CS5014 Machine Learning
Practical 2 Report

University of St Andrews - School of Computer Science

Student ID: 150014151

5th May, 2020

Contents

1 Introduction

This practical covers the exploration of various machine learning practices used when dealing with real-life data. The data in question documents different types of seal pups found in cropped aerial imagery obtained during seasonal surveys of islands. Various data visualisation techniques, data processing steps and classification models are experimented with to design a final pipeline for making predictions on the types of seals observed in the images. The predictions are made by training multiple classification models, which are all evaluated to determine their performance.

The report is separated in four sections, starting with a general introductory section before exploring the diverse array of methodologies and design decisions made during the development of the code in Python, followed by an evaluation of the final classifier and a critical reflection on the findings.

2 System Architecture

2.1 Tools used

- Programming language: Python 3.7.
- Python libraries used: SciKit-Learn [1], Pandas [2], Matplotlib [3], NumPy [4], Seaborn [5] and Imbalanced-Learn [6].
- Editor: PyCharm[1].
- Version controlling: Git and GitHub (private repository).

2.2 Project structure

The project is organised into the following python modules:

- '*main.py*': contains functions to parse command line arguments and control the different sections of the code.
- '*data_visualisation.py*': contains functions to visualise the data's features and distribution.
- '*data_manipulations.py*': contains functions to process the features and labels of the data before feeding it to the classifiers.
- '*classifiers.py*': contains a *Classifier* class to create a Scikit-Learn classifier, fit it, tune it and evaluate it.
- '*helpers.py*': contains general functions to perform general operations on the data or print statements on the command line.
- '*config.py*': contains project-wide variables that are set by the command line arguments.

[1]PyCharm: `https://www.jetbrains.com/pycharm/`

2.3 Execution flow

A command-line interface, implemented through Python's native library *argparse*[2], controls which dataset to use and which part of the program to run:

- Visualising the data sets.
- Training different classification models.
- Running a hyperparameter tuning algorithm (grid search or randomised search) on promising models.
- Making predictions using the final model.

Read the *"README.txt"* file provided in the submission for instructions on how to install and run the code.

3 Methodology & data-driven design decisions

3.1 Datasets

3.1.1 Data description

The data comes in two flavours: *binary* and *multi*, where the "*binary*" data contains two labels, one for images of backgrounds and the other images of seals, and the "*multi*" data contains five labels, one for images of backgrounds and the four others for types of seals (whitecoat, moulted pup, dead pup and juvenile).

3.1.2 Data loading

Each dataset contains three distinct CSV files:

- "*X_train.csv*", containing the features describing the images, which are analysed in further sections.
- "*Y_train.csv*", containing the class/label for each image.
- "*X_test.csv*", containing the features of the images to test. These are used for making the final predictions with the optimal trained classifier.

The provided CSV files are directly loaded into distinct Pandas DataFrames[3] by using the *read_csv* function. However, this method takes a long time due to the large file size (e.g. X_train.csv is 1.5GB large). To speed up the process, the DataFrames are therefore serialised and saved using Pickle[4], boosting future loading times from 11.3 to 0.6 seconds.

[2]argparse: `https://docs.python.org/3.7/library/argparse.html`
[3]Pandas DataFrames: `https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html`
[4]Pickle: `https://docs.python.org/3.7/library/pickle.html`

3.2 Data visualisation & analysis

This data visualisation step[5] is crucial to understand the data and its underlying patterns before fitting a classification model to the training data.

3.2.1 Data overview

Storing the features and labels in a *DataFrames* provides a valuable array of functions that can be used to gain high-level insights on the data. Using the *pandas.DataFrame.info* and *pandas.DataFrame.describe* functions reveals important information such as:

- There are 62210 samples in the training set, each described by 964 features (columns).
- There are no missing values in the entire dataset as each feature has 62210 non-null values (number of images in the training set).
- All training values are numerical (floats).

This initial overview confirms that no data encoding is required to use the features and that the data is considered to "clean", thus requiring no processing.

3.2.2 Classes distribution

Because this is a classification task, it is important to visualise the distribution of classes in the training set to determine whether the data is skewed or not (whether some classes are much more frequent than other classes [7]). This is achieved by plotting the labels found in "*Y_train.csv*" in bar charts (see Figure 1).

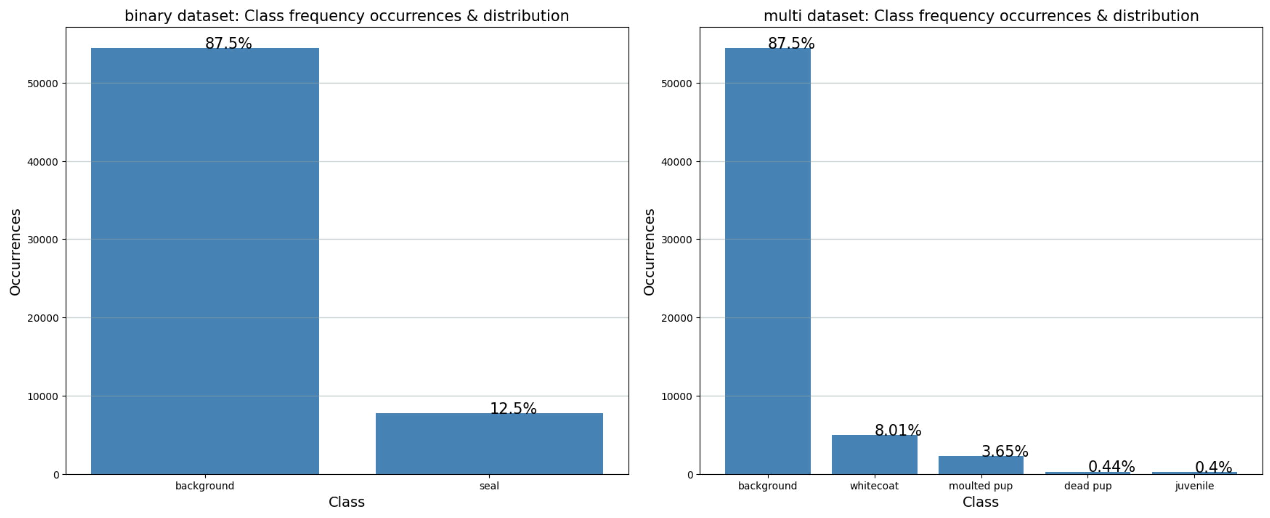


Figure 1: Class distribution of the binary (left) and multi (right) datasets.

[5]Note: running the code to visualise all the figures and tables produced takes roughly 80 seconds.

These bar charts reveal that both datasets are heavily unbalanced, which must be taken into account when analysing the classifiers' scores. Indeed, using an evaluation metric such as accuracy would be misleading as it would not be representative of how well the classifier fitted the data. For instance, if a dumb classifier that always classified an image as "background" was created, it would achieve 87.5% accuracy on the binary dataset. Therefore, other metrics such as confusion matrices, precision, recall and F1 scores could be used [7]. These are further explored in Section 3.4.2. A potential solution to counter the imbalance would be to oversample the dataset [8].

3.2.3 Features & correlation

There are three different types of features describing each image (row) in the datasets:

- *Histogram of oriented Gradients (HoG)*, which correspond to a type of feature often used in computer vision. These extracting HoGs are histograms counting the occurrences of gradient orientations (9 orientations) in 2x2 blocks [9]. In the "*X_train.csv*" datasets, they correspond to the first 900 columns.

- *Samples from a normal distribution* with parameters $\mu = 0.5$ and $\sigma = 2$. These correspond to columns 900 to 916 amongst the features dataset.

- 16-bin *RGB colour histograms*, corresponding to the last 48 columns in the features dataset.

Not all 964 features should be used due to the large complexity introduced by the number of dimensions the data has, also known as the curse of dimensionality [7]. A preliminary check to determine which features can be considered "important" is conducted by calculating the standard correlation coefficient, a value ranging between -1 and 1, between the labels in ("*Y_train.csv*") and all the features "*X_train.csv*". A value close to 1 indicates a strong positive linear correlation, while a correlation close to -1 indicates a strong negative linear correlation. However, a correlation close to 0 indicates a lack of relationship in the data. According to existing literature, features are considered to have a high correlation when they have values in the region of $[|0.5, 0.7|]$ [7] [10]. To determine which features show the highest correlation with the class labels, the *pandas.DataFrame.corr* function is used on a concatenated DataFrame of the features and the labels. The ten features with the highest positive and negative correlation are shown in Appendix A.

A high positive correlation is achieved for features ranging between columns 916-964, which corresponds to RGB colour histograms, while high negative correlation is achieved for features ranging between columns 0-900, which corresponds to HoGs. However, because no feature from the normal distribution have a high correlation with the class labels, it can be dropped altogether.

3.2.4 Visualising features

HoG features To directly visualise some information about the images, the HoG features are reconstructed into images by being reshaped into 30x30 images, as seen in Figure 2.

Visualising the reconstructed HoG images confirms how it is impossible to discern the different classes with the human eye alone, and depicts how it will be impossible to infer anything from the reconstructed images of misclassified predictions.

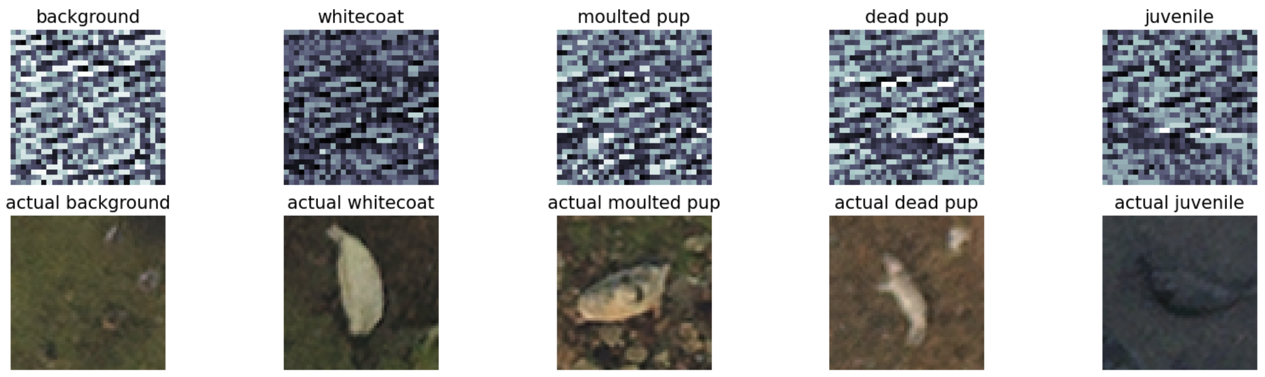


Figure 2: Comparison of images reconstructed images from HoG features (top and middle rows) with actual images (bottom row).

RGB colour histograms The features corresponding to the RGB colour histograms are visualised in Figure 3, revealing that there are more low-intensity pixels than high-intensity pixels.

Due to the considerable spread across the X and Y axes witnessed in the range of values for HoGs and RGB histograms (data distribution is not bell-shaped as some values are more concentrated on extremities), the data will require some feature transformation such as standardising the dataset.

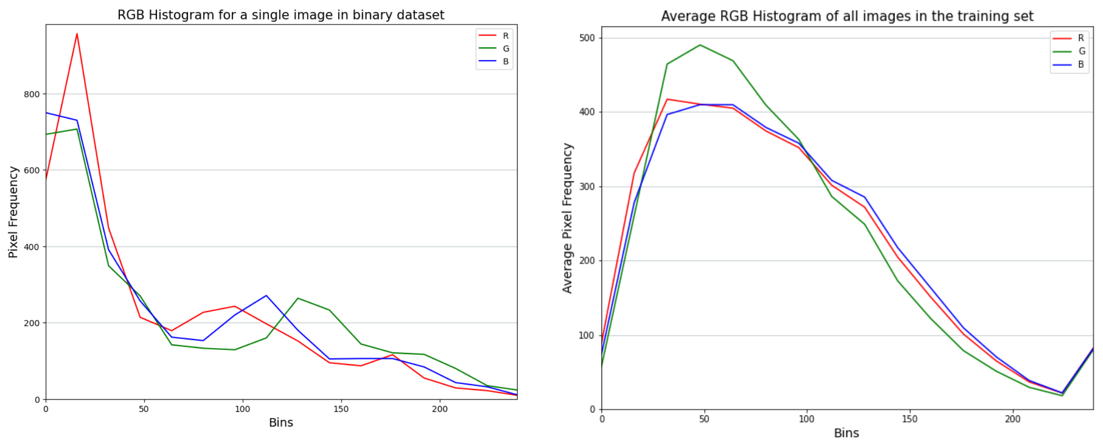


Figure 3: RGB histogram for a single random image (left) and average of all images in the training set (right).

3.3 Data pre-processing

With the data visualisation and analysis steps completed, the training set can now be pre-processed to optimise the classification models' fit on the data. Based on the conclusions made from the data analysis, five pre-processing steps are carried out.

3.3.1 Step 1: SMOTE oversampling

To solve the issue of skewed datasets raised in Section 3.2.2, one potential solution is to undersample or oversample the data. Undersampling the data is considered inefficient on its own as it may discard useful information from the training set. Regarding oversampling, many techniques exist, with the most simple one consisting in duplicating the samples from the infrequent classes to diminish the imbalance. However, such an approach would likely cause the classifier to overfit the data as samples from infrequent classes would have a very large weights, causing the classifiers to learn these examples "by heart", thus not generalising well to unseen data, even if it is similar [8].

An alternative is to produce synthetic data based on the infrequent classes by creating "similar" copies. This is achieved through the Synthetic Minority Over-Sampling Technique (SMOTE), which uses the K-Nearest Neighbours (kNN) algorithm to create similar data. These newly created samples are therefore not copies of the original ones, meaning the classifier will be better able to generalise [11].. The changes to both datasets after oversampling with SMOTE are shown in Figure 4 (compare with the class distributions found in Figure 1).

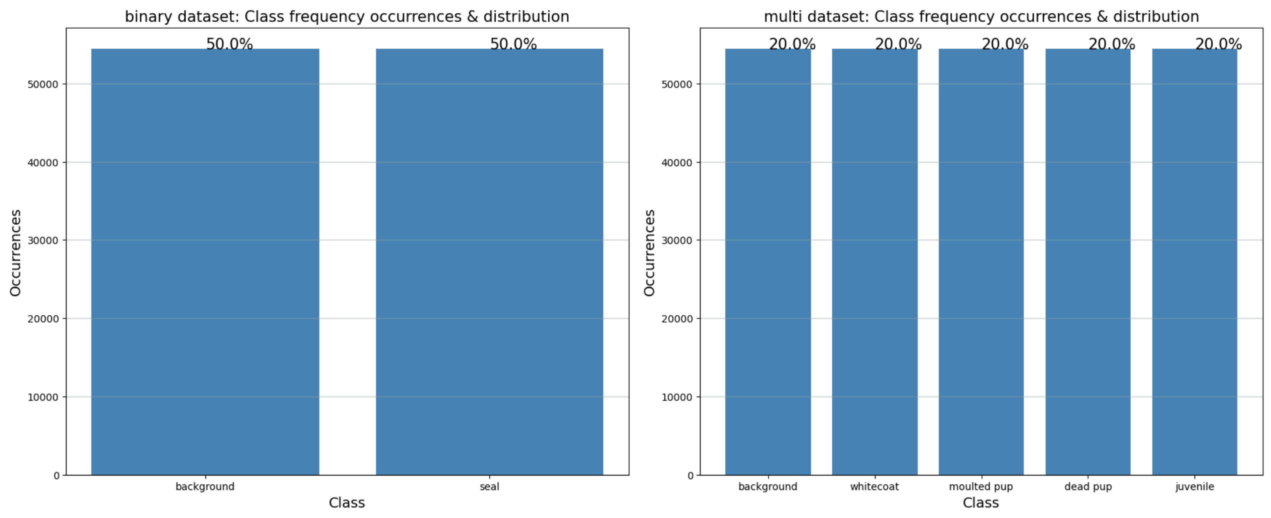


Figure 4: Class distribution of the binary (left) and multi (right) datasets after SMOTE oversampling.

The number of least frequent classes are all brought up to the number of the most abundant class, which is "*background*". This could improve the binary dataset by rising the number of "*seals*" from 7,778 to 54,432 for a total of 108,864 samples, which is acceptable. However, it causes the multi dataset to become too large, rising from 62,210 samples to 272,160. Additionally, in the case of the least frequent class, "*juvenile*", over 54,184 new samples are created from only 248 samples, raising doubt about how close to the original samples the synthetic ones are. Indeed, performing a quick cross validation test using the oversampled data on the multi dataset to confirm the aforementioned doubts reveals that the classifier largely overfits the oversampled multi dataset, reaching 97% accuracy using cross-validation, but only 22% accuracy on its predictions on the test data (checked through the online leaderboards tool provided[6]). Therefore, SMOTE oversampling is only applied to the binary dataset.

3.3.2 Step 2: Dropping features with poor correlation

Based on the correlation results, the normal distribution features can be dropped from the training dataset. This brings down the number of dimensions in the training data from 964 to 948.

3.3.3 Step 3: Standardising the dataset

Next, visualising the values of the dataset revealed how the data distribution is not bell-shaped and has very different scales between the HoG features and the RGB histograms. Indeed, HoG values are floats nearing 0, whereas the 16-bin RGB histogram values range from 0 to 255. Therefore, the features are standardised using the *sklearn.preprocessing.StandardScaler* class, resulting in a distribution with unit variance a shape closer to a bell (the two properties of standardised distributions are a mean of all values equal to 0 and a standard deviation equal to 1) [7] [12]. The *StandardScaler* instance used is saved to a Pickle file in the "*transform_pipeline*" directory to eventually apply the same the transformations on the test data.

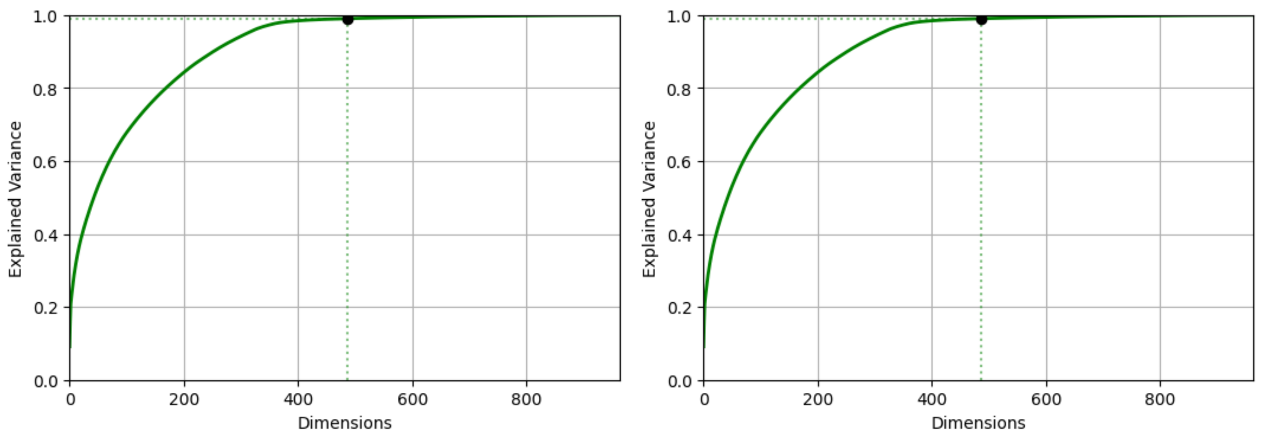
3.3.4 Step 4: Principal Component Analysis

However, the curse of dimensionality remains as there are still 948 columns in the dataset. The most popular solution to counter this problem is to use Principal Component Analysis (PCA) to reduce the number of dimensions via projection. This is achieved by using the *sklearn.decomposition.PCA* class. In order to choose the right number of dimensions to keep to preserve 99% of the training set's explained variance, the explained variance is plotted against the number of dimensions in Figure 5. The plot shows that to maintain 99% variance in the binary dataset, 492 dimensions can be kept (Figure 5a), while 475 dimensions can be kept in the multi dataset (Figure 5b). Diminishing the number of dimensions any lower would reduce the variance under 99% [7]. The training set is therefore reduced to almost half the size of the original dataset. The *PCA* instance used is also saved to a Pickle file in the "*transform_pipeline*" directory to apply the same the dimension reduction on the test data.

3.3.5 Step 5: Label preparation

Basic label processing is conducted for the binary data, converting categorical input to numerical input by converting "*backgrounds*" to 0 and "*seal*" to 1. The array holding the labels

[6]The terms used in this paragraph are all explained in further sections.



(a) Binary dataset: 492 dimensions to keep. (b) Multi dataset: 475 dimensions to keep.

Figure 5: Explained variance plotted against number of dimensions to reveal the number of dimensions that are required to maintain 99% variance in both datasets.

is then flattened using the *numpy.ravel* function in order to be compatible with Scikit-Learn's classifiers.

3.4 Model training

3.4.1 Model selection strategy

At this point, the training data is ready to be fed into the classifiers. The classification models will fit the transformed features from "*X_train.csv*" before making their own predictions, which will be compared with the real labels found in "*Y_train.csv*" for evaluation.

According to A. Géron, an efficient approach to model selection is to try out multiple classifiers, with hyperparameters close to the default values recommended by Scikit-Learn. The goal of this method is to gain a high-level intuition of which classification models to further explore and tweak [7].

After briefly evaluating a Stochastic Gradient Descent (SGD) classifier, One-versus-Rest (OvR) Logistic Regression, Linear and Polynomial (3rd degree) Support Vector Classifiers (SVC), Decision Tree and Multi-Layer Perceptron[7] (MLP), using 3-fold cross validation on all, the MLP was selected due to its consistency across different scoring methods and its lower runtime. "Simpler" models such as the SGD or OvR logistic regressor were not customisable enough, running the risk of poorly adapting to the large dataset, whereas Support Vector Machine-based and Decision Tree-based classifiers took a very long time to fit and are likely to overfit the data if poorly tuned. Therefore, neural networks provided the perfect middle ground for being highly customisable (23 hyperparameters) and quicker than the other models. See Appendix B for the results in tabular format.

[7]Multi-Layer Perceptron are also known as Neural Networks

3.4.2 Classification performance metrics

As mentioned in Section 3.2.2, simply using accuracy can be misleading. Therefore, other metrics are used, namely confusion matrices and the F1 score.

Confusion matrix This metric plots the number of predictions made for each class for each possible class in a table, with each row corresponding to the actual labels (the ones found in "*Y_train.csv*") and each column corresponding to a prediction. It is very useful for detecting which actual classes are being detected the most, and what predicted classes are misclassified as. To further highlight the misclassifications, the confusion matrix can be normalised to show a percentage [7].

F1 Score *Precision* corresponds to the number of positive predictions for a class, whereas *recall* is the number of positive instances that are correctly predicted. Together, they can be combined into a more concise metric, which is the *F1 score* (see Equation 1). It corresponds to the harmonic mean of precision and recall, meaning that to achieve a high F1 score, both precision and recall must be high (unlike a regular mean). Because when the precision goes down the recall goes up, and vice versa, the F1 score is a reliable metric to evaluate a classifier since a high F1 score means a balance has been found between precision and recall [7].

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = \frac{TP}{TP + \frac{FN+FP}{2}} \tag{1}$$

3.4.3 K-fold cross validation

K-fold cross validation consists of a representative way of evaluating the trained regression model. It divides the training set into K subsets (e.g. K=3) and evaluates the model K times. Using the *sklearn.model_selection.cross_val_predict* function, predictions can be made on each subset, ultimately resulting in a clean prediction for each sample in the training set (the predictions are made by a model that never saw the data during training) [7]. Despite the advantages of running k-fold cross validation, it is not always possible due to the expensive cost of multiplying the runtimes by the number of folds.

With the research made on the use of cross validation through the *cross_val_predict* function, the motivation to not split the training data into training/validation sets was made. Indeed, splitting the imbalanced training dataset using the popular 80%/20% split would have considerably diminished the number of samples in infrequent classes, giving the pre-processing algorithms and classification models even fewer samples to learn these rare classes. For example, the number of "*juvenile*" class samples would be lowered from 248 to 198 in the training dataset, which could negatively affect the SMOTE oversampling (the synthetic copies would not be varied enough) and perhaps the models' correctness.

Additionally, the multi-layer perceptron classifiers being the quickest models support the choice of using cross validation (i.e. it would have been unrealistic to use cross validation with a polynomial SVC). Validation sets are usually created to ensure that the model can generalise to unseen data [7], but both the use of the *cross_val_predict* function (which allows clean predictions to be made) and the metrics mentioned in Section 3.4.2, coupled with the availability of the online leaderboards tool provided to view the overall accuracy of the final model's predictions made on "*X_test.csv*", are enough to determine if the model generalises well to unseen data.

3.4.4 Neural network hyperparameter tuning

The hyperparameters of Scikit-Learn's neural network classifier class *sklearn.neural_network. MLPClassifier* are manually tried out to briefly evaluate which have an impact on the evaluation. Next, two hyperparameter tuning algorithms are run to converge towards optimal classifiers for each dataset. An initial *randomised search* algorithm is run to shortlist hyperparameters that work well through the *sklearn.model_selection.RandomizedSearchCV* class. This algorithm is first chosen due to the large hyperparameter space of neural networks [7]. Next, the neural network is fine-tuned with the promising hyperparameters found from the randomised search by running a *grid search* algorithm through the *sklearn.model_selection.GridSearchCV* class. The randomised search evaluate the model using cross-validation with random hyperparameter values, while the grid search evaluates the model with all possible combinations of hyperparameters that are specified, making it the most efficient way of fine-tuning the regression models as manually finding and testing different combinations of parameters is very time-consuming [7].

Neural networks have complex layered structures. Figure 6 depicts a potential neural network that could be used to learn the data, containing an input layer a number of neurons equivalent to the number of features used (492 for the binary dataset, 475 for the multi), a hidden layer (the default Scikit-Learn size is 100 neurons) and an output layer with as many neurons as there are classes.

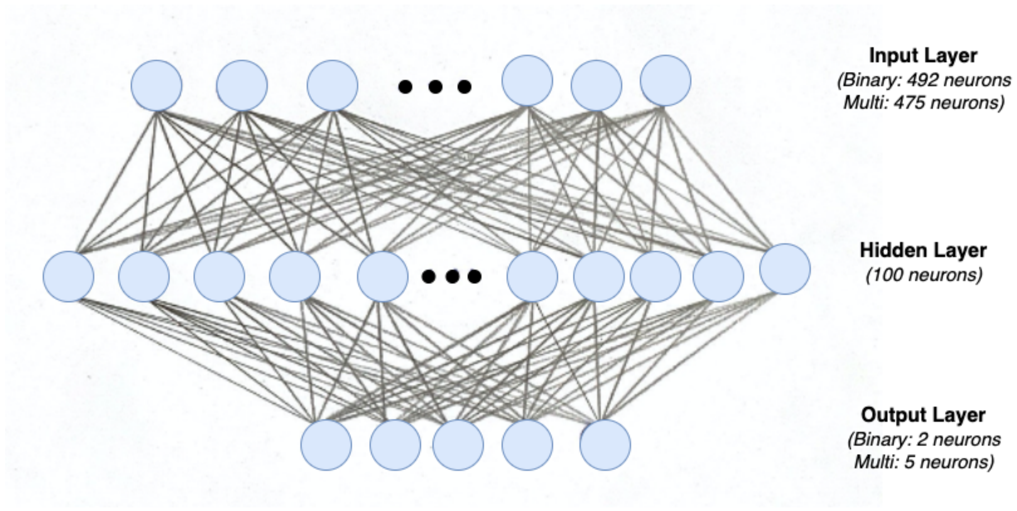


Figure 6: Visualisation of a potential neural network to learn the data.

To account for the "deep" structure of the neural network which could easily overfit the data, two of the hyperparameters explored is the amount of regularisation "alpha" and the number of neurons in the hidden layer. Additionally, because the neural network uses a stochastic gradient descent-based approach to minimise the error at each epoch (iteration) and backpropagation to adjust the weights accordingly at each epoch [13], the learning rate and momentum hyperparameters are explored as well. The most important different combinations of parameters explored across the binary and multi datasets through randomised and grid searches are located in Appendix C.

The results are all reported back into CSV files containing information about each combination of hyperparameters. The various models evaluated using 3-fold cross validation are ranked by their F1 score to determine how well the hyperparameters worked on them. The full results of each search are stored in csv files in the "*results/grid_search*" directory. In total, four randomised searches and three grid searches were conducted, resulting in 449 models trained over the course of 964 minutes. Among these, 251 were dedicated to the multi dataset and 198 to the binary, depicting the additional effort required to fine-tune the neural network for more complex data.

3.4.5 Full training approach

Training is tackled as follows:

1. The dataset's features and labels (either the binary of multi) are loaded into memory (see Section 3.1.2).
2. The features are pre-processed according to the steps mentioned in Section3.3, which include oversampling (for the binary dataset only), dropping features with low correlation, standardising the data and applying a PCA projection to reduce the number of dimensions. The labels are processed as well.
3. An instance of the custom class *Classifier* is created, in which either:
   - a hyperparameter-tuning algorithm is run and its results are analysed in Excel.
   - a classification model is created based on the hyperparameter-tuning algorithm results, fitted and evaluated using cross validation and multiple performance metrics. The model is saved in a Pickle file for future reference and for the final testing phase in the directory "*trained_classifiers*".

4 Conclusion: Evaluation & Critical Discussion

4.1 Evaluating the optimal neural networks

The training execution flow mentioned in Section 3.4.5 is followed for both the binary and multi datasets. Table 1 documents the cross-validation performances for the two optimal neural networks trained for each dataset.

The table indicates that the neural networks determined via hyperparameter fine-tuning perform well, achieving high F1 scores. Further analysis can be conducted by observing their

Dataset	Model Accuracy	Model F1 Score	Fine-tuned hyperparameters
Binary	99.07%	99.08%	- Hidden layers: 2 hidden layers of 114 neurons each - Alpha: 0.0001 - Learning rate: 0.001 - Momentum: 0.9
Multi	94.94%	94.44%	- Hidden layers: 1 hidden layers of 100 neurons - Alpha: 0.9 - Learning rate: 0.001 - Momentum: 0.1

Table 1: 3-fold cross-validation results on the optimal neural networks.

normalised confusion matrices in Figure 7 (see Appendix D for non-normalised confusion matrices).

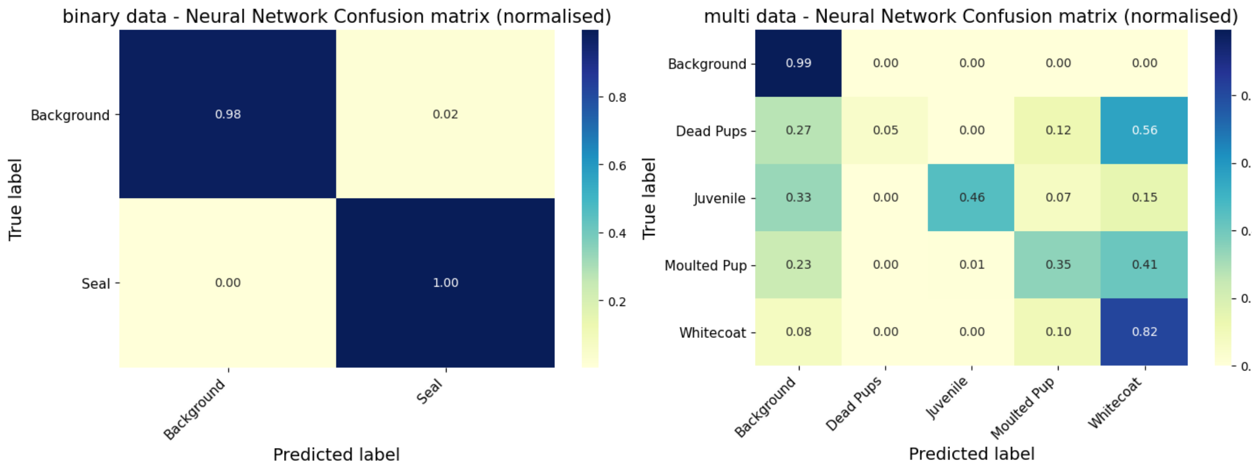
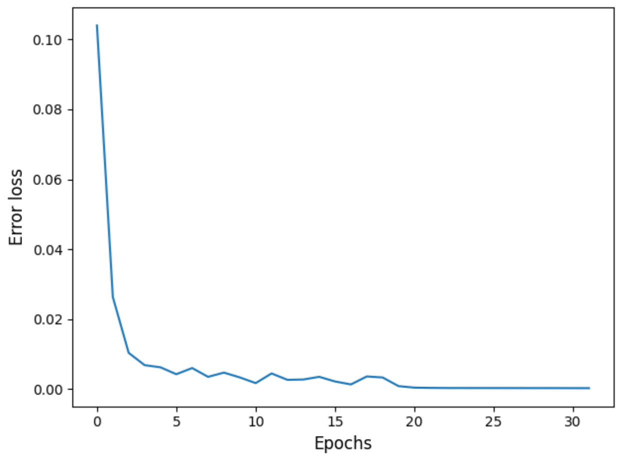


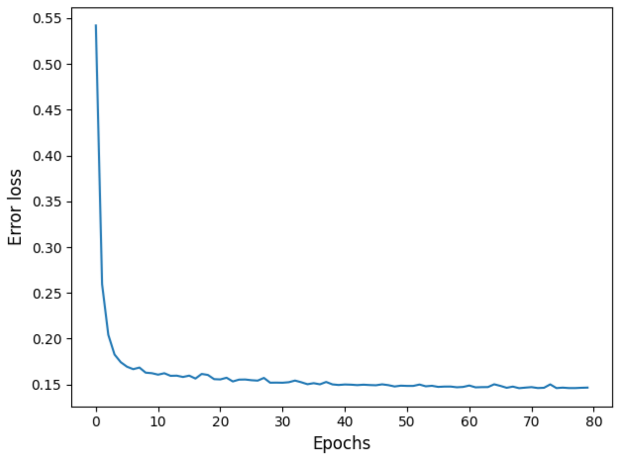
Figure 7: Normalised confusion matrices for optimal neural network models.

The normalised confusion matrices for the binary dataset reveal that all background predictions are correct, while only a few backgrounds are misclassified as seals. This could perhaps indicate that the synthetic seal samples created are not as good as the original background samples, pointing towards using a more efficient oversampling algorithm. Concerning the multi dataset, there are a lot more misclassifications due to confusion amongst the different types of seals. Indeed, there is often confusion between which type of seal is in the image, especially when whitecoat predictions are made (the whitecoat column is often filled). However, there are two pleasant surprises. The first one is how the infrequent classes, dead pups and juvenile, are often correctly classified when predicted, and the second is how any type of seal prediction is never classified as a background, indicating despite being confused about the type of seal, the neural network never classified a seal as a background.

Finally, the error loss of both neural networks is plotted in Figure 8. They both show the desired behaviour: quickly converging towards the designated error. However, the error loss observed for the binary neural network less smooth than the one observed for the multi neural network. These bumps could be due to the high momentum (0.9) used for the binary one compared to the low momentum for the multi one (0.001), suggesting that the momentum could be slightly tuned down.



(a) MLP on binary dataset



(b) MLP on multi dataset

Figure 8: Plotted error loss for optimal neural network models.

4.2 Final predictions & Critical discussion

Finally, the test features are loaded memory in the same fashion the training data was. This is achieved by loading back the *StandardScaler* (see Section 3.3.3) and *PCA* (see Section 3.3.4) instances that initially learned the training data to transform the test data. The "*transform*" function is used rather than the "*fit_transform*" function to do so.

The predictions made on the testing data found in "*X_test.csv*" are saved into a new CSV file named "*Y_test.csv*", one for each dataset. The binary and multi prediction files are uploaded to the online leaderboards tool provided in order to get the final accuracy of the neural network:

- Binary: 98.21% accuracy
- Multi: 97.58% accuracy

The accuracy of the predictions made by the two neural networks confirms that they fitted the training data well without underfitting nor overfitting it, and generalised well to unseen data. Further improvements could have potentially been made by exploring other advanced models proven to work with classification tasks, such as support vector machines, random forests or XGBoost [7]. Based on the brief testing conducted earlier in Section 3.4.1, the models seemed to either overfit the data or took too long to converge, depicting that deeper knowledge of these models is required to fine-tune their hyperparameters in order to correctly use them. For this reason, familiar models were chosen, with the neural network ultimately selected.

References

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[2] Jeff Reback; Wes McKinney; jbrockmendel; Joris Van den Bossche; Tom Augspurger; Phillip Cloud; gfyoung; Sinhrks; Adam Klein; Matthew Roeschke; Simon Hawkins; Jeff Tratner; Chang She; William Ayd; Terji Petersen; Marc Garcia; Jeremy Schendel; Andy Hayden; MomIsBestFriend; Vytautas Jancauskas; Pietro Battiston; Skipper Seabold; chris-b1; h-vetinari; Stephan Hoyer; Wouter Overmeire; alimcmaster1; Kaiqi Dong; Christopher Whelan; Mortada Mehyar. pandas-dev/pandas: Pandas, February 2020.

[3] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[4] Travis Oliphant and NumPy developers. NumPy. `https://numpy.org/`, 2020. [Online] Accessed: 2020-05-04.

[5] Waskom Michael. seaborn: statistical data visualization. `https://seaborn.pydata.org/`, 2020. [Online] Accessed: 2020-05-04.

[6] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5, 2017.

[7] Aurelien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow.* O'Reilly Media, 2nd edition, 2019.

[8] Nick Becker. The Right Way to Oversample in Predictive Modeling. `https://beckernick.github.io/oversampling-modeling/`, 2020. [Online] Accessed: 2020-05-05.

[9] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Proceedings - 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005*, volume I, pages 886–893, 2005.

[10] Will Badr. Why Feature Correlation Matters .... A Lot! `https://towardsdatascience.com/why-feature-correlation-matters-a-lot-847e8ba439c4`, 2019. [Online] Accessed: 2020-04-26.

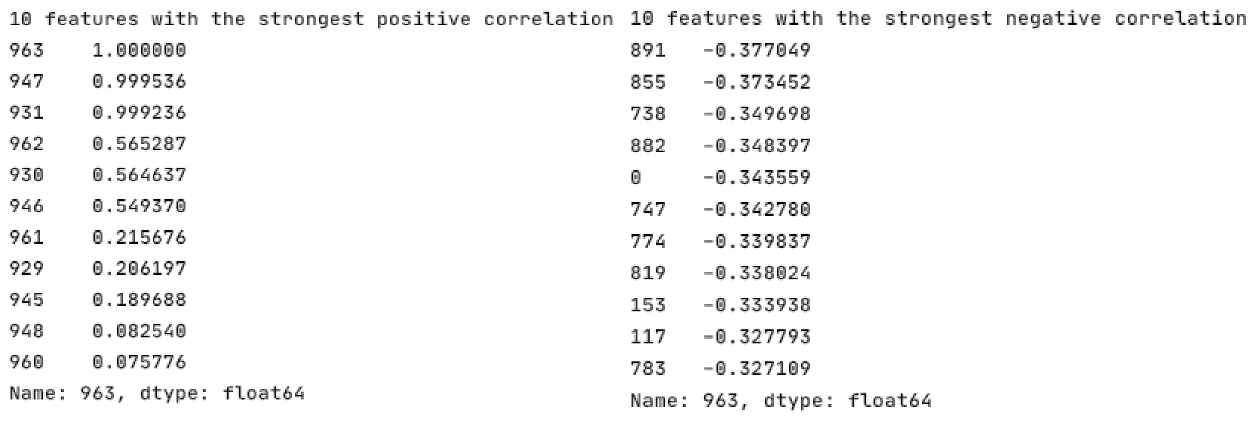
[11] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[12] Stephanie Glen. Standardized Beta Coefficient: Definition & Example. `https://www.statisticshowto.datasciencecentral.com/standardized-values-examples/`, 2014. [Online] Accessed: 2020-04-26.

[13] Stuart J Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education Limited, 2016.

Appendix A Features With Highest Correlation

Top 10 training dataset column indexes (features) with the highest standard correlation coefficient. Left shows positive correlations, right shows negative correlation.



```
10 features with the strongest positive correlation
963    1.000000
947    0.999536
931    0.999236
962    0.565287
930    0.564637
946    0.549370
961    0.215676
929    0.206197
945    0.189688
948    0.082540
960    0.075776
Name: 963, dtype: float64
```

```
10 features with the strongest negative correlation
891   -0.377049
855   -0.373452
738   -0.349698
882   -0.348397
0     -0.343559
747   -0.342780
774   -0.339837
819   -0.338024
153   -0.333938
117   -0.327793
783   -0.327109
Name: 963, dtype: float64
```

Figure 9: Positive correlations (left) and negative correlations (right).

Appendix B Initial Default Classifiers Evaluation

Model	**SGD***	**Logistic* (OvR)**	**Linear SVC**	**Polynomial SVC***	**Decision Tree**	**Neural Network**
Accuracy	95.88%	96.62%	95.89%	31.58%	92.58%	93.59%
Precision	87.42%	91.14%	87.22%	14.96%	70.63%	75.98%
Recall	78.27%	80.86%	78.70%	95.46%	69.59%	71.28%
F1 score	82.59%	85.69%	82.74%	25.87%	70.11%	73.55%
Runtime (seconds)	70.28	104.41	160.09	252.95	442.02	13.58
Max # iterations	10,000	100	1,000	1,000	∞	100
Convergence warnings	No	Yes	Yes	Yes	No	No

Table 2: Initial classification models evaluation using 3-fold cross validation on the binary dataset.
**Using all available processors for quicker runtimes.*

Model	**SGD***	**Logistic* (OvR)**	**Linear SVC**	**Polynomial SVC***	**Decision Tree**	**Neural Network**
Accuracy	93.02%	93.95%	93.67%	N/A	88.49%	91.85%
Precision	91.41%	93.26%	92.42%	N/A	88.79%	90.63%
Recall	93.02%	93.95%	93.67%	N/A	88.49%	91.86%
F1 score	91.69%	93.50%	92.71%	N/A	88.64%	90.80%
Runtime (seconds)	294.35	136.55	656.8	N/A	540.11	187.61
Max # iterations	10,000	100	1,000	10,000	∞	100
Convergence warnings	Yes	Yes	Yes	Too long	No	Yes

Table 3: Initial classification models evaluation using 3-fold cross validation on the multi dataset.
**Using all available processors for quicker runtimes.*

Appendix C Hyperparameter Tuning Combinations

```
# Initialise Grid Search.
if config.is_grid_search:
    if config.dataset == "binary":
        parameters = {
            "hidden_layer_sizes": [(98,), (98, 98), (114,), (114, 114)],
            "learning_rate_init": [0.001, 0.03, 0.04, 0.1],
            "alpha": [0.0001, 0.26, 0.96]
        }
    elif config.dataset == "multi":
        parameters = {
            "hidden_layer_sizes": [(68,), (68, 68), (100,), (100, 100)],
            "learning_rate_init": [0.001, 0.01, 0.1],
            "momentum": [0.1, 0.9],
            "alpha": [0.0001, 0.1, 0.9]
        }
    searchCV = GridSearchCV(param_grid=parameters, estimator=self.clf, cv=self.
   folds, scoring=scoring)
# Initialise Randomised Search.
elif config.is_randomised_search:
    parameters = {
        'hidden_layer_sizes': (sp_randint(1, 150)),
        'learning_rate_init': sp_uniform(0.001, 1),
        'momentum': sp_uniform(0.1, 0.9),
        'alpha': sp_uniform(0.0001, 1)
    }
    searchCV = RandomizedSearchCV(param_distributions=parameters, estimator=
   self.clf, n_iter=100, cv=self.folds, scoring=scoring)
gs_results = searchCV.fit(self.X, self.y)
```

Appendix D Final Neural Network Confusion Matrices (Not Normalised)

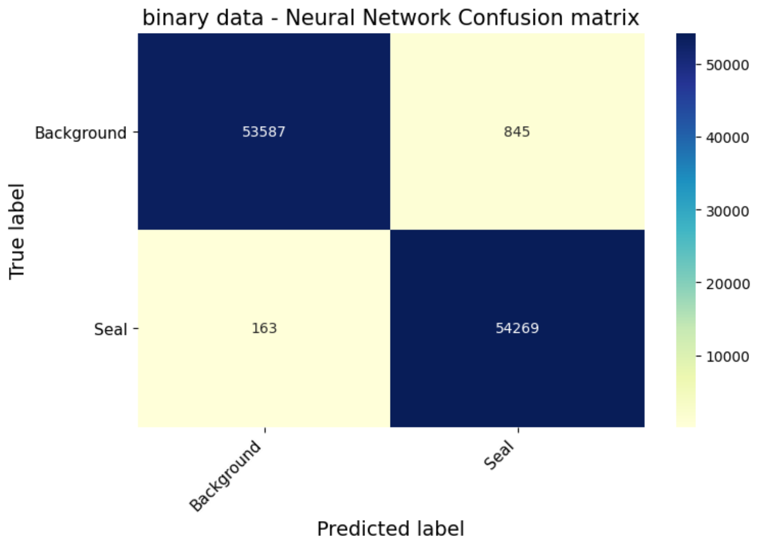


Figure 10: Confusion matrix for binary dataset.

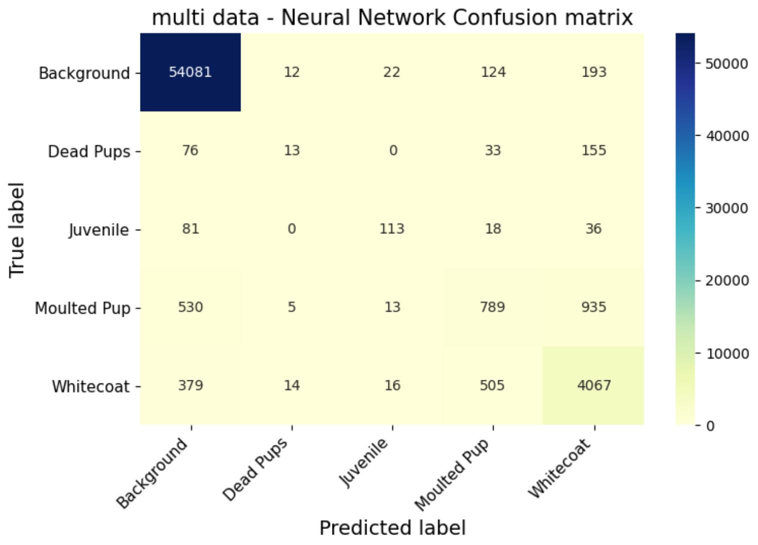


Figure 11: Confusion matrix for multi dataset.