Machine Learning Project Report

Beatriz Mateus Pereira (95922), Max Nobre Supelnic (95960)
Instituto Superior Técnico
Universidade de Lisboa, 2023/2024

1 Part 1: Regression with Synthetic Data

1.1 First Problem: Linear Regression

The first problem's objective was to train a linear regression model to predict the outcomes of the test data most accurately. The metric to analyze the model's performance was the Sum of Squared Errors (SSE) which closely relates to the commonly used Mean Squared Error (MSE), differing only by a constant multiplication.

For this problem, we explored three models: **Linear Regression**, **Ridge Regression** and **Lasso Regression**. The main difference between them is whether or not the model is penalized for its weights, i.e, the β parameter calculation.

Both Ridge and Lasso have a regularization term, which penalizes the use of large coefficients, i.e, doesn't allow the model to place large weights on specific features. The λ parameter represents a trade-off between fitting the data (smaller λ) and keeping the model simple (larger λ). Ridge and Lasso aim to reduce the model's complexity and prevent overfitting which can occur as a result of simple linear regression. The difference between Ridge and Lasso is that the first uses L2 norm and the latter L1 norm. Put simply, the fundamental difference between the norms is that the L2 norm is proportional to the square of β values while the L1 norm is proportional to the absolute value of the values in β. In practice, a couple of important consequences of this difference are that the L1 norm is more robust than the L2 norm and that Lasso, by using the L1 norm, has inbuilt feature selection.

The models were trained using the provided train dataset, which was comprised of 15 samples across 10 features, a rather small training dataset. Additionally, by plotting a heat map, we found that some of the features were highly correlated. For both of these reasons, we hypothesized from the start that a model with a regularization technique was expected to perform better when encountering new data. Specifically, given the correlation between some features, Lasso regression could prove to perform better since it can give sparse weights, performing feature selection.

Linear regression was implemented through algebraic calculations using the normal equations (Equation 1).

$$\textbf{Parameter estimates: } \hat{\beta} = \left(X^T X\right)^{-1} X^T y \quad \textbf{Prediction: } \hat{y} = X\hat{\beta} \tag{1}$$

For both Ridge and Lasso, hyperparameter optimization was performed using sklearn's function *GridSearchCV()* which performs an exhaustive search over specified parameter values and outputs the hyperparameter which minimizes the score (which was MSE), the cross-validation strategy employed was Leave One Out.

It should be noted that to employ regularization methods, the data must be centered, which was not the case. When using functions *Ridge()* and *Lasso()* from sklearn, *fit_intercept* parameter was always set to True, so the function would automatically center the data when fitting.

For Ridge, the optimal λ found was approximately 2.47 (Figure 1, left). For Lasso, the optimal λ found was approximately 0.084 (Figure 1, right).

Additionally, to better understand the impact of the λ parameter, we plotted Ridge and Lasso's regularization paths (not included due to lack of space), which allowed us to see how the coefficient values changed for each λ value. For high λ values, the coefficients are driven towards 0, actually reaching 0 in the Lasso regularization path and generating a simpler model (with less impactful features). For low λ values, the

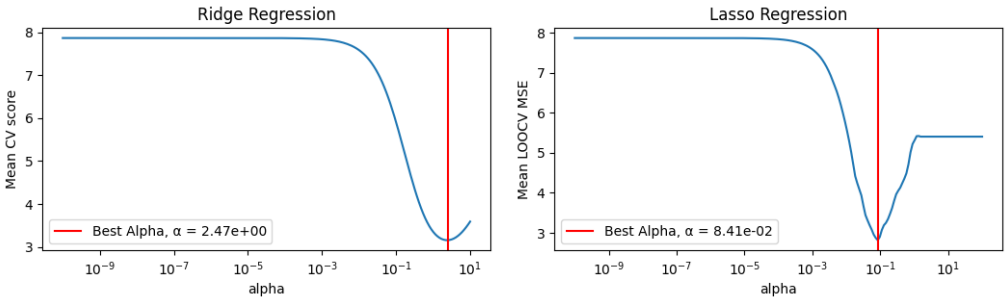

Figure 1: Hyperparameter estimation study for Ridge (left) and Lasso (right) Regression.

model becomes closer to a regular Linear Regression. The features corresponding to the slowest decaying coefficients can be interpreted as the most important ones.

Although we built our own Leave One Out function, for code simplicity and uniformity, we chose to use the sklearn function *cross_val_score()* function to evaluate each model with the Cross Validation strategy of *LeaveOneOut()*, since the training dataset was very small. The performance of each model is presented in Table 1.

Table 1: MSE for each model using Leave One Out Cross Validation.

	Linear Reg.	Ridge Reg.	Lasso Reg.
MSE	7.864	3.154	2.814

Given the MSE values for each model, we chose Lasso Regression as the model to be used. Finally, we trained the model using the whole train samples and the optimal λ value using the sklearn *Lasso()* function and generated the test predictions.

Our submission model got an SSE of 1821.94 (ranking 25^{th}). To better our submission results, we could've used Lasso solely for feature selection, i.e., remove the features with $\beta = 0$ and run that dataset through linear regression. This would probably result in the best solution since the dataset provided had 10 features but only 15 samples, suffering from the well-known "curse of dimensionality". Having more features than observations can lead to several problems such as: data sparsity, increased computation, overfitting, performance degradation and visualization challenges[1]. Correcting this problem by performing feature selection would've resulted in a better solution.

It is worth mentioning that, formally, when estimating hyperparameters and validating a model nested cross-validation should be used. In nested cross-validation, the model is validated using a test set (outer loop) that has never been seen, not even during hyperparameter tuning (which used an inner loop). This way, nested cross-validation provides a more rigorous and unbiased way to validate a model, often obtaining scores closer to reality. However, in this case, since the training set was rather small, we chose not to apply nested cross-validation since that would imply training the model with fewer samples during hyperparameter estimation and model validation.

1.2 Second Problem: Linear Regression with Two Models

The second problem is similar to the first one but the available data contains examples that are generated by two different probabilistic models. Fitting a single model to the data would result in poor results therefore the usage of two models was necessary.

First, we started by understanding the data at hand by calculating some metrics (Table 2) as well as visualizing the data (Figure 2)

From Table 2, we can see that the mean for each feature is relatively small (around the first and second decimal places) while the standard deviation surpasses 1, meaning a large variation in the data, highlighting

Table 2: Mean and standard deviation for each feature.

	Feature 1	Feature 2	Feature 3	Feature 4
Mean	-0.02	0.04	-0.14	-0.22
Standard Deviation	1.04	1.03	1.02	1.01

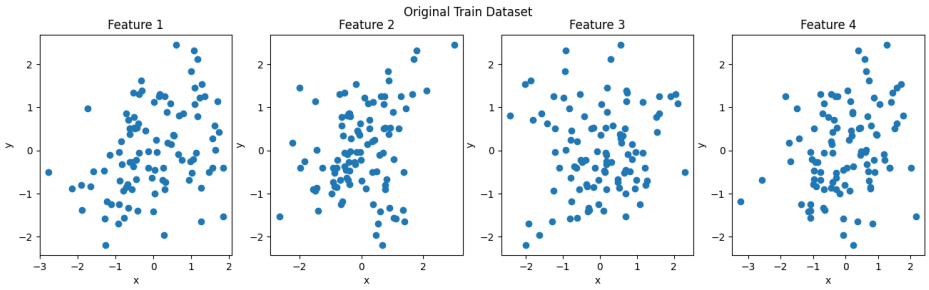

Figure 2: Scatter plot of the complete (normalized) training data.

the two different distributions. In Figure 2, especially in the plots for Features 2 and 3, we can clearly see two different distributions.

To separate the training data and build two separate models, we first attempted to use **K-Means Clustering** using the function *KMeans()* from sklearn. The KMeans algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares [2]. More simply put, 2 initial cluster centroids are selected using sampling based on an empirical probability distribution of the points' contribution to the overall inertia (greedy k-means), and then each data point is assigned to the closest centroid based on the Euclidean distance. After that the centroids are updated by calculating the mean of each cluster and the process is repeated until a maximum number of iterations is reached. The subsets resulting from clustering are plotted in Figure 3.

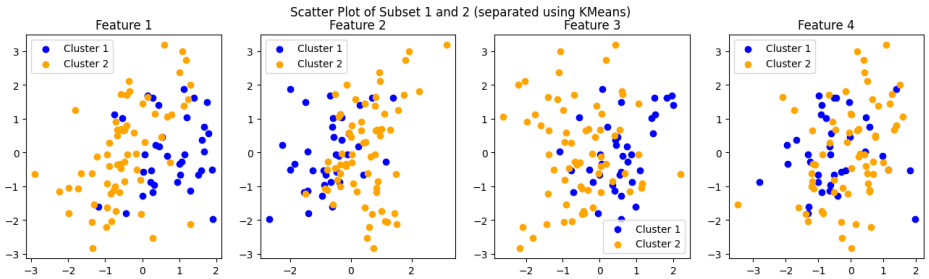

Figure 3: Scatter plot of the clusters created using KMeans (data non-normalized).

As we can see from Figure 3, the clusters don't appear to separate the two distributions accurately. Yet, we still tested Linear Regression and, after optimizing the hyperparameters, Lasso and Ridge Regression on each of the Clusters. None of these methods resulted in satisfying results. The resulting subsets from the K-means method are highly dependent on the distance metric used, which in this case was the Euclidean distance. The Euclidean distance is not appropriate to separate this dataset, as we can see from the scatter plots, often two points which are close to one another do not belong in the same subset.

After unsatisfying results, we explored other methods to separate the different distributions, namely **robust regression method**s capable of identifying outliers. Even though we're not specifically dealing with outliers, we can interpret one of the distributions as the outliers and the other as inliers.

The first method tested to divide the dataset was **Huber regression**. Huber regression is an L2-regularized regression which is robust to outliers, i.e, admits the existence of outliers and assigns less weight to them. The Huber Loss function identifies outliers by calculating each data point's residual and checking if it's above a certain threshold [3]. The two subsets resulting from the inliers and outliers identified by Huber regression are presented in Figure 4.

Although the division seems a bit better than the KMeans clusters, the first subset still presents some data points that seem to belong to the second subset. This is likely the reason why all the models had satisfying results for the second subset but never for the first, as it still had a couple of data points from two distributions. Once again, we tested Linear Regression, Ridge and Lasso. For the second subset, the optimal hyperparameter of each regularization technique was small enough that the model behaved like a normal Linear Regression. The results are presented in Table 3.

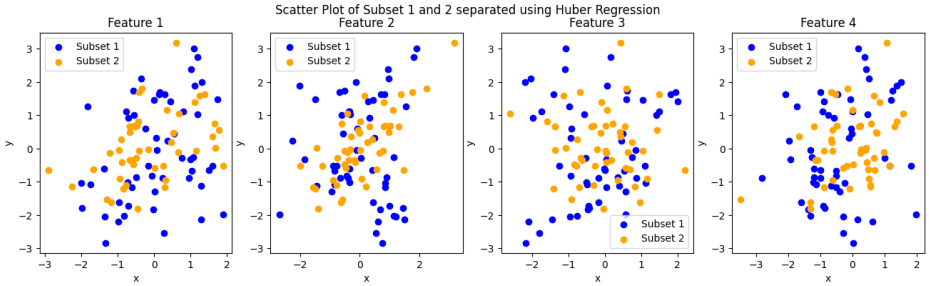

Figure 4: Scatter plot of the clusters created using Huber Regression (data non-normalized).

Table 3: MSE (from LOOCV) and R^2 (from 5-fold CV) for each of the subsets from Huber subsets.

	MSE		R^2	
	S1	S2	S1	S2
Linear Regression	2.414	0.032	-0.142	0.957
Ridge	2.287	0.032	0.097	0.078
Lasso	2.339	0.032	-0.068	0.957

Then, we tested the **Random Sample Consensus Algorithm**, also known as the RANSAC method. The RANSAC algorithm attempts to separate inliers from outliers and then considers only the inliers to fit the model. The steps in this iterative algorithm are the following [4]:

1. Select a minimum number of samples to build a model;
2. Fit a model to the random subset;
3. Classify each data point as inliers or outlier by calculating residuals and comparing to the threshold;
4. Save the fitted model as the best model if the number of inliers is maximal.

The subsets built from the inlier and outlier masks are plotted in Figure 5 (S1: 56 samples, S2: 44 samples). At first glance, the division seems more coherent, which was confirmed when the results from fitting Linear Regression models to each subset were satisfying. Additionally, Ridge and Lasso models were also tested. However, the optimal hyperparameters for the regularization techniques were small enough that the model behaved like a normal Linear Regression. The results are presented in Table 4.

Table 4: MSE (from LOOCV) and R^2 (from 5-fold CV) for each of the subsets from RANSAC subsets.

	MSE		R^2	
	S1	S2	S1	S2
Linear Regression	0.116	0.065	0.883	0.956
Ridge	0.116	0.065	0.077	0.078
Lasso	0.116	0.065	0.883	0.956

Given the results, we chose the Linear Regression models as the best, in Figure 5 we can also observe the different regression lines for each subset on each feature. As suspected, the regression lines for features 2 and 3 are the ones that differ the most.

Interestingly, although both Huber and RANSAC classify outliers using the residuals and a threshold, only RANSAC resulted in a good division. This could be due to the fact that Huber considers all the points (inliers and outliers) when fitting a model prior to calculating the first residuals, additionally, it receives an λ hyperparameter (since it's an L2 regularization technique) which was not optimized for the data. On the other hand, RANSAC uses only 2 data points at a time (minimum to build a Linear Regression line) and calculates the residuals based on this line, the iteration with the most inliers is the final one (or when a stop condition is reached, which in our case was when 50 inliers were reached which resulted in the best MSE).

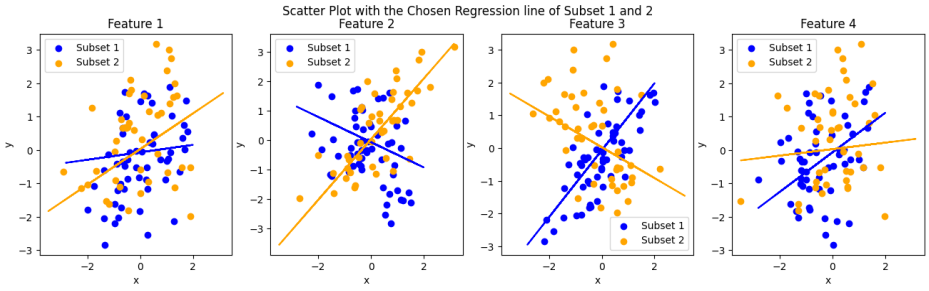

Figure 5: Scatter plot of the clusters created using RANSAC overlapped with the Regression lines from the chosen model.

To further test our results, we removed a sample from the training set, built the subsets once more using RANSAC and trained the Linear Regression models. Then, knowing to which subset the removed sample belonged to, we used both models to predict the y value and compared it to the real one. The model from the correct subset obtained the least squared error. Finally, to build the submission matrix, we classified the whole test set with each one of the regression models and stacked the vectors (1000×1) side by side to originate a 1000×2 matrix. Our submission model got an SSE of approximately 44.70 (scoring 5^{th}).

2 Part 2: Image Analysis

2.1 First Task

The first task consisted of the creation of a model to predict whether a dermoscopy image is from a melanoma (label 1) or a nevu (label 0). We started by pre-processing and visualizing the data. We were told the train data set was imbalanced, which we confirmed: out of 6254 train examples, only 896 were melanomas resulting in 14.32% of the train dataset. We used sklearn's function *train_test_split()* to get a validation set with 10% of the train data and we made sure the imbalance in the validation set was similar to the one observed in the whole train set, in order to get a validation set that accurately represented the reality. Given the imbalance, one of the most appropriate metrics to evaluate performance is Balanced Accuracy.

For balancing, the function *SMOTE()* from the imblearn package was used on the train data. SMOTE is an oversampling technique where the synthetic samples are generated for the minority class. The minority class is over-sampled by taking each minority class sample and introducing synthetic examples along a line which joins that sample to its k-nearest neighbours. The synthetic instances are generated as a convex combination of the two chosen samples [5]. In the present work, we applied *SMOTE()* from imblearn using the default parameters which over-sample all but the majority class (which in our case was the only minority class) with 5 nearest neighbours used to define the neighbourhood of samples to use to generate the synthetic samples. Using too many neighbours can result in abnormal images that do not mirror the reality. By checking the source code [6], we know that the synthetic samples are appended to the end of the array, so we plotted 50 random synthetic samples generated from *SMOTE()* to ensure that the images generated were within what was expected compared to the originals.

We started by testing some models from the sklearn package such as **Naive-Bayes**, **Support Vector Machines** (SVM) and **Multi-layer Perceptron** (MLP).

Naive Bayers classifiers are based on applying the Bayes' theorem which calculates the probability of a certain event based on prior knowledge while assuming conditional independence between every pair of features. For classification, the class with the highest posterior probability is chosen. We applied the *GaussianNB()* from sklearn and got a Balanced Accuracy of 0.722 after testing with the validation set.

Support Vector Machines classifiers find a hyperplane in an N-dimensional space that divides that data points into classes. The advantages of SVMs are their effectiveness in high-dimensional spaces, even when the number of dimensions is greater than the number of samples. Since it uses a subset of training points, the support vectors, it is also memory efficient. Additionally, by manipulating the Kernel functions, SVMs are very versatile classifiers. We applied the C-Support Vector Classification from the SVM class (*svm.SVC()*) from the sklearn package and got a Balanced Accuracy of 0.782 after testing with the validation set.

The Multi-layer Perceptron Classifier is a Neural Network with at least three layers (the input layer, the output layer and the hidden layer). The MLP is fully connected, so for large images, it results in too many parameters which can lead to redundancy and inefficiency and as a result, overfitting. We applied the *MLP-Classifier()* function from sklearn with the activation function ReLu and using the Adam solver and got a Balanced Accuracy of 0.756 after testing with the validation set.

The Balanced Accuracies for each classifier are presented in Table 5

Table 5: Balanced Accuracy for each Classifier from sklearn for the first task.

	Naive-Bayes	SVM	MLP
Balanced Accuracy	0.722	0.782	0.756

Directly using the classifiers from sklearn resulted in acceptable results, especially with SVM. However, the current favourite models for image classification are **Convolutional Neural Networks** (CNNs). CNNs can account for local connectivity, the weights are smaller and shared which results in a model more easily trained and more effective than MLPs. Moreover, since the layers are sparsely connected rather than fully, the model can go deeper without the disadvantages mentioned for MLPs.

To build CNNs we used the Keras package. Throughout the process of finding the best model, several parameters were changed, such as: the number of convolutional and dense layers, batch normalization layers, number of units in each layer, kernel regularization, optimizers when compiling the model, number of epochs, callback conditions, percentage of dropouts and addition of pre-processing layers (which performed data augmentation for example).

The best architecture is presented in Figure 6 (Left) and detailed in Table 6. The architecture consists of three sets of a Convolutional Layer (with ReLu as the activation function) followed by a MaxPooling layer and a Dropout (the first two of 20% and the last one of 40%). Each convolution layer had an increasing number of filters (32, 64, 128) the reason being that as the network goes deeper and we've extracted the initial information from the noise, we want to extract more complex details from the data. Dropout layers are a regularization method to avoid overfitting the model by nullifying the contribution of some neurons towards the next layer. Although during our research we found that dropouts are only usually applied after fully-connected layers (like a dense layer) we also found that they can improve the performance of the model if they're placed with low dropout rates after a Convolutional layers [7], which was what we observed in practice. Batch normalization was also tested after each Convolutional layer. When training a model with images, weights might become too large and produce feature maps with pixels spread across a wide range which could difficult the optimization process and lead to unstable gradients. In order to prevent this problem, we tested Batch Normalization layers after each Convolution layer. Although in the best model Batch Normalization did not improve its performance, in other architectures tested it did, highlighting its importance. Lastly, we flattened the data and added two Dense (i.e fully connected) layers, the first with 128 units and the last with only 1 and a sigmoid activation function to result in a number between 0 and 1, which would be interpreted as a probability, if it were above 0.5 the classifier would attribute class 1 (melanoma).

When compiling the models, several optimizers were tested for each architecture, for the best model SGD resulted in the best Balanced Accuracy. When fitting the model, we used early stopping to further prevent overfitting with a patience of 5 epochs while monitoring the loss, this means that if the loss did not improve for 5 epochs the model would stop. For the best model, we had a maximum number of 100 epochs and the callback stopped the model in epoch 92. In addition to the CNN layers, when adding a pre-processing layer that performed Data Augmentation (by randomly flipping and rotating the images) at the beginning of the model the Balanced Accuracy increased. Ultimately, this model resulted in a Balanced Accuracy of 0.806. In Figure 6, we can also see that although the validation loss had some spikes (which is likely due to the fact that we were only testing with 10% and had an unstable validation set), it tended to be close to the training loss which led us to believe the model was not overfitted.

Layer (type)	Output Shape
Sequential	(None, 28, 28, 3)
Conv2D	(None, 26, 26, 32)
MaxPool	(None, 13, 13, 32)
Dropout	(None, 13, 13, 32)
Conv2D	(None, 13, 13, 64)
MaxPool	(None, 7, 7, 64)
Dropout	(None, 7, 7, 64)
Conv2D	(None, 7, 7, 128)
MaxPool	(None, 4, 4, 128)
Dropout	(None, 4, 4, 128)
Flatten	(None, 2048)
Dense	(None, 128)
Dense	(None, 1)

Table 6: Best CNN architecture.

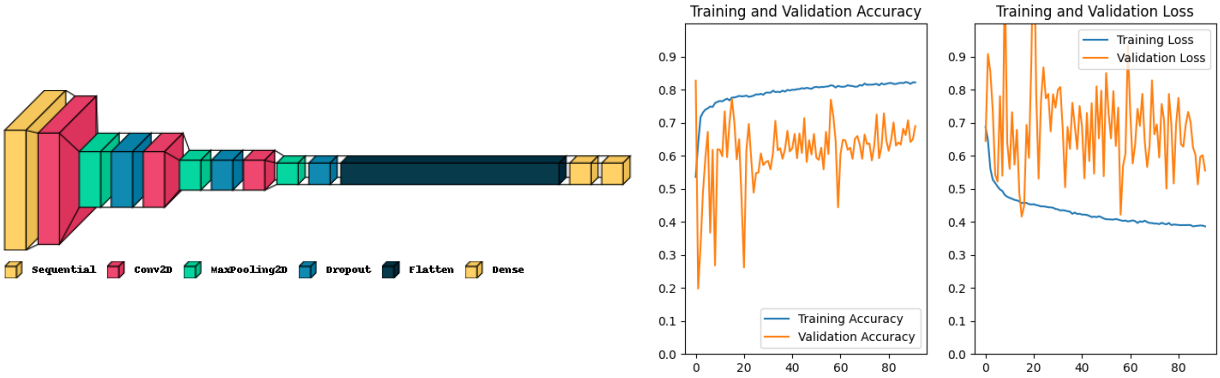

Figure 6: Left: Visual representation of the best CNN (pre-processing data augmentation layers not represented). Right: Training vs Validation Accuracy and Loss for the best CNN.

Our chosen model found 739 melanomas (41.8%) in the test dataset resulting in a Balanced Accuracy of 0.7919 (ranking 53^{th}) which is not too far off from the expected Balanced Accuracy of 0.806. Moreover, when testing with a dataset from another hospital, our model obtained a Balanced Accuracy 0.6781. This large decrease could be due to a number of reasons: the second hospital might treat different cases (more/less complex) that the model did not encounter during training or perhaps our model was too biased for the first hospital's cases and did not generalize well. Another hypothesis could be due to different images from each hospital, for example, if a lack of standardization is present the second hospital might photograph or process the images differently, leading to different features.

We encountered some difficulties which impaired our testing process. Although we defined a global seed to account for CNN's randomization, we were not aware that the Jupyter notebook cell with the global seed had to be run prior to running the other cells each time. So the results we obtained were after initializing, compiling and fitting the model a couple of times which meant Jupyter was no longer using the seed we had introduced and we couldn't reproduce the results. To work around this, every time we got an improvement in the Balanced Accuracy of the model we saved the model as a *.keras* file which allowed us to later import it if we were not able to reproduce it again since we didn't know which seed had been used.

One thing we could've improved would be to also have included random shifts in the data augmentation. By analyzing the train set images, the nevus/melanomas were sometimes more to the left or more to the right. Although flipping and rotations proved to be helpful, shifting could have also made sense. Moreover, we would have liked to be more careful with the seeds issue raised above in order to have made the testing

process easier and more reproducible.

2.2 Second Task

The second task also consisted of the classification of 2D medical images. However, this time the images have six possible classes and come from two different datasets (a dermoscopy and blood cell microscopy).

To begin, we visualized the data and concluded that the images from the two datasets were very distinguishable. Since the dataset was imbalanced we used the SMOTE technique once more to generate synthetic images of the minority classes. To make sure the synthetic images were within of what was expected we visualized them as well.

Once more, to establish baseline values, we tested **some classifiers from sklearn**. The results are presented in Table 7.

Table 7: Balanced Accuracy for each Classifier from sklearn for the second task.

	Naive-Bayes	SVM	MLP
Balanced Accuracy	0.643	0.793	0.745

As an initial approach, we applied the **best model from the previous exercise** (Figure 6). Since this problem consisted of a multi-class classification, some changes had to be made such as switching the number of neurons in the last Dense layer from 1 to 6 and replacing the sigmoid activation function with a softmax activation function. The softmax function outputs six values which can be interpreted as the probability of the input belonging to that class. The loss function was also altered to the Categorical Cross Entropy since the Binary loss was no longer applicable. This model resulted in a Balanced Accuracy of 0.885.

However, since we're dealing with a multi-class classification problem, interpreting the Balanced Accuracy is no longer so simple. To get a better idea of how the model was behaving, we calculated the recall for each one of the six classes. The recall tells us how good the model is at correctly identifying positive instances from all the positive instances in each class. A model with a great Balanced Accuracy can perform outstandingly in some classes and very poorly in others.

The recall values for each class for the model from the previous task are presented in Table 8. We can see that the model performs very well for the last three classes, less well for the first three and has a particularly low recall for the first class. This tells us that although the model achieved a good Balanced Accuracy value, it is not exceptionally good at classifying instances from the first dataset and especially struggles to correctly classify instances from the first class.

Table 8: Recall values for each class using the model from the first task. In yellow are the classes from the dermoscopy dataset, in blue are the classes from the blood cell microscopy dataset.

	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5
Recall	0.710	0.896	0.857	0.924	0.978	0.948

To better our results, we thought of strategies to use the dataset label. Two different models were built using the dataset label: a **hierarchical model** comprised of three CNNs and a **multi-input CNN**.

The hierarchical model started with a CNN trained with the whole dataset with the goal of predicting the dataset label. The architecture used was the same as in the first task, with only one neuron in the last Dense layer with a sigmoid function and binary cross entropy as the loss function. This CNN predicted the dataset label with a Balanced Accuracy of 1 which was somewhat expected since the images are very different. This also assured us that most likely no error would be propagated to the next CNNs. After this, we divided the train and validation datasets according to the dataset they belonged to, with the objective of training two different CNNs each with only one dataset and predicting between 3 classes. We hypothesized that this way the models would be able to focus on specific features of each dataset and better classify the images.

First, we built both CNNs with the same architecture, the one used in the first task. However some parameters were then tweaked in the CNN for the dermoscopy dataset in order to obtain better results (for example, dropout layers between convolution layers were replaced with batch normalization layers and a dropout was placed before the last dense layer). The CNN that predicted the classes in the dermoscopy dataset presented a Balanced Accuracy of 0.838 and the CNN that predicted the class in the blood cell microscopy dataset had a Balanced Accuracy of 0.954. These results highlight once more that the first dataset is harder to predict than the second. Graphs showing the training and validation loss and accuracy for each model are presented in Figure 8. Ultimately, each sample goes through the first CNN and based on its output goes through one of the other CNNs (Figure 7). The outputs are then assembled in the same vector with integer values from 0-5.

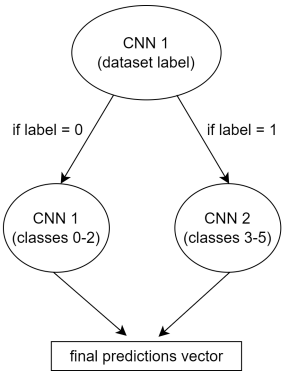

Figure 7: Hierarchical model.

This hierarchical model resulted in a Balanced Accuracy of 0.896. The recall values for each class are presented in Table 9. In this case, we can see that the recall values for the first dataset, which had proved to be problematic, are much more balanced than before, confirming our hypothesis that the separate model was able to better learn the features of the dermoscopy dataset.

Table 9: Recall values for each class using the hierarchical model. In yellow are the classes from the dermoscopy dataset, in blue are the classes from the blood cell microscopy dataset.

	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5
Recall	0.826	0.831	0.857	0.969	0.967	0.927

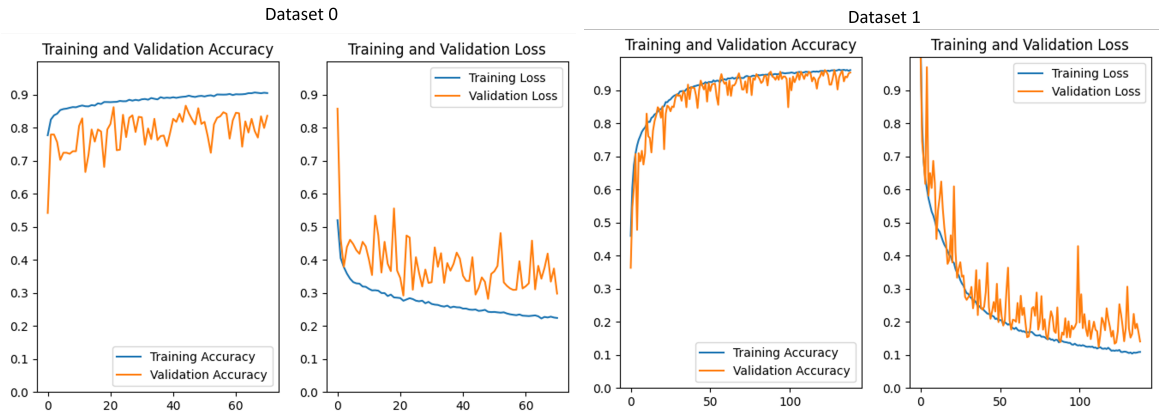

Figure 8: Training vs Validation Accuracy and Loss for the CNN that predicted classes from the dermoscopy dataset and the blood cell microscopy dataset, respectively.

The other approach consisted of a multi-input CNN. This CNN received the training array (with both datasets) and extracted its features, then, after flattening, the label predicted from the dataset prediction model would be added to the flattened vector (Figure 9). This way, the model would gather information from the images altogether while taking into consideration which dataset they belonged to. The architecture used for the CNN was the same as in the first task, however, when using the keras package, multi-input CNNs are built with the Functional API. For this reason, the first layer which consisted of a pre-processing data augmentation layer (implemented using the keras Sequential API) was not implemented in this approach. This likely explains the lower Balanced Accuracy of 0.809. Due to a lack of time and having already obtained satisfying results, we did not implement another way of data augmentation compatible with the Functional API.

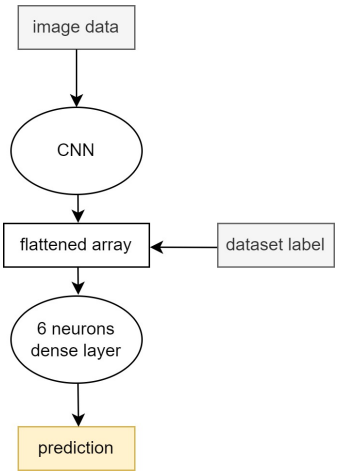

Figure 9: Multi-input model. Inputs in grey, outputs in yellow.

The Balanced Accuracies from each approach are summarized in Table 10.

Table 10: Balanced Accuracy for each model.

	Model from First Task	Hierarchical Model	Multi-input Model
Balanced Accuracy	0.885	0.896	0.809

Given these results and the recall's impact, we applied the hierarchical model to predict the test set classes. The percentage distribution of each class in the predicted test labels was close to the one observed in the train set and a Balanced Accuracy of 0.871 was obtained (ranking 26^{th}), relatively close to what was expected. To better our results, similarly to the first task, we could've used shifting for data augmentation and tried to implement data augmentation for the multi-input model to check if the results improved.

References

[1] Data Camp. The Curse of Dimensionality in Machine Learning: Challenges, Impacts, and Solutions. `https://www.datacamp.com/blog/curse-of-dimensionality-machine-learning`. [Accessed 28-10-2023].

[2] K-means clustering – scikit-learn. `https://scikit-learn.org/stable/modules/clustering.html#k-means`. [Accessed 7-10-2023].

[3] Huber Regression – scikit-learn. `https://scikit-learn.org/stable/modules/linear_model.html#huber-regression`. [Accessed 7-10-2023].

[4] RANSAC: RANdom SAmple Consensus – scikit-learn. `https://scikit-learn.org/stable/modules/linear_model.html#ransac-regression`. [Accessed 28-10-2023].

[5] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, June 2002.

[6] Imblearn. SMOTE source code. `https://github.com/scikit-learn-contrib/imbalanced-learn/blob/20ba698/imblearn/over_sampling/_smote/base.py#L221`. [Accessed 20-10-2023].

[7] BlogAI. Applying Dropout in Convolutional Neural Nets: Where and to what extent? `https://nchlis.github.io/2017_08_10/page.html`. [Accessed 20-10-2023].