Machine Learning for Economics Assignment

Leo Crellin 1948853, Yitong Zhu 2638495, Nandini Sunil 2631619,
Thomas Trainor-Gilham 1864173

Contents

1 Classification Analysis

1.1 Data Description

Our initial dataset comprises of 303 observations (patients) and 14 features. These features include patient demographics such as age and sex, clinical measurements such as resting blood pressure and serum cholesterol, categorical descriptors of patient status such as resting ECG results and binary indicators such as whether the patient experiences exercised induced angina. The target variable is whether heart disease is present (referred to as 'AHD' in the data), where 'Yes' indicates presence and 'No' indicates absence of heart disease. Abbreviations and descriptions are described in Table 1.

The average patient is approximately 54.4 years (ranging from 29 to 77), average cholesterol is 246.7 mg/dl and average resting blood pressure is 131.7mmHg. Table 2 provides summary statistics for the numerical variables and Table 3 shows the distribution for categorical variables in the original dataset.

Table 1: Variable Abbreviations and Descriptions

Abbreviation/Code	Full Name / Description
`AHD`	Heart Disease Presence
`Age`	Age (years)
`Ca`	Number of Major Vessels (colored by fluoroscopy)
`Chol`	Serum Cholesterol (mg/dl)
`ChestPain`	Chest Pain Type
`ExAng`	Exercise Induced Angina (1=Yes, 0=No)
`Fbs`	Fasting Blood Sugar (1= $>$ 120 mg/dl, 0= $\leq$ 120 mg/dl)
`MaxHR`	Maximum Heart Rate Achieved
`Oldpeak`	ST Depression induced by exercise relative to rest
`RestBP`	Resting Blood Pressure (mm Hg)
`RestECG`	Resting Electrocardiographic Results
`Sex`	Sex (1=Male, 0=Female)
`Slope`	Slope of the Peak Exercise ST Segment
`Thal`	Thallium Stress Test Result

Table 2: Summary statistics for numerical features (Original Data).

Feature	Count	Mean	Std Dev	Min	25%	50%	75%	Max
Age	303	54.4	9.0	29.0	48.0	56.0	61.0	77.0
Resting Blood Pressure (mm Hg)	303	131.7	17.6	94.0	120.0	130.0	140.0	200.0
Serum Cholesterol (mg/dl)	303	246.7	51.8	126.0	211.0	241.0	275.0	564.0
Maximum Heart Rate Achieved	303	149.6	22.9	71.0	133.5	153.0	166.0	202.0
ST Depression (exercise vs rest)	303	1.0	1.2	0.0	0.0	0.8	1.6	6.2

Std Dev: Standard Deviation. 25%, 50%, 75%: Percentiles.

1.2 Preprocessing

Data preprocessing was performed to prepare the features for classification modelling.

First, data cleaning addressed missing values. The 'Ca' column was converted to numeric, coercing non-numeric entries to NaN. Subsequently, rows containing any missing values (in 'Ca' or 'Thal') were removed, resulting in a dataset of 297 observations. The binary target variable, 'AHD' (Heart Disease Presence), was label encoded, mapping 'No' to 0 and 'Yes' to 1.

Next, the data was split into training (75%) and testing (25%) sets using stratification based on the encoded target variable to ensure representative class distributions in both sets.

A feature preprocessing pipeline (`ColumnTransformer`) was then defined and applied separately to numerical and categorical predictors:

Table 3: Frequency distribution for categorical features (Original Data).

Feature	Category	Count	Percentage (%)
Sex	1 (Male)	206	68.0
	0 (Female)	97	32.0
Chest Pain Type	asymptomatic	144	47.5
	nonanginal	86	28.4
	nontypical	50	16.5
	typical	23	7.6
Fasting Blood Sugar	0 ($\leq$ 120 mg/dl)	258	85.1
	1 ($>$ 120 mg/dl)	45	14.9
Resting ECG Results	0	151	49.8
	2	148	48.8
	1	4	1.3
Exercise Induced Angina	0 (No)	204	67.3
	1 (Yes)	99	32.7
Slope of Peak Exercise ST Seg	1	142	46.9
	2	140	46.2
	3	21	6.9
Num. Major Vessels (Ca)	0	176	58.1
	1	65	21.5
	2	38	12.5
	3	20	6.6
Thallium Stress Test (Thal)	normal	166	54.8
	reversable	117	38.6
	fixed	18	5.9
Heart Disease Presence (AHD)	No	164	54.1
	Yes	139	45.9

Total N = 303. Missing values: Ca (N=4), Thal (N=2). Percentages calculated on N=303.

1. **Numerical Features:** `Age`, `RestBP`, `Chol`, `MaxHR`, and `Oldpeak` were standardised. This scales the data to have approximately zero mean and unit variance.

2. **Categorical Features:** `Sex`, `ChestPain`, `Fbs`, `RestECG`, `ExAng`, `Slope`, `Thal`, and `Ca` (treated as categorical after cleaning) were one-hot encoded using `OneHotEncoder`. To prevent multi-collinearity, the first category encountered for each feature during fitting was dropped.

The entire preprocessing pipeline (scaler and encoder) was fitted only on the training data to prevent data leakage from the test set. The fitted pipeline was then used to transform both the training and test sets consistently.

This preprocessing resulted in training and testing datasets with 20 predictor features, ready for model input. The structure of the final processed feature set is summarised in Table 4.

1.3 Classifier Construction and Evaluation

The objective was to construct and evaluate three different classifiers to predict heart disease, selecting the best based on test set accuracy. We chose Logistic Regression (LR), K-Nearest neighbours (KNN), and Linear Discriminant Analysis (LDA). The cleaned and preprocessed data (N=297) was split into a training set (222 observations, 75%) and a test set (75 observations, 25%) using stratification based on the target variable ('AHD'). For each classifier, optimal hyperparameters were determined using GridSearchCV with 5-fold stratified cross-validation on the training set, optimising for accuracy.

Table 4: Summary Statistics of Full Processed Data

Feature	Count	Mean	Std Dev	Min	Median	Max
Age	297	-0.015	1.007	-2.857	0.147	2.484
Resting Blood Pressure (mm Hg)	297	-0.025	0.990	-2.127	-0.120	3.783
Serum Cholesterol (mg/dl)	297	-0.043	0.964	-2.292	-0.123	5.827
Maximum Heart Rate Achieved	297	-0.038	1.045	-3.618	0.117	2.348
ST Depression (Oldpeak)	297	0.009	0.991	-0.888	-0.208	4.381
Sex: Male (Sex_1)	297	0.677	0.468	0.000	1.000	1.000
Chest Pain: Non-anginal	297	0.279	0.449	0.000	0.000	1.000
Chest Pain: Non-typical	297	0.165	0.372	0.000	0.000	1.000
Chest Pain: Typical	297	0.077	0.268	0.000	0.000	1.000
Fasting Blood Sugar > 120	297	0.145	0.352	0.000	0.000	1.000
Resting ECG: 1	297	0.013	0.115	0.000	0.000	1.000
Resting ECG: 2	297	0.492	0.501	0.000	0.000	1.000
Exercise Induced Angina: Yes	297	0.327	0.470	0.000	0.000	1.000
Slope ST Seg: 2	297	0.461	0.499	0.000	0.000	1.000
Slope ST Seg: 3	297	0.071	0.257	0.000	0.000	1.000
Thallium Stress: Normal	297	0.552	0.498	0.000	1.000	1.000
Thallium Stress: Reversible	297	0.387	0.488	0.000	0.000	1.000
Num. Major Vessels: 1	297	0.219	0.414	0.000	0.000	1.000
Num. Major Vessels: 2	297	0.128	0.335	0.000	0.000	1.000
Num. Major Vessels: 3	297	0.067	0.251	0.000	0.000	1.000

Note: Numerical features (Age, RestBP, Chol, MaxHR, Oldpeak) were standardised using parameters derived from the training set. Categorical features shown are one-hot encoded (values 0 or 1); the mean represents the proportion of samples in that specific category within this full cleaned dataset (N=297). The Median column represents the 50th percentile. Reference categories dropped during encoding ('drop='first'') were: Sex=0 (Female), ChestPain=asymptomatic, Fbs=0 (≤120), RestECG=0, ExAng=0 (No), Slope=1, Thal=fixed, Ca=0.

Classifier Tuning Results (Based on Cross-Validation on Training Set):

- **Logistic Regression:** The best performance (Mean CV Accuracy: 0.8284) was achieved with L2 penalty and C=10.0.
- **K-Nearest neighbours:** The best performance (Mean CV Accuracy: 0.8241) was achieved with k=17 neighbours, using Manhattan distance ('manhattan') and uniform weights.
- **Linear Discriminant Analysis:** The standard SVD solver (with no shrinkage) yielded the best performance (Mean CV Accuracy: 0.8240).

Test Set Evaluation: The models, refit on the entire training set using these optimal parameters, were evaluated on the held-out test set (N=75).

Confusion Matrices: Table 5 summarises the performance on the test set. Both Logistic Regression and LDA achieved the highest accuracy (84.0%). KNN achieved a slightly lower accuracy of 82.7%. LDA correctly identified slightly more true negatives (No AHD: 36) than LR (35), while LR correctly identified slightly more true positives (Yes AHD: 28) than LDA (27).

Table 5: Confusion Matrices on Test Set (Rows: True Label, Columns: Predicted Label)

	LR (Acc: 84.00%)		KNN (Acc: 82.67%)		LDA (Acc: 84.00%)	
True Label	Pred. No	Pred. Yes	Pred. No	Pred. Yes	Pred. No	Pred. Yes
No (0)	**35**	5	**34**	6	**36**	4
Yes (1)	7	**28**	7	**28**	8	**27**

ROC Curves: The Receiver Operating Characteristic (ROC) curves (see Figure 1) visually compare the trade-off between true positive rate (sensitivity) and false positive rate (1-specificity) across different classification thresholds. The Area Under the Curve (AUC) provides a single measure of overall

discriminatory power. Linear Discriminant Analysis (LDA) achieved the highest AUC (0.9343), followed closely by Logistic Regression (LR) (0.9286), and then KNN (0.9004). This suggests LDA has a slight edge in distinguishing between positive and negative cases across the range of thresholds, despite tying with LR on accuracy.

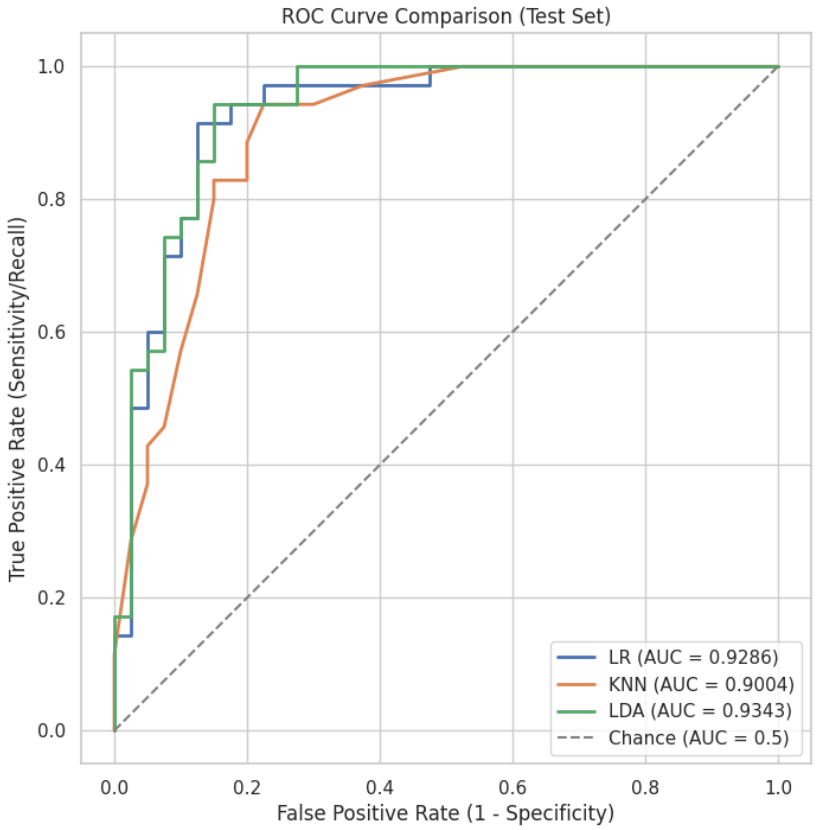


Figure 1: ROC Curve Comparison on Test Set.

Accuracy Comparison & Best Classifier: Based on the primary evaluation metric - test set accuracy - both Logistic Regression and Linear Discriminant Analysis performed best, achieving 84.0%. KNN followed closely at 82.7%. Since they tied on accuracy, we select LDA as the final model, due to edging LR in its AUC value.

1.4 Prediction for Specific Patient

Using the best identified classifier (Linear Discriminant Analysis), we predict the heart disease status for the specified patient: a 55-year-old woman with 'typical' chest pain, 'normal' thallium stress test results, resting blood pressure of 130 mmHg, serum cholesterol of 246 mg/dl, fasting blood sugar of less than 120 mg/dl, RestECG=2, MaxHR=150, ExAng=1, Oldpeak=1.0, Slope=2, and Ca=0.

After applying the identical preprocessing steps (one-hot encoding for categorical features including 'Ca', standardising all 5 numerical features 'Age', 'RestBP', 'Chol', 'MaxHR', 'Oldpeak') to this patient's data, resulting in 20 features, the selected LDA model predicts that the patient does not have heart disease (AHD=No). The model assigns a probability of approximately 4.64% for the patient having heart disease (AHD=Yes).

2 Regression Analysis

2.1 Summary statistics

The summary statistics (Table 6) for the eight continuous variables across 400 observations reveal significant variability, indicating a heterogeneous sample in terms of sociodemographic characteristics,

income, and credit availability.

To predict Balance, the regression model incorporates 11 main predictors and 55 interaction terms. Interactions (e.g., Income * Limit) allow the effect of one predictor to depend on the level of another, capturing non-linear relationships and improving model fit.

Preprocessing involved dummy encoding categorical variables (Gender, Student, Married, Ethnicity) and standardising numerical variables (Income, Limit, Rating, Cards, Age, Education, Balance). All predictors and interactions mentioned above were included. No imputation was required as their were no missing values in the dataset.

Table 6: Summary Statistics for Continuous Variables

	ID	Income	Limit	Rating	Cards	Age	Education	Balance
Count	400	400	400	400	400	400	400	400
Mean	200.50	45.22	4735.60	354.94	2.96	55.67	13.45	520.02
Std	115.61	35.24	2308.20	154.72	1.37	17.25	3.13	459.76
Min	1.00	10.35	855.00	93.00	1.00	23.00	5.00	0.00
25%	100.75	21.01	3088.00	247.25	2.00	41.75	11.00	68.75
50%	200.50	33.12	4622.50	344.00	3.00	56.00	14.00	459.50
75%	300.25	57.47	5872.75	437.25	4.00	70.00	16.00	863.00
Max	400.00	186.63	13913.00	982.00	9.00	98.00	20.00	1999.00

2.2 Lasso Estimation using Cross-Validation

2.2.1 Cross-Validation

In 5-fold cross-validation, we divide the dataset into five equal parts, using four parts for training the model and the remaining part for validation. This process is repeated five times, each time using a different fold for validation, and the cross-validation errors are averaged. When applying this to lasso regression, we test different values of the regulari sation parameter λ to see which one results in the lowest average prediction error. The optimal λ is chosen as the one that minimises the mean cross-validation error to balance model complexity and predictive performance.

Figure 2 plots the cross-validated MSE against $-\log\lambda$. The minimum MSE occurs at the λ indicated by the vertical dashed line.

2.2.2 Coefficient Plot

Figure 3 displays the Lasso coefficient paths as λ increases (shrinkage increases from left to right). At $\lambda = 0$, coefficients are at their Ordinary Least Squares values. As λ increases, coefficients shrink towards zero, with predictors like 'Cards' being penalised earlier than stronger predictors like 'Limit'. As the penalty increases, some coefficients such as Cards and Rating can momentarily bump upwards once a more dominant variable has been eliminated as the Lasso seeks to redistribute explanatory power among the remaining active features. Eventually, all coefficients are reduced to zero.

Under the optimal tuning penalty ($\lambda = 0.916$) shown by the black vertical dashed line in Figure 3 which is calculated using 5-fold cross-validation Lasso with random_state = 0 for both, 50% of the coefficients are shrunk to zero.

2.2.3 Prediction

Using the optimal Lasso model (tuned via 5-fold CV, 'random_state = 0'), we predict their balance to be: \$287.30.

The MSE of this Lasso model is 4,715.30. The square root of the MSE is therefore around $\sqrt{4715.30} \approx 68.67$, indicating that this model leads to predictions that are on average approximately within around \$68.67 of the true balance value.

Since question 1 focuses on model estimation and parameter selection rather than model evaluation, we have used the entire dataset for training the final model after selecting λ through 5-fold cross-validation.

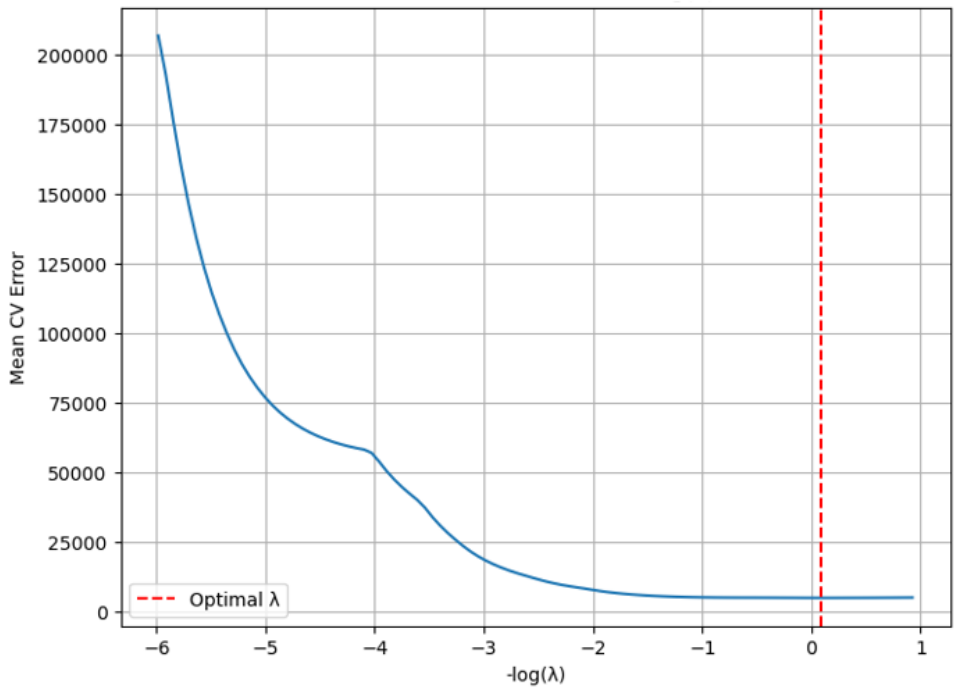


Figure 2: Cross Validation Error vs -log(λ)

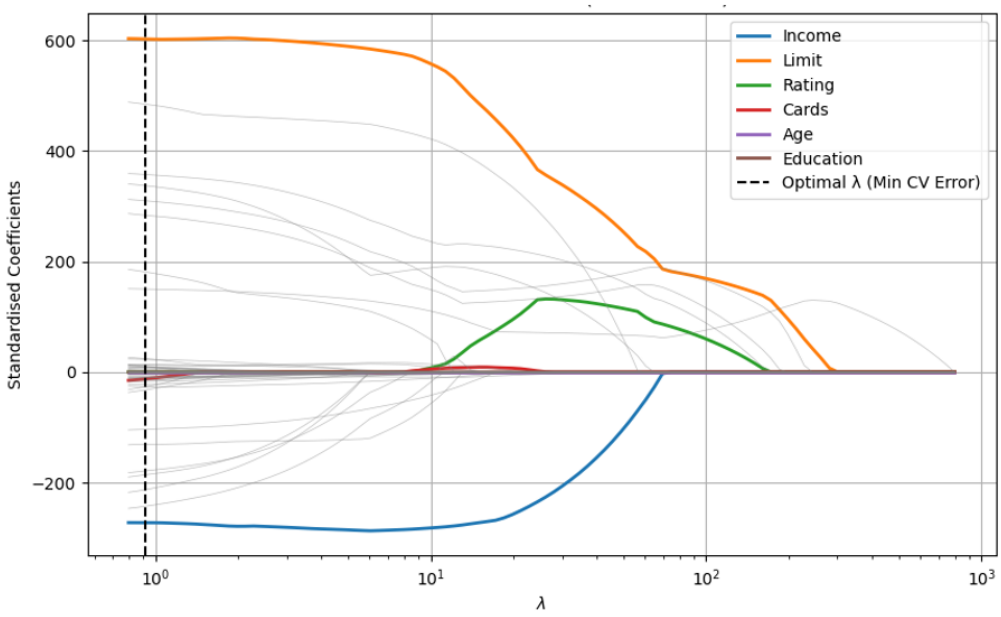


Figure 3: Lasso Coefficient Paths (Standardised)

2.3 Tree-Based Methods

2.3.1 Tree Visualisation

Looking at the decision tree regressor visualisation for predicting credit card balance in Figure 4, several key patterns emerge in how customer characteristics influence average credit card debt.

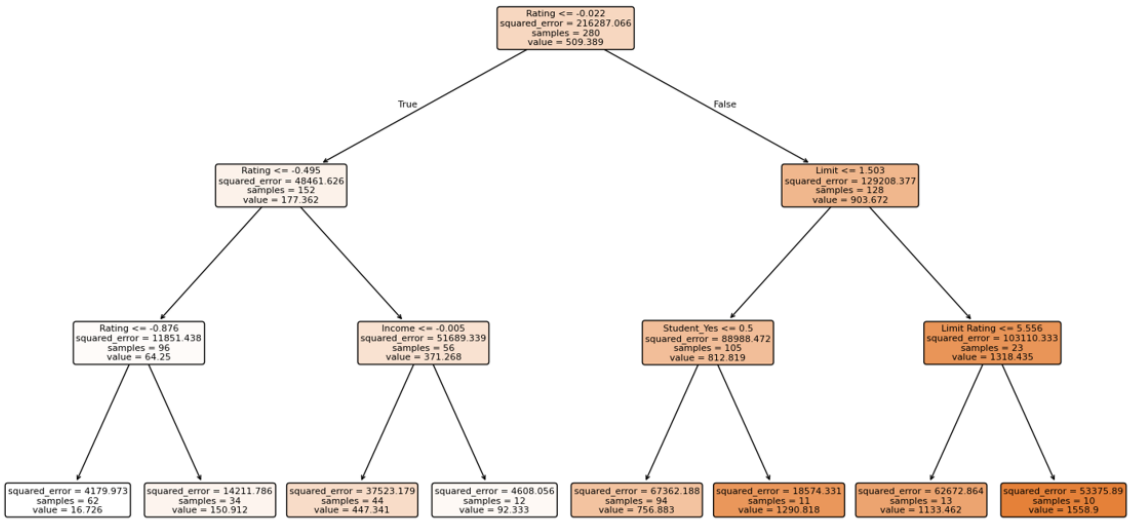


Figure 4: Decision Tree with Max Depth = 3

The tree shows that rating is the most important initial splitting factor, with a threshold of ≤ 0.022. For customers with lower ratings (≤ 0.022), their average balance is around \$509. This group is then further split based on rating again, with those having very low ratings (≤ -0.495) averaging lower balances of about \$177. For customers with better ratings (> 0.022), credit limit becomes the next important factor.

When examining the deeper nodes, we see that users with low ratings, but higher income (> -0.002) have higher balances (\$371) compared to their lower-income counterparts. On the right side of the tree, student status becomes important - non-students with higher credit limits tend to carry higher balances (\$1,318) than students (\$812). The highest average balances (\$1,319) appear among customers with higher limits and high rating scores (> 5.556).

Overall, this tree suggests that credit rating, credit limit, income, and student status are key predictors of credit card balance, with interactions between these variables creating distinct customer segments with varying debt levels. The model effectively identifies several customer profiles with substantially different average balances, from the lowest around \$64 to the highest exceeding \$1,300.

2.3.2 Random-Forest

The data below in Table 7 compares test MSEs for random forests with max_depth=3 and no depth limit (max_depth=None) across varying numbers of trees (1, 5, 10, 50, 100, 200), alongside a single decision tree with max_depth=3. For max_depth=3, MSE drops sharply from 71,255.06 (1 tree) to 30,725.85 (5 trees) but stabilises around 31,560–32,217 beyond 10 trees. For max_depth=None, MSE decreases steadily from 58,810.41 (1 tree) to 11,964.14 (200 trees), showing consistent improvement with more trees. The single tree's MSE is 49,005.80, serving as a baseline.

The plot in Figure 5 visualises these trends: the max_depth=None random forest achieves the lowest MSEs, while the max_depth=3 random forest plateaus at a higher MSE. The single tree (green dot at 0 trees) has an MSE between the two random forest configurations at one tree but is outperformed as the number of trees increases. Analysis reveals that more trees reduce MSE by averaging errors, with the largest gains early on (1 to 5 trees). Unconstrained depth (max_depth=None) consistently outperforms max_depth=3, as deeper trees capture more patterns without overfitting. Random forests with multiple trees surpass the single tree's performance, especially with no depth limit. A Random

Forest model with 200 trees and no maximum depth has the lowest Test MSE (11,964.14) of the options.

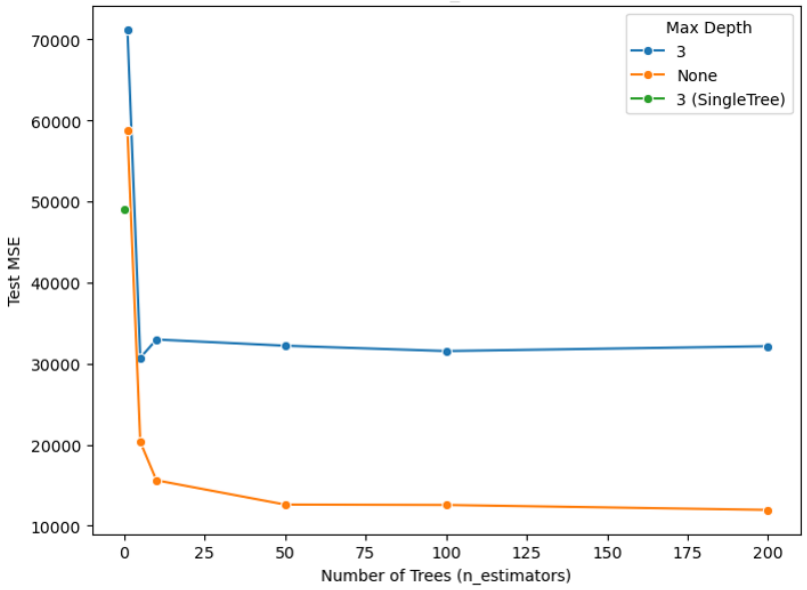


Figure 5: Test MSE for Random Forests (max_depth=3 vs. None) + Single Tree

Table 7: Test MSE Comparison for Different Tree Models

Number of Trees	Maximum Depth	Test MSE
200	None	11964.14
100	None	12571.01
50	None	12617.77
10	None	15615.06
5	None	20337.84
5	3	30725.85
100	3	31560.52
200	3	32159.4
50	3	32217.44
10	3	32984.54
0	3 (SingleTree)	49005.8
1	None	58810.41
1	3	71255.06

2.3.3 Prediction

Using the optimal Random Forest model (200 trees, no max depth, 'random_state = 0'), we predict their balance to be: \$613.87. The test MSE of this model is 11,964.14. The square root of the test MSE is therefore around $\sqrt{11964.14} \approx 109.38$, indicating that this model leads to predictions that are on average approximately within around \$109.38 of the true balance value.

2.3.4 Random Forest Variable Importance

Variable importance for the optimal RF model was assessed using the mean decrease in impurity (MDI) metric, calculated across all 200 trees. Figure 6 and Table 8 display the top 12 most important features. The model identifies 'Rating' as the most influential predictor (Importance = 0.529), followed by 'Limit' (0.218). Other features contributing notably include the interaction 'Limit Rating' (0.061), 'Income'

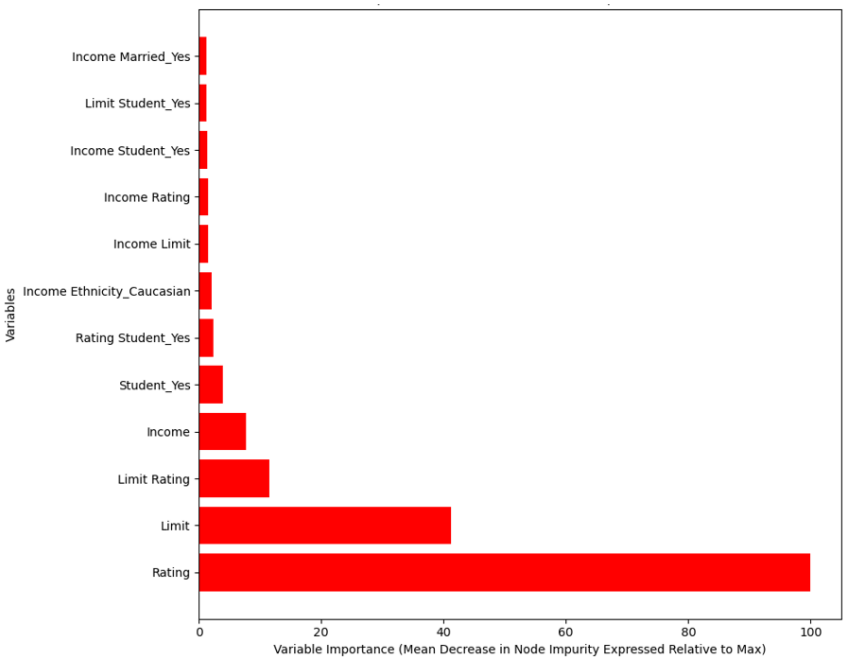


Figure 6: Random Forest (200 Trees, no maximum depth) Variable Importance

(0.041), and 'Student_Yes' (0.021). Several interaction terms also appear among the top predictors, such as 'Rating Student_Yes' and 'Income Ethnicity_Caucasian', suggesting the RF effectively captures complex, non-linear relationships inherent in the data, relying heavily on 'Rating' and 'Limit' to make predictions.

Table 8: Random Forest Top 12 Feature Importance (MDI)

RF Feature	Importance (MDI)
Rating	0.529
Limit	0.218
Limit Rating	0.061
Income	0.041
Student_Yes	0.021
Rating Student_Yes	0.013
Income Ethnicity_Caucasian	0.011
Income Limit	0.008
Income Rating	0.008
Income Student_Yes	0.008
Limit Student_Yes	0.007
Income Married_Yes	0.007

2.3.5 Differences between the Lasso and Random Forest Model

The random forest model places 'Rating' as the most important feature, while Lasso prioritises 'Limit' according to coefficient magnitude (see Table 9). This discrepancy may arise because 'Rating' and 'Limit' are correlated, and random forests handle collinearity better by splitting on either feature across trees, whereas Lasso may distribute effects between correlated variables, emphasising 'Limit'.

The random forest also includes a broader set of interactions (e.g., 'Income Ethnicity_Caucasian', 'Income Married_Yes') among the top 12, reflecting its flexibility in capturing non-linear effects without explicitly defining interactions in the feature set.

Table 9: Lasso Non-Zero Coefficients (Top 12 by Absolute Value)

Lasso Feature	Lasso Coefficient
Limit	521.858
Student_Yes	434.212
Income	-270.831
Limit Student_Yes	136.302
Limit Rating	116.194
Income Rating	-87.568
Rating	76.201
Income Student_Yes	-28.479
Cards	18.422
Limit Gender_Female	14.479
Age	-13.400
Rating Age	-12.458

The random forest has a test MSE of 11,964.14, while the Lasso model achieves a lower MSE of 4,715.30. Despite the random forest's ability to model non-linearities and interactions, the Lasso model outperforms it in predictive accuracy. This suggests that the underlying relationships in the data may be well-approximated by a linear model with selected interactions, or that the random forest (with no depth limit) may overfit, even with 200 trees averaging predictions.

2.4 Conclusion

Comparing the two approaches, the Lasso model (MSE = 4,715.30) demonstrated better predictive accuracy on this dataset compared to the unconstrained Random Forest (Test MSE = 11,964.14). While Random Forest identified 'Rating' as most important and captured complex interactions, the Lasso model's emphasis on 'Limit', 'Student_Yes', and 'Income' within a more constrained framework yielded better performance. This suggests the underlying data structure might be well-approximated by Lasso's selected features and interactions, or that further tuning (e.g., limiting depth) could be necessary for the Random Forest to generalise better. Based on these results, the Lasso model provides a more effective solution for predicting balance in this specific instance.

3 Image Classification

The classifier adopted in our analysis was a convolutional neural network (CNN), designed to extract hierarchical spatial features from font image crops and accurately predict the font identity. The model operates on a dataset comprising 2000 to 3000 distinct font classes. Through systematic hyperparameter tuning guided by validation performance, as detailed in Section 3.2.2, an optimised model was developed. The best configuration achieved a final test accuracy of 80.19%.

3.1 Image Preprocessing and Data Construction

The initial dataset consisted of '.bmp' images, each containing two lines of pangrams rendered in a specific font style. From a random sample of 2000-3000 source images, we extracted fifty 32×32 pixel grayscale crops per font image to capture distinctive typographic elements while remaining computationally efficient.

Rather than relying purely on random sampling of crop locations, a sliding window approach with a step size of 12 pixels was used to systematically extract candidate regions. Crops were prioritized from areas containing sufficient visual content (defined as having $\geq$20 non-white pixels) to ensure they captured meaningful font features rather than empty space.

All crops derived from a single source image were assigned the same font class label. The dataset was partitioned into training (70%), validation (15%), and test (15%) sets, ensuring each font was represented across splits. This division allowed for model training, hyperparameter tuning on the

validation set, and final unbiased performance evaluation on the test set. Prior to model input, pixel intensity values were normalized to the [0,1] range via scalar division by 255.0 to aid convergence.

During the training phase only, on-the-fly data augmentation was applied to diversify the training crops and improve model generalisation. These augmentations included random rotations (e.g., up to ±20°), translations (e.g., up to ±25% horizontally and vertically), scaling (randomly between 0.9 and 1.1), and potentially minor brightness/contrast adjustments (e.g., 0-0.1, based on experimentation) with the specific optimal parameters determined during hyperparameter tuning as detailed in Section 3.2.2. Validation and test sets used the identical cropping strategy but without any augmentation. This methodical preprocessing and augmentation approach generated a robust training corpus, enabling the network to learn font-specific features effectively.

3.2 Training a Classifier

3.2.1 Model Architecture

Our classification framework employs a convolutional neural network (CNN) optimized for 32×32 pixel grayscale font crops, discriminating between 2000-3000 distinct typographic classes. The architecture comprises three sequential convolutional blocks for hierarchical feature extraction, followed by fully connected layers for classification.

The initial block applies 32 convolutional filters (3×3, padding=1) to the input, followed by batch normalization to stabilize training dynamics, ReLU activation for non-linearity, and max-pooling (2×2), reducing feature maps to 16×16. The second block processes these representations through 64 filters (3×3, padding=1) with identical normalization, activation, and pooling operations, yielding 8×8 feature maps. The third block employs 128 filters (3×3, padding=1) with subsequent processing, culminating in 4×4 feature maps.

These multi-dimensional features (*batch size*, 128, 4, 4) are flattened into 2048-dimensional vectors. This vector is processed through a fully connected layer reducing dimensionality to 512 features, followed by batch normalisation and ReLU activation. For regularisation, a Dropout layer with a probability of 0.4-0.6 is applied to the 512 features. Finally, the output Linear layer maps these features to produce the final classification scores for each of the 2000 -3000 possible font classes.

3.2.2 Parameter Optimisation Methodology

The selection of optimal hyperparameters was systematically refined through validation set performance analysis, focusing on maximising generalisation performance. Key aspects included the optimisation strategy, learning rate schedule, regularisation techniques (dropout, L2, label smoothing), data augmentation parameters, and early stopping criteria.

To further refine the optimisation process more, we employed a OneCycleLR learning rate scheduler which implements a cyclical pattern of learning rate adjustments beginning with a starting learning rate of 0.001, progressively increasing to a maximum of 0.005, and then gradually decreasing. This oscillation helps the optimisation process escape local minima and traverse saddle points more effectively than a constant learning rate would allow.

Label smoothing with a factor of 0.1 was applied to the cross-entropy loss function. This regularisation technique prevents the model from becoming overconfident by slightly softening the target labels, which can improve generalisation.

Based on empirical evaluation across multiple runs (summarised in Table 10), a dropout rate of 0.6 often yielded strong performance, particularly when combined with aggressive data augmentation. Explicit L2 regularisation (weight decay) was explored, but results indicated it provided no significant benefit beyond dropout for the best-performing models; hence, it was set to 0 in the final selected configuration.

Comprehensive data augmentation protocols were implemented during training, as described in the preprocessing section. The optimal parameters found (associated with the 80.19% test accuracy model) involved rotations up to ±20° and translations up to ±25%.

Early stopping, based on monitoring the validation loss, was crucial for preventing overfitting. Training was terminated when the validation loss did not improve for a specified number of epochs (patience). The optimal patience varied across runs (e.g., 68 epochs for the model achieving 80.19% accuracy).

3.2.3 Training Curves

The efficacy of the implemented training methodology is quantitatively visualised in Figure 7, which illustrates the temporal evolution of both training and validation performance metrics throughout the optimisation process for a representative training run.

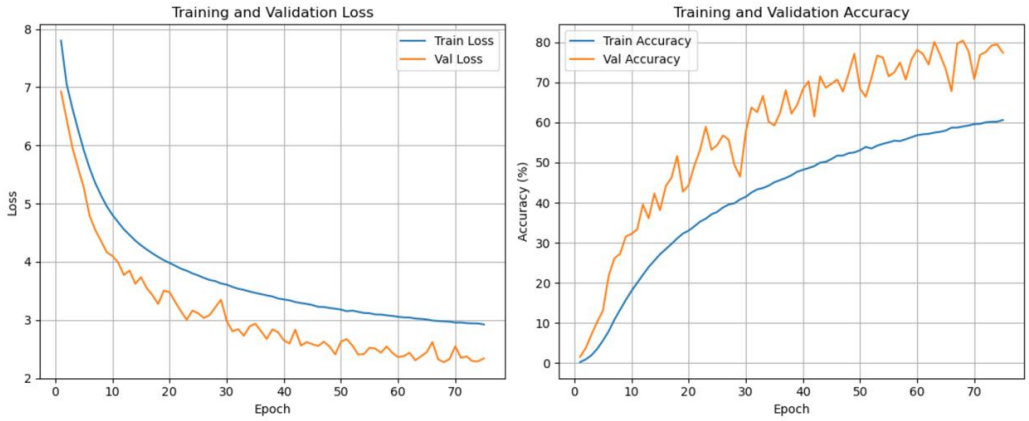


Figure 7: Training and validation loss (left) and accuracy (right) curves during model training.

Typically, both training and validation losses exhibit a downward trend, while validation accuracy increases, indicating successful learning. For the best performing model (80.19% test accuracy), the final validation accuracy reached approximately 80.41%. The fact that validation/test accuracy exceeded the final training accuracy (59% for that specific run) suggests the strong regularisation effect of dropout and data augmentation significantly enhanced generalisation to unseen data. Early stopping prevented the model from excessively fitting the augmented training data.

3.3 Evaluate the Classifier

3.3.1 Test Set Evaluation

After training and selecting the best model based on validation performance, the model was evaluated on the held-out test set. Detailed parameters and results for several experimental runs are listed in Table 10.

Table 10: Tuning Parameters and Accuracy Metrics Across Models

Model No.	Sample Size	Dropout Rate	L2 Reg.	Learn. Rate	Epochs	Patience	Train Acc.	Val Acc.	Test Acc.
1	3000	0.6	0	0.001	68	6	59.00%	80.41%	**80.19%***
2	2000	0.6	0	0.001	143	6	57.34%	77.02%	76.72%
3	2000	0.6	0	0.001	70	5	65.00%	78.00%	76.14%
4	2000	0.5	0.0001	0.001	100	7	61.08%	76.72%	75.99%
5	2000	0.4	0	0.001	52	5	58.78%	75.69%	74.88%
6	2000	0.6	0.0001	0.001	52	3	74.81%	72.01%	68.72%
7	2000	0.6	0	0.001	12	3	71.66%	65.46%	65.88%
8	2000	0.6	0	0.001	24	3	81.96%	65.45%	65.45%

Notes: * indicates the model configuration chosen based on achieving the highest test accuracy in this set of experiments (80.19%). Sample size refers to the number of unique font images used. All runs used 50 crops per image.

The best model identified (Model 1 in the table) performed well, achieving a test accuracy of 80.19%. The high top-5 accuracy observed in similar runs (e.g., 95.6% for the 80.19% model) suggests the model is often very close to the correct prediction even when the top prediction is incorrect.

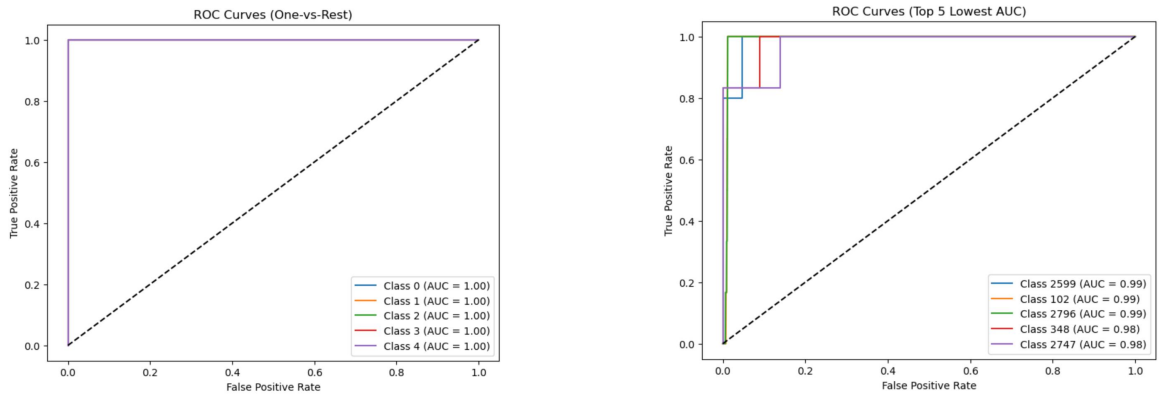
3.3.2 ROC Curve Analysis

Given the high-dimensional classification space (2000-3000 classes), Receiver Operating Characteristic (ROC) analysis provides further insight into model performance. We generated One-versus-Rest (OvR) ROC curves to assess class-specific performance, shown in Figure 8.

Figure 8a displays the OvR ROC curves for five specific classes (Class 0, 1, 2, 3, and 4). These curves demonstrate perfect classification performance for these examples, each achieving an Area Under Curve (AUC) of 1.00. The model achieved an exceptional overall micro-averaged AUC of approximately 0.9998, indicating outstanding overall discriminative capability.

Figure 8b presents the OvR ROC curves for the five classes identified with the lowest AUC values. These classes (2599, 102, 2796, 348, and 2747) still exhibit near-perfect discrimination, with AUCs of 0.99, 0.99, 0.99, 0.98, and 0.98 respectively.

The near-perfect individual ROC curves observed across different performance levels, from the examples in Figure 8a to the lowest-performing ones in Figure 8b, confirm the model's strong ability to effectively distinguish specific font categories from the rest.



(a) One-vs-Rest ROC curves (AUC $\approx$ 0.9998)

(b) ROC Curves (5 Lowest AUC)

Figure 8: ROC Curve Analysis

In conclusion, the optimised CNN successfully classifies the font crops with 80.19% test accuracy and has excellent discriminative ability (AUC $\approx$ 0.9998).

A Code

```
###################################
##   Part I: Classification Analysis
###################################

import numpy as np
import pandas as pd
import sklearn.linear_model as skl_lm
from sklearn.model_selection import train_test_split, GridSearchCV,
   StratifiedKFold
from sklearn.preprocessing import StandardScaler, LabelEncoder,
   OneHotEncoder
import sklearn.neighbors as neighbors
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.metrics import accuracy_score, classification_report,
   confusion_matrix, roc_curve, auc
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
import matplotlib.pyplot as plt
import seaborn as sns
import warnings

warnings.filterwarnings('ignore', category=UserWarning, module='
   sklearn')
warnings.filterwarnings('ignore', category=FutureWarning, module='
   sklearn')

sns.set(style="whitegrid")

## 1. Load and Explore Data
df = pd.read_csv("Heart(1).csv")

df = df.drop('Unnamed: 0', axis=1, errors='ignore')

print(df.shape)
df.info()
print(df.describe(include='all'))
print("\n--- First 5 Rows ---")
print(df.head())

print(f"\n--- Sex ---")
print(df['Sex'].value_counts())
print(f"\n--- ChestPain ---")
print(df['ChestPain'].value_counts())
print(f"\n--- Fbs ---")
print(df['Fbs'].value_counts())
print(f"\n--- RestECG ---")
print(df['RestECG'].value_counts())
print(f"\n--- ExAng ---")
print(df['ExAng'].value_counts())
print(f"\n--- Slope ---")
print(df['Slope'].value_counts())
print(f"\n--- Thal ---")
print(df['Thal'].value_counts())
print(f"\n--- AHD ---")
```

```
print(df['AHD'].value_counts())
print(f"\n--- Ca ---")
print(df['Ca'].value_counts())

## 2. Data Preprocessing

print(df.isnull().sum())
df['Ca'] = pd.to_numeric(df['Ca'], errors='coerce')
df_cleaned = df.dropna().copy()
print("\nMissing Values After Handling:")
print(df_cleaned.isnull().sum())
print(f"\nShape after dropping NaNs: {df_cleaned.shape}")

y = df_cleaned['AHD']
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
target_classes = label_encoder.classes_
positive_class_label = 'Yes'
positive_class_encoded = label_encoder.transform([positive_class_label
    ])[0]
print(f"\nTarget variable 'AHD' encoded. Classes: {target_classes}.
    Positive class '{positive_class_label}' is {positive_class_encoded
    }.")

X_features = df_cleaned.drop('AHD', axis=1)

numerical_features_to_scale = ['Age', 'RestBP', 'Chol', 'MaxHR', '
    Oldpeak']
categorical_features = ['Sex', 'ChestPain', 'Fbs', 'RestECG', 'ExAng',
     'Slope', 'Thal', 'Ca']
print(f"\nNumerical features to scale: {numerical_features_to_scale}")
print(f"Categorical features to encode: {categorical_features}")

X_features['Sex'] = X_features['Sex'].astype(str)
X_features['Fbs'] = X_features['Fbs'].astype(str)
X_features['RestECG'] = X_features['RestECG'].astype(str)
X_features['ExAng'] = X_features['ExAng'].astype(str)
X_features['Slope'] = X_features['Slope'].astype(str)
X_features['Ca'] = X_features['Ca'].astype(str)

X_train, X_test, y_train, y_test = train_test_split(
    X_features,
    y_encoded,
    test_size=0.25,
    random_state=42,
    stratify=y_encoded
)
print(f"\nData split into Training set ({X_train.shape[0]} samples)
    and Test set ({X_test.shape[0]} samples).")

numeric_transformer = Pipeline(steps=[('scaler', StandardScaler())])
categorical_transformer = Pipeline(steps=[('onehot', OneHotEncoder(
    drop='first', sparse_output=False, handle_unknown='ignore'))])
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numerical_features_to_scale),
```

```
        ('cat', categorical_transformer, categorical_features)
    ],
    remainder='drop'
)


X_train_processed = preprocessor.fit_transform(X_train)
X_test_processed = preprocessor.transform(X_test)

processed_feature_names = preprocessor.get_feature_names_out()
X_train_processed = pd.DataFrame(X_train_processed, columns=
   processed_feature_names, index=X_train.index)
X_test_processed = pd.DataFrame(X_test_processed, columns=
   processed_feature_names, index=X_test.index)

print(f"Processed training data shape: {X_train_processed.shape}")
print(f"Processed test data shape: {X_test_processed.shape}")

## 3. Model Training, Tuning, and Evaluation

cv_strategy = StratifiedKFold(n_splits=5, shuffle=True, random_state
   =42)
print(f"\nCross-validation strategy: {cv_strategy}")

results = {}

# --- 3.1 Logistic Regression ---

log_reg_model = skl_lm.LogisticRegression(random_state=42, solver='
   liblinear', max_iter=1000)
param_grid_lr = {
    'C': np.logspace(-3, 3, 7),
    'penalty': ['l1', 'l2'],
    'class_weight': [None, 'balanced']
}
grid_search_lr = GridSearchCV(log_reg_model, param_grid_lr, cv=
   cv_strategy, scoring='accuracy', n_jobs=-1, refit=True)
grid_search_lr.fit(X_train_processed, y_train)
lr_best_model = grid_search_lr.best_estimator_
pred_lr = lr_best_model.predict(X_test_processed)
prob_lr = lr_best_model.predict_proba(X_test_processed)[:,
   positive_class_encoded]
acc_lr = accuracy_score(y_test, pred_lr)
cm_lr = confusion_matrix(y_test, pred_lr)
report_lr = classification_report(y_test, pred_lr, target_names=
   target_classes)
fpr_lr, tpr_lr, _ = roc_curve(y_test, prob_lr, pos_label=
   positive_class_encoded)
auc_lr = auc(fpr_lr, tpr_lr)
results['LR'] = {
    'model': lr_best_model, 'best_params': grid_search_lr.best_params_
       ,
    'best_cv_score': grid_search_lr.best_score_, 'test_accuracy':
       acc_lr,
    'predictions': pred_lr, 'probabilities': prob_lr,
    'fpr': fpr_lr, 'tpr': tpr_lr, 'auc': auc_lr,
```

```
    'confusion_matrix': cm_lr, 'report': report_lr
}
print(f"Best Parameters: {results['LR']['best_params']}")
print(f"Best CV Accuracy: {results['LR']['best_cv_score']:.4f}")
print(f"Test Set Accuracy: {results['LR']['test_accuracy']:.4f}")
print(f"Test Set AUC: {results['LR']['auc']:.4f}")

# --- 3.2 K-Nearest Neighbors (KNN) ---

knn_model = neighbors.KNeighborsClassifier()
param_grid_knn = {
    'n_neighbors': range(1, 30, 2),
    'weights': ['uniform', 'distance'],
    'metric': ['minkowski', 'manhattan']
}
grid_search_knn = GridSearchCV(knn_model, param_grid_knn, cv=
   cv_strategy, scoring='accuracy', n_jobs=-1, refit=True)
grid_search_knn.fit(X_train_processed, y_train)
knn_best_model = grid_search_knn.best_estimator_
pred_knn = knn_best_model.predict(X_test_processed)
prob_knn = knn_best_model.predict_proba(X_test_processed)[:,
   positive_class_encoded]
acc_knn = accuracy_score(y_test, pred_knn)
cm_knn = confusion_matrix(y_test, pred_knn)
report_knn = classification_report(y_test, pred_knn, target_names=
   target_classes)
fpr_knn, tpr_knn, _ = roc_curve(y_test, prob_knn, pos_label=
   positive_class_encoded)
auc_knn = auc(fpr_knn, tpr_knn)
results['KNN'] = {
    'model': knn_best_model, 'best_params': grid_search_knn.
       best_params_,
    'best_cv_score': grid_search_knn.best_score_, 'test_accuracy':
       acc_knn,
    'predictions': pred_knn, 'probabilities': prob_knn,
    'fpr': fpr_knn, 'tpr': tpr_knn, 'auc': auc_knn,
    'confusion_matrix': cm_knn, 'report': report_knn
}
print(f"Best parameters found: {results['KNN']['best_params']}")
print(f"Best CV accuracy: {results['KNN']['best_cv_score']:.4f}")
print(f"Test Set Accuracy: {results['KNN']['test_accuracy']:.4f}")
print(f"Test Set AUC: {results['KNN']['auc']:.4f}")

# --- 3.3 Linear Discriminant Analysis (LDA) ---

lda_model = LinearDiscriminantAnalysis()
param_grid_lda = [
    {'solver': ['svd'], 'shrinkage': [None]},
    {'solver': ['lsqr', 'eigen'], 'shrinkage': [None, 'auto'] + list(
       np.linspace(0.1, 0.9, 5))},
]
grid_search_lda = GridSearchCV(lda_model, param_grid_lda, cv=
   cv_strategy, scoring='accuracy', n_jobs=-1, refit=True)
grid_search_lda.fit(X_train_processed, y_train)
lda_best_model = grid_search_lda.best_estimator_
pred_lda = lda_best_model.predict(X_test_processed)
```

```
prob_lda = lda_best_model.predict_proba(X_test_processed)[:,
    positive_class_encoded]
acc_lda = accuracy_score(y_test, pred_lda)
cm_lda = confusion_matrix(y_test, pred_lda)
report_lda = classification_report(y_test, pred_lda, target_names=
    target_classes)
fpr_lda, tpr_lda, _ = roc_curve(y_test, prob_lda, pos_label=
    positive_class_encoded)
auc_lda = auc(fpr_lda, tpr_lda)
results['LDA'] = {
    'model': lda_best_model, 'best_params': grid_search_lda.
        best_params_,
    'best_cv_score': grid_search_lda.best_score_, 'test_accuracy':
        acc_lda,
    'predictions': pred_lda, 'probabilities': prob_lda,
    'fpr': fpr_lda, 'tpr': tpr_lda, 'auc': auc_lda,
    'confusion_matrix': cm_lda, 'report': report_lda
}
print(f"Best parameters found: {results['LDA']['best_params']}")
print(f"Best CV accuracy: {results['LDA']['best_cv_score']:.4f}")
print(f"Test Set Accuracy: {results['LDA']['test_accuracy']:.4f}")
print(f"Test Set AUC: {results['LDA']['auc']:.4f}")


## 4. Results Comparison

# --- 4.1 ROC Curve Comparison ---

res_lr = results['LR']
plt.plot(res_lr['fpr'], res_lr['tpr'], lw=2, label=f"LR (AUC = {res_lr
    ['auc']:.4f})")
res_knn = results['KNN']
plt.plot(res_knn['fpr'], res_knn['tpr'], lw=2, label=f"KNN (AUC = {
    res_knn['auc']:.4f})")
res_lda = results['LDA']
plt.plot(res_lda['fpr'], res_lda['tpr'], lw=2, label=f"LDA (AUC = {
    res_lda['auc']:.4f})")

plt.plot([0, 1], [0, 1], linestyle='--', color='grey', label='Chance (
    AUC = 0.5)')
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity/Recall)')
plt.title('ROC Curve Comparison (Test Set)')
plt.legend(loc='lower right')
plt.grid(True)
plt.axis('square')
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.show()

print(f"AUC Score (LR): {results['LR']['auc']:.4f}")
print(f"AUC Score (KNN): {results['KNN']['auc']:.4f}")
print(f"AUC Score (LDA): {results['LDA']['auc']:.4f}")

# --- 4.2 Confusion Matrix Comparison ---
print("\n" + "="*30)
```

```
print("--- Confusion Matrix Comparison ---")
print("="*30)
fig, axes = plt.subplots(1, 3, figsize=(18, 5), sharey=True)
colors = ['Blues', 'Greens', 'Oranges']

# Plot LR CM
name_lr, res_lr = 'LR', results['LR']
cm_lr_plot = res_lr['confusion_matrix']
sns.heatmap(cm_lr_plot, annot=True, fmt='d', cmap=colors[0], ax=axes
    [0], cbar=False, xticklabels=target_classes, yticklabels=
    target_classes, square=True)
axes[0].set_title(f'{name_lr}\nTest Acc: {res_lr["test_accuracy"]:.3f}
     | AUC: {res_lr["auc"]:.3f}')
axes[0].set_xlabel('Predicted Label')
axes[0].set_ylabel('True Label')
# Plot KNN CM
name_knn, res_knn = 'KNN', results['KNN']
cm_knn_plot = res_knn['confusion_matrix']
sns.heatmap(cm_knn_plot, annot=True, fmt='d', cmap=colors[1], ax=axes
    [1], cbar=False, xticklabels=target_classes, yticklabels=
    target_classes, square=True)
axes[1].set_title(f'{name_knn}\nTest Acc: {res_knn["test_accuracy"]:.3
    f} | AUC: {res_knn["auc"]:.3f}')
axes[1].set_xlabel('Predicted Label')
# Plot LDA CM
name_lda, res_lda = 'LDA', results['LDA']
cm_lda_plot = res_lda['confusion_matrix']
sns.heatmap(cm_lda_plot, annot=True, fmt='d', cmap=colors[2], ax=axes
    [2], cbar=False, xticklabels=target_classes, yticklabels=
    target_classes, square=True)
axes[2].set_title(f'{name_lda}\nTest Acc: {res_lda["test_accuracy"]:.3
    f} | AUC: {res_lda["auc"]:.3f}')
axes[2].set_xlabel('Predicted Label')
plt.tight_layout()
plt.show()

## 5. Identify Best Model and Detailed Analysis
# --- 5.1 Identify Best Model ---
print("\n" + "="*30)
print("--- Accuracy Comparison ---")
print("="*30)
print("Test Set Accuracies:")

print(f"- LR: {results['LR']['test_accuracy']:.6f}")
print(f"- KNN: {results['KNN']['test_accuracy']:.6f}")
print(f"- LDA: {results['LDA']['test_accuracy']:.6f}")

best_model_name = 'LDA'

# --- 5.2 Analysis of LDA ---
final_best_model_results = results[best_model_name]
final_best_model = final_best_model_results['model']
final_cm = final_best_model_results['confusion_matrix']

print(pd.DataFrame(final_cm, index=target_classes, columns=
    target_classes))
```

```
tn, fp, fn, tp = final_cm.ravel()

print(f"\nTrue Negatives (TN): {tn} (Correctly predicted '{
    target_classes[0]}')")
print(f"False Positives (FP): {fp} (Incorrectly predicted '{
    target_classes[1]}' - Type I Error)")
print(f"False Negatives (FN): {fn} (Incorrectly predicted '{
    target_classes[0]}' - Type II Error)")
print(f"True Positives (TP): {tp} (Correctly predicted '{
    target_classes[1]}')")

sensitivity = tp / (tp + fn)
specificity = tn / (tn + fp)
precision = tp / (tp + fp)
f1_score = 2 * (precision * sensitivity) / (precision + sensitivity)

print(f"\nSensitivity (Recall for '{positive_class_label}'): {
    sensitivity:.4f}")
print(f"Specificity (Recall for '{target_classes[0]}'): {specificity
    :.4f}")
print(f"Precision (for '{positive_class_label}'): {precision:.4f}")
print(f"F1-Score (for '{positive_class_label}'): {f1_score:.4f}")

print("\nFull Classification Report (LDA):\n",
    final_best_model_results['report'])

## 6. Prediction on New Patient Data
patient_data = {
    'Age': 55, 'Sex': '0', 'ChestPain': 'typical', 'RestBP': 130, '
        Chol': 246,
    'Fbs': '0', 'RestECG': '2', 'MaxHR': 150, 'ExAng': '1', 'Oldpeak':
         1.0,
    'Slope': '2', 'Ca': '0', 'Thal': 'normal'
}

new_patient_df = pd.DataFrame([patient_data])

new_patient_df_ordered = new_patient_df[X_features.columns]

new_patient_processed = preprocessor.transform(new_patient_df_ordered)

# --- Prediction ---
pred_class_encoded = final_best_model.predict(new_patient_processed)
    [0]
pred_label = label_encoder.inverse_transform([pred_class_encoded])[0]
pred_probs = final_best_model.predict_proba(new_patient_processed)[0]

pred_prob_positive = pred_probs[positive_class_encoded]

print(f"\nPredicted Probability of Heart Disease ('{
    positive_class_label}'): {pred_prob_positive:.4f}")
print(f"Predicted Class for the patient: {pred_label} (Encoded as: {
    pred_class_encoded})")

###################################
```

```
# ## Part II: Regression Analysis
###################################

# #### Data Preprocessing

# In[3]:


#Set WD
import os
#PC os.chdir(r"C:\Users\qq18295\OneDrive - University of Bristol\TB2\
   ML TB2\Group Coursework")
os.chdir('/Users/thomastrainor-gilham/Library/CloudStorage/OneDrive-
   UniversityofBristol/TB2/ML TB2/Group Coursework')


# In[9]:


import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split, cross_val_score,
    KFold
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.linear_model import LassoCV, Lasso
import matplotlib.pyplot as plt
import seaborn as sns

# Load the data
#mac: df = pd.read_csv('/Users/thomastrainor-gilham/Library/
   CloudStorage/OneDrive-UniversityofBristol/TB2/ML TB2/Group
   Coursework/CW Data/Credit.csv')
df = pd.read_csv('/Users/thomastrainor-gilham/Library/CloudStorage/
   OneDrive-UniversityofBristol/TB2/ML TB2/Group Coursework/CW Data/
   Credit.csv')
df.head()


# In[11]:


# Display summary statistics for continuous variables
print("Continuous Variables Summary:")
print(df.describe())

# Display unique values for potential categorical variables
potential_categorical_cols = ['Cards', 'Education', 'Gender', 'Student
   ', 'Married', 'Ethnicity']
for col in potential_categorical_cols:
    print(f"\nColumn: {col}")
    print(df[col].value_counts(dropna=False))


# In[13]:
```

```
# Check for missing values
print("Missing Observations by column:")
print(df.isna().sum())

# Remove rows with missing values
df_clean = df.dropna().copy()

print("\nShape before cleaning:", df.shape)
print("Shape after cleaning:", df_clean.shape)


# In[15]:


# Drop ID column
df_clean_v2 = df_clean.drop(columns=['ID'])

# Convert Categorical columns to dummies- drop first to avoid
    multicollinearity
df_clean_v2 = pd.get_dummies(df_clean_v2, columns=['Gender', 'Student
    ', 'Married', 'Ethnicity'], dtype=int, drop_first=True)
df_clean_v2.head()


# In[17]:


from sklearn.preprocessing import StandardScaler, PolynomialFeatures

# define continuous variables
continuous_vars = ['Income', 'Limit', 'Rating', 'Cards', 'Age', '
    Education']

# create Standardised df
scaler = StandardScaler()
X_cont_scaled = scaler.fit_transform(df_clean_v2[continuous_vars])

# Create DataFrame for interaction terms [xij * xik]
X_cont_df = pd.DataFrame(X_cont_scaled, columns=continuous_vars)

# Create categorical dummy variables df
X_cat = df_clean_v2.drop(columns=['Balance'] + continuous_vars)

# Create final full df with interaction terms
X_full = pd.concat([X_cont_df, X_cat.reset_index(drop=True)], axis=1)
X_full.shape


# In[19]:


# Generate interaction terms
poly = PolynomialFeatures(degree=2, interaction_only=True,
    include_bias=False)
X_poly = poly.fit_transform(X_full)
```

```
# Create interaction column names
feature_names = poly.get_feature_names_out(X_full.columns)

# Create Final DataFrames with interaction terms
X_std = pd.DataFrame(X_poly, columns=feature_names)
y = df['Balance']

#Standardise y for coefficient paths analysis
#scaler_y = StandardScaler()
#y_std = scaler_y.fit_transform(y.values.reshape(-1, 1)).ravel()

X_std.head()


# In[21]:


X_std.shape


# In[23]:


from sklearn.model_selection import train_test_split

# Split the data (70% training, 30% testing)
X_train, X_test, y_train, y_test = train_test_split(X_std,
                                                    y,
                                                    test_size=0.3,
                                                    random_state=0)

print(f"Training sample size: {X_train.shape[0]}, Testing sample size:
    {X_test.shape[0]}")


# #### Part II. 1. (b)  "Conduct the estimation of the model using
   lasso. In doing so, we want to choose the optimal lambda using a 5-
   fold CV"

# In[26]:


from sklearn.linear_model import LassoCV, Lasso

kfold = KFold(5,
              random_state=0,
              shuffle=True)

# Fit Lasso regression model with 5-fold CV
lasso_cv = LassoCV(cv=kfold,
                   random_state=0)

lasso_cv.fit(X_std, y)
# The model will be trained on 4 folds and tested on the remaining
   fold, repeating this process 5 times
```

```
# so that each fold is used as a test set once.

# Find Optimal lambda
optimal_lambda = lasso_cv.alpha_
print(f"Optimal lambda: {optimal_lambda}")


# In[28]:


# Assuming X_std is your feature matrix and feature_names are the
    columns of X_std
feature_names = X_std.columns  # Adjust this if you're using NumPy
    array or something else

# Get coefficients corresponding to the optimal lambda
coefficients = lasso_cv.coef_

# Create a DataFrame to display feature names and their coefficients
coeff_df = pd.DataFrame({
    'Feature': feature_names,
    'Coefficient': coefficients
})

# Sort the coefficients by absolute value to see which features are
    most influential
coeff_df_top12 = coeff_df.reindex(coeff_df.Coefficient.abs().
    sort_values(ascending=False).index).head(12)
coeff_df_top12


# In[30]:


from sklearn.linear_model import ElasticNet

#Extract the array of coefficients corresponding to the solutions
    along the regularization path

# Create 100 lambdas (alphas) to test
lambdas = 10**np.linspace(8, -2, 100) / y.std()

# Run Lasso path (l1_ratio = 1.0 = pure Lasso)
soln_array = ElasticNet.path(X_std,
                             y,
                             l1_ratio=1.0,
                             alphas=lambdas,
                             max_iter=10000)[1]

soln_array.shape


# In[35]:


soln_path = pd.DataFrame(soln_array.T,
```

```
                          columns=X_std.columns,
                          index=-np.log(lambdas))
soln_path.index.name = 'negative log(lambda)'
soln_path


# In[37]:


import matplotlib.pyplot as plt
from matplotlib.pyplot import subplots

# Define the variables you want to plot
continuous_vars = ['Income', 'Limit', 'Rating', 'Cards', 'Age', '
   Education']

# Create the plot
path_fig, ax = subplots(figsize=(10, 8))

# Plot only the selected variables
for col in soln_path.columns:
    if col in continuous_vars:
        ax.plot(soln_path.index, soln_path[col], label=col, linewidth
           =2, linestyle='--')
    else:
        ax.plot(soln_path.index, soln_path[col], color='gray',
           linewidth=0.5, alpha=0.5)

# Labeling
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Standardised coefficients', fontsize=20)
ax.legend(loc='upper left')


# In[43]:


# Determine MSE of Lasso with lambda chosen by 5-fold cross-validation
   :
lasso_mse = np.min(lasso_cv.mse_path_.mean(1))
lasso_mse_sqrt = lasso_mse ** 0.5
print("Lasso MSE Square Root:", lasso_mse_sqrt)


# In[45]:


# plot of the cross-validation error

lassoCV_fig, ax = subplots(figsize=(8,8))
ax.errorbar(-np.log(lasso_cv.alphas_),
            lasso_cv.mse_path_.mean(1),
            yerr=lasso_cv.mse_path_.std(1) / np.sqrt(5))
ax.axvline(-np.log(lasso_cv.alpha_), c='k', ls='--')
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Cross-validated MSE', fontsize=20);
```

```
# In[47]:


#Calculate what % of coefficients that are exactly zero due to the
   Lasso regularization (L1 penalty)
coeffs = lasso_cv.coef_

# Calculate the percentage of zero coefficients
zero_coeffs = np.sum(coeffs == 0)
total_coeffs = len(coeffs)
percentage_zero = (zero_coeffs / total_coeffs) * 100

print(f"{percentage_zero:.3f}% of the coefficients are shrunk to zero
   under optimal lambda.")


# #### Cross-Validation MSE vs. log(lambda) Plot

# In[50]:


import matplotlib.pyplot as plt

mse_path = lasso_cv.mse_path_.mean(axis=1)
alphas = lasso_cv.alphas_

plt.figure(figsize=(8, 6))
plt.plot(-np.log(alphas), mse_path)
plt.axvline(-np.log(optimal_lambda), color='red', linestyle='--',
   label='Optimal lambda')
plt.xlabel('-log(lambda)')
plt.ylabel('Mean CV Error')
plt.title('Cross-Validation Error vs. -log(lambda)')
plt.legend()
plt.grid(True)
plt.show()


# In[56]:


from sklearn.linear_model import lasso_path

# lasso_path returns a list of alpha values
# and the corresponding coefficients for each alpha
# Get the lasso path
alphas, coefs, _ = lasso_path(X_std, y)

plt.figure(figsize=(10,6))

#plot just continuous vars
continuous_vars = ['Income', 'Limit', 'Rating', 'Cards', 'Age', '
   Education']
```

```
for i in range(coefs.shape[0]):
    colname = X_std.columns[i]
    if colname in continuous_vars:
        plt.plot(alphas, coefs[i], label=colname, linewidth=2,
            linestyle='-')
    else:
        plt.plot(alphas, coefs[i], color='gray', linewidth=0.5, alpha
            =0.5)

plt.xscale('log')
#plt.gca().invert_xaxis()
plt.axvline(optimal_lambda, color='black', linestyle='--', label='
    Optimal lambda (Min CV Error)')
plt.xlabel(r'$\lambda$')
plt.ylabel('Standardised Coefficients')
plt.title('Lasso Coefficient Paths (Standardised)')
plt.legend()
plt.grid(True)
plt.show()


# In[62]:


# Create a DataFrame with one row
person = pd.DataFrame([{
    'Income': 100,
    'Limit': 6000,
    'Rating': 500,
    'Cards': 3,
    'Age': 70,
    'Education': 12,
    'Gender_Female': 1,
    'Student_Yes': 0,
    'Married_Yes': 1,
    'Ethnicity_Asian': 1,
    'Ethnicity_Caucasian': 0
}])

# Standardise continuous variables using the same scaler
continuous_vars = ['Income', 'Limit', 'Rating', 'Cards', 'Age', '
    Education']
person_scaled = scaler.transform(person[continuous_vars])
person_cont_df = pd.DataFrame(person_scaled, columns=continuous_vars)

# Merge with dummies
person_cat = person.drop(columns=continuous_vars)
person_full = pd.concat([person_cont_df, person_cat], axis=1)
# Ensure person_full has columns in same order as training data
person_full = person_full.reindex(columns=X_full.columns, fill_value
    =0)


# Match feature set
person_poly = poly.transform(person_full)
```

```
#Double check interaction terms generated correctly
print("person_poly shape:", person_poly.shape)
print("training X_poly shape:", X_poly.shape)

# Predict
predicted_balance = lasso_cv.predict(person_poly)
print(f"Predicted Balance: ${predicted_balance[0]:.2f}")


# ## Part II. Question 2: Tree-based Methods

# In[66]:


from sklearn.tree import (DecisionTreeRegressor, plot_tree,
   export_text)
from sklearn.model_selection import train_test_split


#Create test-train split
X_train, X_test, y_train, y_test = train_test_split(X_std,
                                                    y,
                                                    test_size=0.3,
                                                    random_state=0)

# Single tree
tree_md3 = DecisionTreeRegressor(max_depth=3,
                                 criterion='squared_error',
                                 random_state=0)
tree_md3.fit(X_train, y_train)

# Predict on test
y_pred_tree_md3 = tree_md3.predict(X_test)
test_mse_tree_md3 = np.mean((y_test - y_pred_tree_md3)**2)
print("Decision Tree (max depth=3) Test MSE:", test_mse_tree_md3)
test_mse_sqrt = test_mse_tree_md3 ** 0.5
print("Decision Tree (max depth=3) Test MSE Square Root:",
   test_mse_sqrt)


# In[50]:


from sklearn.tree import plot_tree

# Visualize the tree
ax = subplots(figsize=(19,10))[1]
plot_tree(tree_md3,
          feature_names=X_std.columns,
          filled=True,
          rounded=True,
          fontsize=9,
          ax=ax);
plt.title("Decision Tree (max depth=3)")
plt.show()
```

```
# In[68]:


from sklearn.ensemble import RandomForestRegressor

n_trees_list = [1, 5, 10, 50, 100, 200]

test_results = {
    'n_trees': [],
    'max_depth': [],
    'test_mse': []
}

for n in n_trees_list:
    # 1) RF with max_depth=3
    rf_md3 = RandomForestRegressor(n_estimators=n,
                                   max_depth=3,
                                   random_state=42)
    rf_md3.fit(X_train, y_train)
    y_pred_rf_md3 = rf_md3.predict(X_test)
    mse_md3 = np.mean((y_test - y_pred_rf_md3)**2)

    test_results['n_trees'].append(n)
    test_results['max_depth'].append('3')
    test_results['test_mse'].append(mse_md3)

    # 2) RF with no max depth
    rf_unlim = RandomForestRegressor(n_estimators=n,
                                     random_state=42)
    rf_unlim.fit(X_train, y_train)
    y_pred_rf_unlim = rf_unlim.predict(X_test)
    mse_unlim = np.mean((y_test - y_pred_rf_unlim)**2)

    test_results['n_trees'].append(n)
    test_results['max_depth'].append('None')
    test_results['test_mse'].append(mse_unlim)

# Convert results to a DataFrame
df_test_results = pd.DataFrame(test_results)

# For convenience, also add the single tree result
df_test_results = pd.concat([
    df_test_results,
    pd.DataFrame({'n_trees':[0], 'max_depth':['3 (SingleTree)'], '
       test_mse':[test_mse_tree_md3]})
], ignore_index=True)

# Plot all test MSE
plt.figure(figsize=(8,6))
sns.lineplot(data=df_test_results, x='n_trees', y='test_mse', hue='
   max_depth', marker='o')
plt.title("Test MSE for Random Forests (max_depth=3 vs. None) + Single
    Tree")
plt.xlabel("Number of Trees (n_estimators)")
plt.ylabel("Test MSE")
```

```
plt.legend(title="Max Depth", loc="best")
plt.show()


# In[70]:


df_test_results.sort_values(by='test_mse')


# In[72]:


# Identify best row
best_row = df_test_results.loc[df_test_results['test_mse'].idxmin()]
best_n_trees = best_row['n_trees']
best_max_depth = best_row['max_depth']
print("Best model:\n", best_row)

# Fit that best model
if best_max_depth == 'None':
    best_max_depth = None  # interpret string to actual None
elif best_max_depth == '3 (SingleTree)':
    # In that edge case, it's not a random forest but a single
       decision tree
    # We'll treat that separately:
    rf_best = tree_md3
else:
    best_max_depth = int(best_max_depth)

if best_max_depth != '3 (SingleTree)':
    rf_best = RandomForestRegressor(
        n_estimators=int(best_n_trees),
        max_depth=best_max_depth if best_max_depth is not None else
           None,
        random_state=0
    )
    rf_best.fit(X_train, y_train)

# Reuse the code from Part 1(c) to assemble the data for the same
   person
# (assuming same feature engineering steps, standardized columns, etc
   .)
# We'll call it "person\_X" to be consistent with the fitted model's
   input structure

person_pred_balance_rf = rf_best.predict(person_poly)
print(f"Random Forest predicted Balance for that person: ${
   person_pred_balance_rf[0]:.2f}")


# In[76]:


#RF with 200 trees, no max depth
rf_200_nd = RandomForestRegressor(n_estimators=200,
```

```
                                        random_state=0)
rf_200_nd.fit(X_train, y_train)
y_pred_rf_200_nd = rf_200_nd.predict(X_test)
mse_rf_200_nd = np.mean((y_test - y_pred_rf_200_nd)**2)

pred_balance_rf_200_nd = rf_200_nd.predict(person_poly)
print(f"Random Forest predicted Balance for that person: ${
   pred_balance_rf_200_nd[0]:.2f}")
test_mse_rf_200_sqrt = mse_rf_200_nd ** 0.5
print("Test MSE Square Root:", test_mse_rf_200_sqrt)


# ### Plot Variable Importance:
#

# In[60]:

feature_imp = pd.DataFrame(
    {'importance':rf_200_nd.feature_importances_},
    index=feature_names)
top_12_features=feature_imp.sort_values(by='importance', ascending=
   False).head(12)
top_12_features

# In[78]:

#Get raw importances
importances = rf_best.feature_importances_

# Normalize to the max = 100
normalised_importances = 100 * importances / importances.max()

indices = np.argsort(normalised_importances)[::-1]  # sort descending
feature_names = X_std.columns

# Take only the top 12 features
top_n = 12
top_indices = indices[:top_n]  # Get indices of top 12 features
top_importances = normalised_importances[top_indices]  # Get
   importance values for top 12

plt.figure(figsize=(10, 8))
plt.barh([feature_names[i] for i in top_indices[::1]], top_importances
   [::1], color='red')
plt.title("Top 12 Random Forest Variable Importances")
plt.xlabel("Variable Importance (Mean Decrease in Node Impurity
   Expressed Relative to Max)")
plt.ylabel("Variables")
plt.tight_layout()
plt.show()

###################################
##   Part III: Image classification
###################################

#!/usr/bin/env python
```

```
# coding: utf-8

# In[1]:


import os
import zipfile

#Set WD
#PC os.chdir(r"C:\Users\qq18295\OneDrive - University of Bristol\TB2\
   ML TB2\Group Coursework")
os.chdir('/Users/thomastrainor-gilham/Library/CloudStorage/OneDrive-
   UniversityofBristol/TB2/ML TB2/Group Coursework')

# Step 1: Define paths
zip_path = '/Users/thomastrainor-gilham/Library/CloudStorage/OneDrive-
   UniversityofBristol/TB2/ML TB2/Group Coursework/Font_Images.zip'
extract_dir = '/Users/thomastrainor-gilham/Library/CloudStorage/
   OneDrive-UniversityofBristol/TB2/ML TB2/Group Coursework/
   font_images'

# Step 2: Create extraction directory
os.makedirs(extract_dir, exist_ok=True)

# Step 3: Extract the ZIP file
with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_dir)

print(f"Files extracted to: {extract_dir}")
print(f"Number of files extracted: {len(os.listdir(extract_dir))}")

#If zip file already extracted
import os

extract_dir = '/Users/thomastrainor-gilham/Library/CloudStorage/
   OneDrive-UniversityofBristol/TB2/ML TB2/Group Coursework/
   font_images'
inner_dir = os.path.join(extract_dir, 'generated_pangrams')
image_dir = inner_dir  # Point to where the .bmp files are

# List all .bmp files
all_images = [f for f in os.listdir(image_dir) if f.endswith('.bmp')]
print("Total images found:", len(all_images))  # Should be 192383
print("Sample images:", all_images[:5])

# In[43]:

import random
import numpy as np
from PIL import Image
import torch
import torchvision.transforms as transforms
from torchvision.transforms import ToTensor
from torch.utils.data import TensorDataset, DataLoader
import torch.nn as nn
import torch.optim as optim
```

```
from torchinfo import summary
from torch.utils.data import Dataset

# Set random seeds for reproducibility
random.seed(42)
np.random.seed(42)
torch.manual_seed(42)

# List all .bmp files and randomly select a subset
all_files = [f for f in os.listdir(image_dir) if f.endswith('.bmp')]
num_fonts = 3000  # Subset size
selected_files = random.sample(all_files, num_fonts)

# Define target crop size, number of crops and step size:
CROP_SIZE = 32
TARGET_CROPS = 50
STEP = 12

# AUGMENTATION STRENGTH ADJUSTMENT:
DEGREES = (-15, 15)
TRANSLATE = (0.25, 0.25)
BRIGHTNESS = 0
CONTRAST = 0

#TUNING PARAMETERS
DROPOUT = 0.6
EPOCHS = 200
L2_REG = 0.000
PATIENCE = 7

# Assign unique numeric label to each font
font_labels = {file: idx for idx, file in enumerate(selected_files)}

# Modified crop.py function to generate multiple 32x32pix crops
def generate_crops(img, crop_size=CROP_SIZE, step=STEP,
   min_black_pixels=20, target_crops=TARGET_CROPS):
    # Extract two lines
    raw_pix = np.array(img)
    line_crops = []
    for pg_line in [0, 1]:
        pix = raw_pix[pg_line * 200:200 + pg_line * 200, :]
        topcol = np.argmax((np.argmax(pix != 255, axis=0) > 0))
        botcol = np.argmax((np.argmax(np.flip(pix, axis=(0, 1)) !=
           255, axis=0) > 0))
        toprow = np.argmax((np.argmax(pix != 255, axis=1) > 0))
        botrow = np.argmax((np.argmax(np.flip(pix, axis=(0, 1)) !=
           255, axis=1) > 0))
        pix = pix[toprow:200 - botrow, topcol:400 - botrow]
        line_crops.append(pix)

    # Generate and deduplicate candidate crops
    candidate_crops = []
    seen = set()
    for pix in line_crops:
        H, W = pix.shape
        if H < crop_size:
```

```
            pix = np.pad(pix, ((0, crop_size - H), (0, 0)), mode='
                constant', constant_values=255)
        elif H > crop_size:
            pix = pix[:crop_size, :]
        H = crop_size

        for x in range(0, max(W - crop_size + 1, 1), step):
            if x + crop_size <= W:
                crop = pix[0:crop_size, x:x + crop_size]
            else:
                # Corrected Padding:  Pad on BOTH sides to center the
                    crop
                pad_left = (crop_size - W) // 2
                pad_right = crop_size - W - pad_left
                crop = np.pad(pix[0:crop_size, 0:W], ((0, 0), (
                    pad_left, pad_right)),
                              mode='constant', constant_values=255)
            num_black = np.sum(crop < 255)
            if num_black >= min_black_pixels:
                crop_tuple = tuple(crop.flatten())
                if crop_tuple not in seen:
                    seen.add(crop_tuple)
                    candidate_crops.append(crop)

    result_crops = candidate_crops
    return result_crops[:target_crops]

# Applies cropping to all fonts
font_crops = {}
for file in selected_files:
    img = Image.open(os.path.join(image_dir, file))
    crops = generate_crops(img)
    font_crops[file] = crops

# Split crops into train, validation, and test sets (70% train, 15%
    val, 15% test)
train_crops, val_crops, test_crops = [], [], []
train_labels, val_labels, test_labels = [], [], []

to_tensor = transforms.ToTensor()
for file in selected_files:
    crops = font_crops[file]
    label = font_labels[file]
    random.shuffle(crops)
    N = len(crops)
    train_end = int(0.7 * N)  # 70% = 35 crops
    val_end = train_end + int(0.15 * N)  # 15% = 7-8 crops

    train_crops.extend(crops[:train_end]) # Store NumPy arrays
    val_crops.extend(crops[train_end:val_end]) # Store NumPy arrays
    test_crops.extend(crops[val_end:]) # Store NumPy arrays

    train_labels.extend([label] * train_end)
    val_labels.extend([label] * (val_end - train_end))
    test_labels.extend([label] * (N - val_end))
# Define transforms
```

```
train_transform = transforms.Compose([
    transforms.RandomRotation(degrees=DEGREES),
    transforms.RandomAffine(degrees=0, translate=TRANSLATE),
    transforms.RandomAffine(degrees=0, scale=(0.9, 1.1)),
    transforms.ColorJitter(brightness=BRIGHTNESS, contrast=CONTRAST)
])


# Convert label lists to PyTorch tensors
train_labels = torch.tensor(train_labels)
val_labels = torch.tensor(val_labels)
test_labels = torch.tensor(test_labels)

class FontCropDataset(Dataset):
    def __init__(self, crops_list, labels, transform=None):
        self.crops = crops_list # Expecting a list of NumPy arrays
        self.labels = labels
        self.transform = transform
        self.to_tensor = transforms.ToTensor() # ToTensor will be
           applied here

    def __len__(self):
        return len(self.crops)

    def __getitem__(self, idx):
        crop_np = self.crops[idx] # Get NumPy array
        label = self.labels[idx]

        # Convert NumPy array to PIL Image FIRST (this fixes the error
           )
        # Assuming images are grayscale (L mode)
        crop_pil = Image.fromarray(crop_np.astype(np.uint8), mode='L')

        # Apply transformations (augmentation for train, None for
           others)
        if self.transform:
            crop_pil = self.transform(crop_pil)

        # Convert the (optionally augmented) PIL Image to a Tensor
        crop_tensor = self.to_tensor(crop_pil)

        return crop_tensor, label

# Create datasets with transforms
batch_size = 64
train_dataset = FontCropDataset(train_crops, train_labels, transform=
   train_transform)
val_dataset = FontCropDataset(val_crops, val_labels, transform=None)
test_dataset = FontCropDataset(test_crops, test_labels, transform=None
   )

train_loader = DataLoader(train_dataset, batch_size=batch_size,
   shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
test_loader = DataLoader(test_dataset, batch_size=batch_size)
```

```
print(f"Training samples: {len(train_dataset)}")
print(f"Validation samples: {len(val_dataset)}")
print(f"Test samples: {len(test_dataset)}")


# In[ ]:


import matplotlib.pyplot as plt
import copy
from torch.optim.lr_scheduler import OneCycleLR


# Define 3-layer CNN model (Conv2d -> BatchNorm -> ReLU -> MaxPool x
    3)
class FontCNN(nn.Module):
    def __init__(self, num_classes=num_fonts):
        super(FontCNN, self).__init__()
        # Input size: [batch_size, 1, 32, 32] (grayscale)
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1) #
            Output: 32x32x32
        self.bn1 = nn.BatchNorm2d(32)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2, 2) # Output: 16x16x32

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1) #
            Output: 16x16x64
        self.bn2 = nn.BatchNorm2d(64)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2, 2) # Output: 8x8x64

        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1) #
            Output: 8x8x128
        self.bn3 = nn.BatchNorm2d(128)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(2, 2) # Output: 4x4x128

        self.flatten = nn.Flatten() # Output: 128 * 4 * 4 = 2048

        self.fc1 = nn.Linear(128 * 4 * 4, 512) # Input: 2048, Output:
            512
        self.bn4 = nn.BatchNorm1d(512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(DROPOUT)

        self.fc2 = nn.Linear(512, num_classes) # Output: num_classes

    def forward(self, x):
        # Applying Conv -> BN -> ReLU -> Pool pattern
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.bn2(x)
```

```
        x = self.relu2(x)
        x = self.pool2(x)

        x = self.conv3(x)
        x = self.bn3(x)
        x = self.relu3(x)
        x = self.pool3(x)

        x = self.flatten(x)

        # Applying Linear -> BN -> ReLU pattern
        x = self.fc1(x)
        x = self.bn4(x)
        x = self.relu4(x)

        x = self.dropout(x)
        x = self.fc2(x)

        return x


# Instantiate the model
model = FontCNN(num_classes=num_fonts)

# Display model summary
X_sample = next(iter(train_loader))[0]  # Sample batch
summary(model, input_data=X_sample, col_names=['input_size', '
    output_size', 'num_params'])

# Define loss function, optimiser, and onecycle scheduler
criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=
    L2_REG) #Adjust
scheduler = OneCycleLR(optimizer, max_lr=0.005, steps_per_epoch=len(
    train_loader), epochs=EPOCHS)

# Training loop with metrics tracking
def train_model(model, train_loader, val_loader,
                criterion, optimizer,
                num_epochs,
                patience,              # how many epochs to wait
                minimize_loss=True):     # True: early-stop on loss,
                    False: on accuracy

    device = torch.device("cuda" if torch.cuda.is_available() else "
        cpu")
    model.to(device)

    best_val_metric = float('inf') if minimize_loss else 0.0
    best_model_wts = copy.deepcopy(model.state_dict())
    epochs_no_improve = 0

    train_losses, val_losses = [], []
    train_accs,  val_accs   = [], []

    for epoch in range(num_epochs):
```

```
        # Training Phase
        model.train()
        running_loss, correct, total = 0.0, 0, 0
        for x, y in train_loader:
            x, y = x.to(device), y.to(device)
            optimizer.zero_grad()
            preds = model(x)
            loss  = criterion(preds, y)
            loss.backward()
            optimizer.step()
            scheduler.step()
            running_loss += loss.item() * x.size(0)
            _, p = preds.max(1)
            correct += (p == y).sum().item()
            total   += y.size(0)

        epoch_train_loss = running_loss / total
        epoch_train_acc  = correct / total * 100
        train_losses.append(epoch_train_loss)
        train_accs.append(epoch_train_acc)

        #  Validation Phase
        model.eval()
        val_loss, val_correct, val_total = 0.0, 0, 0
        with torch.no_grad():
            for x, y in val_loader:
                x, y = x.to(device), y.to(device)
                preds = model(x)
                loss  = criterion(preds, y)
                val_loss += loss.item() * x.size(0)
                _, p = preds.max(1)
                val_correct += (p == y).sum().item()
                val_total   += y.size(0)

        epoch_val_loss = val_loss / val_total
        epoch_val_acc  = val_correct / val_total * 100
        val_losses.append(epoch_val_loss)
        val_accs.append(epoch_val_acc)

        #  Check for improvement
        if minimize_loss:
            improved = epoch_val_loss < best_val_metric
            current = epoch_val_loss
        else:
            improved = epoch_val_acc > best_val_metric
            current = epoch_val_acc

        if improved:
            best_val_metric = current
            best_model_wts = copy.deepcopy(model.state_dict())
            torch.save(best_model_wts, "best_model.pth")  #Save best
               model
            epochs_no_improve = 0
        else:
            epochs_no_improve += 1
```

```
        print(f"Epoch {epoch+1}/{num_epochs} | "
              f"Train Loss: {epoch_train_loss:.4f}, Acc: {
                 epoch_train_acc:.2f}% | "
              f"Val  Loss: {epoch_val_loss:.4f}, Acc: {epoch_val_acc
                 :.2f}% | "
              f"NoImprove: {epochs_no_improve}/{patience}")

        if epochs_no_improve >= patience:
            print(f"Early stopping triggered. Restoring best model
               from epoch {epoch+1-epochs_no_improve}.")
            break

    # restore best weights before returning
    model.load_state_dict(best_model_wts)

    # Save best model to file
    torch.save(model.state_dict(), "best_model.pth")

    # Plot learning curves
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    #Make length of epochs equal where early stopping cutout
    epochs = range(1, len(train_losses) + 1)

    # Align lengths
    train_accs = train_accs[:len(epochs)]
    val_accs = val_accs[:len(epochs)]

    # Loss plot
    ax1.plot(epochs, train_losses, label='Train Loss')
    ax1.plot(epochs, val_losses, label='Val Loss')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss')
    ax1.set_title('Training and Validation Loss')
    ax1.legend()
    ax1.grid(True)

    # Accuracy plot
    ax2.plot(epochs, train_accs, label='Train Accuracy')
    ax2.plot(epochs, val_accs, label='Val Accuracy')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Accuracy (%)')
    ax2.set_title('Training and Validation Accuracy')
    ax2.legend()
    ax2.grid(True)

    plt.tight_layout()
    plt.show()

    return train_losses, val_losses, train_accs, val_accs

# Run training and get metrics
train_losses, val_losses, train_accs, val_accs = train_model(
    model, train_loader, val_loader, criterion, optimizer, num_epochs=
       EPOCHS, patience=PATIENCE
)
```

```
# In[41]:


from sklearn.metrics import accuracy_score, roc_curve, auc
import matplotlib.pyplot as plt
import torch
from collections import Counter


# TEST MODEL
def evaluate_model(model, test_loader):
    device = torch.device("cuda" if torch.cuda.is_available() else "
       cpu")
    model.eval()
    all_preds = []
    all_labels = []
    all_probs = []

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            probs = nn.functional.softmax(outputs, dim=1)
            all_probs.extend(probs.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    all_probs = np.array(all_probs)
    all_labels = np.array(all_labels)
    all_preds = np.argmax(all_probs, axis=1)


    accuracy = accuracy_score(all_labels, all_preds)
    print(f"Test Accuracy: {accuracy * 100:.2f}%")

     # Check samples per class
    label_counts = Counter(all_labels)
    #print("Number of samples per class in test set:", label_counts)


    # ROC Curve (One-vs-Rest for multi-class)
    n_classes = num_fonts
    fpr = {}
    tpr = {}
    roc_auc = {}
    for i in range(n_classes):
        # Binarize labels for class i
        labels_bin = [1 if l == i else 0 for l in all_labels]
        fpr[i], tpr[i], _ = roc_curve(labels_bin, [p[i] for p in
           all_probs])
        roc_auc[i] = auc(fpr[i], tpr[i])

    # Plot ROC curves for a few classes (first 5)
    plt.figure(figsize=(8, 6))
    for i in range(min(5, n_classes)):
```

```
    plt.plot(fpr[i], tpr[i], label=f'Class {i} (AUC = {roc_auc[i
        ]:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves (One-vs-Rest)')
plt.legend(loc='best')
plt.show()

# After calculating roc_auc
# Convert roc_auc dictionary to a list of (class, auc) pairs and
    sort
auc_sorted = sorted(roc_auc.items(), key=lambda x: x[1], reverse=
    True)  # Descending order

# Select top 5 (highest AUC) and bottom 5 (lowest AUC)
n_to_plot = 5
highest_auc_classes = auc_sorted[:n_to_plot]  # Top 5
lowest_auc_classes = auc_sorted[-n_to_plot:]  # Bottom 5

# Plot highest AUC classes
plt.figure(figsize=(8, 6))
for class_idx, auc_value in highest_auc_classes:
    plt.plot(fpr[class_idx], tpr[class_idx], label=f'Class {
        class_idx} (AUC = {auc_value:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves (Top 5 Highest AUC)')
plt.legend(loc='best')
plt.show()

# Plot lowest AUC classes
plt.figure(figsize=(8, 6))
for class_idx, auc_value in lowest_auc_classes:
    plt.plot(fpr[class_idx], tpr[class_idx], label=f'Class {
        class_idx} (AUC = {auc_value:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves (Top 5 Lowest AUC)')
plt.legend(loc='best')
plt.show()

# Micro-average ROC curve
labels_onehot = np.zeros((len(all_labels), n_classes))
for idx, label in enumerate(all_labels):
    labels_onehot[idx, label] = 1
fpr_micro, tpr_micro, _ = roc_curve(labels_onehot.ravel(), np.
    array(all_probs).ravel())
roc_auc_micro = auc(fpr_micro, tpr_micro)
print("Micro-average AUC:", roc_auc_micro)

macro_auc = np.mean(list(roc_auc.values()))
print("Macro-average AUC:", macro_auc)
```

```
    plt.figure(figsize=(8, 6))
    plt.plot(fpr_micro, tpr_micro, label=f'Micro-average (AUC = {
       roc_auc_micro:.2f})')
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Micro-Average ROC Curve')
    plt.legend(loc='best')
    plt.show()

    # Check Top 5 & 10 Accuracy
    probs_tensor = torch.tensor(all_probs)
    top5_preds = torch.topk(probs_tensor, 5, dim=1).indices
    top10_preds = torch.topk(probs_tensor, 10, dim=1).indices

    all_labels_tensor = torch.tensor(all_labels)

    top5_correct = sum([label in top5 for label, top5 in zip(
       all_labels_tensor, top5_preds)])
    top10_correct = sum([label in top10 for label, top10 in zip(
       all_labels_tensor, top10_preds)])

    top5_acc = top5_correct / len(all_labels)
    top10_acc = top10_correct / len(all_labels)

    print(f"Top-5 Accuracy: {top5_acc * 100:.2f}%")
    print(f"Top-10 Accuracy: {top10_acc * 100:.2f}%")

# Evaluate the best model from training
model.load_state_dict(torch.load("best_model.pth"))
evaluate_model(model, test_loader)


# In[45]:

# Function to visualise a sample of crops
def visualize_crops(font_crops, num_samples_per_font=3,
   num_fonts_to_show=5):

    # Select a subset of fonts
    selected_fonts = list(font_crops.keys())[:num_fonts_to_show]

    # Set up the plot
    fig, axes = plt.subplots(num_fonts_to_show, num_samples_per_font,
                             figsize=(num_samples_per_font * 3,
                                num_fonts_to_show * 3))
    fig.suptitle("Sample Crops from Font Images", fontsize=16)

    for i, font_file in enumerate(selected_fonts):
        crops = font_crops[font_file]
        # Ensure we don't exceed available crops
        num_crops = min(num_samples_per_font, len(crops))
        if num_crops == 0:
            print(f"No crops available for {font_file}")
            continue
```

```
        sample_crops = crops[:num_crops]  # Take the first few crops
        for j in range(num_samples_per_font):
            ax = axes[i, j] if num_fonts_to_show > 1 else axes[j]
            if j < num_crops:
                # Display the crop (grayscale image)
                ax.imshow(sample_crops[j], cmap='gray', interpolation
                    ='none')
                ax.set_title(f"Font: {font_file.split('.')[0]}\nCrop {
                    j+1}")
            else:
                ax.axis('off')  # Hide empty subplots
            ax.set_xticks([])
            ax.set_yticks([])

    plt.tight_layout(rect=[0, 0, 1, 0.95])  # Adjust layout to fit
        title
    plt.show()

# Call the visualisation function after generating crops
visualize_crops(font_crops, num_samples_per_font=50, num_fonts_to_show
    =5)


# In[ ]:
```

B Contribution Breakdown

Leo	Developed the code and description for Part 1, contributed to the brainstorming and development process for Part 3, and managed the final document compilation, including proofreading and revisions.
Nandini	Contributed to the writing and structuring of Part 2 of the code and report. Supported the coding efforts where required and suggested some model options for Part 3.
Thomas	Helped code and answer Parts 2 and 3. Built the model we used for Part 3
Yitong	Conduct part1 code review and correct. Build part of part3 model, and responsible for the writing of part3.

Table 11: Member Contributions