ML Practical 1

PART 1

a) Loading and Cleaning Data

Loading

The first step I took was to carefully read the dataset's description to fully understand its features and how we intend to use them. To load the data the panda's *read_csv* method was used which allowed me to read the comma-separated value file and save it in a DataFrame type (df). The given dataset file did not contain column names and as such, I created an array containing each feature's name. This array was passed in the *read_csv* method. The initial dataset contained 32 features.

Cleaning

Before the data could be used by a model, a few procedures were done to clean it:

- The **data type** of each feature and whether there were any **missing values** were determined. The personality features that will be used (indexed 1-12) all have the **same data type** (float64) and therefore do not require any data type conversions. It was also found that the dataset has **no missing values**, hence there was no need to impute or drop incomplete data.
- To increase readability, the values for the Country and Ethnicity characteristics were converted from integers to strings (mapping their original labels from the dataset). This is an important preparatory step for one-hot encoding that will be performed later (See 1 c.) to those features, as it will allow us to identify which columns represent which attributes (e.g, countryUSA).
- The values in the given dataset were previously standardised by the creators. This might be a significant problem moving forward as all the values were **standardised** based on the **mean and standard deviation of all the data**, including the ones that will be used for testing (the split did not happen yet). This is a very subtle case of **possible** data leakage, as the data came like this from the source. To fix this problem, I remapped the values for Nscore, Escore, Oscore, Ascore, and Cscore to their original recorded value (found on the UCI page). Only these features were remapped because the remaining were either encoded (at a later stage) or their values were measured using a specific metric (e.g., BIS-11 for Impulsiveness) that was not transformed by the creators. Please note that I acknowledge that if I standardise the data again (after split) the values will be recalculated. However, I still suspect that it is likely that some data leakage would remain from the original standardisation by the creators.
- All the people (rows) that said they used the Semer drug, were **removed** from the dataframe. Semer is a **fictious** drug and therefore nobody ever used it before. If they are lying that they used it, then they might be lying about the other categories as well and therefore they are dropped.
- Columns indexed 13 to 32 were dropped using panda's *drop* method. This is done since they are not personality types, and as such irrelevant to our goal. Including them would most likely result in an entirely different model and possibly a weaker classifier.
- The ID feature (index 0) was also removed because it is irrelevant to the objective - not a personality feature and has no association with whether someone uses tobacco.

b) Split Data

As soon as the data was cleaned, the immediate next step was to **split** the dataset into **training** and **testing** sets. The features (X) used are columns indexed 1-12 on the original dataset and the target (y) variable is Nicotine consumption.

Sklearn's *train_test_split* function was used to split the data, passing in the X and y values and performing an 80 per cent training data and 20 per cent testing data split. This split gives us a lot of training data (relative to how much is available) but also sufficient testing data. The stratify parameter of *train_test_split* was also used, passing in the y value. This ensures that the **proportion of values for y** in the training and testing samples produced will be the same as the proportion of values in the full dataset. The training dataset will be used to train and evaluate the algorithm using 5-fold cross-validation. The testing dataset will be used as the **final test** to check how the algorithm performs on **completely unseen** data.

c) Encoding, Conversion, Scaling

Encoding

The features Age, Gender, Education, Country, and Ethnicity were all categorical features that were transformed into numeric values. Age and Education represent relative **ordinality order** (e.g., higher education standard - higher value) and as such Ordinal Encoding was used by using sklearn's *OrdinalEncoder*. In this dataset, Gender is binary and as such Ordinal Encoding could also be used. Country and Ethnicity features, on the other hand, **should not convey any ordinality** as there is no natural order there whatsoever, yet the original data did (e.g., Australia had a lower value than the UK). This could lead to the model capturing a wrong relationship such as Australia < UK, something that would lead to false conclusions and lead to a **weaker classifier**. To fix that, one-hot encoding was used which creates additional **binary** features for each attribute of the categorical features chosen (e.g., countryUK). Panda's *get_dummies* method was used to achieve that. After these encoding transformations, our training set contained 24 features.

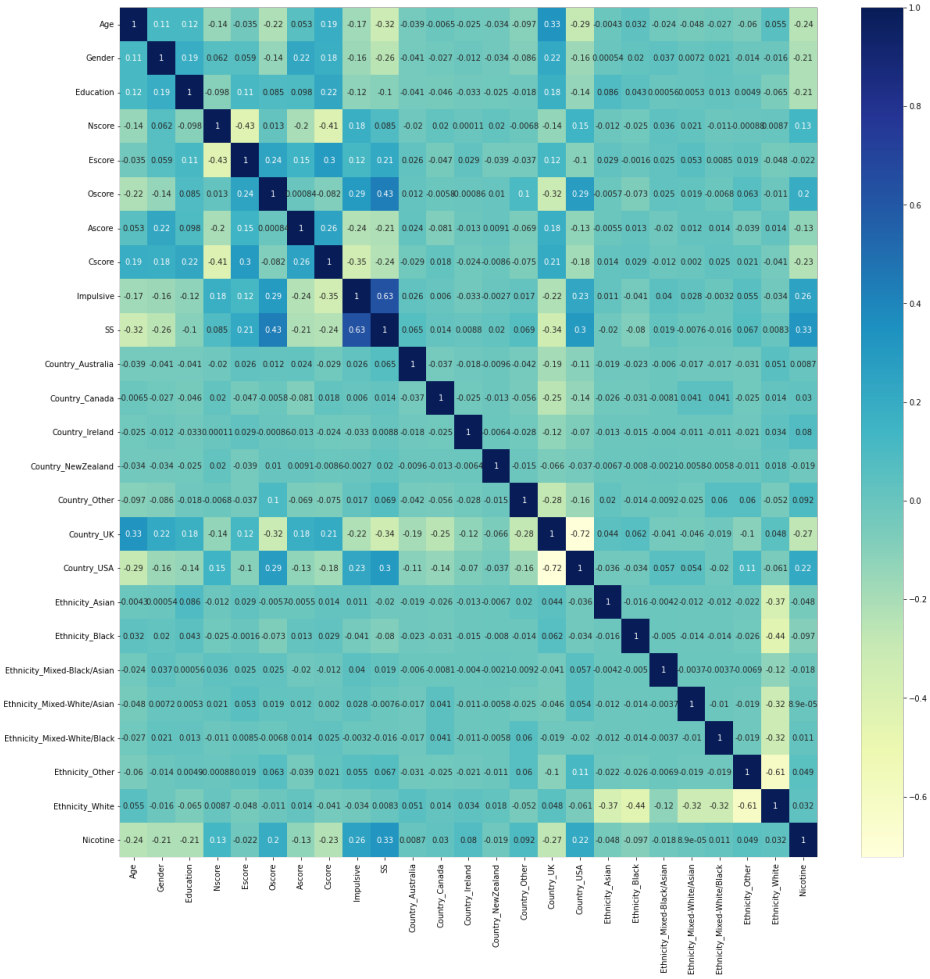


Figure 1: Correlation matrix between the features.

Feature Selection

The correlation matrix (Figure 1) shows the correlation between each pair of features in the training set, allowing us to identify which features are correlated with one another and how strong that correlation is. This is especially beneficial because some machine learning models, including logistic regression, can perform worse if the data contains strongly correlated input variables [7]. To deal with this, I generated a new correlation matrix by using both the X_train and y_train (label of training features) to compute the correlation matrix which also includes Nicotine feature. I then created a new function called *remove_highly_correlated_features*, which takes this correlation matrix and discovers pairs of features that are highly correlated (above a 50% correlation threshold) between them and then drops the feature (from the pair) that is least correlated with the Nicotine feature. This is done to ensure that the strongest correlation with the target is retained while simultaneously eliminating the strong pair correlation. Following this process, the Impulsive, Country_USA, and Ethnicity_White features are removed from the dataset (21 features remaining).

Scaling

Standardization scales each input variable separately by subtracting the mean and dividing by the standard deviation to shift the distribution to have a mean of zero and a standard deviation of one. Standardization is **not required** by Logistic Regression (unlike some more complex algorithms) as if the data is not standardised it will just select a smaller θ, getting the same slope. For this reason, I initially **did not** scale the data. However, my goal for this practical was to create a **solid methodology** that is also **future proof**. Knowing that it is common in Machine Learning to start with a simple model (e.g., Logistic Regression) and gradually explore more complex ones, I eventually decided to standardise the data as if I continued this project (in a real-life scenario), I would most likely experiment with some model that **requires** standardisation (e.g., SVM). Furthermore, another reason that I decided to use standardisation is that sklearn suggests standardising the feature matrix X (input) before fitting the model, to ensure that the penalty treats features equally [4]. Finally, it sometimes helps the algorithm converge faster [3].

To do standardization, I used sklearn's *StandardScaler* before training each algorithm, putting the whole process together in an sklearn Pipeline.

d) Data Leakage

Data leakage is when test data is leaked to the model that gives it an unrealistic advantage to make better predictions in the test data but often cannot replicate this on real-world unseen data (data not included in the testing set). An example of data leakage can be seen when standardization of input variables occurs before the data is split into training and testing sets. As mentioned, standardization necessitates first estimating the mean and standard deviation values from the domain and using them to scale the values of the variables. After the data is split into training and testing datasets, the data in the training set knows something about the data in the testing set as they have been scaled using the same mean and standard deviation. To avoid this, I split the data **immediately** after the data was cleaned and **before conducting transformation, encoding and standardisation**. Additionally, I used 5-fold cross-validation which randomly separates the training data into 5 distinct sets and then treats and evaluates the model 5 times, selecting a new fold for evaluation every time and training on the other 4 folds. This enabled the models to only **see the testing data at the very end** ("final test"), after they were trained and validated, further minimizing the risk for data leakage. The use of sklearn's pipeline also allows the sequence of data preparation steps to be performed within each cross-validation fold, further reducing cross-validation data leakage. Furthermore, as previously mentioned, at the very beginning I reversed the standardisation of the data created by the creators as it might have introduced data leakage in the whole dataset.

PART 2

a) Train using penalty='none' and class_weight=None

An initial logistic regression model was trained using penalty='none' and class_weight=None by importing and using *LogisticRegression* from sklearn. I used the *make_pipeline* function to create a pipeline that first uses *StandardScaler* to standardize the data and then passes them to Logistic Regression.

I expect the best performance we can get using a logistic regression without basis expansion to be around 50%, being able to predict most CL6 and CL0 labels correctly. In terms of other classifiers, I would expect a non-linear model to be able to perform extremely well having accuracy of more than 90%. The worst performing logistic regression would make random guesses or learn the data so wrong that it would perform even worse than random guesses. Another possibility is that we make a "dumb classifier" model that only predicts the majority class - CL6. Interestingly such model would still be right 32% of the time.

The basic model developed after 5-fold cross-validation scoring on training data, has 38.6% accuracy with a standard deviation of 1.2%. The model developed is, therefore, better than a model just predicting CL6 all the time, but it is still a weak classifier that fails to learn the features well. This was expected as the Nicotine consumption feature has a very unbalanced distribution of classes (Figure 2), and the data is non-linearly separable making a linear model like Logistic Regression highly unsuitable for such a task.

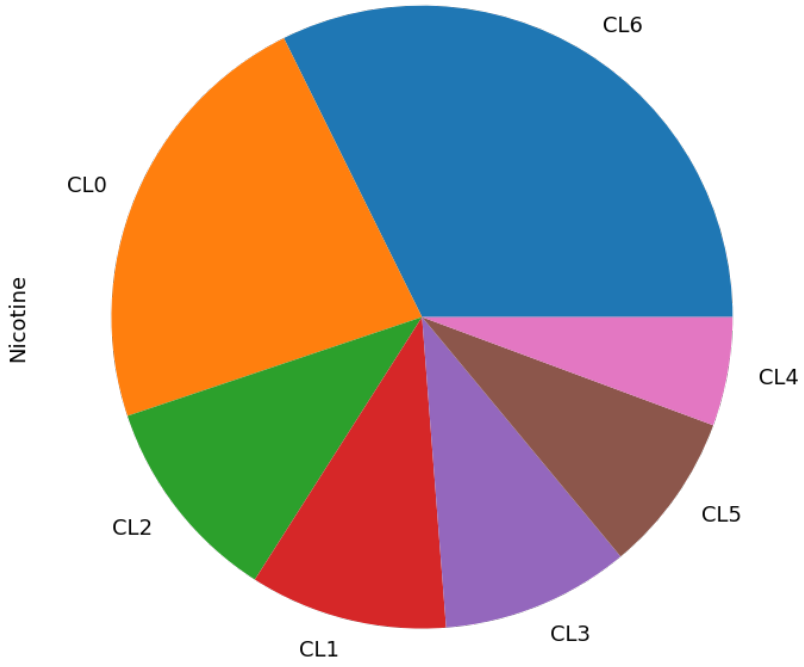


Figure 2: Imbalance of Nicotine consumption cases.

b) Logistic Regression Parameters

Penalty: The use of penalty function is a form of regularization that imposes a cost on the optimization function to make the optimal solution unique, introduce bias and decrease variance to the model [1]. The four penalty options of sklearn's Logistic Regression are none, l2, l1, and elasticnet, are explained in Table 1.

Penalty Options Logistic Regression		
Penalty	**Equation**	**Explanation**
None	-	No penalty applied.
L2	$Llog + \lambda\sum\beta j^2$	L2 is calculated as the square root of the sum of the vector values squared. The number of parameters/coefficients is j, and log is the loss function (see Equation). Therefore, when the penalty is added, the only way for the optimization technique to maintain the overall loss function as minimal as possible is to assign smaller values to the coefficients. The λ parameter controls how much emphasis is given to the penalty term. The higher the λ value, the more coefficients in the regression will be pushed towards zero. However, they will never be exactly zero, which is not desirable if we want the model to select important variables [1].
L1	$Llog + \lambda\sum \lvert\beta j\rvert$	Small modification of L2. Instead of squared values, L1 uses the absolute values of Bj. This penalty usually picks one variable at random when predictor variables are correlated. In this situation, it appears that one of the factors is unimportant, however it may still be predictive [1].
elasticnet	$Llog + \lambda p\sum(\alpha\beta_j^2 + (1-\alpha)\lvert\beta j\rvert)$	Uses both L1 and L2 penalties to get best of both worlds. The additional parameter *a* can have values between 0 and 1 and controls the weight given to L1 or L2 [1].

Table 1: The penalty options for logistic regression.

Tol: When the algorithm is training its objective is to minimize the loss. On each iteration it checks its convergence by computing the difference between its present iteration to its previous. This residual value is called tolerance and when the specified tolerance value is met by the algorithm, then it stops searching for the optimal point. If the value of tol is too large, then the algorithm may stop before it can converge resulting in a weak classifier. If we choose a too small value, then the algorithm will take more time to converge.

Max_iter: This is the maximum number of iterations that the algorithm does during optimisation – tries to converge. Max_iter should be higher when we have a very large amount of data as the default value of 100 might not be enough for the solver to converge.

c) Train using class_weight='balanced'

As mentioned earlier, the target class's frequency is highly imbalanced i.e., the occurrence of some classes is very high compared to the other classes (See Figure 2). This means that there is a bias or skewness towards the classes that have more samples (CL6 and CL0). This leads to the model having adequate information about these classes with a lot of samples but insufficient information about the rest. This is likely to cause high misclassification errors in our minority classes.

We can modify the Logistic Regression algorithm to **consider the skewed distribution** of our classes by giving **different weights** to the classes depending on their number of samples. This difference in weights will influence the classification of the classes during the training phase. It will penalize the misclassifications that Logistic Regression will make in the minority classes by setting higher weights for those classes and lower weight for the majority classes. This is useful because by giving more weightage to the minority

classes, the algorithm will provide a higher penalty to these classes and therefore reduce its errors. When the *class_weight* is assigned to 'balanced' then the model automatically assigns the class weights inversely proportional to their respective frequencies [2].

To demonstrate this, I manually calculate the weights assigned when class_weight='balanced' to the class with the most samples -CL6, and the one with the least - CL4. There are 7 target variable classes (CL0 - CL6). The total number of samples in the training dataset is 1501, from which 486 belong to CL6, and 85 to CL4. Figure 3 shows my calculations to figure out the weight assigned to CL6 and CL4 by a balanced model.

$$w_{class} = n\_samples / (n\_classes * class\_frequency)$$

$$w_{CL6} = 1501 / (7 * 486)$$

$$= 0.44 \quad (2\ d.p.)$$

$$w_{CL4} = 1501 / (7 * 85)$$

$$= 2.52 \quad (2\ d.p.)$$

Figure 3: Weights assigned for CL6 and CL4, manual calculation.

As evident, the weight assigned to CL4 (2.52) is more than 5 times bigger than the weight assigned to CL6 (0.44). Smaller weights result in a small penalty and small update to the model's coefficients. On the contrary, larger weights lead to larger penalty and updates in the model's coefficients.

A new Logistic Regression model was trained using class_weight = "balanced". The accuracy of the model decreased (now at only 24.7% with standard deviation of 2.1%) but the recall, precision, and f1 scores improved considerably. I believe that these metrics are **more suitable** than accuracy for problems with significant class imbalance, and as such our model improved.

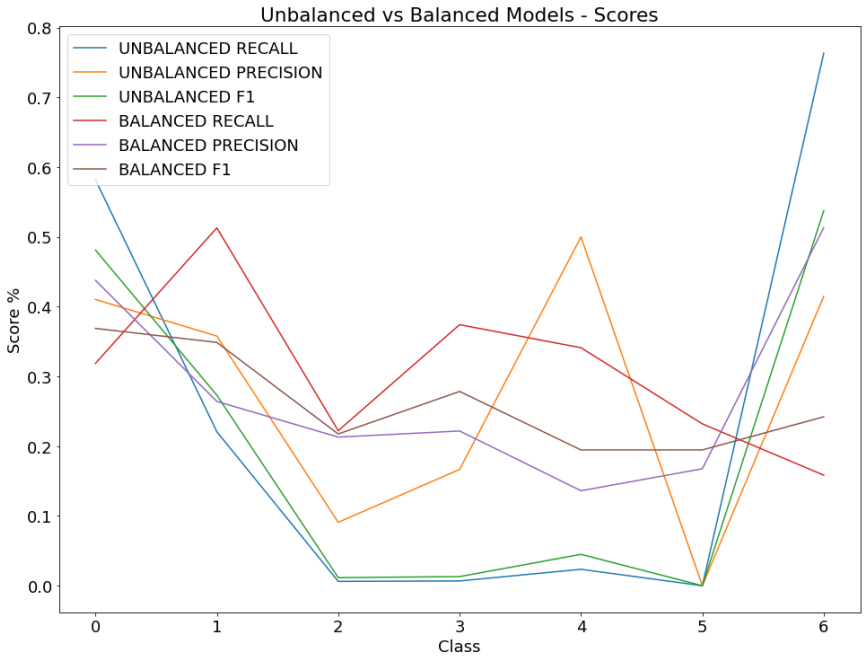


Figure 4: Unbalanced vs balanced models F1, recall, and precision scores. Training data.

Figure 4 compares recall, precision, and f1-score, between unbalanced (2.a) and balanced (2.c) models on the **training data**. As evident, the unbalanced model performed better in terms of recall and f1-score on labels with the most data (CL0 and CL6). In the minority classes (CL1-CL5), on the other hand, the balanced model demonstrated greater overall performance (f1 score - combining recall and precision). This means that the balanced model learned the minority classes' relations in the training data significantly better than the unbalanced model.

d) Classification Results Functions

The *predict* function is used to predict the **label** (class) over a new set of data (e.g., CL0). When given the X_test dataset (containing features) it returns an array containing the predicted label for each data point in the X_test. The *predict_proba* function returns an array containing the probability of the instance being in each of the classes. For example, given a single data point, it might return [0.6, 0.1, 0, 0, 0.2, 0.1, 0] which denotes that the class probability for CL0 is 0.6, for CL1 is 0.1, etcetera. The *predict* function returns the class that has the highest probability (*predict_proba)*. The *decision_function* returns an array with a "confidence score" where every element represents the distance between a predicted sample and its separating hyperplane (different hyperplanes for different labels). For example, if the array has the highest value (confidence score) at index 1 (indexes 0-6), then it means that the sample is the closest with the hyperplane of CL1.

PART 3

a) Classification Accuracy

Classification accuracy is calculated by diving the number of **correct** predictions with the **total** number of predictions. It gives us an overall measure of how much the model is correctly predicting on the entire set of data.

The formula considers the **sum** of True Positive and True Negative elements at the numerator and the sum of all entries (True Positive, True Negative, False Positive, and False Negative) of the confusion matrix at the denominator. For multi-class classification the accuracy is the average of accuracy over all classes.

$$Accuracy = \frac{TP + TN}{P + N}$$

The classification accuracy of both models on the **testing** dataset was calculated using sklearn's *classification_accuracy* method. The unbalanced model has 36.9% classification accuracy, while the balanced model has only 26.9% classification accuracy. It is not possible to identify the classes where the algorithm performs poorly using this metric. Accuracy is not a reliable metric for our problem since it assumes that the samples are equally distributed, even though the dataset contains significant class imbalance. Remember, a model just "predicting" CL6 would still get 32%.

b) Balanced Accuracy

Balanced Accuracy returns the average of recall obtained on each class. This is a better metric for unbalanced datasets like our own since it avoids inflated performance estimates. The balanced accuracy is the raw accuracy where each sample is weighted according to the inverse prevalence of its true class.

$$Balanced\ Accuracy = \frac{1}{M}\left(\sum_{m=1}^{M} \frac{TP_m}{TP_m + FN_m}\right)$$

The **balanced accuracy** of the models is 21.22% and 30.42% for the unbalanced and balanced models respectively, calculated using sklearn's *balanced_accuracy_score* function.

c) Confusion Matrix

A confusion matrix is a N x N matrix, with N being the number of classes, that categorizes the model's correct and incorrect classifications and assigns them to the appropriate label. It gives us further insight into how well the model performs when predicting each class and also where it gets confused. From the confusion matrix we can find the True-Positive, True-Negative, False-Positive, and False-Negative for each individual class.

Unbalanced

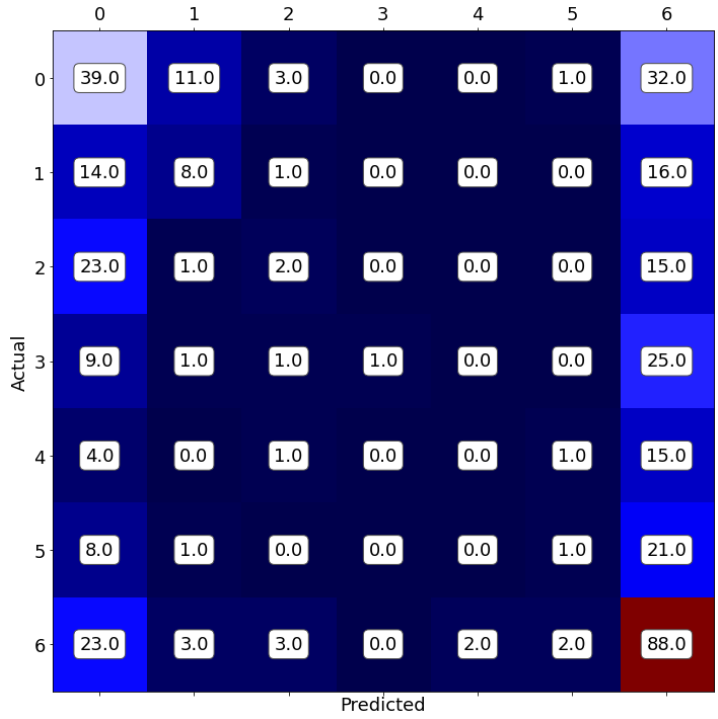


Figure 5: Confusion matrix for unbalanced model (untuned).

The unbalanced model is very skewed in predicting either CL0 or CL6 and often confuses between them (Figure 5). The model performs extremely poorly for classes 2-5, correctly classifying only 2 cases for CL2, and misclassifying everything else most of the times as either CL6 or CL0. For CL1, it performs slightly better correctly identifying 8 of 39 samples. The model performs extremely poorly and failed to capture the relationship between inputs and outputs accurately (underfitting), making a lot of mistakes, especially in the classes with less samples.

Balanced

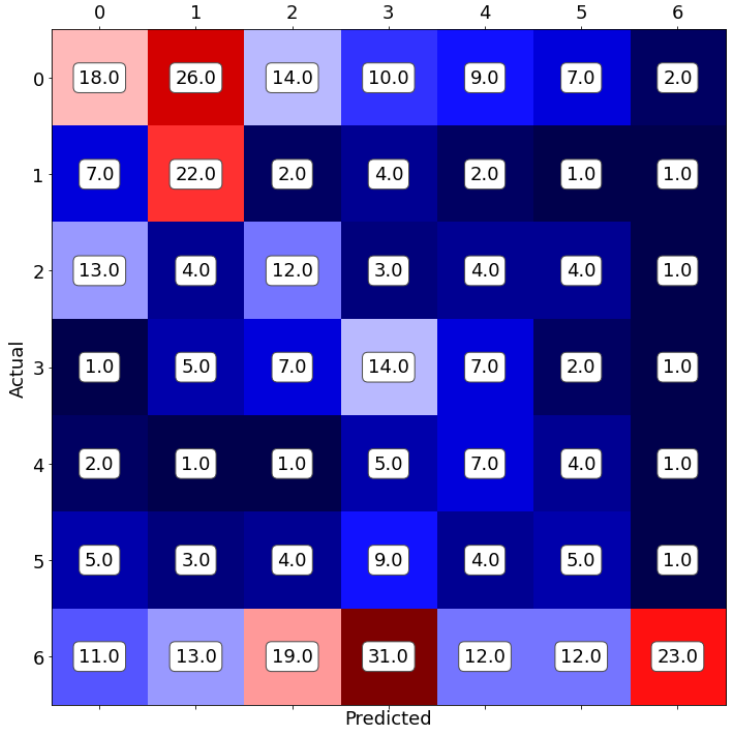


Figure 6: Confusion matrix for balanced model.

Figure 6 shows the confusion matrix for the balanced model. As immediately evident this model performs significantly better for the classes that do not have a lot of samples (CL1-CL5). However, it appears that the balanced model performs worse for the majority classes CL6 and CL0 than the unbalanced model. It now misclassifies CL6 and CL0 as CL3 and CL1 respectively, more times than it correctly classifies them.

d) Precision and Recall

$$Precision = \frac{TP}{TP + FP}$$

Precision is the proportion of true positives in comparison with the number of **total positives predicted** (true positives plus false positives).

$$Recall = \frac{TP}{TP + FN}$$

Recall is the proportion of **true positives that are correctly identified**; calculated by diving the true positives with the true positives plus the false negatives.

To calculate precision and recall for each class, the *precision_score* and *recall_score* functions were used. The micro and macro averages for each were calculated by passing in the average parameter 'micro' and 'macro' respectively. Recall and precision are often more representative of the model's performance when they are combined, creating the f1-score.

Unbalanced

	precision	recall	f1-score	support
CL0	0.33	0.45	0.38	86
CL1	0.32	0.21	0.25	39
CL2	0.18	0.05	0.08	41
CL3	1.00	0.03	0.05	37
CL4	0.00	0.00	0.00	21
CL5	0.20	0.03	0.06	31
CL6	0.42	0.73	0.53	121

Figure 7: Precision & Recall for unbalanced model

Figure 7 shows the precision, recall, f1-score, and support (number of samples) for each class. As expected CL0 and CL6 have significantly better precision and recall metrics than the other classes (due to more support). Interestingly, the model appears to pick up on the association for CL1 (weakly) as its scores are higher than those of the other classes (except CL0 and CL6), even though several of them have similar support, with CL2 having even more. Precision 1 for CL3 can be explained by considering the Precision formula, which divides the correctly identifies samples by the total number of samples predicted as this class (including wrong predictions). By looking at the confusion matrix (Figure 3) we can see that the model only predicted one CL3, which was a correct prediction, giving it a precision of 100%. The recall score of 0 for CL4 shows that no sample was correctly identified as CL4.

Precision & Recall Micro and Macro Averages – Unbalanced Model	
Metric	**Percentage**
Precision Micro Average	36.96%

Precision Macro Average	35%
Recall Micro Average	36.96%
Recall Macro Average	21.34%

Table 2: Precision and Recall Micro and Macro averages - Unbalanced model

A macro-average (both for Precision and Recall) computes the metric independently for each class and then takes the average (therefore treating all classes equally) [6]. A micro-average aggregates all classes' contribution to compute the average value [6]. If there is class imbalance in a multi-class classification setting, then the **micro-average** is preferred. To illustrate why this is the case, I did the calculations for the precision micro and macro averages by hand, to show what happens under the hood.

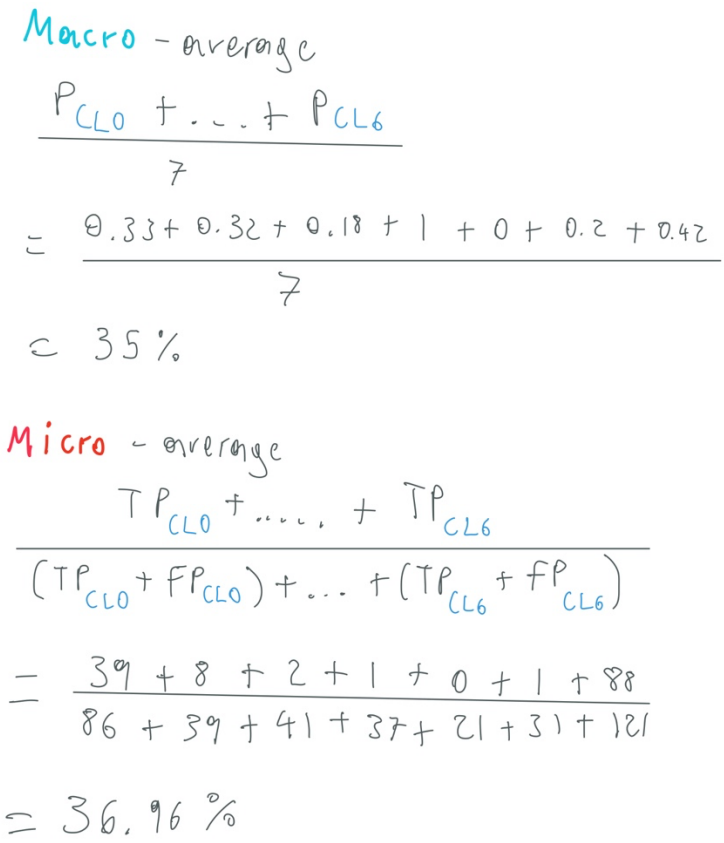


Figure 8: Precision micro and macro average manual calculation.

As evident, the 0.42 precision for CL6 used on the macro average **only contributes 1/7** towards the average **despite constituting 23.7%** of the data. This showcases how the Macro average can be unrepresentative and why Micro should be preferred when dealing with imbalanced datasets. The same of course applies for the recall metric.

Balanced

	precision	recall	f1-score	support
CL0	0.32	0.21	0.25	86
CL1	0.30	0.56	0.39	39
CL2	0.20	0.29	0.24	41
CL3	0.18	0.38	0.25	37
CL4	0.16	0.33	0.21	21
CL5	0.14	0.16	0.15	31
CL6	0.77	0.19	0.30	121

Figure 9: Precision & Recall for balanced model

When compared with the unbalanced class, a few conclusions can be made. Precision and recall improved significantly for the classes CL2-CL4, with CL4's f1-score going from 0 to 0.21. For CL1 and CL5 the recall was considerably improved but the precision got slightly worse. For CL6 the recall became much worse (0.19 from 0.73) but precision increased significantly. This means that the balanced model does not predict CL6 as often as it should but whenever it does its much more likely to be a correct prediction. For CL0 all the metrics worsened.

Precison & Recall Micro and Macro Averages – Balanced Model	
Metric	**Percentage**
Precision Micro Average	26.86%
Precision Macro Average	29.5%
Recall Micro Average	26.86%
Recall Macro Average	30.4%

Table 3: Precision and Recall Micro and Macro averages - Balanced model

The **differences** between micro and macro averages on recall have been reduced when compared with the unbalanced model. The micro averages and precision macro averages are lower than the unbalanced model, but the recall average is better.

PART 4

a) L2 Penalty

A new logistic regression model was created, passing in the L2 penalty. I chose to also use balanced class weights as they are better suited for our imbalanced data (as previously explained). This model has a 24.8% accuracy on training data, slightly lower than the balanced model developed for 2c.

Equation 1 shows the equation of the Logistic Regression cost function when using L2 penalty.

Cost function - L2

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left[-y^{(i)} \log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^{m} \theta_j^2$$

Equation 1: Logistic Regression Cost function L2

The gradient of this L2 regularized cost is calculated as shown in Equation 2.

Gradient

$$\delta_\theta J = \delta_\theta \frac{1}{m} \sum_{i=1}^{m} \left[-y^{(i)} \log(h_\theta(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right] + \delta_\theta \frac{\lambda}{2m} \sum_{j=1}^{m} \theta_j^2$$

$$\delta_\theta J = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)} + \frac{\lambda}{m} \theta$$

Equation 2: Gradient of L2.

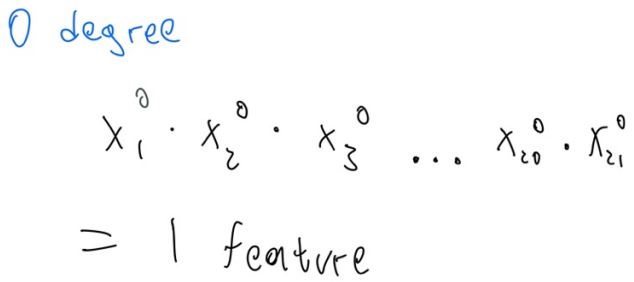
b) 2nd degree Polynomial Expansions

Logistic Regression performs best with linearly separable data. A popular approach to handle nonlinear datasets is to add more features by using polynomial feature expansion that generates new polynomial and interaction features. The new feature matrix is made up of all polynomial combinations of features that have a degree less than or equal to the given degree.

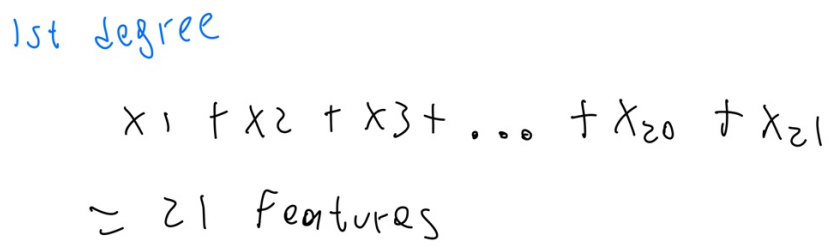
It is worth noting that at a low polynomial degree, this approach cannot handle very complex datasets, while at a high polynomial degree, it generates a large number of features, slowing down the model. To implement this sklearn's *PolynomialFeatures* function was used, specifying a 2nd degree polynomial expansion. The number of features increased from 21 to 253.

To calculate how many dimensions, it produces we must calculate every combination of those features with a degree less than or equal to 2.

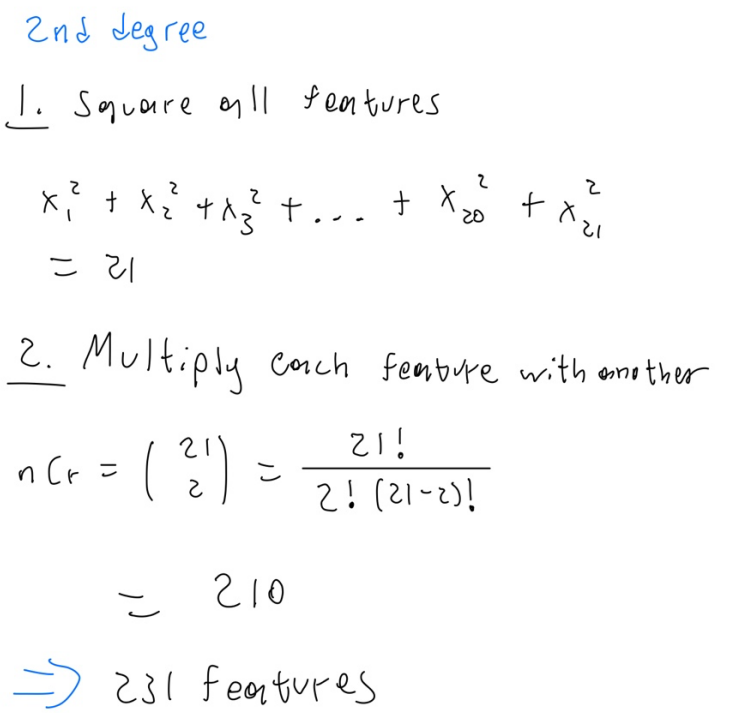
The 0 degree just adds one feature – multiplying each feature at the zero power.



The 1st degree just adds all the features in the dataset ➔ producing 21 features.



To calculate the 2nd degree, we must first square all the features (power of 2) and then use the nCr combinations formula to find all the possible feature combinations when features are multiplied between them. When added together, the 2nd degree produces 231 features.



Therefore, when all the features produced by the degrees are added together (1+21+231), we get 253 features, confirming our sklearn polynomial expansion was done correctly.

c) Regularised vs Unregularized Classifiers on Expanded Data

Four different models were developed to compare the performance of regularised and unregularized classifiers on the expanded data, summarised in Table 4.

Model	Parameters	Train Accuracy	Test Accuracy	Precision Weighted Average	Recall Weighted Average
Unregularised & Unbalanced	class_weight=None, penalty='none'	32.9%	32.71%	30%	33%
Unregularised & Balanced	class_weight='balanced', penalty='none'	23.1%	24.73%	37%	25%
Regularised & Unbalanced	class_weight=None, penalty='l2'	33.2%	33.24%	30%	33%
Regularised & Balanced	class_weight='balanced', penalty='l2'	22.7%	24.2%	35%	24%

Table 4: Summary of four different models developed for task 4c.

When the class weight is None, then the regularised model has a slightly better accuracy on both the training and testing sets. Interestingly, the weighted precision and recall values are the same. When the class weight is balanced, then when the L2 penalty is applied all the metrics being compared slightly worsen.

These results do not match my expectations as I was expecting the penalty to decrease training accuracy but improve testing accuracy. The models still do not capture the relation even after the 2nd degree polynomial expansion. Higher order (3rd or 4th degree) basis expansion might help with that but at the cost

of creating a much slower model that might overfit. Given more time, I would experiment with different degrees of polynomials.

d) Precision-Recall Plots

The precision-recall curve illustrates the trade-off between precision and recall at various thresholds [8]. To create separate precision and recall plots for each class of nicotine consumption, a function called *precision_recall_plots_each_class* was created. This function takes a model and testing data and then uses predict_proba and precision_recall curve functions to compare the model precision and recall performance for each class with the true labels.

Figure 10 shows an example of the precision recall plots for each class for the regularised and balanced model developed for 4 c.

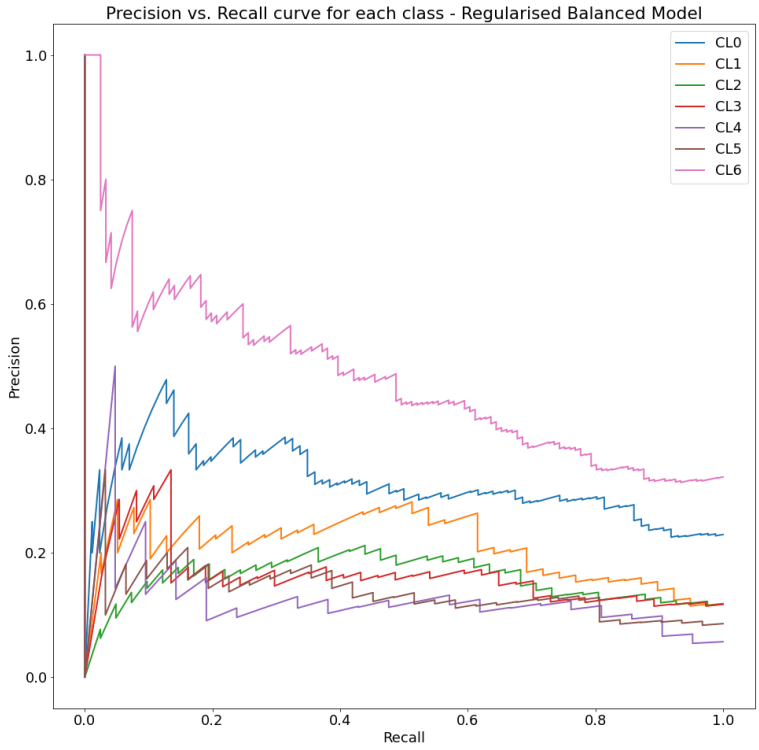


Figure 10: Precision and recall plots for regularised balanced model.

A high area under the curve indicates both high recall and high accuracy, with high precision corresponding to a low false positive rate and high recall corresponding to a low false negative rate. A good classifier would have high scores for both indicating that it is producing accurate results (high precision) and predicting the majority of all the samples (high recall) for each class [8]. From Figure 10, we can again confirm that the model preforms the best for CL6, followed by CL0 and CL1. We can also see how at lower recall thresholds, the precision values tend to be higher, tor all the classes except CL2.

e) Balanced and Unbalanced Models Hyperparameter Tuning

BayesSearchCV from scikit-optimize was used to tune the parameters for both **balanced (2a.)** and **unbalanced (2c.)** models. Often, hyperparameter tuning is done via grid search, in which every parameter combination is explored, and random search, in which parameter combinations are chosen at random. I did not use GridSearchCV as I wanted to test a relatively large grid (parameter space), which would mean that the number of parameter choices would be incredibly large (exponential growth). This is a problem due to the limited resources and time I have available. BayesSearchCV, like random search, does not check every possible combination, instead only exploring a predefined set of combinations. The distinction between Bayesian optimisation search and random search is that it **incorporates previous evaluations** when selecting the next hyperparameter set to assess. It focuses on the regions of the parameter space where it expects the validation scores will be the most promising, minimising the iterations required to arrive at the optimal hyperparameter values, reaching at the optimal solutions faster.

Table 5 shows the parameter space tried. As evident, I experimented with a wide variety of values for all the parameters to discover ones that would increase the model's performance. Fifty distinct hyperparameter combinations (n iter = 50) were tested.

Parameter Space – Bayesian Search Optimisation	
Parameter	**Values Tried**
penalty	'none', 'l1', 'l2'
Tol	0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000
Max_iter	1, 10, 20, 50, 100, 200, 400, 800, 1000
C	0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000
Multi_class	'ovr', 'multinomial'
solver	'newton-cg', 'lbfgs', 'liblinear', 'saga', 'sag'

Table 5: Parameter Space Tried.

On training data the unbalanced tuned model has a slightly better accuracy at 39.9%. Similarly, the balanced model's performance also improved after tuning, reaching an accuracy of 33.8%.

On testing data, the unbalanced model has 38.8% accuracy and 19.06% balanced accuracy. The balanced model has 34.84% accuracy and 27.52% balanced accuracy. Interestingly the tuning conducted decreased the balanced_accuracy for both models but increased the classification accuracy.

The best parameters found from the Bayesian optimisation are shown below:

Unbalanced:

```
OrderedDict([('logisticregression__C', 0.001),
('logisticregression__max_iter', 1000),
('logisticregression__multi_class', 'multinomial'),
('logisticregression__penalty', 'l2'), ('logisticregression__solver',
'lbfgs'), ('logisticregression__tol', 10.0)])
```

Balanced:

```
OrderedDict([('logisticregression__C', 0.01),
('logisticregression__max_iter', 20),
('logisticregression__multi_class', 'ovr'),
('logisticregression__penalty', 'l2'), ('logisticregression__solver',
'liblinear'), ('logisticregression__tol', 0.0001)])
```

```
CodeText
```

Please be aware that if you rerun this you will most likely get different results as bayesian search does not always find the same solutions (due to its random nature).

Performance Difference on testing data

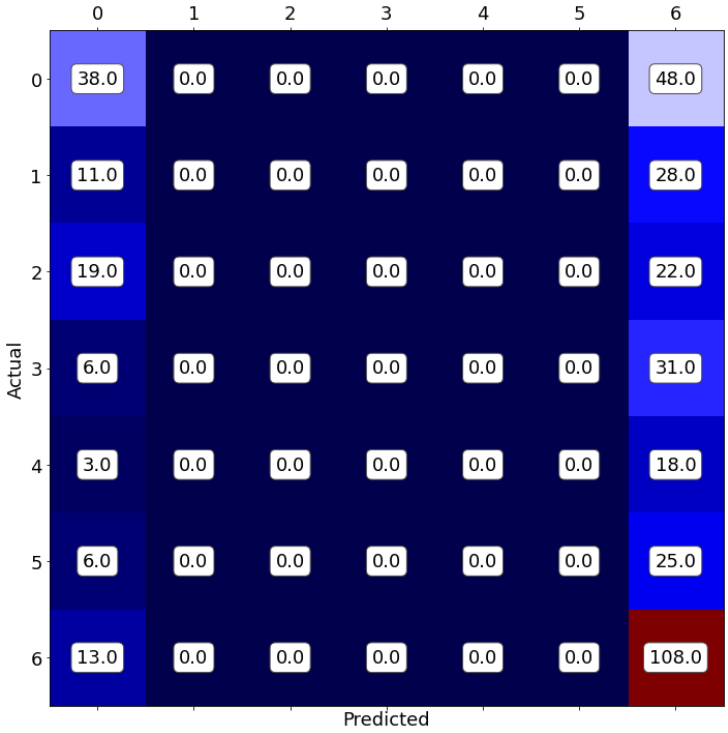


Figure 11: Confusion matrix for unbalanced tuned model.

From Figure 11 we can see that the unbalanced **tuned** model always predicts either CL0 (0) or CL6 (6). Out of 121 actual CL6 samples, it correctly identifies 108 of them, with the remaining 13 being predicted as CL0. For CL0, it correctly classifies 38 out of 86 samples, with the 48 remaining ones being misclassified as CL6. For classes CL1-CL5 it **always misclassifies** them as either CL0 or more frequently as CL6. Comparing this confusion matrix with the unbalanced untuned model (Figure 4) we can conclude that the model predicts more often CL6 than before, signalling that this was caused by the tuning and that accuracy is misleading in imbalanced datasets. Even though the tuned model has higher accuracy (due to misclassification), it is actually an even **worse** model.

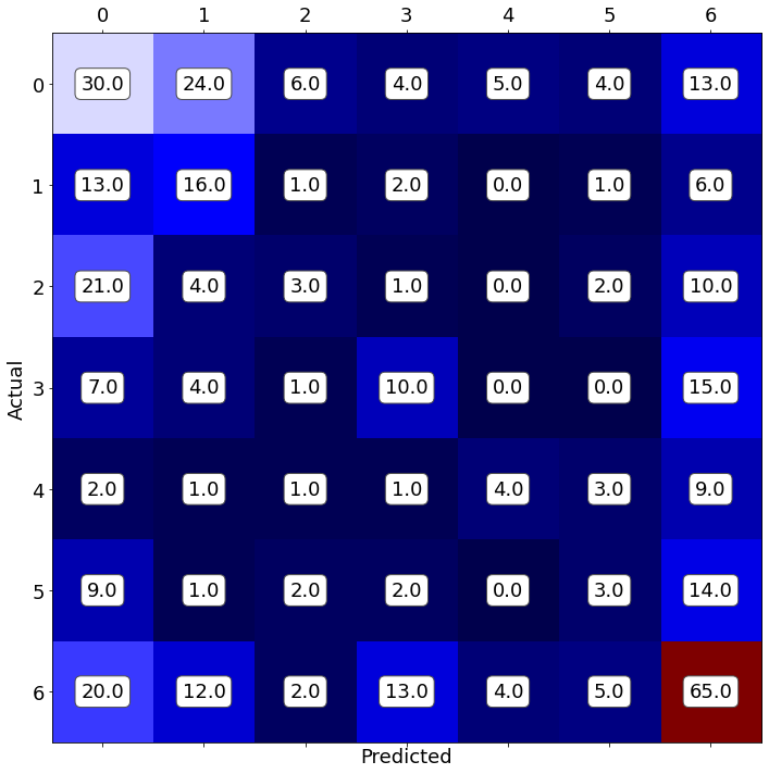


Figure 12: Confusion matrix for balanced tuned model.

When compared with the balanced model before tuning (Figure x), we can see that the model improved in correctly predicting CL0 and CL6. The model now confuses more often CL1 with CL0. Its performance decreased noticeably for CL2 as before it was correctly predicting the label 12 times and now only 3. Its performance also decreased for classes CL3-CL5. The tuning definitely affected the model in choosing more times CL6 and CL0, and effectively making the model less balanced.

REFERENCES:

[1] Akalin, A., 2022. *5.13 Logistic regression and regularization | Computational Genomics with R*. [online] Compgenomr.github.io. Available at: https://compgenomr.github.io/book/logistic-regression-and-regularization.html

[2] scikit-learn. n.d. *sklearn - LogisticRegression*. [online] Available at: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[3] Analytics Vidhya. 2020. *Feature Scaling | Standardization Vs Normalization*. [online] Available at: https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/

[4] scikit-learn. n.d. *Linear Models*. [online] Available at: https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

[5] Kraus, M., 2019. *Using Bayesian Optimization to reduce the time spent on hyperparameter tuning*. [online] Medium. Available at: https://medium.com/vantageai/bringing-back-the-time-spent-on-hyperparameter-tuning-with-bayesian-optimisation-2e21a3198afb

[6] knowledge Transfer. 2021. *Micro and Macro Averages for imbalance multiclass classification*. [online] Available at: https://androidkt.com/micro-macro-averages-for-imbalance-multiclass-classification/

[7] Brownlee, J., 2019. *Visualize Machine Learning Data in Python With Pandas*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/visualize-machine-learning-data-python-pandas/

[8] scikit-learn. n.d. Precision-Recall. [online] Available at: https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html