondition(columnName="installer",NotInSet,[RWE, DWE,
Action: DataAction(CategoricalToOneHotTransform(columnName="installer"))
```

Figure 34: The following screenshot illustrates that the transform process correctly pre-processes the training set data (`task2_train.csv`). The transform process produces records with 274 columns after pre-processing, which is as expected. There are columns for each of the 6 numerical features, plus a total of 268 one-hot encoded columns from the categorical features. It was also verified that the one-hot encoding transform correctly produces values in the one-hot encoded columns of a categorical feature, even though we can assume the library used (`DL4J`) to be correct.

```
Training Set Iterator For 'input/task2_train.csv' Produces Pre-processed Records With 274 Columns.
Testing Set Iterator For 'input/task2_test.csv' Produces Pre-processed Records With 274 Columns.
```

Figure 35: The following screenshot shows that a NN is successfully created for task 2. The number of hidden layers, hidden neurons within a layer, and the output layer are unchanged from task 1 for evaluation purposes. There are significantly more input features with task 2 (due to one-hot encoding), so there are more input neurons in the input layer, which is as expected. The number of input neurons is equivalent to the number of numerical columns in the transform, plus all of the one-hot encoded columns derived from the categorical features.

```
Created Neural Network For Training Data In 'input/task2_train.csv':

=================================================================
LayerName (LayerType)   nIn,nOut   TotalParams   ParamsShape
=================================================================
layer0 (DenseLayer)     274,20     5,500         W:{274,20}, b:{1,20}
layer1 (DenseLayer)     20,20      420           W:{20,20}, b:{1,20}
layer2 (OutputLayer)    20,2       42            W:{20,2}, b:{1,2}
-----------------------------------------------------------------
```

C.4 Task 3

Task 3 uses the implementation from task 2 but with different input files, which are unbalanced. The only testing required was to ensure that the Task 2 implementation could be executed with the different data sets from task 3, which was trivial (figure 36).

Figure 36: The following screenshots illustrate the working order of the task 2 implementation with task 3 data sets. For example, the top screenshot shows that task 3 is being executed with the correct corresponding arguments. As seen in the bottom screenshot, the task 2 implementation accepts these arguments and starts the training process by successfully establishing a schema for the data set.

```
task3 train input/task3_train.csv task3_NN -o
```

```
Started Training On: 'input/task3_train.csv', Saving NN To: 'task3_NN' File(s)...
Created Initial Schema For Training Data In 'input/task3_train.csv':
Schema():
idx   name                    type          meta data
0     "id"                    Integer       IntegerMetaData(name="id",)
1     "amount_tsh"            Double        DoubleMetaData(name="amount_tsh",a
2     "date_recorded"         Categorical   CategoricalMetaData(name="date_rec
```

C.5 Task 4

Table 5: Table outlining the high-level, core correctness testing carried out for task 4.

Test Description	Passed?	Proof
Test weights are being assigned to the cross-entropy loss function when using a weighted loss function for handling class imbalance.	Yes	Figure 37
Test under-sampling correctly drops negative class samples at random to balance a small test CSV file.	Yes	Figure 38
Test under-sampling correctly balances the task 3 training data set CSV file.	Yes	Figure 39
Test over-sampling correctly duplicates positive class samples at random to balance a small test CSV file.	Yes	Figure 40
Test over-sampling correctly balances the task 3 training data set CSV file.	Yes	Figure 41

Figure 37: Test showing that weights for the weighted loss function are correctly assigned. The first index is the weight for the '0' class, which is the negative, majority class and so is given a minuscule weight. In contrast, the positive, minority class ("functional needs repair", which is mapped to '1') is given a high weight.

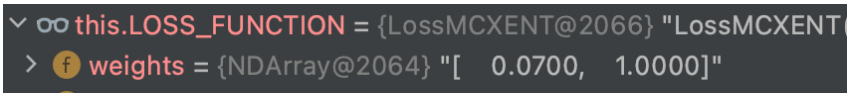


Figure 38: The following screenshots demonstrate that the implementation for under-sampling a given input file to balance the classes is correct. The bottom-left screenshot shows a small example CSV file, which is imbalanced. The top screenshot illustrates the arguments given to the task 4 implementation, to show that under-sampling is specified for use. The bottom-right screenshot shows the resulting file of the under-sampling. As expected, the majority negative class has had samples removed at random to balance the output classes.

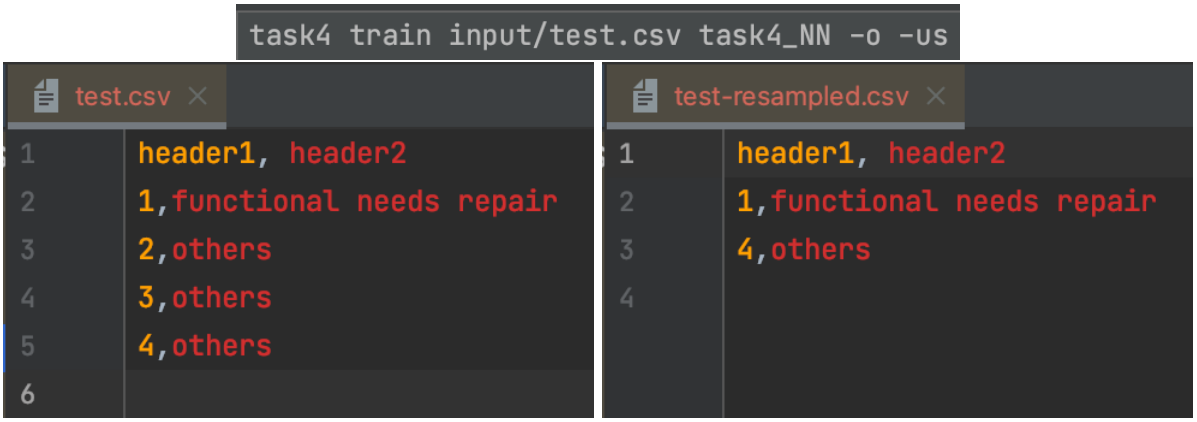


Figure 39: The following screenshot shows the counts of samples for the re-sampled CSV file created by under-sampling the `task3_train.csv` input data set. As expected, the positive and negative classes have been balanced by removing from the majority negative class repeatedly at random.

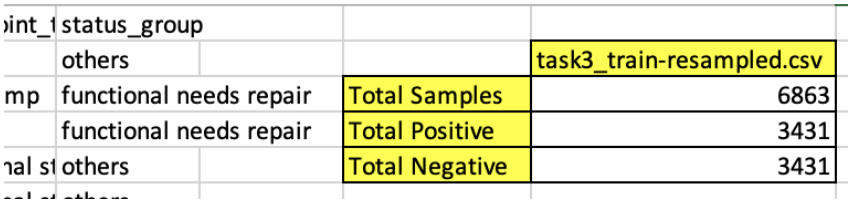


Figure 40: The following screenshots demonstrate that the implementation for over-sampling a given input file to balance the classes is correct. The bottom-left screenshot shows a small example CSV file, which is imbalanced. The top screenshot illustrates the arguments given to the task 4 implementation, to show that over-sampling is specified for use. The bottom-right screenshot shows the resulting file of the over-sampling. As expected, the minority positive class has had samples duplicated at random to balance the output classes.

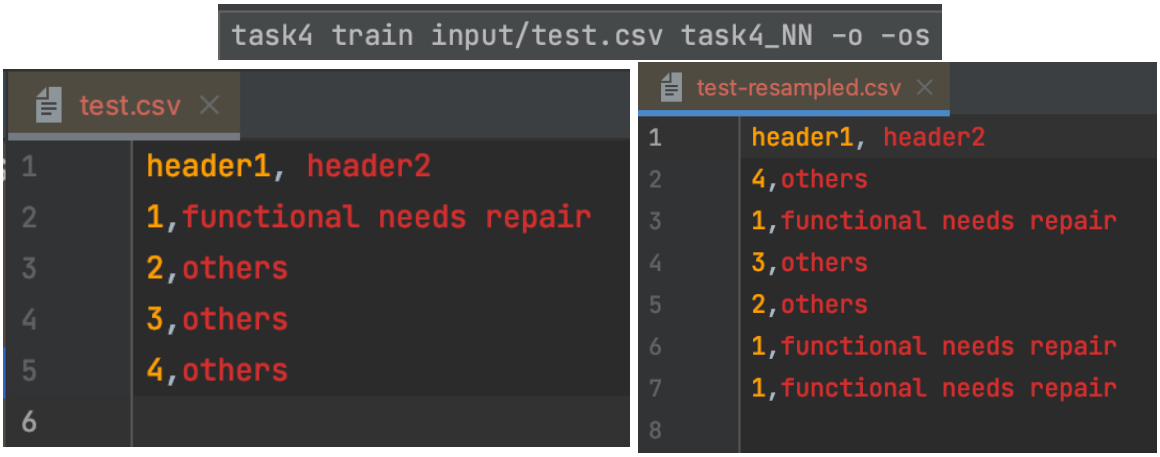
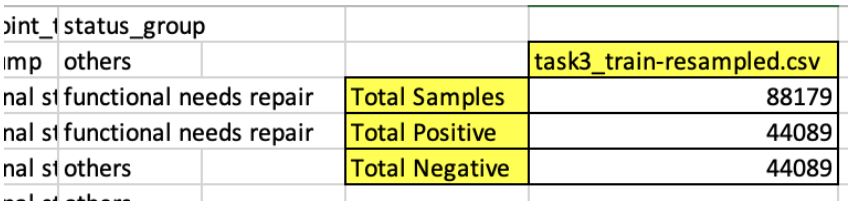


Figure 41: The following screenshot shows the counts of samples for the re-sampled CSV file created by over-sampling the `task3_train.csv` input data set. As expected, the positive and negative classes have been balanced by duplicating the minority positive class repeatedly at random.



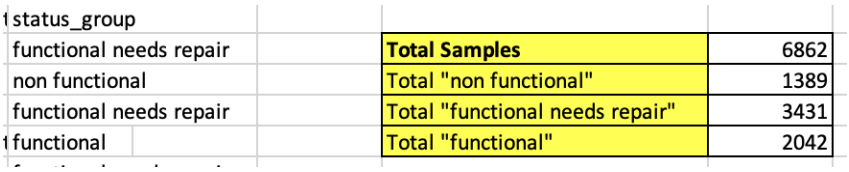
C.6 Task 5

Table 6: Table outlining the high-level, core correctness testing carried out for task 5.

Test Description	Passed?	Proof
Test weights are being assigned to the cross-entropy loss function when using multi-class classification on an imbalanced data set.	Yes	Figure 42
Test under-sampling correctly balances the task 5 training data set CSV file.	Yes	Figure 43
Test over-sampling correctly balances the task 5 training data set CSV file.	Yes	Figure 44
Test that the 'test' functionality correctly makes predictions which are of the three output classes.	Yes	Figure 45

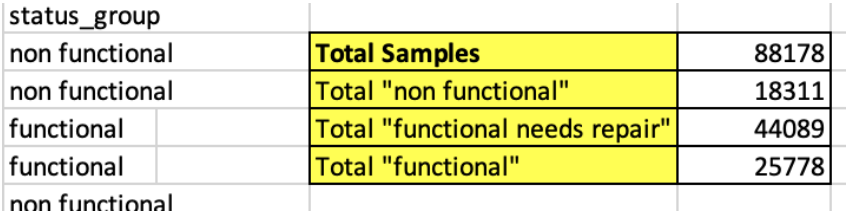
Figure 42: Test showing that weights for the weighted loss function are correctly assigned. The first index is the weight for the '0' class, which is a majority class ("non functional") and is therefore given a minuscule weight. The positive, minority class ("functional needs repair", which is mapped to '1') is given a high weight. The final '2' class is a majority class ("functional") and is also given a minuscule weight. The output class weights are assigned inversely proportional to their share of the distribution of data within the training set.

Figure 43: The following screenshot shows the counts of samples for the re-sampled CSV file created by under-sampling the `task5_train.csv` input data set. As expected, the positive and negative classes have been balanced by removing the majority, positive classes repeatedly at random. Class imbalance has been reduced, especially for the priority output class ("functional needs repair").



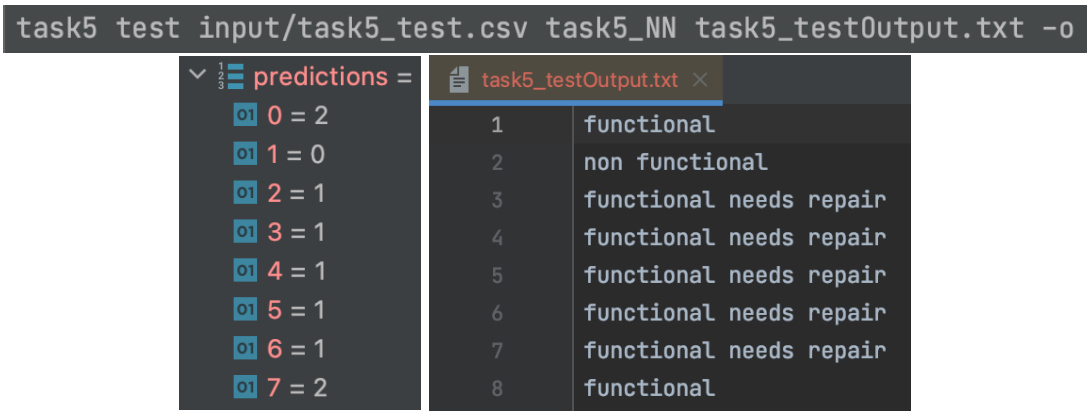
status_group		
functional needs repair	**Total Samples**	6862
non functional	Total "non functional"	1389
functional needs repair	Total "functional needs repair"	3431
functional	Total "functional"	2042

Figure 44: The following screenshot shows the counts of samples for the re-sampled CSV file created by over-sampling the `task5_train.csv` input data set. As expected, the positive and negative classes have been balanced by duplicating the minority, positive class repeatedly at random. Class imbalance has been reduced, especially for the priority output class ("functional needs repair").



status_group		
non functional	**Total Samples**	88178
non functional	Total "non functional"	18311
functional	Total "functional needs repair"	44089
functional	Total "functional"	25778
non functional		

Figure 45: The following screenshots serve to show the working 'test' functionality for task 5. The top screenshot shows the arguments specified to invoke the 'test' functionality, given that a NN has been trained for the task 5 training data. The bottom-left screenshot demonstrates the program predictions for the task 5 test data. The bottom-right screenshot shows these predictions are being correctly mapped to corresponding output class labels and are written to the file specified. In the program, "non functional" maps to 0, "functional needs repair" maps to 1, and "functional" maps to 2.



D Bibliography

The required content for this assessment was acquired from the specification and module material.