Report - Machine Learning

Catarina Franco (102678) & Mariana Henriques (103165)

1 Introduction

This report presents a comprehensive exploration of machine learning applications. It is structured into two main parts, each addressing distinct challenges within the domain of regression and image analysis.

In the first part, the focus is on regression analysis using synthetic data. We deal with tackle issues such as outlier detection and model optimization, employing various regression techniques including multiple linear regression, Ridge, Lasso, and others. Also, we use robust statistical methods and hyperparameter optimization frameworks like *Optuna* to enhance model performance.

The second part shifts to image analysis, covering both image classification and segmentation tasks. The report details the implementation of *Convolutional Neural Networks* (CNN) and *Vision Transformers* (ViT) for image classification, emphasizing data augmentation and model fine-tuning. For image segmentation, we employ advanced models like *SegFormer*, demonstrating the efficacy of deep learning techniques in accurately identifying and classifying features in images.

Throughout the report, critical aspects such as data preprocessing, model evaluation, and optimization are highlighted, providing valuable insights into the practical application of machine learning methodologies across different data-driven tasks.

2 Part 1 - Regression with Synthetic Data

2.1 First Problem - Multiple Linear Regression with Outliers

We first organized the data into a pandas DataFrame, defining five predictor variables (X_1 to X_5) and one dependent variable (y). Then, we conducted exploratory data analysis, which included visualizing the distributions of the predictor variables using histograms. The data showed a Gaussian-like distribution for the predictors, while the dependent variable exhibited a non-Gaussian pattern (figure 1). The outliers were inspected using boxplots, but only for the dependent variable (y). This decision was made because human error was found to affect approximately 25% of the samples of the dependent variable, while the independent variables were largely unaffected. As such, the primary concern regarding outliers lay with the dependent variable, making it unnecessary to perform outlier detection for the predictors (figure 2). Furthermore, missing values were identified and counted, concluding the absence of missing values throughout the entire dataset.

- **First Approach:**

  Our initial approach began with fitting a linear regression model using the entire training dataset (*xtrain*). To detect potential outliers, we compared the model's predicted values with the actual values of the dependent variable (*ytrain*). Based on prior knowledge that approximately 25% of the data in y were impacted by human error, we established a threshold for the prediction error that would capture exactly 50 outliers (25% of the data points). The data points with prediction

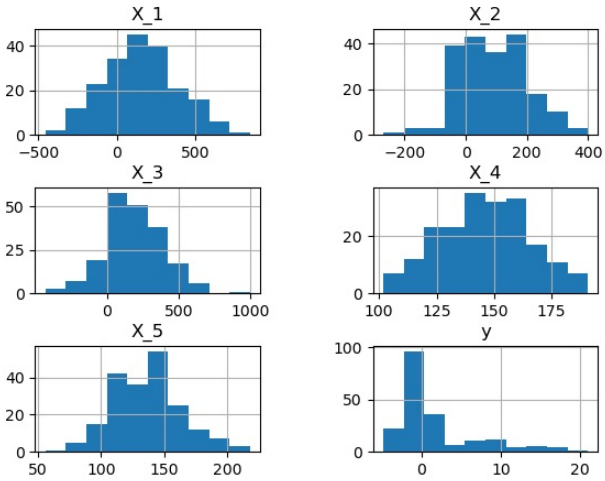

Figure 1: Histograms for each variable.

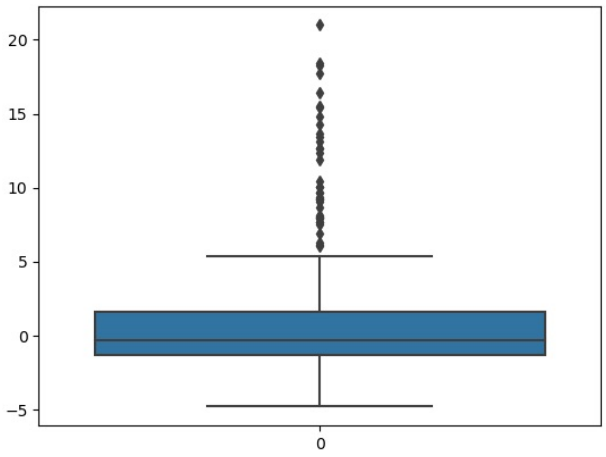

Figure 2: Boxplot of y.

errors greater than this threshold were classified as outliers. These outliers were removed from both the predictor (*xtrain*) and the target (*ytrain*) datasets, resulting in the creation of new, cleaned datasets (*xtrain_new*, *ytrain_new*). To evaluate the model's performance after removing outliers, we split the new dataset into training and validation sets using the *train_test_split* function, ensuring that 80% of the data was used for training, while 20% was reserved for validation. This step is crucial for assessing how well the model generalizes to new data that was not used in the training process. Once the data was split, a new linear regression model was fitted using the training set (*xtrain_new1*, *ytrain_new1*). The model was then used to make predictions on the validation set (*x_newval*), and its performance was assessed by calculating the Mean Squared Error (MSE). The MSE was computed using the actual values (*y_newval*) and the predicted values ($\hat{y}$) generated by the model, yielding a value of **0.2563**. Unsatisfied with this result, we explored another approach to achieve better accuracy.

- **Second Approach:**

  We began by employing the *MinCovDet* class [3] to compute a robust covariance estimate of our dataset, which allowed for effective outlier detection using Mahalanobis distance. The Mahalanobis distance is a measure that accounts for correlations in the data and is particularly effective in identifying multivariate outliers.

  To identify outliers, we set a target of exactly 50 outliers (25% of the data in y that were impacted by human error) to remove from our dataset. We initialized our search for an appropriate threshold for outlier detection by defining a range of thresholds, from 0 to the maximum Mahalanobis distance calculated from the dataset. A binary search approach was implemented to iteratively adjust this threshold until the number of detected outliers matched our target. The process involved fitting the *MinCovDet* model and assessing the number of removed outliers at each iteration. Once we identified the outliers, we cleaned the dataset by removing these problematic data points, resulting in a refined training dataset. We then split this cleaned dataset into features and the target variable, preparing it for modeling. Next, we trained a linear regression model on the cleaned data to establish a baseline performance. The model's effectiveness was evaluated again using Mean Squared Error (MSE).

  To further improve model performance, we employed *Optuna* [1], a hyperparameter optimization framework, to explore various regression models, including Ridge, Lasso, ElasticNet, Huber Regressor, and Linear Support Vector Regression (SVR). An objective function was defined for

each model type, which allowed us to tune critical hyperparameters such as alpha and epsilon based on the model's MSE. For each model, we created an Optuna study to optimize the hyperparameters over 1000 trials, seeking to minimize the MSE. This systematic exploration enabled us to identify the model that delivered the best performance. Ultimately, Ridge Regression emerged as the best model, which was then fitted to the split training data. This resulted in a MSE of **0.000245158**, which was our best result, leading us to submit the prediction values ($\hat{y}$) generated by this model.

2.2 Second Problem - The ARX Model

We began by standardizing both the input and output data using *StandardScaler*, ensuring that the values have a mean of 0 and a standard deviation of 1. This preprocessing step enhances the model's performance and convergence. Following this, we implemented a function named *create_regression_matrix* to generate the regressor matrix X and the output vector Y. These matrices are constructed based on three key parameters:

- n: the number of previous output values (autoregressive terms) included in the model;
- m: the number of previous input values (exogenous terms) considered;
- d: the time delay applied to the input values before they are used.

The function iterates through the dataset, extracting the necessary historical output and input values to construct the regression matrix, preparing the data for model training. This function was used in both models that we will discuss.

- **First Approach:**

  The dataset was divided into training and testing sets, with the first 2000 samples used for training and the remaining samples allocated for testing. Next, we systematically explored various combinations of the parameters n, m, and d (with *create_regression_matrix* function), each ranging from 0 to 9. For every combination, we generated training data, fit the ARX model using the least squares method to estimate the coefficient vector θ, and made predictions on the test set. The performance of each combination was evaluated by calculating the Sum of Squared Errors (SSE) on the last 40 test samples. If a particular combination produced a lower SSE than the previous best, we updated our record of the best parameters.

  To enhance our workflow, we implemented a progress bar using *tqdm* to visualize the completion of the parameter search process. After evaluating all possible combinations, we printed the optimal values of n, m, and d, as well as the corresponding SSE. Finally, we created a plot comparing the actual output values to the predicted values for the best model parameters, allowing us to assess the model's predictive accuracy.

  Our best combination, which resulted in the lowest Sum of Squared Errors (SSE), was found with $n = 9$, $m = 9$, and $d = 5$, achieving an SSE of **0.5751**.

  Below, we can see the plot of the actual vs predicted output values for this combination (figure 3).

  Despite the strong visual alignment between actual and predicted outputs, the SSE value points to room for improvement. For this reason, we decided to explore an alternative approach to minimize the error.

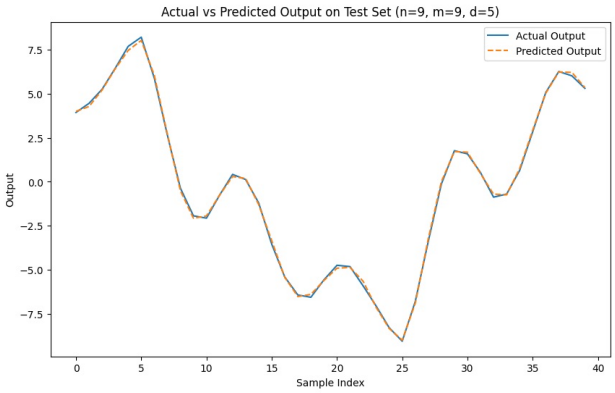

Figure 3: Comparison of actual vs predicted output values for the best model with $n = 9$, $m = 9$, and $d = 5$.

- **Second Approach:**

  We implemented a function called *is_stable* to ensure that the models we built were stable. Stability was checked by verifying the roots of the characteristic polynomial, derived from the autoregressive coefficients. If all roots were inside the unit circle, the model was considered stable.

  To test the predictive ability of our models, we implemented the *one_step_ahead_prediction* function, which predicted one value ahead at each time step using the learned parameters from the ARX model.

  After preparing the data and defining the necessary functions, we split the dataset into a training set (containing 1800 samples) and a testing set (containing the remaining 240 samples). For model selection, we employed three different regression models:

  – Lasso Regression,

  – Ridge Regression,

  – Linear Regression.

  To find the best parameters, we used *Optuna*, an optimization library, to search for optimal values of n, m, and d (with *create_regression_matrix* function), as well as the regularization parameter α for both Lasso and Ridge regression. The objective was to minimize the Mean Squared Error (MSE) on the testing set, while ensuring that the model remained stable.

  For each model (Lasso, Ridge, and Linear Regression), we defined an *Optuna objective function*. These functions explored different combinations of n, m, d, and α (for Lasso and Ridge), and for each combination, we trained the model and evaluated its stability. If the model was not stable, we returned a large error value to discard that combination. Otherwise, the MSE for the predictions was computed and returned as the objective value.

  We ran 500 trials for each model using Optuna to find the best parameters. After optimization, we recorded the best parameters and the corresponding Sum of Squared Errors (SSE) for each model. The best overall model was chosen based on the lowest SSE value across all three models. After identifying the best model (whether it was Lasso, Ridge, or Linear Regression), we retrained it on the entire training dataset using the optimal parameters. We then checked the model's stability once again to ensure that it met the required conditions.

  Finally, we made predictions on both the training set and the test set. For the test set, we predicted the output values based on the scaled input values from the test data (*u_test*).

We compared the predicted values with the actual values, and for visualization, we plotted the actual versus predicted output for both the training and test datasets. Additionally, we calculated the Mean Squared Error for the last 50 samples of the training set to evaluate the model's performance on recent data points. We plotted the actual and predicted values for these last 50 samples to provide a clearer view of the model's performance on the training set. Lastly, the predictions for the last 400 test samples were saved to a file to submit. This method allowed us to systematically find the best model while ensuring both accuracy and stability in the predictions.

Our best-performing model was determined to have the parameter combination $n = 9$, $m = 9$, and $d = 6$, achieving an MSE of **0.0183151**.

Below are two plots illustrating the performance of our model (figure 4 and figure 5). The last plot suggests that the model is performing well on the test set, with high accuracy and generalization, and is capable of predicting values closely aligned with the actual outputs. Therefore, the predictions generated by this model were the ones submitted for evaluation.

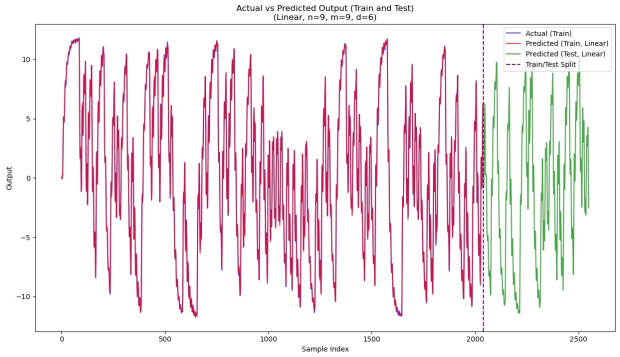

Figure 4: Comparison of actual vs predicted output for the best model with $n = 9$, $m = 9$, and $d = 6$.

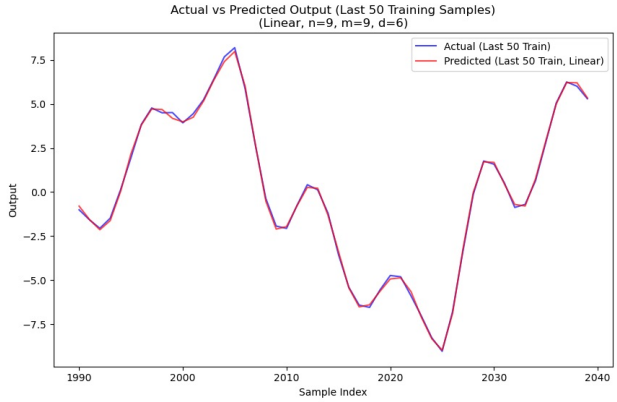

Figure 5: Performance on the last 50 samples of the training set.

3 Part 2 - Image Analysis

3.1 First Problem - Image classification

- **First Approach:**

  We started by implementing a Convolutional Neural Network (CNN) to classify images into two categories (binary classification) using *TensorFlow* and *Keras*. The initial step involved

preparing the image data (X). The images were normalized to a range of 0 to 1 by dividing pixel values by 255. This normalization process enhances the model's convergence speed during training. The images are then reshaped to include the channel dimension, accommodating the requirements of the CNN architecture. To ensure a robust evaluation, the dataset is split into training and testing sets using stratified sampling. Also, the class weights were computed and data augmentation techniques were applied, such as rotation, width and height shifts, and horizontal flips (using the *ImageDataGenerator*), ensuring that the model gives appropriate importance to the under-represented class during training. This strategy generates variations of the training data, enabling the model to generalize better to unseen data and reducing the risk of overfitting. Then, the CNN model was constructed using a sequential architecture, comprising several types of layers:

1. Convolutional layers to extract hierarchical features from the input images;
2. Max pooling layers to downsample the feature maps, reducing dimensionality while preserving essential information;
3. Dropout layers to mitigate overfitting by randomly disabling a fraction of neurons during training.

The model concludes with fully connected layers, using a sigmoid activation function in the output layer to produce probabilities for binary classification. Following the training phase, the model's performance is evaluated on the test set, with the metrics:

- Receiver Operating Characteristic (ROC) curve and Area Under the Curve (AUC) in order to assess the model's classification performance across different thresholds;
- F1 score and Recall in order to evaluate the model's effectiveness in handling class imbalance and its ability to correctly identify positive cases.

To provide a clearer insight into the model's behavior, two plots were generated: the ROC curve, depicts the trade-off between the true positive rate (sensitivity) and the false positive rate, offering a graphical view of classification performance at different thresholds; the training and validation loss curves, giving a visual representation of the model's learning progression over epochs. These curves are crucial for diagnosing overfitting and assessing the model's generalization ability.

When performed with our data, not only the curve's proximity to the top-left corner and the area under the curve is 0.93 indicate a high level of classification accuracy (figure 6), but also the alignment of training and validation loss curves indicates that the model generalizes well without overfitting, maintaining consistent performance across different datasets (figure 7). Additionally, the model achieved an F1 Score of **0.93** and a Recall of **0.95**. However, we were not fully satisfied with these results, so we actively sought a more powerful model to enhance our performance further.

- **Second Approach:**

  We followed a comprehensive process that involved a pre-trained Vision Transformer model, specifically the *ViTForImageClassification* (ViT) [2] from *Hugging Face's Transformers* library. This model is designed for image classification tasks and was fine-tuned for our specific problem of classifying crater images. To enhance training efficiency, we used two GPUs (Nvidia T4) to significantly speed up the supervised training and evaluation with ViT. This acceleration is crucial for handling the computationally intensive tasks involved in processing high-dimensional

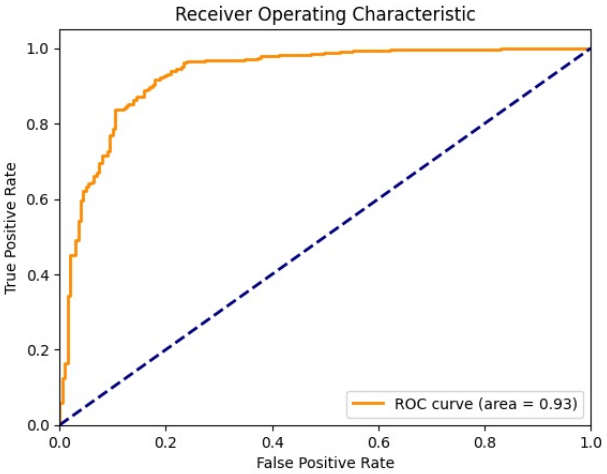

Figure 6: ROC curve.

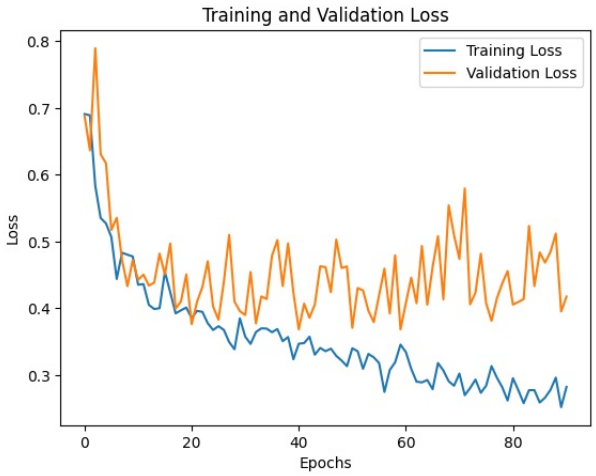

Figure 7: Training and Validation Loss curves.

image data and running deep learning models. Additionally, we implemented data parallelism to distribute its computations across available GPUs, significantly reducing training time.

To prepare the dataset for model training, we split the data into training and validation sets using an 80-20 ratio while ensuring that the distribution of classes was maintained through stratification. This step is crucial in classification tasks to avoid class imbalance in the training and validation datasets.

Then, a directory structure was created to store the images in separate folders based on their labels. This structure included a training directory with subdirectories for each class (0 and 1) and a validation directory following the same format. We then iterated through the training and validation datasets, reshaped the images, and saved them in the appropriate directories as PNG files. This organization facilitated easier access during model training.

To efficiently handle image loading during training and validation, we implemented a custom dataset class, *CraterDataset*, extending PyTorch's Dataset class. This class included methods for fetching the length of the dataset and loading individual images and their corresponding labels. The images were converted to RGB format to ensure compatibility with the Vision Transformer model, which expects three-channel input. With the dataset prepared, we established data loaders for both the training and validation datasets to handle batching and shuffling, which are essential for efficient training. By setting a batch size of 64, we ensured that the model received data in manageable chunks, thereby speeding up the training process. To ensure compatibility with the ViT model, each image underwent a series of transformations:

- *transforms.Resize*: The ViT model requires a consistent input size for all images and is typically pre-trained on images sized 224×224 pixels. To match this requirement, we resized all images to these dimensions, ensuring compatibility with the model's input layer and preserving the spatial information within a standardized frame.
- *transforms.ToTensor*: This transformation converts the image data from a NumPy array to a PyTorch tensor, which is essential for deep learning operations. It also scales pixel values from the original range of 0-255 to a normalized range of 0-1, making it easier for the model to process. This normalization step is crucial for improving the stability and convergence of the model during training.

With these transformations applied, we passed the dataset through PyTorch's DataLoader, which handles batching and shuffling. Batching optimizes GPU utilization, allowing the model to process multiple images simultaneously, while shuffling enhances data randomness during each epoch, preventing the model from learning any order-based biases. By combining these transformations and DataLoader, the pipeline ensures that each image is resized, normalized, and delivered to the model in batches, significantly contributing to more effective and stable training.

Finally, we set up our ViT model. The core of our implementation was the train_model function, which orchestrated the training process. During each epoch, the model underwent a forward pass to compute predictions, followed by loss calculation using the Cross-Entropy Loss function. After calculating the loss, we performed backpropagation to adjust the model's weights based on the calculated gradients. The model was then evaluated on the validation dataset, where we computed the validation loss and the F1-score and a recall. The model's weights were saved whenever an improvement in F1-score was achieved, ensuring that we retained the best-performing model. The best validation F1-score obtained was **0.9913**, making this model the top choice for our final submission.

3.2 Second Problem - Image segmentation

The dataset contains grayscale images of craters and corresponding binary masks that indicate crater locations.

- **First Approach:**

  We defined a model, *model_a*, associated with the data format A (a 7×7 pixel neighborhood around each pixel as features).

  This model is a Convolutional Neural Network (CNN) that takes the 7×7 neighborhood around each pixel and classifies the central pixel as either crater or background. Here, we used *Conv2D*, which is a convolutional layer with 32 filters, each of size 3×3, and which extracts small features from the 7×7 input, like edges or textures, helping identify whether the central pixel belongs to a crater. Using a flatten layer, we converted the two-dimensional features maps into a single vector, which prepares it for the fully connected, or dense, layers. Regarding dense layers, the first (fully connected) has 64 neurons with a *relu* activation function, which allows it to learn more patterns, while the final has a single neuron with a *sigmoid* activation function, outputting a probability value between 0 and 1 (if the output is closer to 1, it indicates the central pixel is part of a crater, otherwise, it's considered background). This model is compiled with *binary cross-entropy* loss (suitable for binary classification), *Adam* optimizer and *accuracy* as a metric to measure performance during training.

  Before training *model_a*, we reshaped the corresponding input data to ensure that it had the shape that is required by the CNN input layer. Then, this models is trained for 10 epochs with a batch size of 32.

  Furthermore, the predictions from this model are evaluated using *Balanced Accuracy* to account for the class imbalance (since the background pixels dominate). Additionally, in evaluation, the model generates predictions for its training data, which are rounded to 0 or 1 to get binary results.

  Lastly, taking into consideration the value of accuracy of the *model_a*, **0.7574**, which is not very satisfactory, we tried another model that presented better segmentation performance.

- **Second Approach:**

In this approach, we defined a model associated with the data format B (the entire 48×48 pixel image along with a segmentation mask for structured prediction). In the dataset preparation process, we start by doing data splitting, where the dataset is split into training and testing sets, with an $80-20$ ratio, using *train_ test_ split*, to train the model and evaluate its performance on unseen data. Then, we reshaped each image and mask to 48×48 pixels, in order to match the model input requirements, and we normalized the data by dividing pixel values by 255, bringing the values into the range $[0, 1]$, which generally helps improve training stability and convergence. Furthermore, we used a SegFormer model [4], which is well-suited for our problem, since it combines effective feature extraction with efficient computation. In particular, we load the pretrained (with weights for image segmentation) SegFormer model, *nvidia/mit-b3*, and configure it with two labels, namely, crater and non-crater. In cases where parameter mismatches arise, textitignore_mismatched_sizes=True allows partial loading of weights. To handle data loading and augmentation for the model, we define also a dataset class called *CreaterDataset*. Specially, we applied *ColorJitter* for data augmentation, which randomly adjusts brightness, contrast, saturation and hue, in order to improve generalization by providing slightly altered versions of the images during training. Since these images can be optionally augmented (in particular, the augmentation is applied with a 50% probability to each image), then the *processor* transforms the image and mask into the expected format. Moreover, the *DataLoader* loads mini-batches of data from the *CraterDataset*. We used a batch size of 8 and enabled shuffling for the training data to improve learning. Regarding optimization, we used *AdamW* as our optimizer, which is a variant of *Adam* that incorporates weight decay, with different learning rates to the encoder and segmentation head to improve training stability. We also applied a *CosineAnnealingWarmRestarts* scheduler, which adjusts the learning rate periodically and improve the convergence by simulating a "restart" of the optimization. Then, we enabled mixed precision training using *autocast* and GradScaler, what reduces memory usage and speeds up training by computing some operations in half precision without losing accuracy. After each epoch, we evaluated the model's performance on the validation set using *calculate_ validation_ accuracy*. In particular,

- for each image in the validation set, the model generates a probability map indicating the likelihood of each pixel belonging to a crater. These probabilities are thresholded to produce binary masks;
- predictions are resized back to 48×48 if necessary, ensuring they match the original dimensions of the masks;
- we calculate balanced accuracy, which takes into account class imbalance (more non-crater than crater pixels), offering a fair assessment of the model's performance: **0.8258**.

After training, we loaded the best model and generated predictions on the test set *X_final*, where the test images are processed similarly to the training and validation images. Besides balanced accuracy, we calculated some metrics, which give a fuller picture of how well the model identifies crater pixels versus non-crater pixels, to gain more insight into the model's performance:

- Sensitivity: **0.7271** (which means that the model correctly identified **72.71**% of actual positive regions (craters));
- Specificity: **0.9246** (which means that the model accurately identified **92.46**% of actual negative regions (non-craters));
- True positives: **54176**;
- True negatives: **165435**;
- False positives: **13495**;

– False negatives: **20334**.

Finally, we saved the binary predictions for all images in *X_final*, allowing further analysis or visualization if needed. We also computed some basic statistics on the predictions, such as the number and percentage of pixels classified as craters:

– Number of positive predictions (craters): **114312**;

– Number of negative predictions (non-craters): **337272**;

– Percentage of positive predictions (pixels classified as craters): **25.31**%.

According to the outputs values, presented above, the model demonstrates strong specificity and reasonable sensitivity, making it well-suited for tasks where it's more critical to avoid false positives than to capture every positive (that is the case). The obtained balanced accuracy suggests the model performs consistently across both classes. Moreover, the distribution of positive predictions in *X_final* (**25.31**%) is in line with the sensitivity and specificity values, indicating that the model's predictions align well with the class distributions in the data. For all these reasons, this model was the preferred option for our final submission.

4 Conclusion

We achieved strong results in regression with synthetic data and image segmentation, underscoring the effectiveness of our chosen methodologies. However, in the ARX Model section, further exploration and refinement could have improved the outcomes. Overall, the findings emphasize the importance of thorough investigation and model tuning to maximize performance across diverse machine learning tasks.

References

[1] Takuya Akiba et al. *Optuna: A Next-generation Hyperparameter Optimization Framework*. 2019. arXiv: 1907.10902 [cs.LG]. URL: https://arxiv.org/abs/1907.10902.

[2] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV]. URL: https://arxiv.org/abs/2010.11929.

[3] Mia Hubert, Michiel Debruyne, and Peter J. Rousseeuw. "Minimum covariance determinant and extensions". In: *WIREs Computational Statistics* 10.3 (Dec. 2017). ISSN: 1939-0068. DOI: 10.1002/wics.1421. URL: http://dx.doi.org/10.1002/wics.1421.

[4] Enze Xie et al. *SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers*. 2021. arXiv: 2105.15203 [cs.CV]. URL: https://arxiv.org/abs/2105.15203.