TÉCNICO LISBOA

Machine Learning

Eletrical and Computer Engineering

Project Report

Students:

Alexandre Reis (100123) alexandre.leite.reis@tecnico.ulisboa.pt
Rodolfo Amorim (100260) rodolfo.amorim@tecnico.ulisboa.pt

Group 34

2023/2024 – First Semester, Q1

1 Tasks

1.1 Task 4.1

For the first task, we decided to use a **Linear Regression Model** [5] for the given training set, which consisted of 15 examples, each with 10 features. Given that the number of examples is close to the number of features, the model, built using the training set as it is, would be prone to **overfit.**

Accordingly, we began by using **Lasso Regularization** [6], since this technique allows us to select the most important features out of the 10, thus reducing our models complexity. After centering our training data, we used **cross-validation** (due to our small sample size), for many different k-folds (from 2 to 15), to select the Lasso's hyperparameter λ_{Lasso}. The best k-fold and λ_{Lasso} were chosen based on the Negative Mean Squared Error, and are on the following table:

k-fold	λ_{Lasso}	NMSE
4	0.05	-2.039

For this $\lambda_{Lasso} = 0.05$ the Lasso Regularization technique gave the following sparse vector of coefficients:

Feature	1	2	3	4	5	6	7	8	9	10
Value	0	-0.36	1.004	-0.236	0	1.64	-0.88	0	0.127	0.64

As we can seen from the previous table, the Lasso Regularization excluded the following features: $[1, 5, 8]$, hence reducing the original number of features to 7.

After having applied the Lasso Regularization, we decided to apply the **Ridge Regularization** [7] and see what value we would get for the NMSE. In contrast to the Lasso Regularization, the Ridge Regularization can't set any coefficient value to zero, however it penalizes big errors more, which leads to small coefficients values. Just as before, we used cross-validation, for many different k-folds and based our decision on the Negative Mean Squared Error, for the Ridge's hyperparameter. The best values were the following:

k-fold	λ_{Ridge}	NMSE
4	2.15	-2.35

Since the Lasso Regularization had an NMSE smaller than that of Ridge Regularization we decided to use the first technique. However, we still wanted to further improve our model, and as such we decided to use the Ridge Regularization after having applied the Lasso Regularization. Once more we searched for the best Ridge hyperparameter and got the following results:

k-fold	λ_{Ridge}	NMSE
4	0.51	-1.45

For a $\lambda_{Ridge} = 0.51$ the Ridge Regularization technique gave the following vector of coefficients:

Feature	2	3	4	6	7	9	10
Value	-0.404	0.97	-0.28	1.62	-0.915	0.242	0.67

Comparing these coefficients to the previous sparse vector of coefficients from Lasso, we can see that there's a decrease in the values of features: [2,3,4,6,7], as expected, and a small increase in the value of features: [9, 10], which is not expected. This may be due to the significance that feature 9 and 10 have on minimizing the NMSE, which in turn allows to further reduce the size of the other coefficients.

To sum up, we show all of our models performances on the following table:

Model	NMSE
Lasso + Ridge	-1.45
Lasso	-2.04
Ridge	-2.18
No Regularization	-30.8

Accordingly, with the previous table, we concluded that using both the **Lasso and Ridge Regularization** on our Linear Regression Model was the best option. Having done the all the Regularization techniques, we moved on to **inverting the pre-processing** [2]and obtain our vector of coefficients, which is expressed bellow, in order to make our predictions for the test set.

$$\boxed{\hat{\beta} = \left[\hat{\beta}_0 \quad \hat{\beta}'^T\right], \hat{\beta}_0 = \bar{y} - \bar{x}^T \hat{\beta}'^T}$$

The value obtained by the teaching team, for the SSE was: **1858.30793**, which translates to an NMSE of: **-1,85830793**. Comparing this value to the previous table, we can see that our model's performance (in terms of NMSE) was worse than what we predicted. However, the difference in performance was small (approximately **0.4083**), and the performance itself was good, which tells that our model's generalization ability was somewhat good.

A possible way of improving our model's performance could be to analyse our trainning data, and remove possible outliers. This would be especially indicated if the amount of training data would be bigger, as if we removed some data from the 15 samples, the training data could become to small and non representative (this is why this was not done in this task).

1.2 Task 4.2

In this task, data is generated with 2 different linear models. If we knew what data was generated by each linear model, a similar approach to Task 4.1. As such, in this problem an additional step has to be done in order to separate the data from each model before performing the linear regressions.

Before trying to separate the data, a benchmark **Linear regression with Ridge Regularization** was performed to the training set and the results were:

	NMSE of Linear Regression	**R^2 of Linear regression**
Ridge linear regression for the entire/ original training set	-1.350	0.298

It is possible to conclude that the model does not fit the data at all (due to the R^2 value being close to 0) . Due to this, changes must be made to our model.

There are several approaches that can be used to separate the data, but the one we used was **Clustering**. This method separates the dataset into n amounts of clusters that share some similarities, which is exactly what we want in this case.

For this task we used two different clustering models:

- **K-means clustering** [4] - This is the standard clustering method and is usually one of the first methods to try when using clustering. This method tries to separate data into n clusters with equal variance, which means that this method separates data accordingly to how close they are together in space.

- **Gaussian Mixture Models** [4] - This method assumes that the data is generated from a mixture of finite Gaussian distributions with unknown parameters.

These clustering methods were trained using a matrix with all of X_{train} features and the y_{train} values. The reason for also having included a column for the y values is that the x values should not depend on the applied linear model, only the y values do. It was considered to give more weight to the y feature column, as it is decisive for the data clustering, but the GMM clustering method showed to have great results without this. Thus, we concluded that this non-weighted format for the training matrix was the correct approach.

For each of these clustering models, the data was divided into **2 different clusters** and then, for each cluster, a Linear Regression with Ridge regularization was applied. Using cross-validation, a Negative Mean Squared Error was obtain for each regression.

The Ridge regularization was chosen over the Lasso regularization, since there were a small number of features (only 4), so feature selection was not needed.

	Fraction of data	**NMSE of Linear Regression**		**R^2 of Linear Regression**	
Clustering Method	**from each linear model**	Linear Model 1	Linear Model 2	Linear Model 1	Linear Model 2
Gaussian Mixture Model	43% - 57%	-0.053	-0.038	0.971	0.980
k-Means	51% - 49%	-0.665	-0.532	0.260	0.195

Table 1: Comparison of the 2 used Clustering models

Analysing the data from table 1 we can conclude that the 2 clustering methods produce very different results in regards to NMSE and R^2. But estimate similar distributions between the 2 classes, in percentage.

For the **k-means**, we can see that the R^2 did not increase from the non-split dataset, and still continues to be close to 0 for the two linear regressions. This means that the data was not split accordingly to the linear models that generated it. This makes sense, as k-mean focuses on clustering data that is close together, based on the centroid of that cluster. Due to this problem's formulation, the training data does not have to be far apart - depending on the model that generated it, it just has to have a different linear relation between features x and y values. Therefore, this clustering method is rejected for this task.

In contrast, the results obtained from the **GMM** clustering method were very good, with R^2 values getting close to 1, which means that the performed linear regressions fits the data very well. This is only possible if the clustering method correctly separated the data . On top of good R^2 values, this method also achieved very good NMSE values, which shows that it should perform well on the test data.

In the end, GMM clustering was used to separate the data, followed by two linear regression with Ridge regularization, one for each cluster, to make predictions. For this regression, several λ ridge hyperparameters were tried and the best one was $\lambda_{Ridge} = 0.01$ (this was done using cross validation).

Using both predicted linear models, y_{test} results were generated and submitted with a classification (SSE=48.084 which is equivalent to NMSE=-0.0481). This was almost exactly what was expected (between -0.053 and -0.038 1) with the validation data set. The results proved that the GMM clustering method was the appropriate one for this task, as was predicted from the definition of the method itself.

1.3 Task 5.1

Both task 5.1 and 5.2 are image classification tasks. After some consideration, it was concluded that the industry standard model for this type of problem is **Convolutional Neural Networks**. As taught in ML lectures [2], these neural networks' hidden layers are composed of: Convolutional layers, with n amount of kernel/filters each; Pooling layers; Flateen layers (that deconstruct a matrix array into a vector that feeds into dense layers) and Dense layers/fully connected layers, each with m amount of nodes. All of the layers mentioned above were used as hidden layers of the developed CNN for this task. A Convolutional layer was also used as an input layer for the 28x3x3 images that we want to classify.

As this is a binary classification problem, there were 2 options for the output layer of our CNN:

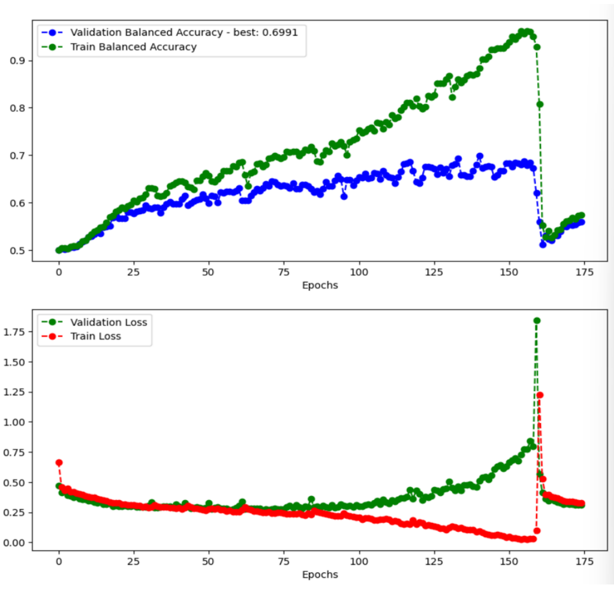
- 1 node, with a **sigmoid activation function**. For this activation function, if it's input is bellow -5, the function returns a value close to zero; and if the input is above 5, it returns a value close to 1. This means that the output of the CNN is a single node with values ranging from 0 to 1, enabling to classify each sample of data with one of the two labels, using this value. In the case of using this output function, the model should use the "Binary Crossentropy" [1] loss function.

- 2 nodes, using a **softmax activation function**. As there is more than one output node, in this case, the "Categorical Crossentropy" [1] loss function should be use to train the model, as there is more than one output node. After predicting test values using the trained model, in this case, an argmax function should be use, to select the class that corresponds to the node with the most softmax value.

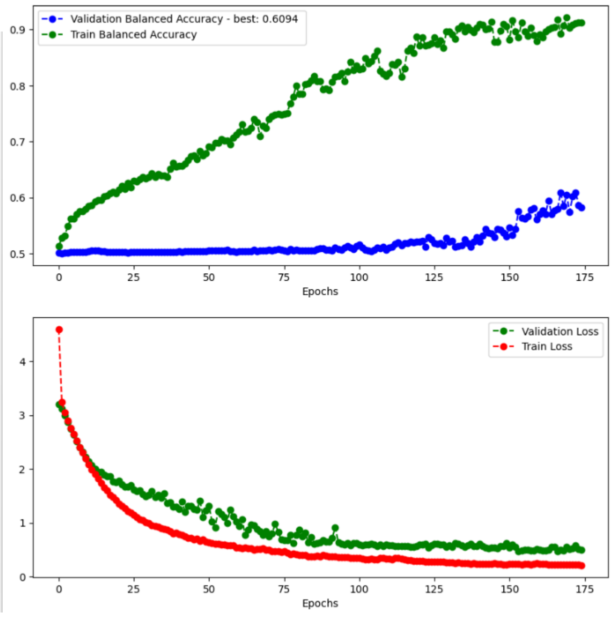
The two methods should be equivalent and, as to prepare for the multi-class classification problem in task 5.2, the **softmax** option was used, because it can be used for any amount of number of labels/ classes. The **"Adam"** [1] was the optimizer of choice for the CNN in these Tasks. This is based on the stochastic gradient descent method.

Using this output layer, an initial CNN was built, with a similar arrangement to well developed CNNs like AlexNet [2]. This had a hidden layer with Convolutional layers activated by the "relu" function, Max Pooling layers, a Flatten Layer and some Dense layers with the same activation function, preceding the 2 node output layer that uses sofmax activation. For the time being this CNN did not include any regularization techniques.

This model was trained with the given training data. The only caution taken was to preprocess all x values to range from 0 to 1, instead of 0 to 255 (RGB value for each colour). The



(a) Original CNN with imbalanced data and without regularization (Model 1)



(b) Original CNN with regularization and Class Weights

Figure 1

result was a model that had 2 clear flaws: It was **overfitting**, as after epoch 100 the loss value of validation data began to increase while the training data loss value continue to decrease (figure 1a); The Confusion Matrix indicates that the model **never predicts any validation data to be of class 1** (melanoma), despite the model's predictions to the validation data-set not having very bad Balanced Accuracy value (Model 1 in table 2).

Model	Best Balanced Accuracy	Confusion Matrix
Model 1	0.699	0.874, 0. 0.126, 0.
Regularization and Class weights	0.712	0.902, 0.435 0.098, 0.565
Regularization and Oversampling	0.698	0.943, 0.558 0.057, 0.442

Table 2: Balanced accuracy and Binary confusion matrix for initial CNN models

To solve the first problem, the CNN was adapted to include several **regularization methods** [1]. The ones that proved to be more effective were: Batch Normalization between convolutional layers; weights Droupout mainly between fully connected layers, but also between some convolutional layers; kernel regularizer for the kernels of each convolutional layer, Activity and Bias regularization for the dense layers. The result was a model that did not tend to overfitting. These same regularization techniques were used in all other models for tasks 5.1 and 5.2 and proved very effective in preventing overfitting.

The cause of the second problem is the fact that the data-set was **imbalanced**. This means that there is a big difference in the amount of samples from one class compared to another

(0.86% class 0 - Nervu; 0.14% class 1 - Melanoma). In practical terms, when the model is training with an unbalanced dataset, it will tend to always predict the majority class, as it will be a correct prediction 86% of the time (in this case). This is a big problem as this model's predictions cannot be trusted. To solve this, there were two different approaches that were tried:

- **Class weights** [3] - For each class a weight is computed, which is inversely proportional to the fraction of samples with a label corresponding to this class in relation to the entire data-set. When training the model, the loss function for each sample will be multiplicated by the weight corresponding to its class. This means that when minimizing the cost function, it is equally as important to predict the majority classes correctly but also the minority classes.

- **Oversampling** with data augmentation - This method's goal is to generate a dataset with equal amount of samples from each class. To do that, there were images generated that belong to the minority class, with a method called Data Augmentation. This is a method that takes images that belong to the original data set and applies transformations in order to generate a new image that is guarantee to represent an object with the same label as the original image, but is considered different in the point of view of the convolutional layers. The transformations that were applied are rotation, width and height shifts, flips, zoom and brightness changes, which all together create new images that visibly look like they are part of the original training set.

No data balancing technique was used on the validation data, as this should represent the real test data with class sample distributions similar to the test ones.

The results were better, solving the overfitting problem (image 1b) and actually predicting some validation samples to be of the minority class (Class Weights and Oversampling Models of table 2). The balanced accuracy is still very bad, which is mainly due to the model getting about half the minority class predictions wrong. As the balanced accuracy is the metric which we are trying to maximize, several parameters were tuned:

Batch size	32	64	128	512	1024	2048
Original CNN with class weights	0.590	0.621	0.654	0.689	0.712	0.6094
Final CNN with class weights	-	-	-	-	0.777	0.781

Shuffled training data	No	Yes
Original CNN with class weights	0.712	0.749
Original CNN with oversampling	0.698	0.730

Table 3: Balanced accuracy values for CNN with different parameters

Different **Convolutional Neural Network layer configurations** were tried until one that improved the metric significantly was reached. The main gains were in reducing the number of dense layers and units in those layers. This makes sense as a binary classification task should be simple, and most of the complexity should be in identifying features using convolutions layers. This CNN is called **"Final CNN"** and was always trained using **shuffled data**, as is is proven in tabel 3 to improve results.

In terms of **batch size**, we decided to use mini-batches as is indicated for most cases. Several values were tried and we concluded that for the Final CNN, the best approach would be using 2048 as the batch size. This value did not increase the training time to much and improved results. This is a big batch size, given the size of the data set and could have lead to a lack of generalization of our model, which could have injured our results in this task.

The 2 discussed solutions for solving the imbalanced data set problem were compared for the "Final CNN" model and the oversampling method with augmented data was chosen. The main reason for this is that the validation balanced accuracy value was slightly higher, reaching a value of **0.767**. In conclusion, the delivered model for this task uses the "Final CNN", with a batch size of 2048. The data used to train it is pre-processed with a augmentation oversampling method that deals with the imbalanced nature of the dataset, besides being shuffled and normalized.

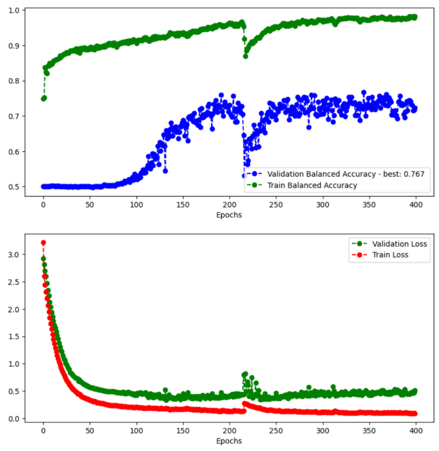


Figure 2: Task 5.1's delivered model"

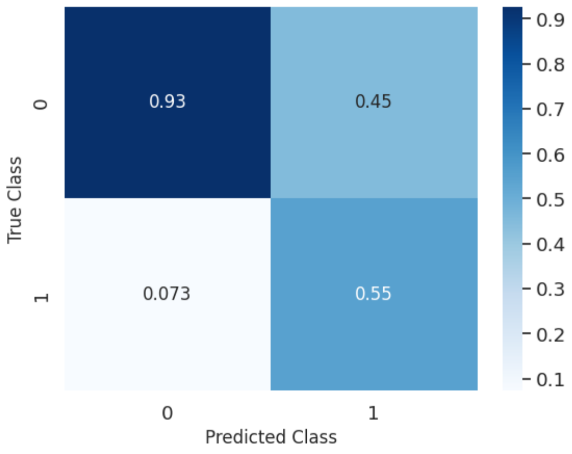


Figure 3: Confusion matrix for Task 5.1's delivered model

The final balanced accuracy result is not great, but it was not clear (at the time) what could be done to improve this value. The main problem with this model is it's ability to predict the minority class. Even thought methods to deal with unbalanced data were used, the confusion matrix clearly shows that there are almost as many true melanomas predicted as nervus as there are melanomas correctly predicted as melanomas. In Task 5.2 some mistakes made in this task were understood and the group results improved significantly.

1.4 Task 5.2

This is a very similar problem to the previous one, but with 6 classes instead of 2. Because of this we tried to analyze our results from the previous task.

Our model's balanced accuracy, for the previous classification task, was around 0.77 whilst our score was 0.62. This difference expresses a lack of generalizing ability from our Convolutional Neural Network, as such, we decided to make the following changes:

We began by reducing our **Batch Size** of 2048 to 256. Even though a big batch size can lead to a faster convergence and training time, it can also lead to a lack of generalization by the model. In order to choose the batch size we did multiple trials using the same CNN and epoch number:

Batch Size	Average Balanced Accuracy (5 trials)
32	0.6367
64	0.6518
128	0.6629
256	0.6723

After changing the batch size, we decided to **reduce our CNN complexity**, which would make it faster to train and less prone to overfitting, since there would be fewer weights and hyperparameters to tune. In order to do so, we experimented changing the number of hidden layers, its hyperparameters and regularization terms. We repeated the previous methodology for multiple trials, by using the same number of epochs and a batch size of 256. At each trial we evaluated the CNN based on its balanced accuracy. While we were experimenting we found

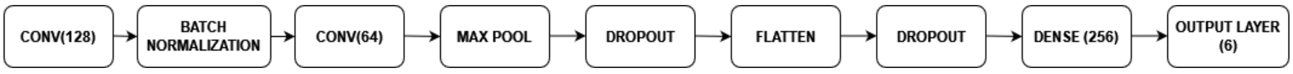


Figure 4: CNN with the highest Balanced Accuracy

two things. Firstly, **shuffling** our X_train and y_train synchronously before giving it to our CNN, improved its balanced accuracy. Upon further research, we saw that shuffling the training set before it is given to a neural network is known to often improve the model's performance. Secondly, **updating the learning rate at each epoch** also lead to better results in terms of balanced accuracy. With some research, we also saw that updating the learning rate as the neural network is trained has been shown to often reduce the training time and improve model's performance. In order to do so, we used the keras function: *LearningRateScheduler()*, and changed the learning rate the following way (with α_t equal to the learning rate at the previous epoch t):

$$\alpha_{t+1} = \begin{cases} \alpha_t, & \text{if } t < 50 \\ 0.95 \cdot \alpha_t, & \text{otherwise} \end{cases}$$

Finally, the last experiment that we did was comparing our model from figure 4 to an **ensemble of four models**, since it could lead to an performance enhancement. The first model of our ensemble is the one displayed on figure 4, with the remaining three being adaptations of the first model. These adaptions include changing parameters, regularization terms and the number of convolutional layers. In order to compare these two models we used the their **confusion matrix** (evaluated on the validation set). Accordingly, their confusion matrix is the following:

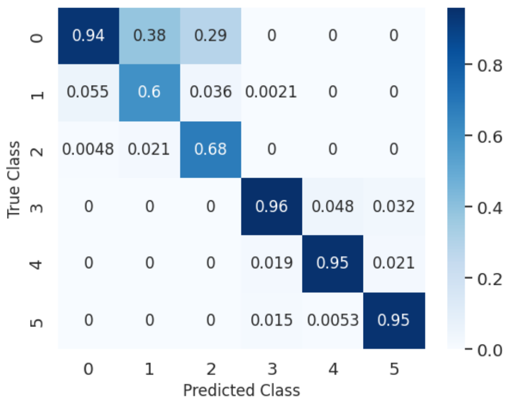


Figure 5: Ensemble Confusion Matrix

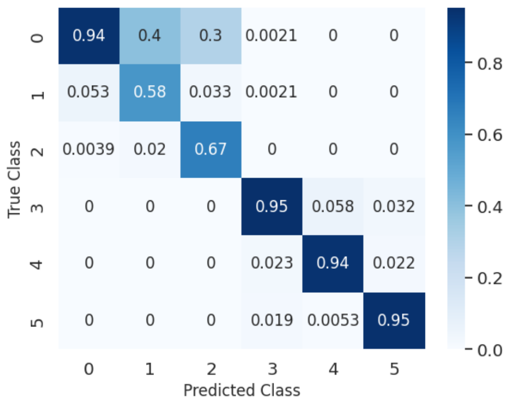


Figure 6: Model 1 Confusion Matrix

By analysing both figure 5 and 6 we can see that the diagonal values of the **ensemble's** confusion matrix (All the confusion matrices in this report were normalized, however so as to make them more readable we rounded up some numbers), are equal to or greater than the diagonal values of model's 1 confusion matrix. Knowing that the diagonal values of a confusion matrix, (for multi-class classification), express correct predictions, we can conclude that our Ensemble has a better performance than our model 1 (figure 4). Additionally, we can also see that both models confuse class 0 with both classes 1 and 2. These two classes are also the two that have the lowest diagonal value in the confusion matrix of both models. This can be in part due to all of these 3 classes being from dermoscopy dataset (making them similar to each other), but mostly due to their low representation in the training set, which class weights tries to compensate but can't fully do (Percentage in relation to the size of the training set: $\text{Class } 1 : 8.37\%, \text{Class } 2 : 1.1\%$).

Using the the ensemble confusion matrix, figure 5, we made a prediction on the expected Balanced Accuracy of our model. As such, we calculated the average of the Balanced Accuracy for each class, which gave us a value of approximately **0.858**. Comparing this value to the one obtained by the teaching team, which was **0.84178792**, we can conclude that our guess for our ensemble's Balanced Accuracy wasn't far off and that, overall, the changes made in this task were helpful.

References

[1] Keras. Keras neural networks documentation. Accessed on October 31, 2023.

[2] Jorge S. Marques and Alexandre Bernardino. *Machine Learning Slides (Lectures)*. 2023.

[3] Scikit learn. Class weights. Accessed on October 31, 2023.

[4] Scikit learn. Clustering. Accessed on October 31, 2023.

[5] Scikit learn. Linear models. Accessed on October 31, 2023.

[6] Scikit learn. Linear models - lasso regularization. Accessed on October 31, 2023.

[7] Scikit learn. Linear models - ridge regularization. Accessed on October 31, 2023.