Final Report
Machine Learning

Instituto Superior Técnico
Group 82
Miguel Lopes | nº93406   Ricardo Silva | nº104077

Regression Task 1

In this first assignment we were given a training set with $n = 100$ samples with 10 features and a test set. We were asked to train predictors $f(x)$ and submit a vector of outcomes $\hat{y} = f(x_{\text{test}})$.

The format of the train data is the following: $\mathcal{T}_{\text{train}} = \{(x^{(1)}, y^{(1)}), \cdots, (x^{(100)}, y^{(100)})\}$, with feature vector $x^{(i)} \in \mathbb{R}^{10}$ and outcome vector $y^{(i)} \in \mathbb{R}$, with $n = 100$ samples. The only difference in the test data is the size of the set, $n' = 1000$ examples. In the code, we imported the data into the `X_import` and `Y_import` arrays, and the test data in the end as `x_eval_import`

1.1. Method to choose the model

In order to train the model, we experimented with various methods. Firstly, we trained just a linear model with all the data provided, with a simple linear model. This resulted in a quite small mean squared error (MSE) leading us to believe that it was already very good.

However, we immediately realized that we wouldn't be able to reliably quantify the model's performance because it may have a greater MSE when evaluating the training set owing to overfitting. Given this, we proceeded to split provided train data into four arrays (`X_train`, `X_test`, `Y_Train`, `Y_test`) using the function `train_test_split()` with the `test_size = 0.2`, meaning that 20% of the imported data was used to the validation set and 80% to the training set. Thus, this was done to later train the models and have a value of SSE/MSE for a validation set to compare between the various models.

It is important to highlight the fact that throughout the code, every time we mention "`_test`", we are mentioning instead the validation set that is part of the train data (we should have called "`_val`").

We also introduced (with exception of the Linear model), data normalization which can be done with `StandardScaler()`. However, for some reason the results were worse with this so we decided not to deliver with this.

Taking this into account, we started by training using three predictive linear models: linear regression, Ridge regression and Lasso regression. These 3 methods are all linear regression with a parameter β. Ridge and Lasso, with the exception of the linear model, need regularization, as discussed in the theoretical lectures. The parameter are then given by:

$$\text{Linear regression:} \quad \hat{\beta}_{\text{linear}} = \arg\min_{\beta} \|y - X\beta\|_2^2$$

$$\text{Ridge regression:} \quad \hat{\beta}_{\text{ridge}} = \arg\min_{\beta} \|y - X\beta\|_2^2 + \lambda\|\beta\|_2^2$$

$$\text{Lasso regression:} \quad \hat{\beta}_{\text{lasso}} = \arg\min_{\beta} \|y - X\beta\|_2^2 + \lambda\|\beta\|_1$$

The commands that we used to implement the methods are from the `scikit-learn` library if estimators: `LinearRegression()`, `linear_model.Ridge()` and `linear_model.Lasso()`.

Regularization

The purpose of the regularization is to limit the coefficient estimation towards zero. This decreases the likelihood of overfitting by discouraging the training of overly complicated models.

Ridge regression operates by changing the cost function and introducing a penalty $(\lambda\|\beta\|_2^2)$, with the goal of preventing large coefficients.

Similar principles apply to the Lasso technique, although there is an additional penalty of $(\lambda\|\beta\|_1)$. As a result, there are significant discrepancies between the two models, which will be seen when the various models are trained and cross-validated.

We used the `GridSearchCV` function of the `scikit-learn` Python package to determine the best value of λ for each of the two models. By increasing it by a certain amount, this approach enables us to cycle through various values of λ between an upper and a lower limit. It does cross validation for each value (which is better covered in a later section). The λ that best fits the model is the one with the lowest mean squared error. The lambda that corresponds to the model with the lowest mean squared error will be the optimal one.

Cross Validation

As we mentioned before, the training set underwent Repeated K-fold Cross Validation to lower the possibility of overfitting caused by the selection of an unsuitable model. The typical K-fold Cross Validation procedure is repeated n times with various splits for each iteration in this kind of cross validation. This approach was selected because it is likely to be more accurate than doing a single K-fold Cross Validation while being less computationally costly or susceptible to variability than the Leave One Out or Leave P Out approaches. The data set was divided into 5 folds (k=5) and performed 5 iterations of the Repeated K-fold CV. A greater value of k may have resulted in possibly an inadequate number of points being set aside for validation, whereas a lower value might have resulted in a reduction of data being utilized for the training. Furthermore, the decision to limit the number of repeats to 5 was taken since more repetitions had little influence on the outcomes whilst adding a greater computing strain.

This technique was put into practice using the `RepeatedKFold` function from the `scikit-learn` package. It is to be noticed that, all these approaches calculates both the MSE and SSE for each test, which are correlated, there when one is minimum the other is also minimum.

1.2. Results

After this, in this section, we will present a short summary of the results obtained for each model. For the model linear progression, with no hyperparameters (λ), we simply performed the standard cross validation, where we used the `X_train[i]`, `Y_Train[i]` to train the model, and `X_test[i]`, `Y_test[i]` to evaluate and obtain the associated MSE and SSE

We tested the polynomial model before moving on to the models with regularization on the results, but we quickly saw that the MSE value increased dramatically with higher order polynomials, so we rejected them (the image that shows this increase in showed in the jupyter notebook).

When dealing with the Ridge and Lasso regressions, as we mention in the cross validation, we started to use the `GridSearchCV` to search for the best hyperparameters, including the λ (`alpha` in the code) and the `solver` used to minimize (which was later discarded since it increase the computation border without improving significantly the model).

We passed a list of with a wide range of values, using `np.linspace(start=0.001, stop=0.1, num=100)`, in order to estimate the value that best fits for each one. Following that, we applied the same cross validation technique to each model and compared the outcomes (MSE and SSE).

The library `sklearn` also has the Ridge and Lasso models with an automatic implementation of cross-validation called `RidgeCV` and `LassoCV` which ran much faster without influencing the results. The results that we obtained are in table 1.

Model	Nº Validation	Best λ	Average MSE
Linear	1000	-	0.01274817157
Ridge	1000	0.057656	0.01274787344
Lasso	1000	0.003139	0.01278741677

Table 1: Results of regression task 1 with corresponding average λ given by GridSearchCV() and the average standard deviation of the validation set over the nº of validations.

The MSE and SSE for the various approaches are all relatively close, according to the results of the cross validation processes, with Ridge regression having the lowest MSE. During numerous tests, these numbers fluctuated quite a deal, but the Ridge model was always marginally superior, so we chose it.

1.3. Conclusion

After choosing the Ridge with $\lambda = 0.05765$ as the best model to predict the test set, we used the entire provided dataset (instead of splitting the data into training and validation set), in order to compute the β vector that best fits a certain data `(X,Y)`. Given the model, we implemented it, and using the `.predict(x_eval_import)` function we obtain the the predicted values $\hat{y}$ and save in the file `predictions_t1.npy`.

Improvements:

After submitting, we noticed that we could have improved the model by when selecting the λ, we could had restricted to a certain range to a point where the iteration step was reasonably small, instead of leaving a large interval (specially, we could reduced the search interval after choosing the best model, in order to reach a more exact value).

Furthermore, in light of the fact that Lasso can zero out coefficients, thus removing them from the model, whereas ridge can only scale them down, we later deduce that Lasso could had been a more accurate model. Effectively, Lasso may be more efficient when it comes to minimizing overfitting risk, since it can set certain coefficients to zero.

Regression Task 2

In this second assignment, it was once again given a training set with $n = 100$ samples and 10 features. However, this time with an unknown proportion (less or equal than 20%) of outliers and a test set, we were asked to search for a method to identify and remove those outliers and a method for train predictors $f(x)$ and submit a vector of $\hat{y} = f(x_{\text{test}})$.

The format of the training data is the following $\mathcal{T}_{\text{train}} = \left\{\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(n)}, y^{(n)}\right)\right\}$, with a feature vector $x^{(i)} \in \mathbb{R}^{10}$ and outcome vector $y^{(i)} \in \mathbb{R}$, with $n = 100$ samples including the the outliers. And the test data was a different size of $n' = 1000$ examples. In the code, we imported the data into the `X_import` and `Y_import` arrays, and the test data in the end as `X_TEST`.

2.1. Choosing the method for removing outliers

In order to train the model, we need first to remove some outliers and there is several ways to remove those. Some methods like deleting the point with the largest standard error like Cook's Distance, RANSAC, Huber Theil and DDFITS (which is very similar to Cook's Distance). Since these methods are very different we decided to implement all of the and compare the indexes that each one identifies and evaluate the results.

In fact, The Local Outlier Factor, Isolation Forest were the first approaches considered. However, we quickly discovered and were advised that employing any of these three strategies would result in very slight gains, due to the fact they only take into account the density of points in x when determining whether or not a point is an outlier. Therefore weren't the most appropriate for outlier identification when the data is distributed like in this case.

2.1.1 Cyclical removal of points with the highest standard error

The mean standard error for a single point is known just as standard error, this method is relatively basic approach for eliminating outliers in which the point with highest error would indicate that its the most likely to be an outlier, giving by following expression 1.

$$e_i^2 = (\hat{y}_i - y_i)^2 \tag{1}$$

We proceeded to create two methods of when to stop the removal, which consist on: first calculating the mean and the standard deviation (σ) of the squared errors, and considered any value over 3σ away to be a potential outlier; the other consisted on creating a loop after fitting the current points with a typical linear regression and deleting the one with the highest error. This cycle is repeated until we reach a threshold of the maximum 20 total outliers (20%).

This method is ideal when the outliers are far apart from the inliers, as they will almost certainly have the biggest errors.

2.1.2 Cyclical removal of points with the highest influence given by Cook's Distance

Cook's distance is another regression analysis approach for identifying significant outliers in a set of predictor variables. We may use this to identify points that impact the regression model and provide poor results. This technique calculates influence as a mix of observation leverage and residual values when performing a regression provided by the equation 2.

$$D_i = \frac{\sum_{j=1}^{n}(\hat{y}_j - \hat{y}_{j(i)})^2}{p s_i^2} \tag{2}$$

where $\hat{y}_{j(i)}$ and $\hat{y}_j$ corresponds to the predicted values of the model trained without and with the point i, $p = 10$ is the number of features per observation and s_i^2 the mean squared error (MSE).

Just like before we have implemented two methods: eliminating all points with a D score above a certain threshold, commonly used as $I = 4/n$; the other on creating a loop were we compute the influence for each point, then eliminate the point with the highest impact until we hit the threshold or the maximum total number of 20 outliers.

2.1.3 Removing of outliers using RANSAC algorithm

RANSAC, which stands for random sample consensus, is an iterative approach for estimating parameters of a mathematical model using a collection of observed data that includes outliers. This approach employs repeated random sub-sampling on the assumption that training comprises both inliers and outliers.

2.1.4 Cyclical removal of points with the highest by Difference in Fits (DDFITS)

This method of removing the points with highest DDFITS it resembles to Cook's Distance however the formula for each point is given by the equation 3:

$$DDFITS_i = \frac{\hat{y}_i - \hat{y}_{(i)}}{s_{(i)}\sqrt{h_{ii}}} \tag{3}$$

and with the threshold at which the points are removed different and given by: $I_{(\text{DDFITS})} = 2\sqrt{\frac{k+2}{n-k-2}}$

2.2. Choosing the model

The procedure for choosing the model here is very similar to Regression Task 1, we spitted the data, after removing the outliers, in `(X_train,X_test,Y_train,Y_test)` with `train_test_split()` with a `test_size=0.2`, thus 20% of the inline training data was used as validation data and 80% to train the set. This was done again to later compare the MSE values between the predictions and validation set and evaluate the best model.

Similarly, we attempted to normalize the data for Ridge and Lasso using `StandardScaler()` to center the data around 0 to increase model performance and convergence time, but it produced poorer results, so we opted not to use it.

We also implemented regularization to find the best parameters for both models and with `RidgeCV()` and `LassoCV()` with the same reason as before.

2.3. Results analysis and conclusions

The results are as follows in the table 2.

Model	**Threshold**	**Outliers**	**MSE**		
			Linear	Ridge	Lasso
No method	None	0	7.561638	7.550554	9.750298
Highest SE removal	$mean \pm 3\sigma$	18	0.634133	0.634055	0.630259
Cyclical highest SE removal	$mean \pm 4.5\sigma$	20	0.011838	0.011838	0.011837
Cooks Distance	1/4n	7	2.500083	2.499542	2.482515
Cyclical Cooks Distance	1/4n	20	0.011877	0.011870	0.011868
RANSAC	None	19	0.032138	0.032135	0.032124
Cyclical DDFITS	$2\sqrt{\frac{k+2}{n-k-2}}$	20	0.011599	0.011599	0.011598

Table 2: There is the mean squared error for each model using the best alpha offered by cross validation in both ridge and lasso regression models, as well as the threshold utilized and number of outliers removed in each method.

The results for mean standard error of each model were given by same procedures as in the Regression Task 1.

As we can see, cyclical removal of highest standard error, cyclical Cook's distance and DFFITS method for removing outliers yielded the greatest results; nevertheless, the differences are only circumstantial because the outliers discovered by both methods were the same. With certainty that both models recognized the same outliers, we eliminated the outliers from the training data and proceeded to model selection. The outlier indexes that were identified in the final were:

$$15, 18, 24, 29, 30, 33, 34, 36, 47, 48, 62, 63, 65, 70, 71, 72, 83, 88, 93, 95$$

We choose to deliver the Lasso Model with a $\lambda = 0.004565$ which results in a final SSE$=$ 13.13089191, that is a very good results but there is a lot of margin for improvement. It was confirmed in class that the outliers we deleted were correct, implying that the loss was in selecting the model; the Lasso model was consistently the best, implying that our λ wasn't likely the best. `LassoCV()` takes as input a list of potential values and attempts to find the best value by iterating over numerous lists with finer and finer values until a good precision on the value is obtained.

Classification Task 1

I this third task we were given a training set with $n = 8273$ images, each image has 30x30 pixels and each pixel has the RGB values (2700 columns = 30x30x3). The training data has images of butterfly wings with two types of wing patterns: spots and eye-spots for this homework. The purpose was to categorize these two types of patterns as 0 (spots) or 1 (eye-spots). Although the data is unbalanced, the metric used to evaluate the model is the F1 score, given by:

$$F_1 = \frac{2}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}} \quad \text{where} \quad \text{Recall} = \frac{\#\text{ True Positives}}{\#\text{Predicted Positives}} \quad \text{Precision } = \frac{\#\text{ True Positives}}{\#\text{ Positives}}$$

Once again, we started by dividing the data set into training and validation sets, with a splitting ratio of 0.8 to 0.2 correspondingly.

3.1. Method for choosing the model

There are several strategies for models to deal with this kind of classification tasks (receive images as inputs and predict a label based on their features) like: Convolution Neural Network (CNN), Multi-layer Perceptron (MLP), Random Forest (RF), Decision Tree (DT), K Nearest Neighbors (KNN), Gaussian Naive Bayes (GNB) and Support Vector Classification (SVM). In our case we decided to implement all of them and compare the results, which are present in the table 4.

3.2. Convolution Neural Network (CNN)

Convolution neural networks are a type of artificial neural network that is commonly used for image analysis. They are regularized versions of multi-layer perceptrons because they use convolution instead of general matrix multiplication in at least one of their layers. They are made up of three layers: an input layer, hidden layers, and an output layer. These hidden layers in CNN include convolution layers.

3.2.1 Convolution Neural Network (CNN) layers

Convolution Neural Networks can have several kind of layers, like 2D Convolution Layers, 2D Max Polling, Dropout, Batch Normalization, Flatten and Dense. We'll now go through the objective of each layer and some of the benefits of employing them.

2D Convolution Layers:

At its core, 2D convolution is a rather straightforward operation: you begin with a kernel, which is essentially a tiny matrix of weights. This kernel "slides" through the 2D input data, doing element-wise multiplication with the portion of the input it is presently on and adding the results into a single output pixel.

The kernel continues this procedure for each spot it glides over, transforming a 2D feature matrix into yet another 2D feature matrix. The output features are simply weighted sums (with the weights being the kernel's values) of the input features situated nearly in the same place on the input layer as the output pixel.

2D Max Polling Layers

If Max Pooling is not used to downscale the image and replace it with Convolution to extract the most important features, it will incur a high computational cost. So we use Max Pooling to extract the most value from the Feature map based on filter size and strides.

In order to downsize an image that is very large, we must use Max Pooling, saving only pixels with the highest value. These values in the Feature map show how important a feature is and where it is located. Taking only the maximum value, then, means extracting the most important feature in a region.

Dropout Layers

In machine learning, "dropout" refers to the practice of disregarding specific nodes in a layer at random during training (set input units to zero). It is used as a regularization strategy to prevent over-fitting by guaranteeing that no units are codependent on one another. When we apply dropout to a neural network, we create a "thinned" network in which unique combinations of the units in the hidden layers are discarded at random during training. We construct a new thinning neural network with various units discarded depending on a probability hyperparameter p each time the gradient of our model is changed.

Training a network with dropout may thus be considered as training loads of various thinned neural networks and combining them into a single network that takes up the important attributes of each thinned network.

Batch Normalization Layers

Batch Normalization is the process of normalizing the activation values of hidden units so that the distribution of these activations remains constant during training. If the hidden activation distribution changes during training of any deep neural network due to changes in the weights and bias values at that layer, they trigger fast changes in the layer above it.

Flatten Layers

Flattening is the process of converting a multi-dimensional Pooled Feature map to a one-dimensional vector. This is significant because we intend to input this Pooled Feature map into a Neural Network, and Neural Networks can only accept One Dimensional input.

Dense Layers

The dense layer is a densely linked neural network layer, which implies that each node in the dense layer receives input from all nodes in the preceding layer.

The dense layer conducts matrix-vector multiplication in the background. The matrix values are really parameters that may be learned and changed via backpropagation. The dense layer produces an 'm' dimensional vector as output. As a result, the dense layer is mostly employed to alter the vector's dimensions.

3.3. Final Arquitecture

In this task the performance of the CNN was measured using F1 score, after several trials the architecture settled was the table 3.

Layer 1	Conv2D(32)	**Layer 8**	Batch Norm.	**Layer 15**	Batch Norm.
Layer 2	MaxPolling2D	**Layer 9**	Conv2D(32)	**Layer 16**	Dense(64)
Layer 3	Dropout(0.2)	**Layer 10**	Dropout(0.2)	**Layer 17**	Dropout(0.2)
Layer 4	Batch Norm.	**Layer 11**	Batch Norm.	**Layer 18**	Dense(1)
Layer 5	Conv2D(64)	**Layer 12**	Flatten	**Layer 19**	Batch Norm.
Layer 6	MaxPolling2D	**Layer 13**	Dense(256)	**Layer 20**	Dense(1, Sigmoid)
Layer 7	Dropout(0.2)	**Layer 14**	Dropout		

Table 3: Convolution neural network architecture used in classification task 1.

This architecture was created using a mix of research and adaption of popular models such as Alex-Net, VGG, and LeNet. We arrived at the table arrangement by repeating a normal configuration of 2D Convolution, 2D MaxPolling, Dropout, and eventually Batch Normalization, followed by a Flatten after some of these cycles. Adding extra layers at this time was significantly increasing our computation cost with negligible gain. As is typical for current neural networks, the *ReLU* function was utilized as the activation function. Because we were dealing with binary classification, the final activation function layer was a `sigmoid` rather than a typical `softmax`.

The data was also **normalized** before any of the other layers, by dividing it's values by 255, so that all values fit into a range of zero to one.

The design was then constructed using an Adam optimizer with a `1E-3` learning rate and a loss parameter called "binary crossentropy." These values were determined by trial and error utilizing all of the Keras options. In order to speed up learning and converge quicker, the data comprised of the RGB channel values of each pixel in the picture.

On top of this, we added **data augmentation** such as rotation and modest degrees of zoom to certain training data in order to try to deal with the unbalanced of the data (this data augmentation is only applied to the training set and not).

Overfiting: To reduce the overfiting, not only was done data augmentation, but it was done the training using a validation set were it was determine the optimal amount of epochs to then train the full set of data. An early stopping criterion was used to determine the exact epoch at which the best validation loss had been obtained, and terminate training after no improvement in the validation loss for 50 epochs. After that, predictions were generated using the most accurate model, which was stored to a `.h5` file using the model `checkpoint function` instead of the model acquired after the final epoch.

3.4. Results analysis and conclusions

Model	F1 Score	Accuracy	Model	F1 Score	Accuracy
SVM (kernel = linear)	0.58076	0.70514	KNN	0.65266	0.75728
SVM (kernel = poly)	0.65477	0.76616	Decision Tree	0.61428	0.70736
SVM (kernel=rbf)	0.73120	0.81208	GNB	0.56031	0.61839
Random Forest	0.67587	0.71092	CNN	0.87060	0.90430

Table 4: Results of classification task 2 with the methods implemented

The images obtain for plotting the epoch versus score is on the figure 1.

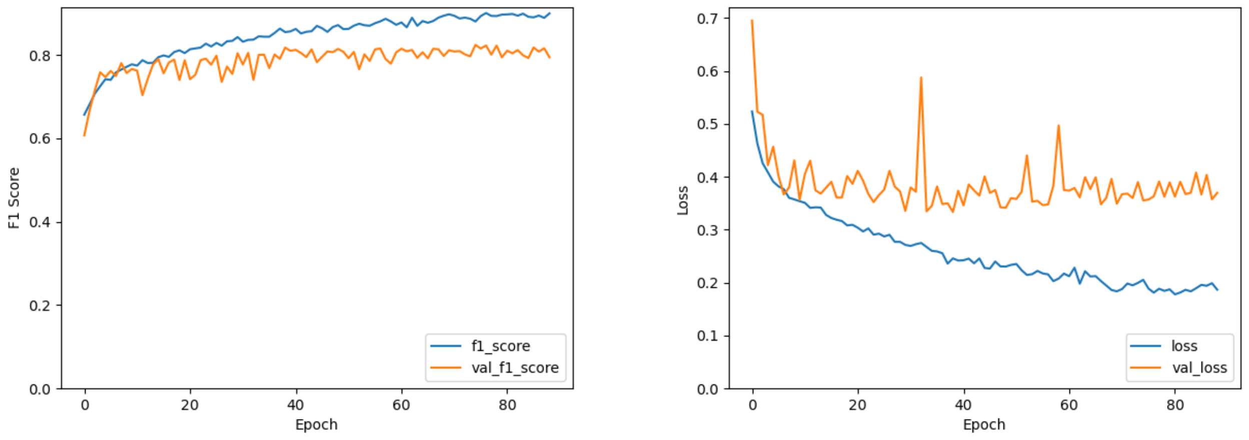

Figure 1: Evolution of F_1 score / validation F_1 score and Loss /validation loss with the nº of epochs

After several experiments, it was decided that 30 proved to be the ideal number of epochs, and this value was used to train the complete dataset. A validation set and any early stopping were not used during the final training because they had already helped to establish the ideal number of training epochs. It's worth noting that the increasing loss values for the validation set, in contrast with the F1-score, that reaches a stagnation value after around 30 epochs.

After submitting the results, the professors assigned an F1 Score of 0.919925512, indicating that there was some room for growth. For one thing, the model's architecture could probably be improved further by changing some of the layers; additionally, this model is very susceptible to statistical fluctuations that can influence the final results; a better approach would be to take an average of multiple runs of the same model to improve the confidence of more difficult images to classify. Furthermore, to deal with the imbalanced data one could also add when fitting the mode a parameter `class_weights` adding the `class_weight = 'balanced'` from the `scikit-learn` package

Classification Task 2

In this fourth task, we were given a training set of $n = 50700$ pictures, each having 5x5 pixels and the RGB values (75 columns - 5x5x3). These 5x5 photos are intern patches of 30x30 eye-spot images produced with the `extract_patches_2d()` method from the `skimage` toolbox. The aim is to recognize the patch as belonging to one of three unique areas: the white ring (0), the black and gold ring combined (1), and the backdrop (2). The categorization is just the classification of the patch's center pixel (2,2).

Similarly to the classification task 1 there are several method to deal with this problem like: Convolution Neural Network (CNN), Multi-layer Perceptron (MLP), Random Forest (RF), Decision Tree (DT), K Nearest Neighbors (KNN), Gaussian Naive Bayes (GNB) and Support Vector Classification (SVM). In our case we decided to implement all of them and compare the results. We uncovered a library called "imblearn" that implements portions of the sklearn libraries particularly for imbalanced data, and we identified two primary methods named "Balanced Bagging" and "Balanced Random Forest," as well as implementations of Over-sampling and Under-sampling our set. The performance of a model is calculated by it's balances accuracy given by equation 4

$$\text{BACC} = \frac{1}{2}\left(\frac{\text{Total True Positives}}{\text{Total Positives}} + \frac{\text{Total True Negatives}}{\text{Total Negatives}}\right) \tag{4}$$

4.1. Choosing the model

After executing all of the models outlined above and regularizing the hyper-parameters, we decided on the SVM model since it produced the best results in terms of balanced accuracy.

The **CNN architecture** used in this case was similar to the one used in the previous task, however in this case not only the data is highly imbalanced, the number of inputs is relatively small thus not being very feasible to perform `Conv2D` and `2DMaxPolling`; To try to solve this last issue, we attempted to implement the U-net architecture, but because our data was 26x26 (due to the way the function `extract_from_patches()` works, there will be a 2 pixel wide border, thus not possible to classify), we had problems implementing it with the original architecture (due to `2DMaxPolling()` when our split reached an uneven number; with some changes to the architecture, we were able to run it, but with sub-optimal performance. Furthermore, we had very few images to train our model with (75 images), thus not being ideal to use, opting against it.

4.2. Out-sampling and Under-sampling

In the `imblearn` library there were a lot of method for Out-sampling(RandomOverSampler, SMOTE, ADASYN among others) and Under-sampling(RandomUnderSampler, NearMiss, TomekLinks,EditedNearestNeighbours among other).

We tried both; out-sampling our data artificially boosted our balanced accuracy, but that higher accuracy did not show up in validation data. We were working with complicated photos, and because this methods attempted to recreate previous data with some variance, the end result was grainy images that had little to do with butterfly wing eye-spot patterns.

Under-sampling improved our balanced accuracy, but it is a bad idea in this situation since we have very little data (75 photos) and it is imbalanced, thus eliminating points will simply exacerbate the problem.

4.3. Support Vector Machine Classifiers

Support vector machines (SVMs, also known as support vector networks) are supervised learning models with associated learning algorithms that examine data for classification and regression analysis in machine learning. SVMs, which are based on statistical learning frameworks or VC theory, are one of the most robust prediction approaches. An SVM training algorithm creates a model that assigns new examples to one of two categories, making it a non-probabilistic binary linear classifier, given a series of training examples, each marked as belonging to one of two categories. SVM assigns training examples to points in space in order to maximize the distance between the two categories.

Then, new instances are mapped into that same space and categorised based on which side of the gap they fall.

SVMs may successfully do non-linear classification in addition to linear classification by applying the kernel approach, which entails implicitly mapping their inputs into high-dimensional feature spaces. It our case we used an RBF kernel, $\gamma = 0.6$ and $C = 1$, these parameters were discovered with KFold and all runs were made with CrossValidation.

4.4. Results and conclusions

Model	Accuracy	Balanced Accuracy	Model	Accuracy	Balanced Accuracy
Bagging	0.83769	0.69794	Decision Tree	0.78905	0.67962
Balanced Bagging	0.78223	0.81137	KNN	0.79974	0.73143
Random Forest	0.84162	0.68421	GNB	0.58606	0.70431
Balanced Random Forest	0.77787	0.81465	CNN	0.84658	0.81634
			SVM	0.79842	0.87965

Table 5: Table of results in Classification Task 2

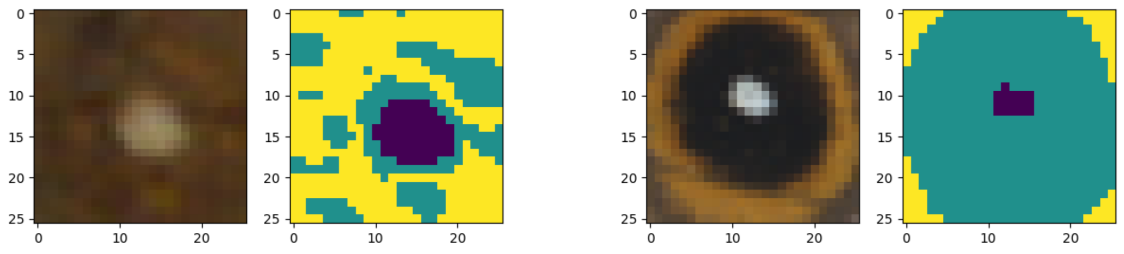

Figure 2: Examples of the results obtained by SVM on two different examples, on purple we have the white ring, on green black and golden ring and on yellow we have the background

The results are shown in table 5; after submitting the results, the professors assigned a balanced accuracy of 0.876130109, which is a very good result; however, there is room for improvement; for example, the γ and C parameters found with the `GridSearchCV` weren't the best ones; by testing with a finer grid of values, we could probably arrive at better parameters and improve the balanced accuracy of our model.