A Level Computer Science

NEA

Project Title:

Mortgage Eligibility Checker

BY:

Aldrich Antonio Fernandes

Candidate Number: 2055

Centre Number: 12532

Contents

Analysis

Background to Project

The real estate sector, marked by its dynamic nature and constant fluctuations in home prices, presents a challenge for individuals aspiring to become homeowners. The uncertainties surrounding the ability to purchase a home often deter potential buyers from applying for loans, driven by a lack of confidence in the likelihood of approval. This hesitation leads many individuals to opt for renting, attracted by the perceived lower economic commitments compared to the complexities of home ownership. Simultaneously, financial institutions, including banks and loan companies, find themselves allocating substantial time and resources to process applications, only to encounter numerous submissions that ultimately prove unsuitable. This situation underscores the critical need for a solution that not only encourages potential homebuyers to apply but also expedites the application assessment process for lending institutions.

In this context, my client, Mr. Jatau Van Weiren, plays a pivotal role within a bank as one of the mortgage loan officers, dedicating his efforts to assist potential borrowers about what type of loan they might be eligible to get. As part of his occupation, he is also incharge of the initial screening process, where he filters out applications that are unlikely to proceed as they won't meet the lender's qualifications.

The primary focus of this project is to develop a predictive system that can be effectively utilised by Mr. Weiren and respective lending institutions alike. This system aims to assess the eligibility of loan applications, distinguishing between weaker and stronger submission, ultimately enhancing efficiency and profitability by enabling a targeted allocation of resources toward applications with higher success rates.

Current System

The current system for processing mortgage applications is a complex one that can use up lots of time for both sides.

A mortgage loan officer is responsible for guiding potential borrowers through the loan application journey, and also acting as an intermediary between the borrower and underwriter (who are responsible for performing risk assessment). Some of the initial steps are as follows:

1). Initial Client Interaction	An initial interview to gauge the applicant's eligibility, discussing factors like income, and overall financial health.
This stage is often completed through an online interview, or application. The loan officer will look through this applicant and choose whether to approve or not it is worth continuing their application or reject it.	

2). Loan Product Guidance	The Loan officer provides information about different products available, explaining terms, interest rates and repayment options.
After the initial interactions, the loan officer will help the client understand what they may qualify for. For instance, how large of a loan can be taken and available payment plans.	

3). Pre-approval Process	An assessment of the borrower's eligibility.
The applicant will need to provide documentation such as payslips, tax returns, employment verifications and more to allow the loan officer to analyse their likelihood of loan approval evaluating factors such as credit score, income stability.	

For Mr. Weiren this process can be easier or more complex depending on the applicant, hence utilising a composition of automated and human systems. For instance, processes such as checking customer credit and reliability history by querying databases, and risk assessments can be undertaken by other programs. However actions such as document verification, collateral appraisal and final proofreading requires human resources.

As mentioned prior, the efficiency of this system is heavily tied to the applicant's ability to provide clear and organised documentations, which can contribute to a smooth process, while disorganisation can introduce delays and complexities.

Identification of Users

End users

The primary users are loan officers like Mr. Weiren, who work for lending institutions, that are seeking to screen out low probability applications in the initial stages of loan applications quicker, to allow for more time and resources to be allocated to more promising applicants.

Prospective users

While the current design is tailored for lending institutions, future instances may include provisions for the public as prospective users, albeit with limited access to different functionalities. For instance, like adjusting and training the model shouldn't be accessible.

Client Interview

For the following questions, all responses will be paraphrased slightly to make it more easier to read.

Could you elaborate the primary goal of the project?

“Sure, I'd say the main goal is to make the client selection process faster and more efficient. Instead of manually considering each factor in every application, having a system where I can input key information and quickly receive a response on whether the client is a good fit would be a significant improvement.”

What are some challenges with the current system?

“Well, the current system can be sluggish, and sifting through each application, especially for complex cases, is a bit tedious. After filtering out the less promising ones, I then have to dive deeper into the remaining applications to ensure they're a good fit, which adds to the time it takes.”

When choosing an applicant for processing what are the critical factors you consider?

“The top three factors are the applicant's income, including any co-applicant's income, their desired loan amount and duration, and their credit history . These factors not only gauge the feasibility of their goals but also give insights into their understanding of the process and financial management, helping identify high-probability applications."

What are some features you would like to be a part of the application?

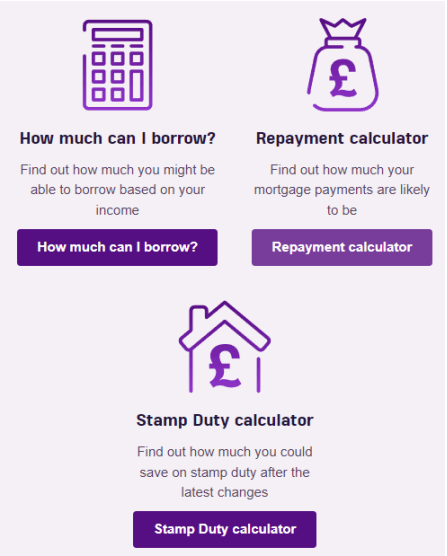
“Given that the project aims to expedite the client selection process, a fast application is crucial. Loading times shouldn't be a bottleneck, although I understand AI learning might take some time. Also, since our criteria may change regularly with market shifts, an easily retrainable application would be handy. I'd prefer a straightforward, user-friendly terminal to avoid getting lost among countless options and having to bother the IT department consistently.”

Similar Systems

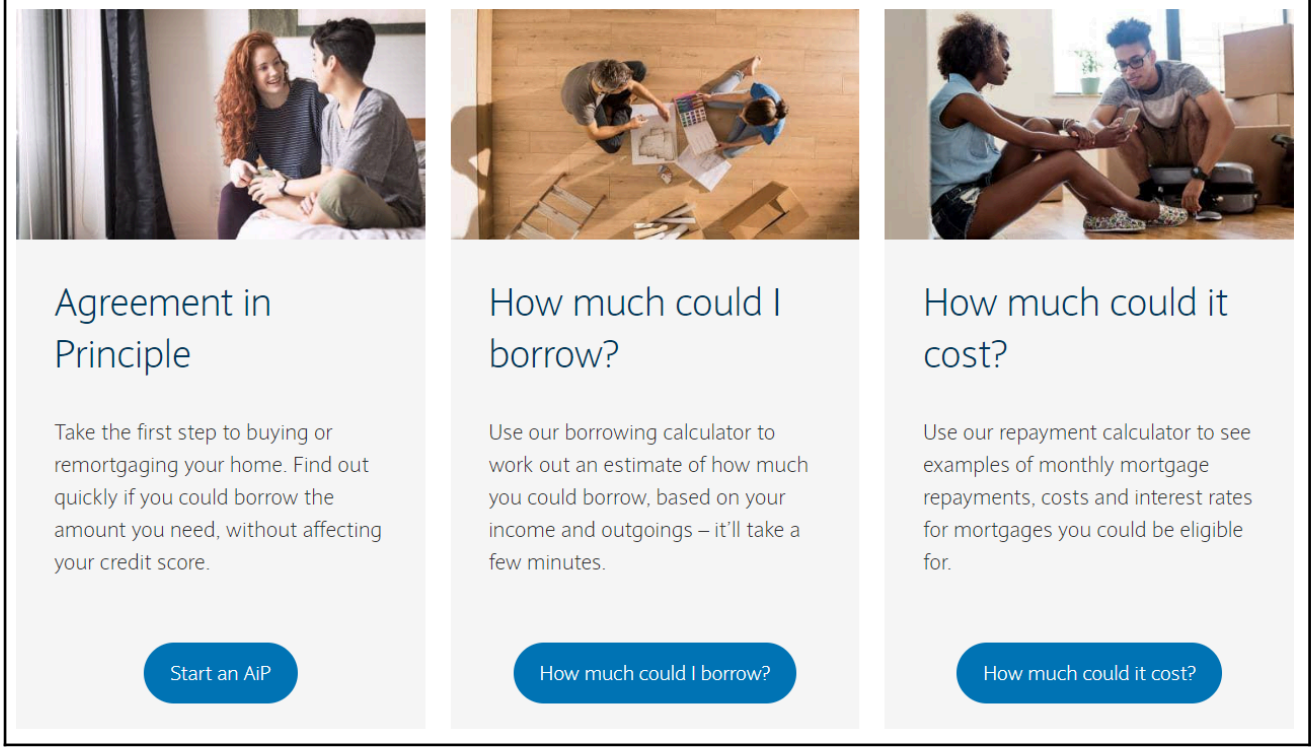
There are already a couple of systems that could help a person evaluate their applications. Some of the applications available are shown below.

Money Super Market

These calculators are directed towards the potential borrowers, who would like to explore more deeply, aspects such as how much money can be borrowed, how much will be overpaid and so on. These programs being supplied with lots of data are often accurate and specific, however for a first time buyer the use of unfamiliar terms and processes can be overwhelming, eventually requiring the assistance of a loan officer to understand this process. Furthermore, even if a client checks all this data beforehand, it stills need to be looked over and selected by a loan officer, among the range of applications they would receive. Overall not providing much assistance to the officer as it does for the client.



The image below, shows another example of a mortgage calculator provided by Barclays bank. As mentioned before, these calculators are more aimed towards potential applicants who are trying to conduct some research, and won't assist my client, who requires a system that informs him about the effectiveness of an application.



Potential Solution

Criteria

After reviewing the interview and similar systems, the algorithm used would require to adhere the following:

- Efficiency
    - Batch Processing: As the market changes, so does the data hence the algorithm used should be able to learn from batches of data to quicken the learning process.
    - Quick Response Time: The complete algorithm should be able to quickly process and relay the results to the user.
    - Saving and Loading: The trained model should not only be capable of saving and loading data to allow a quicker launch, but also utilise minimal memory.
- Adaptability:
    - Retrainable: As requested by the client, they should be capable of easily retraining the algorithm, hence the model should be able to work without needing to be altered from the source code itself.
    - Data Errors: The algorithm should be capable of not only automatically dealing with errors in the raw data but also prevent other data based errors within the algorithm. For instance, parameter adjustments for the algorithm.

Viable Algorithms

Logistic Regression (with Neural Network)

Overview:	Uses a neural network to process multiple inputs and calculate the probability of belonging to the positive class.
Strengths:	● Easy to break down and build. ● Highly customizable to different needs with a range methods of evaluating, optimising and fine tuning the algorithm. ● Using neural networks allows the algorithm to capture intricate relationships which logistic regression wouldn't be capable of by itself.
Consideration:	● Requires lots of data (in this context about 100-500 entries). ● Highly sensitive hyperparameters (configuration variables) are time consuming and tedious to adjust. Furthermore, the model ○ Can easily overfit (memorise the data). ○ Can converge on a single prediction everytime the model is run.

Decision Tree

	Overview:	Uses tree graphs where each node represents a decision based on a feature (data column), where each leaf is a final prediction.
	Strengths:	● Can handle non-linear relationships and is easy to interpret. ● Provides insights into feature importance.
	Consideration:	● Prone to overfitting. ● Sensitive to small variations in the data. ● As the tree grows and gets deeper, its interpretability decreases.

Random Forest Classification

	Overview:	Utilises multiple decision trees and combines each of their predictions through averaging.
	Strengths:	● More robust than a decision tree. ● Reduces overfitting by utilising averages of multiple trees.
	Consideration:	● Can be computationally intensive. ● The rate at which the interpretability decreases is greater than compared to a single decision tree.

Support Vector Machine (SVM)

	Overview:	Finds a hyperplane, a decision boundary that divides a space into two or more regions corresponding to a respective class.
	Strengths:	● Effective in higher dimension spaces (larger number of features). ● Can handle non-linear relationships using kernel functions (maps a lower dimension feature to a higher dimension).
	Consideration:	● Doesn't perform well with large datasets. ● Sensitive to noisy data (that that hasn't been clean and prepared properly) ● Same as in neural networks, hyperparameter tuning is sensitive and crucial.

Pilot Program

After analysing the different methods, the two algorithms that would be effective for this project are logistic regression with neural networks and random forest classification.

Having worked with neural networks in the past, I am more confident in creating a neural network compared to a random forest. The pilot code below, compares both algorithms, by having them work with the same loan dataset I am going to use (which will be analysed in the next section).

Pilot.py

```python
# Models used
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
# Unitilites
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# From my programs
from DataHandle import Preprocess

# random forest classification - utilises tree
def RandForest(X_train, X_test, y_train, y_test):
    acc = []
    for _ in range(10):     # Generates 10 different models to get average accuracy.
        # Initialising a model
        model = RandomForestClassifier(n_estimators=100, random_state=42)

        # Training the model
        model.fit(X_train, y_train)

        # Testing performance
        predictions = model.predict(X_test)
        accuracy = accuracy_score(y_test, predictions)
        acc.append(accuracy)

    return round(sum(acc)/len(acc), 4)

# Logistic regression model - utilities neural networks
def LogReg(X_train, X_test, y_train, y_test):   # Same as RanForest()
    acc = []
    for _ in range(10):
        model = LogisticRegression()
        model.fit(X_train, y_train)
        predictions = model.predict(X_test)
        accuracy = accuracy_score(y_test, predictions)
        acc.append(accuracy)

    return round(sum(acc)/len(acc), 4)

def testModels():
    Preprocessor = Preprocess()
    LogRegAcc = []
    RandForestAcc = []

    for x in range(10): # Trains models with 10 different datasets.
        print(f"Training dataset {x+1}")
        Preprocessor.newDataset()
        TrainX, TrainY, _, _ = Preprocessor.getData(split=0)
        X_train, X_test, y_train, y_test = train_test_split(TrainX, TrainY, test_size=0.2, random_state=42)

        LogRegAcc.append(LogReg(X_train, X_test, y_train, y_test))
        RandForestAcc.append(RandForest(X_train, X_test, y_train, y_test))
```

```
    # Average accuracy after 100 tests each.
    print(f"Logistic Regression     | Accuracy: {round(sum(LogRegAcc)/len(LogRegAcc), 4)}")
    print(f"Random Forest           | Accuracy: {round(sum(RandForestAcc)/len(RandForestAcc), 4)}")

testModels()
```

Results

To assess the program's performance and establish a benchmark, I developed a simple logistic regression application for evaluating the potential accuracy of the system. As shown above, utilising the sklearn machine learning module in Python, I conducted 10 iterations using different random dataset generated from the same source for both logistic regression and random forest models. The outcomes revealed comparable accuracy levels following an average of 100 tests each.

Average Accuracy:

- Logistic Regression: 70.2%
- Random Forest Classifier: 67.4%

This analysis provides insights into the effectiveness of the models, showcasing their performance in handling the dataset across multiple trials.

Re-Interview

Questions

As done before, all responses will be paraphrased slightly to make it more easier to read.

What are your thoughts on the overall concept and approach outlined in the proposal?

"I find the overall concept and approach outlined to be promising and well thought through. Your choice of using a neural network appears to have been strategically chosen to meet the criteria we discussed earlier. The emphasis on the project being modular to allow for easier testing and reusability, I believe will be appreciated greatly by our IT department."

Does this proposed project alight with your objectives?

"Absolutely, this project hits the nail on the head when it comes to our objectives of making our client selection process more efficient and accurate. Using advanced machine learning techniques to speed up decision-making while maintaining accuracy is exactly what we're aiming for."

Is there anything that you would like to change or add that is not mentioned in my proposal?

"While the proposal covers a lot of ground, there are a couple of things I think we should consider adding. Firstly, given the constantly changing nature of our data and the market, having a way to fine-tune the model's performance over time could be really valuable. This could involve building in features for ongoing adjustments or recalibrations to keep the model performing at its best. Additionally, exploring ways to incorporate real-time data sources could give us even more accurate predictions and make the system even more powerful in real-world situations."

Are there any specific criteria I should consider when developing the project?

"Certainly, two specific criteria that should be prioritised during development of the project are accuracy and efficiency. While accuracy is obviously crucial for making reliable decisions, we also need to make sure the system runs as efficiently as possible to minimise processing time and keep things moving smoothly. Finding the right balance between accuracy and efficiency will be key to making this project a success. And of course, we should keep scalability in mind to ensure the system can handle whatever comes its way as we grow and evolve."

Adjustments and Limitations

According to this interview, the following adjustments / additions should be implemented. The following will also be reiterated and integrated into the objectives.

- Hyperparameter tuning option.
    - Implement a feature allowing clients to adjust hyperparameters related to both the optimizer and network architecture, providing greater flexibility and control over model performance.
    - Develop a user-friendly interface for this purpose, to include hyperparameters such as learning rate, regularisation strength, and epochs.
        - Aspects such as changing the number of layers will not be accommodated as a logistic regression model utilises only an input and output layer.
            - However, a method to do so is implemented in the model class for alternate/further uses.
    - This will also ensure that the model can accommodate larger datasets and complex model architectures as needed in the future.

- Average algorithm accuracy.
    - Ensure the algorithm achieves an accuracy of greater than 70%, as determined in the pilot study
    - Minimising the time required to (re)train the algorithm.
        - Utilise low training dataset sizes, batch sizes, and epochs to reduce processing time while preventing overfitting.
    - This may require the need to retrain the neural network multiple times due to the nature of the dataset and model until a suitable model is found.
        - This is because if the model trained is unsuitable, it will need to restart the training process with a fresh dataset otherwise the model will just overfit or converge on a single output for all predictions.
    - This approach aims to maintain a high level of accuracy while minimising training time.

- Real-time data
    - Due to the nature of the type of data (applicant personal data) my program processes, as a student I don’t have access to such data.
        - As a result the data to be used must come from open sources databases such as Kaggle.
        - This allows for the utilisation of realistic data while ensuring compliance with academic constraints.

These additions and improvements aim to enhance the functionality, effectiveness, and user experience of the mortgage loan eligibility checker, aligning with the adjusted objectives outlined in the interview feedback.

Final Proposal

Algorithm

To process user applications, I plan to perform binary classification to provide an estimate of the strength of the application. As mentioned before, there are a number of methods to perform binary classification, however after considering different methods such as neural networks, random forests and SVM, I decided to implement a logistic regression algorithm utilising neural networks.

A neural network is a machine learning model that is inspired by the human brain; composed of layers of 'neurons' which can be trained and adjusted to perform classification on a set of data. I will be making a neural network that consists of the following:

- Dense layers - A collection of neurons, where each one is connected to every neuron in the next layer.
- Activation functions - Adjust a neuron's output between a range.
- Loss Functions - Quantifies the difference between the predicted and real output.
- Optimizers - Adjusts the weights and biases of neurons

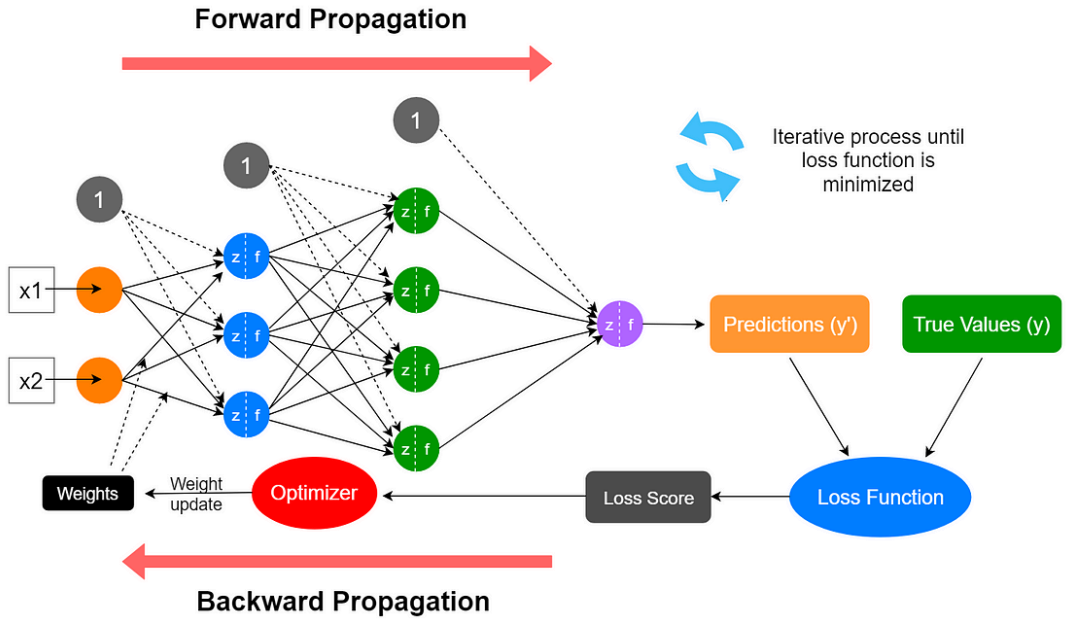
In the context of the application, utilising a neural network will allow me to easily implement several of the aspects Mr. Weiren has requested, in addition to functionalities that will benefit the overall application. For example, being able to program the model completely modular allows each part of the algorithm to be tested, updated and adapted easily to keep up with constantly changing data, surrounding the real estate sector. As long as the data have been organised and labelled in the same format as the original training .csv file used in this project, the program can be utilised with any dataset. Furthermore, utilising a neural network allows a developer to automate or improve processes other than just predicting eligibility - such as multiclass classification and natural language processing and sequence modelling to name a few - by replacing or adding the necessary classes and adjusting the model hyperparameters. This is much more beneficial compared to random forest, which despite providing roughly the same level of accuracy, is mainly utilised for classification and is limited with further utilisation.

This will be beneficial for my NEA as it will allow me to implement these different components as their own classes to introduce a complex class relation. Furthermore, neural networks are composed of several smaller functions that utilise A-level maths and further maths such as differentiation and matrices. Each part of the algorithm will be explained more in depth in the Design and Technical Solution section of this documentation.

The application will be written in python and will utilise Tkinter for the GUI. The GUI will need to be easy to understand, able to handle user input errors and contain different functionality. Such functionalities can include, loading a pre-trained model to allow for quicker execution as well as being able to train, save and load new models.

Model

During development, I will base my neural network on the following diagram, as a guideline. This will be explored in more depth in the Design section of this document.



Rukshan Pramoditha, 'Overview of a Neural Network's Learning Process', Medium, 'https://medium.com/data-science-365/overview-of-a-neural-networks-learning-process-61690a502fa

Data

Source

As I plan on using a neural network, the chosen data had to be categorical and have a binary value (loan approved or not) to be calculated. After some research, I decided to utilise a dataset sourced from Kaggle (https://www.kaggle.com/datasets/rishikeshkonapure/home-loan-approval last accessed 20/01/24) by Rushikesh Konapure.

This dataset would also include many issues such as a mix of categorical, quantitative and missing data, class imbalance, feature scaling and skew (to name a few), that will need to be dealt with prior to developing the network, same as when expected training would have.

Example of the Dataset

Below is an example of the dataset I am going to use (The example does not consist of all the available features, only the most important ones).

Dependents	Education	ApplicantI...	Coapplica...	LoanAmou...	Loan_Amo...	Credit_His...	Property_...	Loan_Status
0	Graduate	5849	0		360	1	Urban	Y
1	Graduate	4583	1508	128	360	1	Rural	N
0	Graduate	3000	0	66	360	1	Urban	Y

Overview of Data

The Data used shown is stored in a csv file and is made up of 13 features (columns) and 614 unique entries (row). As mentioned before, the two main types of data found in this dataset are categorical (eg. gender, education) and numerical (eg. applicantIncome, loanAmount). Across the dataset there are also 134 entries that contain missing values.

To save time I reviewed an analysis on the dataset conducted by Sirkanth (https://www.kaggle.com/code/srikanth917/home-loan-approval-prediction) prior to selecting the dataset. As a result when prepping the dataset to be used for training, I will be using the following features only:

ApplicantIncome	CoapplicantInco	LoanAmount	Loan_Amount_T	Credit_History	Property_Area	Loan_Status
3036	2504	158	360	0	Semiurban	N
4006	1526	168	360	1	Urban	Y
12841	10968	349	360	1	Semiurban	N
3200	700	70	360	1	Urban	Y
2500	1840	109	360	1	Urban	Y
3073	8106	200	360	1	Urban	Y

Objectives

1). Data Selection and Processing

1. Obtaining training data
    a. Working with training data is a delicate process especially when it comes to privacy.
        - As such the data used for the algorithm needs to be not only anonymous, if not have no link to a person.
            - Sources such as Kaggel, UCI machine learning repository and government open data portals can be used for finding such data.
    b. Selecting a training dataset.
        - The dataset should be relevant to home loan applications and be big enough to allow the algorithm to find correlations.
            - It should at least contain features such as income, credit history, loan amount and duration as identified by Mr. Weiren. in the interview.
        - The dataset should preferably be in a .csv format which will allow to separate and load the data in their respective features easier.

2. Data needs to be altered so it can be processed by the algorithm in an efficient manner. (eg, should consist of numerical standardised data) and address common data related issues.
    a. Some issues that may be encountered are:
        - Null data - Some entries of data may include data that is empty or can't be processed.
        - Feature Scaling - A large spread of data with multiple anomalies can make it difficult for the model to find patterns.
        - Data Imbalances - These cause an unfair and biassed training process as one data may overpower the other.

    These will be explored more in detail in the Design section of this documentation.

3. The data should be processed effectively and efficiently.
    a. To improve time efficiency:
        - Data should be processed as a matrix.
        - Data entries are effectively randomised each epoch.
            - This will also ensure that the model doesn't overfit for the dataset (produce a model that is specific for the dataset and isn't generalised).

2). Neural Network

1. Designing the model
    a. The model should include dense layers, activation functions, loss functions and optimiser as core modules.
    b. Implement appropriate functions to use for logistic regression.
    c. Develop using an object-oriented approach (OOP) to build the neural network.
        - To further this the different components of the neural networks can be organised as packages, rather than just files.

2. User Prediction
    a. The user should have an easy to use interface to input their data and be processed.
        - This could include different methods of inputs (such as entries and radio buttons) to further simplify this.
    b. The data should provide a correct and appropriate prediction with valid data.
    c. The model should reject and not process invalid user inputs relaying a relevant message.
    d. If extreme / unrealistic data is provided a prediction of 0% should be made.

3. Model Control
    a. The user should be able to control what model is used and be relayed an appropriate message indicating what model is being used.
    b. A default model should be loaded when the application is run to improve launch time.
        - If the generated model doesn't achieve this accuracy; weights and biases should be reset; and new data generated to be processed.
    c. A newly generated model should be indicated as unsaved and also display its accuracy.
        - This newly generated model should have an accuracy of greater than 72%.

4. Hyperparameter tuning
    a. The User Interface (UI) could incorporate a separate tab to allow for easier testing of the model with different parameters
    b. A new model is loaded/trained when the hyperparameters have been changed.
    c. The entries shouldn’t try to retain a model with invalid hyperparameter.
        - This includes values that shouldn’t be used such as decay or regularisation straight that is greater than 1 or a negative value.

5. Saving a model:
    a. If the model name already exists, it shouldn't accept and override the file.
    b. When saving the model only save necessary data such as:
        - Weights and biases for each layer.
        - Scales for each feature to apply to user data.
    c. A previously trained model should be able to be loaded.
        - And the user should receive an acknowledgement of this.
        - A nonexistent model shouldn’t be attempted to be loaded.

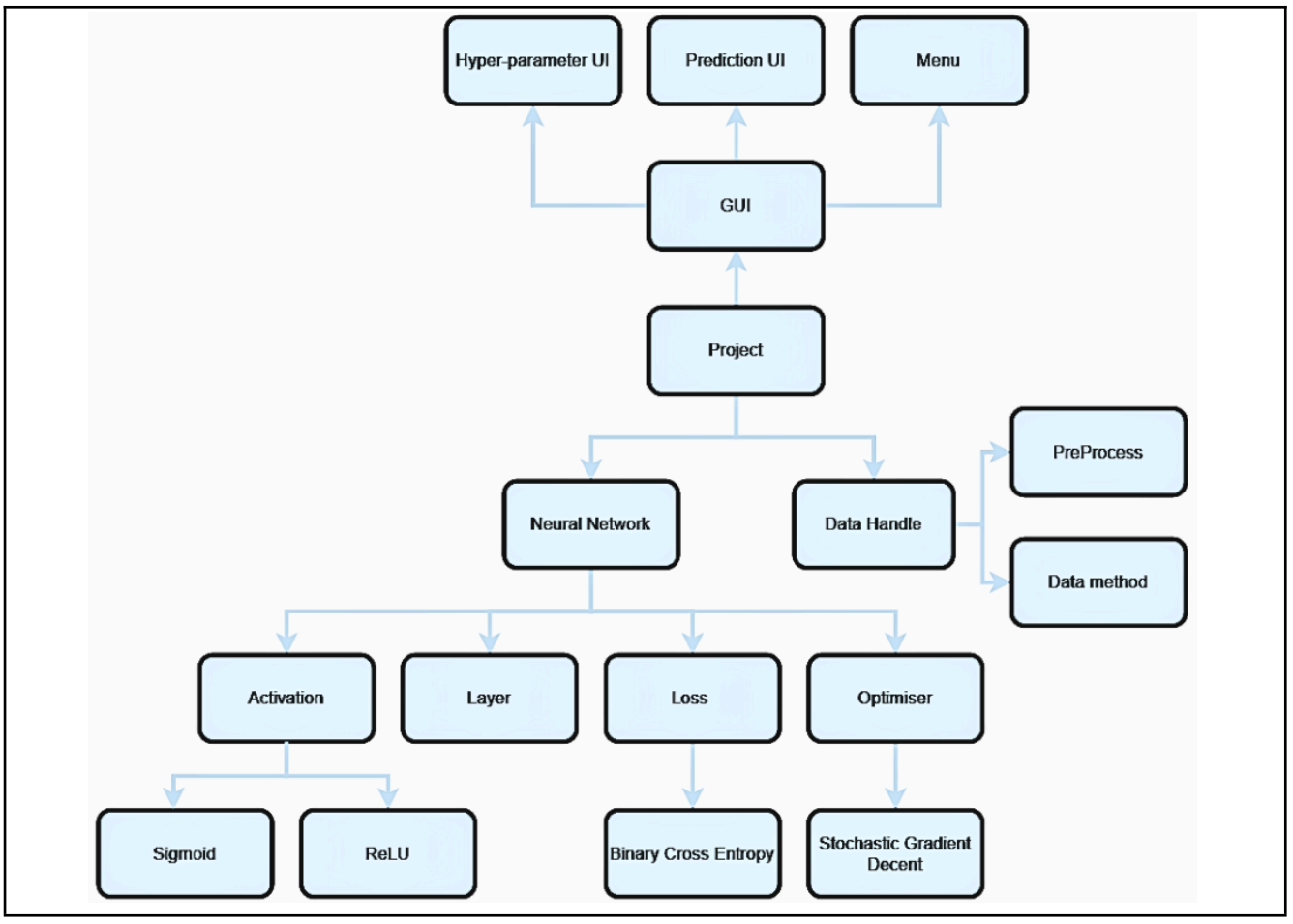
Design

Overview of system

As mentioned previously, in the potential solution section in the Analysis, the application will utilise python to create a neural network, and Tkinter (a built in package) for the graphical user interface (GUI). As such the project will be broken down into 3 core parts: data handling, the neural network itself, and the GUI.

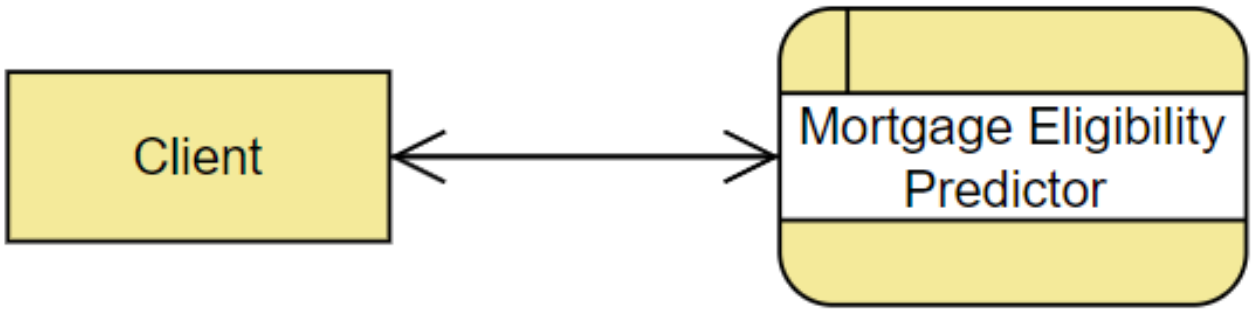
Each part should be developed in that respective order, for ease of testing throughout the development. This is because processes such as data handling will also be needed to verify the system's efficiency and benchmark, during the pilot run, prior to the development of the neural network itself.

Hierarchy chart of system

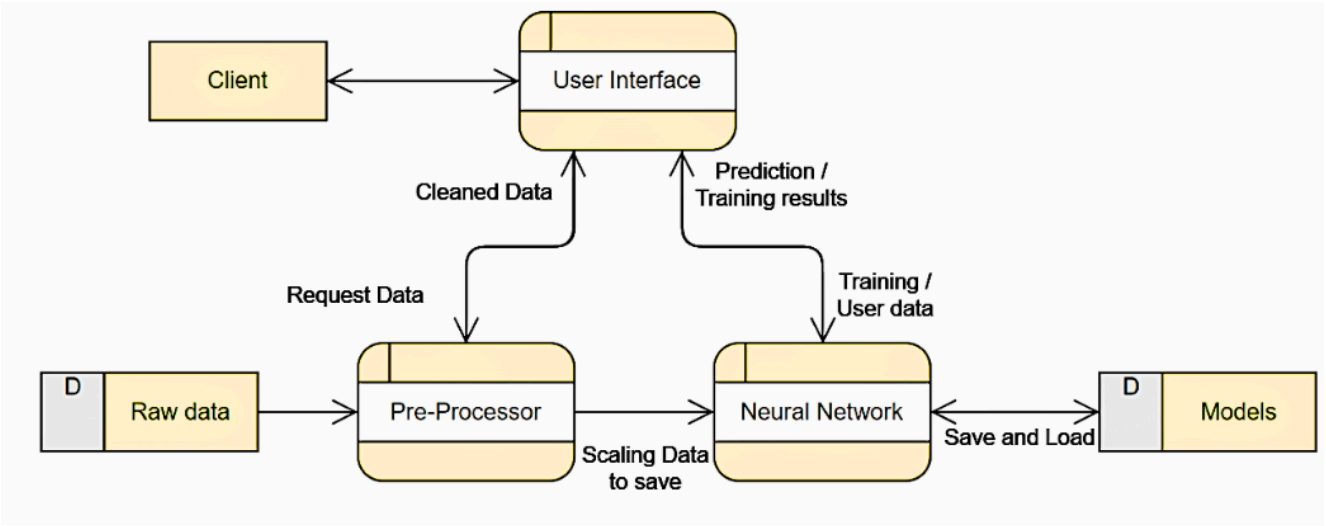


Data Flow Diagrams (DFD)

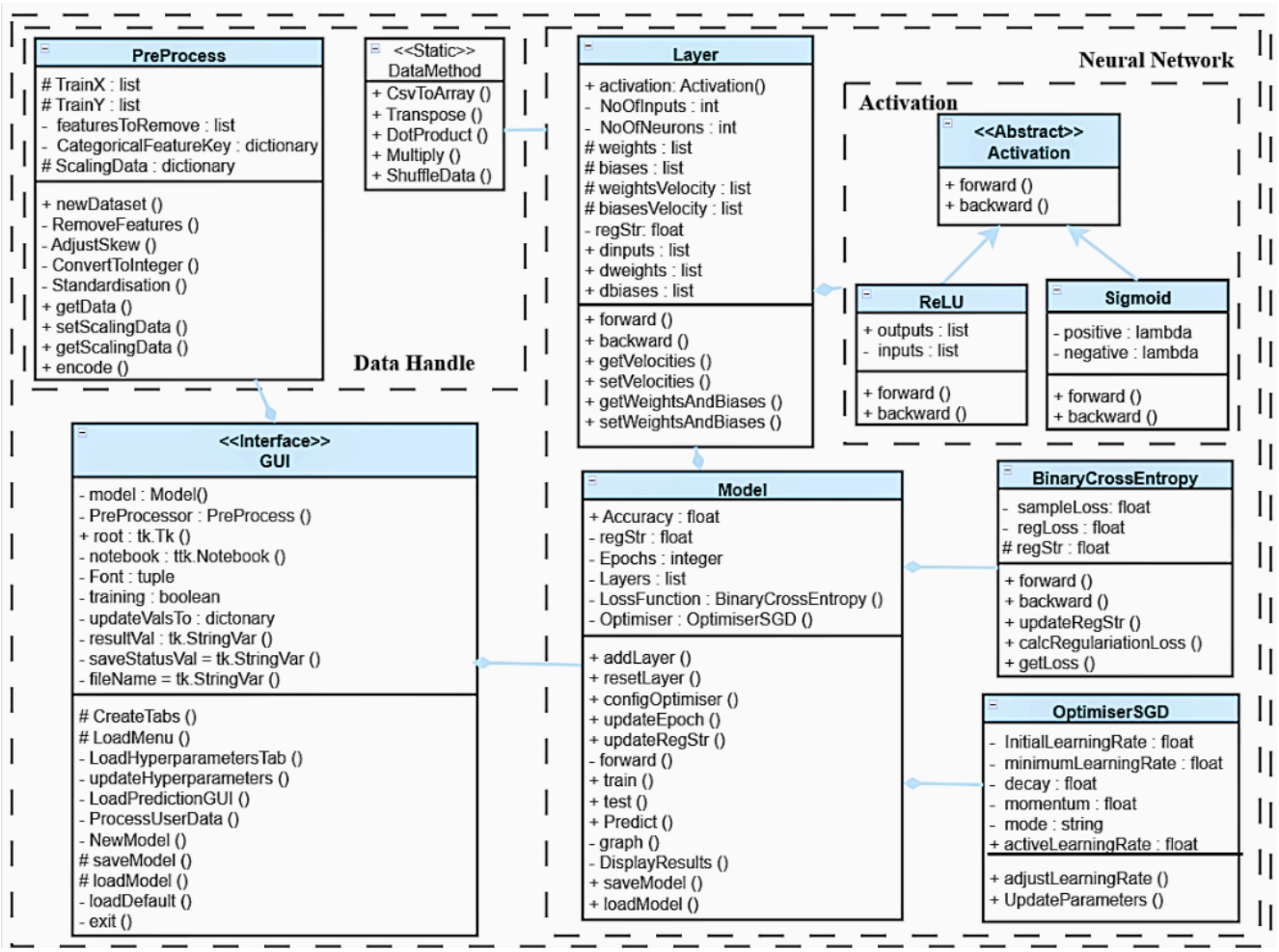
Level 0



Level 1



Final UML (Class) Diagram

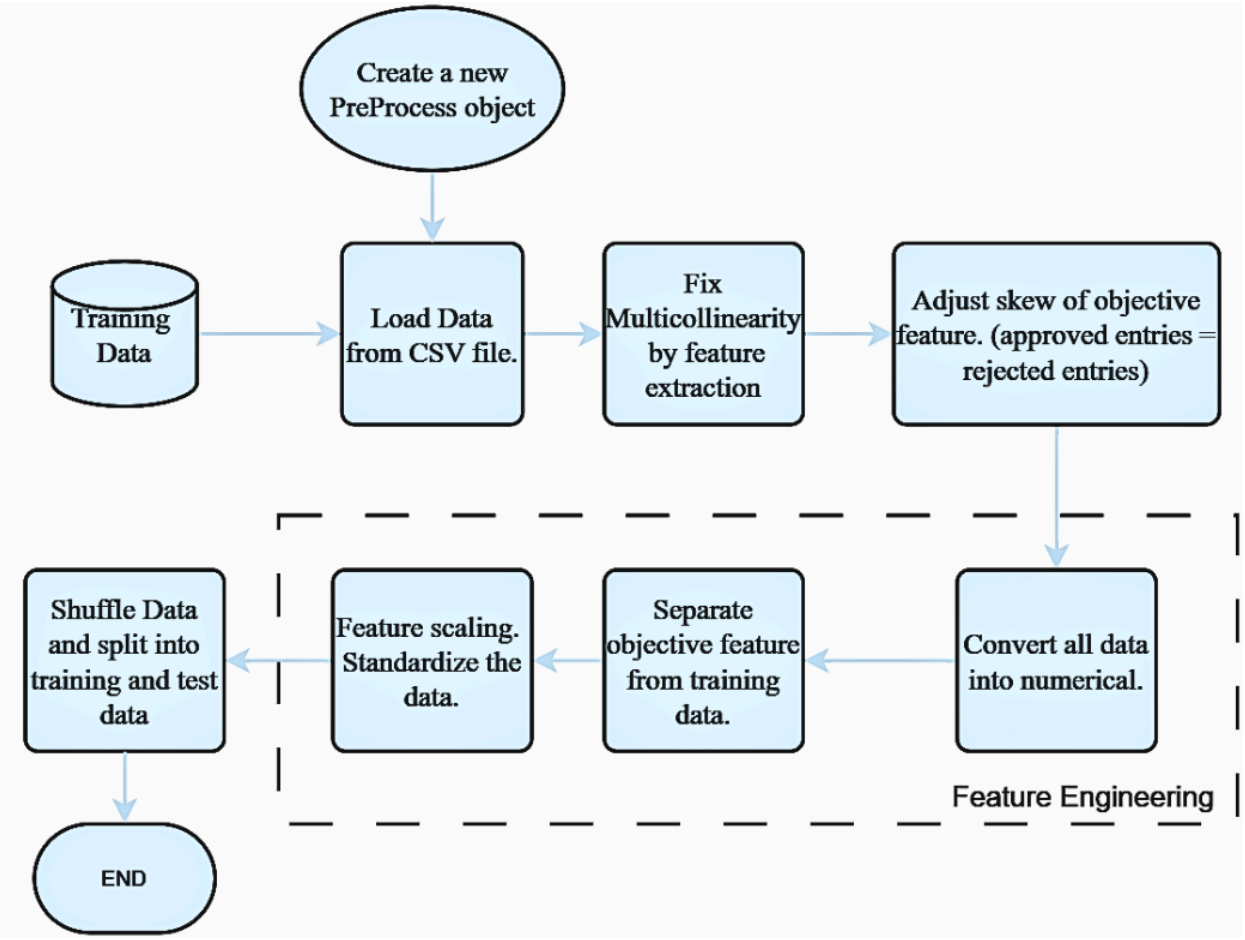


Breakdown of Data Handling

The purpose of this part is to assist the use of data in the neural network. This will be done in various ways such as preparing the data to train the model with and providing the system with functions to make mathematical processes easier.

Pre-process

As mentioned previously, the neural network cannot process raw data that consists of a mixture of strings and real numbers. This data needs to be cleaned and prepped for this purpose. The following diagram shows the rough process the Preprocess class will undertake.



A more detailed description and solution to achieve this can be found below:

1). Missing Data	These can cause TypeErrors during processing as missing data can be classed a NoneType which cannot be processed.
Solutions: ● Entries with missing data can be removed. ● Missing data can be replaced with the most common string or modal value of that feature.	

2). Multicollinearity	Some features may have a negative impact on the model if it noisy data (data with a large amount of additional meaningless information)

Solutions:

- Features that have low / no impact on the outcome should be removed. For example, a feature that acts as a primary key has no influence on the probability of the application being approved hence should be removed.

3). Non-numerical data	Data that is not a real number. (eg, strings).

Solutions:

- Convert all data into a numerical format to allow for model training.
    - This can be done by utilising their a character's unicode or ASCII.
- A key can also be generated or attributed to the Preprocess to replace such values with a numerical one.

4). Data Imbalance	Data Imbalances cause an unfair and biassed training process as one data may overpower the others. For instance, if there were 80 approved entries compared to 20 rejected entries, the model will actively predict approval, regardless of the data as it would be the most probable. Balance the data to allow for a fair and biassed less training less process.

Solutions:

- Provide an equal amount of entries for each outcome to ensure the model does not train to favour one outcome.
- This can also be used to choose the number of entries used for the training.

5). Feature scaling	A large spread of data with multiple anomalies can make it difficult for the model to find patterns. In addition, it can cause issues similar to data imbalance.

Solutions:

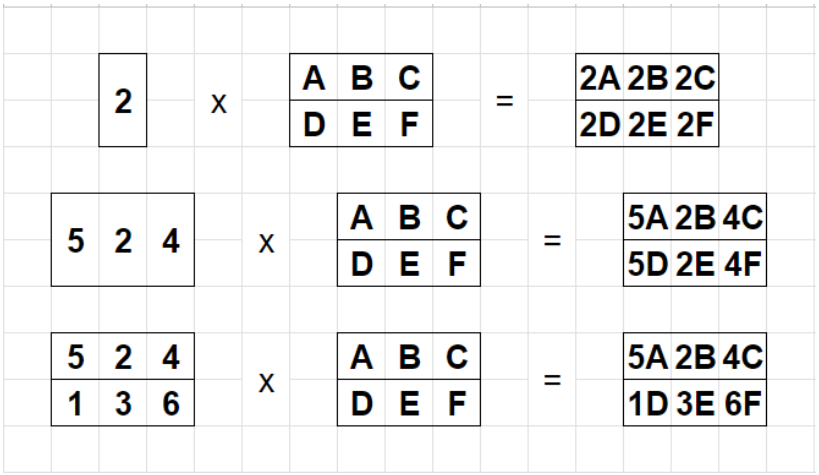
- Standardise / normalise all data in a feature. This will make it easier for the model to find collaborations between the features.
    - This can be done with Z - score normalisation, which standardised features by subtracting the mean and dividing the standard deviation.
        - This allows the features to have a mean of 0 and standard deviation of 1.

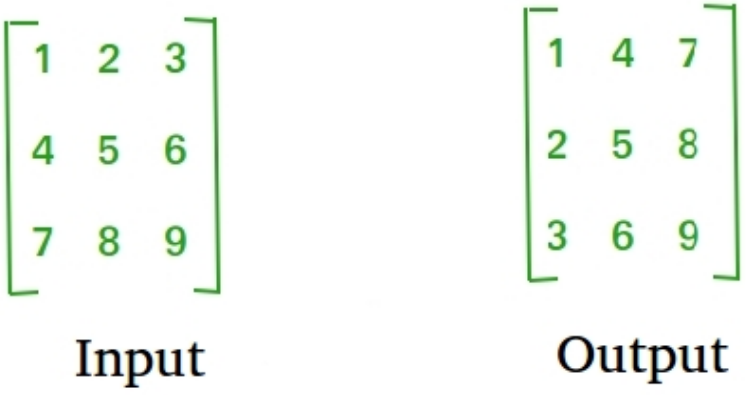
Data Methods

The neural network makes use of multiple different matrix processes such as transpose and dot product, in its algorithm. This is because it also makes it easier and quicker to process chunks of data such as training data, weights and multiple userdata.

Some useful methods to program are shown below:

Dot product (matrix multiplication)	$\begin{bmatrix} a & b \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax+by \end{bmatrix}$ $\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax+by \\ cx+dy \end{bmatrix}$ $\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw+by & ax+bz \\ cw+dy & cx+dz \end{bmatrix}$
Multiplies two matrices by calculating the sum of a row times a column as shown. Used when performing forward propagation.	

Multiplication (for scaling)	
Multiplies an array with a scalar, list, or another array of equal dimensions as shown. This can also be used to scale arrays	

Transpose	
Used to swap the dimensions of a matrix. This can help turn the data from list of entries to list of features.	

Pseudocode

DotProduct

```
FUNCTION DotProduct(arr1, arr2)
    result ← # Create a zeros matrix with the dimensions of (arr1Rows, arr2Columns)

    arr2_T ← Transpose(arr2)
    FOR i FROM 0 TO LENGTH(arr1)-1
        FOR j FROM 0 TO LENGTH(arr2_T)
            Result[i][j] ← sum( Multiply( arr1[i], arr2_T[j ] ) )
        END FOR
    END FOR

    RETURN Result
END FUNCTION
```

Multiply

```
FUNCTION Multiply(arr1, arr2)
    IF arr2 is a 2d array
        IF arr2[0] is a list THEN
            FOR i FROM 0 TO LENGTH (arr2)-1
                RETURN MULTIPLY (arr1[i], arr2[i])
            END FOR
        ELSE
            FOR row IN arr2
                RETURN Multiply (arr1, row)
            END FOR
        END IF
    ELSE
        Result ← []
        FOR i FROM 0 TO LENGTH(arr2)-1
            Result.APPEND(arr1[i] * arr2[i])
        END FOR
        RETURN Result
    END IF
END FUNCTION
```

Transpose

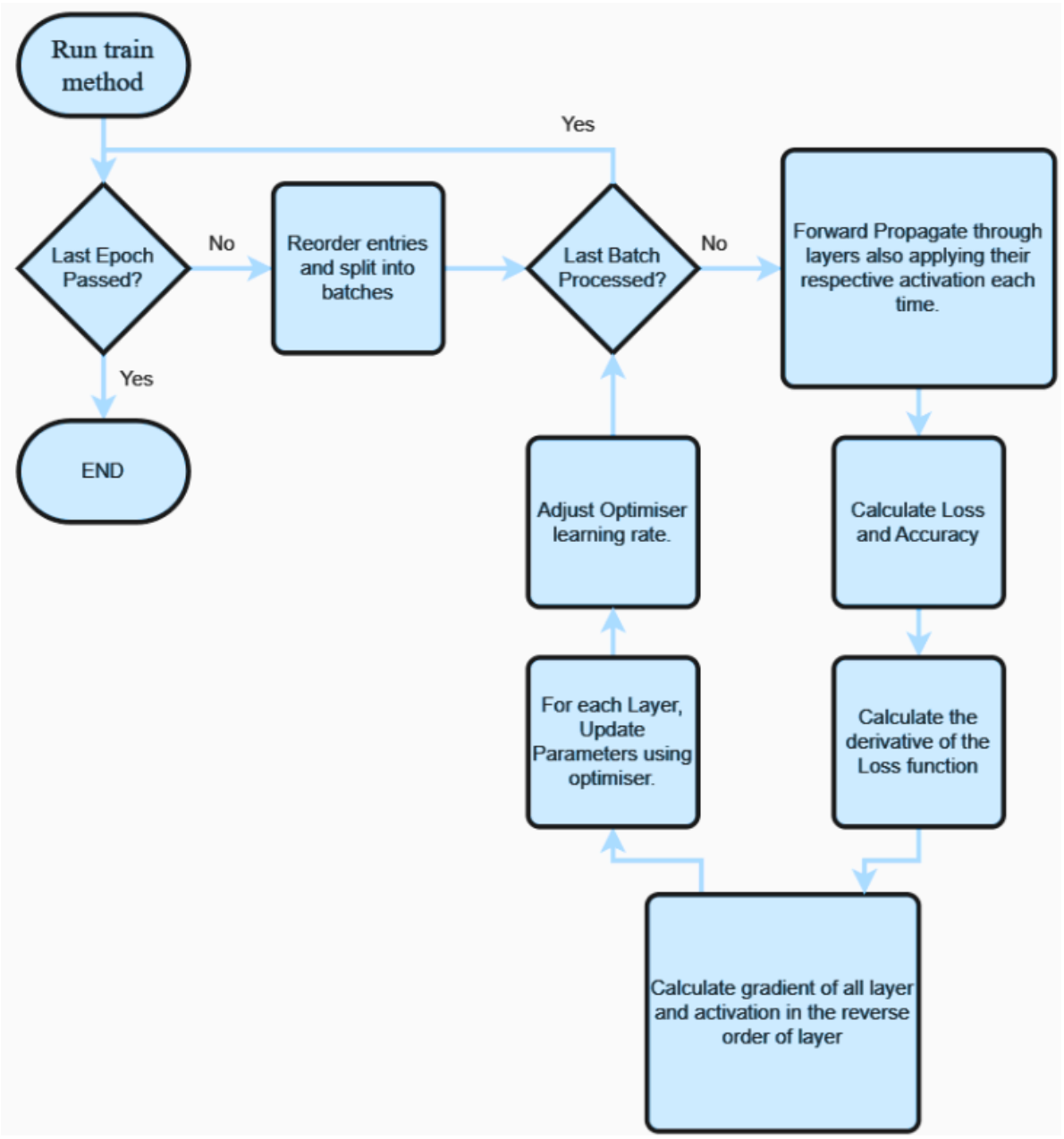
```
FUNCTION Transpose(arr)
    Result ← # empty array of dimensions as arr
    FOR y FROM 0 TO LENGTH (arr[0])
        FOR x FROM 0 TO LENGTH (arr)
            Result[x][y] ← arr[x][y]
        END FOR
    END FOR
    RETURN Result
END FUNCTION
```

Breakdown of Neural Network

This part aggregates multiple classes and is the most complex aspect of this project. This is because a neural network is made up of various components, as explained previously, such as layers, different types of activations and optimiser, all of which have to be carefully considered and selected before the tedious process of tuning all the hyperparameters involved.

Model

The model class acts as the central control for the algorithm, handling and housing the neural network and its components. The main purpose of this class is to handle the neural network performing tasks like training the data provided, graphing the process and so on. The diagram below shows the training process.



Layer

A layer is a collection of neurons, which is responsible for transforming the inputs into a form that is more useful for the following layers. Typically neural networks are formed of multiple layers, however for Logistic Regression, we will only utilise one (output) layer to prevent overfitting (model memerises the data).

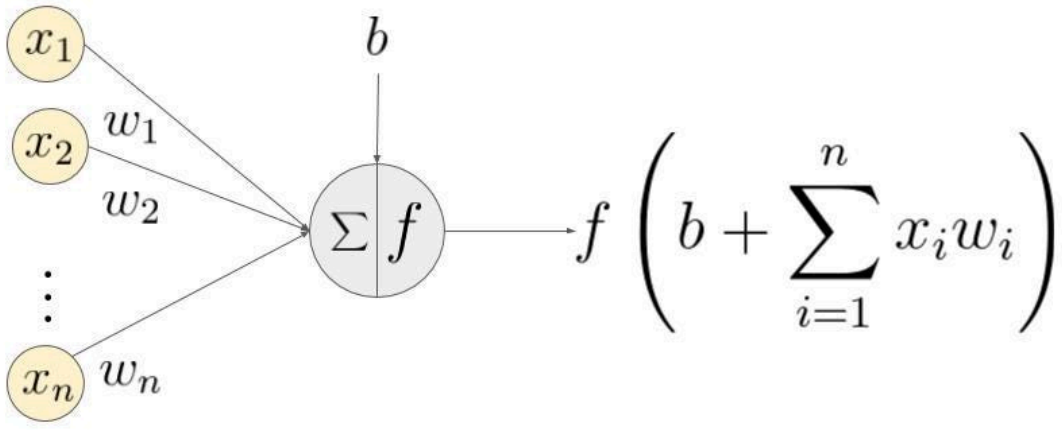
There are 3 main types of Layers:

1. Input - The first layer and acts as an entry point for input data. Each neuron in this layer corresponds to a feature.
2. Hidden - These layers are responsible for learning and increasing the complexity of the network. Hence there are usually more than one hidden layer.
3. Output - This layer produces the final prediction of the network. The number of neurons in thai layer corresponds to the number of possible classes, where the output of each neuron is probability of it being that class.

Forward Propagation

A neural network can be visualised as a graph, where neurons (nodes) in each layer are connected by weights (edges). During forward propagation, the output from a layer of neurons becomes the input for the next layer, and the process of weighted sum calculation and activation function application is repeated in subsequent layers. Until the final output layer proceeds the network's prediction.

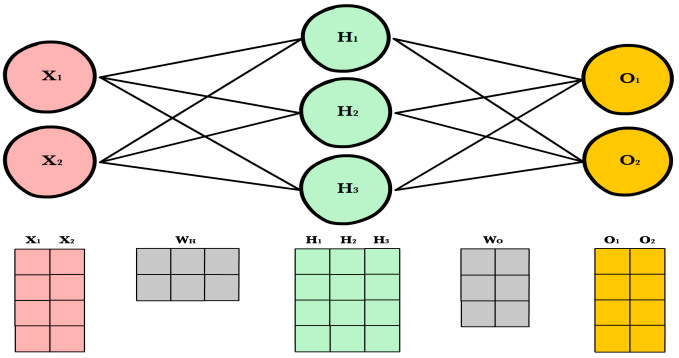
The following diagram displays the how the output of a neuron in a layer is calculated before being activated:



An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.

Aditya Sharma, 'Activation Function in Deep Learning - A Complete Overview', LearnOpenCVhttps://learnopencv.com/understanding-activation-functions-in-deep-learning/

The neurons in each layer are stored as a matrix. This is so that when the layers are propagated through, the matrix shape will automatically change. The following diagram visualised what this looks like.



Back Propagation

For backpropagation through a layer, there are 3 different gradients that need to be calculated for the data. The following gradients that need to be calculated are for the inputs, weight and biases.

The layer's gradients (dvalues), which will be passed to the next layer. This can be calculated using the following formula:

$$dinputs = v \cdot Transpose(w)$$

The other 2 gradients that need to be calculated are for the weights and biases. These aren't passed along like other gradents but are to be utilised by the optimiser to adjust the weights and biases for that layer. Their formulas are:

$$dweights = (Transpose(i) \cdot v) + 2(\lambda \times w)$$
$$dbiases = sum(Transpose(v))$$

Where;

- 'v' is the gradient from the layer's actionion function.
- 'w' is the layer's weight.
- 'i' is the input value received during forward propagation.
- 'λ' is the L2 regularisation strength.

The use of 'λ' will be explained in the Loss section, where it is utilised.

Pseudocode

Forward Propagation

```
FUNCTION forward(inputs)
    weightedSum ← DotProduct(inputs, weights)
    outputs ← []
    FOR sample IN weightedSum
        sampleOutput ← []
        FOR neuronOutput IN sample
            sampleOutput.APPEND(neuronOutput + bias)
        END FOR
        weightedSum.APPEND(sampleOutput)
    ENDFOR
    RETURN weightedSum
END FUNCTION
```

Backpropagation

```
FUNCTION backward(dvalues)
    dinput s← DotProduct(dvalues, Transpose(weights))

    dweights ← DotProduct(Transpose(inputs), dvalues)

    dbiases ← []
    FOR row IN Transpose(dvalues)
        dbiases.APPEND( SUM(row) )
    END FOR
END FUNCTION
```

L2 Regularisation

```
FUNCTION regularisation()
    DweightsSquared ← Multiply (regularisationStrength, weights)

    newDweights ← []
    FOR x FROM 0 TO LENGTH(Dweights)-1
        row ← []
        FOR y FROM 0 TO LENGTH(dweights)-1
            row.APPEND( dweights[x][y] + (2 * DweightsSquared[x][y])
        END FOR
        newDweights.APPEND(row)
    END FOR
```

Activation

Activation functions play a crucial role in neural networks, by introducing non-linearity to the model, helping it learn complex patterns from data.

Rectified Linear Unit (ReLU)

This activation is a commonly used function, which outputs the input value if it is positive or zeros if it is negative.

ReLU is defined as:

$$R(Z) = \max(0, Z)$$

It's derivative for backpropagation is the following:

$$R'(Z) = \begin{cases} 0 & Z < 0 \\ 1 & Z > 0 \end{cases}$$

Where

- R(Z) is the ReLU function
- Z in an input value.

Sigmoid

This activation is mostly used for the output layer, a binary classification, as it squashes input values to the range of [0,1]. This also helps smoothen out extreme values.

Sigmoid is defined as:

$$f(x) = \frac{1}{1+e^{-x}} \text{ or } f(x) = \frac{e^{x}}{e^{x}+1}$$

It's derivative can be equated to the following:

$$f'(x) = \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) = f(x)(1 - f(x))$$

Where:

- f(x) is the sigmoid function.
- 'x' is an input value.

When implementing this backpropagation, The output will also have to be multiplied by the respective gradient (dvalue) from the previous layer, in order to produce its own gradient (dvalue) to pass along.

Pseudocode

ReLU Forward

```
FUNCTION forward(inputs)
    Outputs ← [ ]
    FOR sample IN inputs
        sampleOutput = [ ]
        FOR value IN sample
            sampleOutput.APPEND ( max(0, value) )
        END FOR
        Outputs.APPEND(sampleOutput)
    END FOR
    RETURN Outputs
END FUNCTION
```

ReLU Backward

```
FUNCTION backwards()
    Dinputs ← [ ]
    FOR sample IN inputs
        sampleOutput = [ ]
        FOR value IN sample
            IF value > 0 THEN
                sampleOutput.APPEND(1)
            ELSE
                sampleOutput.APPEND(0)
            END IF
        END FOR
    END FOR
    RETURN Dinputs
END FUNCTION
```

Sigmoid Forward

```
FUNCTION forward(inputs)
    Outputs ← [ ]
    FOR value IN inputs
        IF value < 0 THEN
            Outputs.APPEND (exp(value) / (exp(value)+1))
        ELSE
            Outputs.APPEND ( 1 / (exp(-value) + 1) )
        END IF
    END FOR
    RETURN Outputs
END FUNCTION
```

Sigmoid backward

```
FUNCTION backward (dvalues)
        Dinputs ← [ ]
        FOR i FROM 0 TO LENGTH(outputs)
                Dinputs.APPEND( [dvalues[i] * outputs[i] * (1 - outputs[i])] )
        END FOR
        RETURN Dinputs
END FUNCTION
```

Loss

The binary cross-entropy loss function, also known as log loss, is a commonly used loss function in binary classification tasks. It measures the difference between two probability distributions: the true distribution of the labels and the predicted distribution generated by the model.

Forward Propagation

The formula used for binary cross entropy loss can be expressed as:

$$L(y, \widehat{y}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot log(\widehat{y_i}) + (1 - y_i) \cdot log(1 - \widehat{y_i})]$$

Where:

- y_i is the true label (0 or 1) for the i-th sample.
- $\widehat{y_i}$ is the predicted probability that the i-th class belongs to the target class.
- N is the total number of samples

Back Propagation

The formula use for backpropagation, doesn't produce a singular value (sum) as gradient is to be passed back through the layers and the layers work with matrices rather than singular values:

$$L'(y, \widehat{y}) = \frac{\widehat{y_i} - y_i}{(1 - \widehat{y_i}) \cdot y_i}$$

Where:

- y_i is the true label (0 or 1) for the i-th sample.
- $\widehat{y_i}$ is the predicted probability that the i-th class belongs to class 1.

L2 Regularisation

L2 regularisation is a technique used to prevent overfitting by adding a penalty term to the loss function that discourages large weights in the network. The regularisation term is defined as:

$$R(W) = \frac{\lambda}{2}\sum_{i=1}^{N} W_i^2$$

Where:

- 'λ' is the L2 regularisation strength.
- 'W' is the weight of the layer.

The results of $L(y, \widehat{y})$ and R(W) are combined to calculate the total loss.

Pseudocode

Forward Propagation

```
FUNCTION forward(predicted, trueValues)
    SampleLoss ← [ ]
    FOR i FROM 0 TO LENGTH(predicted)-1
        T ← trueValue[i]
        P ← predicted[i]
        SampleLoss.APPEND( -(T * log(P)) + (1 - T) * log(1 - P) )
    END FOR
    RETURN SampleLoss
END FUNCTION
```

Backpropagation

```
FUNCTION backward(predicted, trueValues)
    Dinputs ← [ ]
    FOR i FROM 0 TO LENGTH(predicted)-1
        T ← trueValue[i]
        P ← predicted[i]
        Dinputs.APPEND( (P - T) / ((1 -P) * P))
    END FOR
    RETURN Dinputs
END FUNCTION
```

Regularisation

```
FUNCTION regularisation(layerWeights)
    weightsSquaredSum ← SUM( Multiply (layerWeights, layerWeights) )
    regLoss ← 0.5 * regularisationStrength * weightsSquaredSum
    RETURN regLoss
END FUNCTION
```

Optimiser

Stochastic Gradient Descent with Momentum

Stochastic Gradient Descent (SGD) is a popular optimization algorithm used for training machine learning models, including neural networks. It is an iterative optimization algorithm that updates the parameters of the model in small steps to minimise a given loss function. When utilising an optimiser each layer is individually passed through the optimiser to have its parameters be updated.

The optimizer is will implement will also include 2 additional functionalities:

- Learning rate decay
  - By gradually decreasing the learning rate we can avoid overshooting the optimal parameters.
  - If the learning rate is too high it can overshoot the optimal parameters however if it is too low the model won't be able to train properly.
- Momentum
  - Convergence and oscillation can be reduced by incorporating momentum, which adds a fraction of the previous update to the current update step.
  - This helps to smooth out the updates and overcome the problem of slow convergence in regions with small gradients.

Learning Rate

The learning rate is the amount by which the optimiser adjusts the weights and biases in a layer. As mentioned before, we allow the optimiser to avoid overshooting the optimal parameters, by reducing the amount by which the weights and biases are changed the more the epochs are run.

The formula for linear decay is:

$$Lr_i = \frac{Lr_1}{(1 + \alpha \times i)}$$

For the optimiser I do not plan to utilise exponential decay, however the formula it is:

$$Lr_i = Lr_1 \times e^{(-\alpha \times i)}$$

Where:

- 'Lr_1' is the initial learning rate
- 'Lr_i' is the current learning rate.
- 'α' is the decay (usually < 0.1).
  - $0 \leq \alpha \leq 0.1$
- 'i' is the number of forward passes completed (current epoch).

Momentum

As mentioned before, momentum works by adding a small fraction (which we will call velocities) of the previous update to promote updates in one direction, thus reducing fluctuations.

The following formula is used to calculate the new velocities for the weights and biases.

$$V_{i+1} = (\gamma \times V_i) - (Lr_i \times dval)$$

Where:

- 'V_i' is the previously calculated velocity.
- 'V_{i+1}' is the new calculated velocity for the layer.
- 'γ' the momentum used (usually ~0.9 to ~0.99).
    - $0 \leq \gamma \leq 1$
- 'dval' is the dweights and dbiases for the layer.

The following formula is used to update the bias and weights:

$$P_{i+1} = P_i - (Lr_i \times V_{i+1})$$

Where:

- 'P_i' is the previous layer parameters.
- 'P_{i+1}' is adjusted layer parameters.
- 'V_{i+1}' is the calculated velocity for the layer.
- 'Lr_i' is the current learning rate.

Pseudocode

Learning Rate Decay

```
FUNCTION decay(iter)
    newLearningRate ← initalLearningRate / (1 + decay * iter)
END FUNCTION
```

Layer Updates (with momentum)

```
FUNCTION updateLayerParameters(layer)
    oldWeightVelocities ← layer.getWeightVelocities ()
    oldBiasVelocities ← layer.getBiasVelocities ()

    oldWeights ← layer.getWeights ()
    oldBiases ← layer.getBiases ()
```

```
        weightUpdates ← Multiply (activeLearningRate, layer.dweights )
        biasUpdates ← Multiple ( activeLearningRate, layer.dbiases )

        velocityWeightUpdates ← Multiply (momentum, oldWeightVelocities )
        velocityBiasUpdates ← Multiply (momentum, oldBiasVelocities )

        newWeightVelocites ← [ ]
        FOR i FROM 0 TO len(weightUpdates)-1
            sampleUpdates ← [ ]
            sampleVelocityUpds ← velocityWeightUpdates [i]
            sampleWeightUpds ← weightUpdates [i]
            FOR j FROM 0 TO len(sampleWeightUpdates)-1
                sampleUpds.APPEND ( sampleVelocityUpds[j] - sampleWeightUpds [j] )
            END FOR
            newWeightVelocites.APPEND (sampleUpds)
        END FOR

        newBiasVelocites ← [ ]
        FOR i FROM 0 TO len(baisUpdates)-1
            sampleVelocityUpdates ← velocityBiasUpdates [i]
            sampleBiasUpdates ← biasUpdates [i]
            newBiasVelocites.APPEND( sampleVelocityUpdates - sampleBiasUpdates  )
        END FOR

        layer.setWeightVelocities (newWeightVelocites)
        layer.setBiasVelocities (newBiasVelocites)

        newWeights ← [ ]
        FOR x FROM 0 TO LENGTH (oldweights)-1
            Hold ← [ ]
            FOR y FROM 0 TO LENGTH (oldWeights[x])-1
                Hold.APPEND (oldWeights[x][y] + newWeightVelocites[x][y]])
            END FOR
            newWeights.APPEND (Hold)
        END FOR

        newBaises ← [ ]
        FOR x FROM 0 TO LENGTH(biases)-1
            newBiases.APPEND ( oldBiases[x] + newBiasVelocites[x] )
        END FOR

        layer.setWeights (newWeights)
        layer.setBiases (newBiases)
    END FUNCTION
```

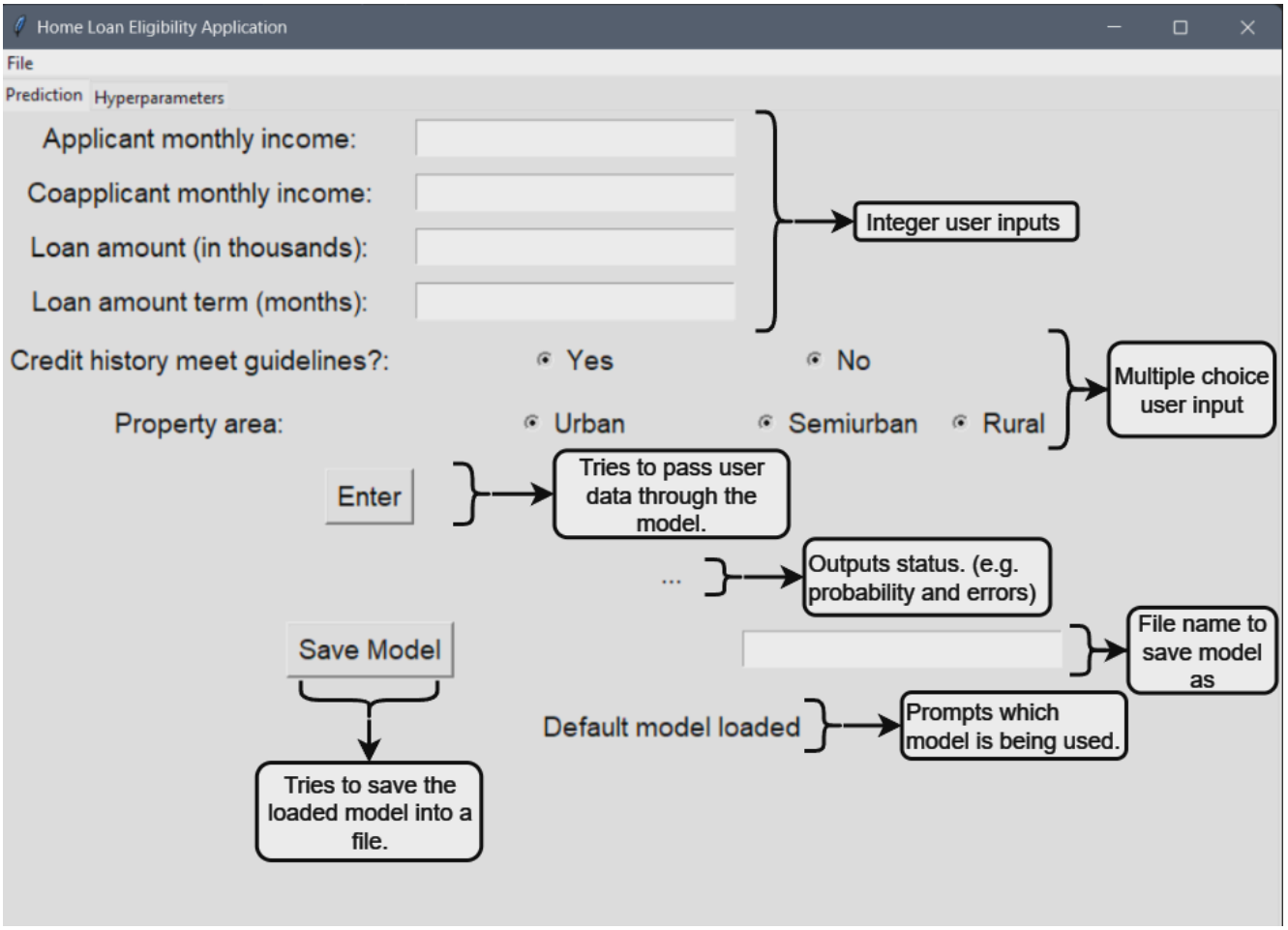
Breakdown of GUI

As requested by the client, Mr. Weiren, the graphical user interface (GUI) needs to be simplistic, easy to use and efficient. As such there will be 3 main aspects to the interface:

1. Prediction UI
2. Hyperparameter adjustment
3. Loaded model control

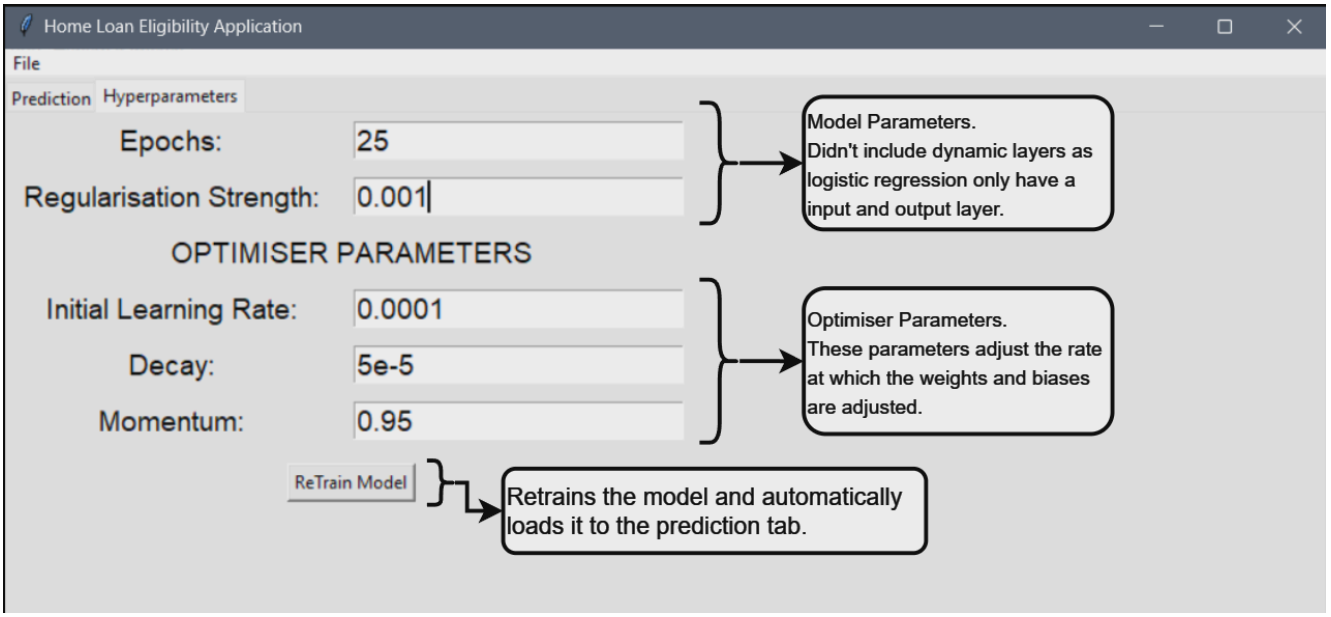
Prediction UI

This section of the GUI will be a tab (as shown below) which will allow the user to enter their data, predict and save a model.



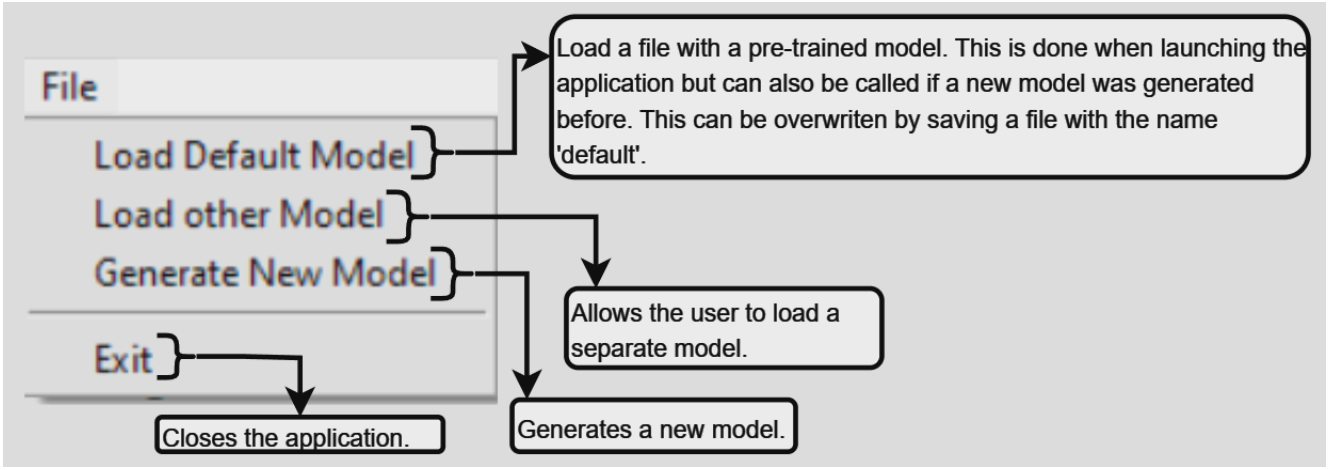
Hyperparameter Adjustment UI

This section of the GUI will also be a tab that allows the user to adjust the hyperparameters of the model and train a new one.



Model Control

This section of the GUI will be a drop down menu that allows the user to change the model that is being used.



Technical Solution

Evidence of Programming Techniques

Technical Skills

Group	Technical Skill	Description	Evidence	Example Pages
A	Lists, graphs, or structures of equivalent standard	Neural networks are implemented as graphs where layers of nodes (vertices) are connected with weights (weighted edges). There is also use of data structures such as lists and matrices used.	The whole Neural Network package Data Handling	62 65 52 - 53
	Complex mathematical model	A neural network is a complex mathematical model consisting of various mathematical functions such as ReLU, Sigmoid, binary cross entropy and stochastic gradient descent and their derivatives.	Neural Network package	61 - 62 65 - 66 67 - 68
	Complex user - defined use of object - oriented programming (OOP) model	Extensive use of classes, inheritance, composition, and polymorphism to implement the neural network model and other classes.	The whole program.	57 63 - 64
	Advanced matrix operations	Program involves matrix operations throughout the program as all data is stored as a matrix. For instance when propagating through the neural network and handling data.	Utilised throughout the program but complex functionality performed through the DataMethods class	52 - 53
	Recursive algorithms	Used to scale matrices of data across the program.	Utilised in the Multiply method in DataMethods class, which is further accessed and used across the program.	53

	Complex user-defined algorithms	Use of optimisation techniques like stochastic gradient descent implemented with momentum and decay. Custom preprocessor to clean and prepare raw data for utilisation and feature engineering	OptimiserSGD class Preprocess class	68 - 69 55 - 57
	Dynamic generation of objects based on complex user-defined use of OOP model	The program involves creating and manipulating objects such as layers, models, and preprocessors.	Whole Program	46 57
B	Dictionaries	Used to store various data such as scaling, mapping, user data to name a few.	Preprocess class GUI class	47 - 48 54
	File(s) organised for sequential access	Utilised to store and load different models, to and from a txt file. Used to save trained models.	CsvToArray module in DataMethod class. LogisticRegression class	52 60
	Text files	Use of .csv files to store training data. Use of .txt files to store models.	LogisticRegression class	60

Coding Style

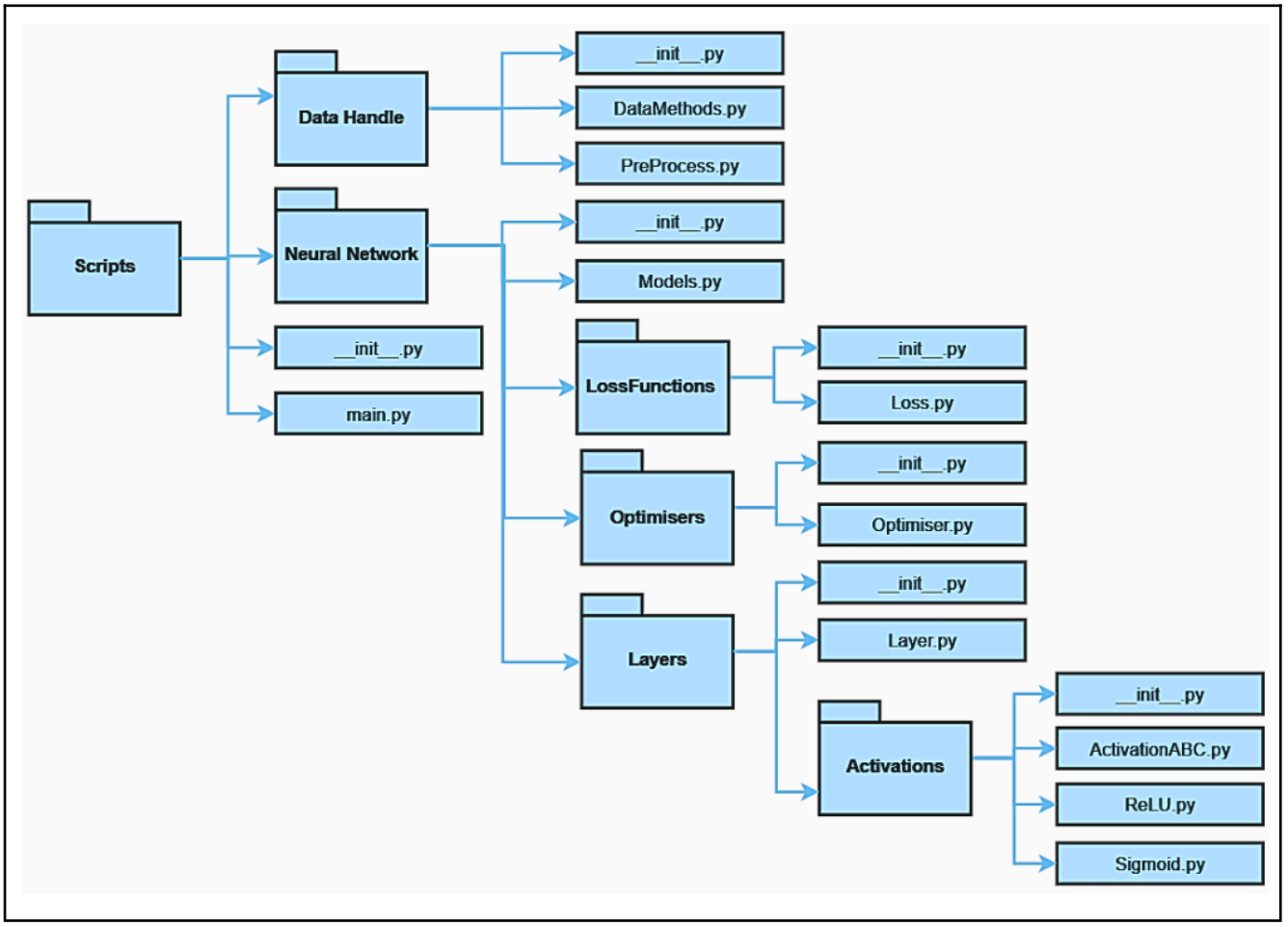
Style	Characteristic	Explanation/Usage
Excellent	Modules (subroutines) with appropriate interfaces	The GUI class serves also as a central controller, interfacing with Preprocess and Logistic Regression classes using Tkinter. For example, functions such as train and predict from the neural network module can be utilised through this.
	Loosely coupled modules (subroutines) – module code interacts with other parts of the program through its interface only.	Modules are packaged and interact through their interfaces only, enhancing modularity. The GUI class allows DataMethod and Neural Network packages to communicate, demonstrating effective use of encapsulation and interface-oriented design.

	Cohesive modules (subroutines) – module code does just one thing.	Each module in the program is designed to perform a specific task or handle a specific aspect of functionality. For example, the DataHandle module includes functions for data preprocessing, while the NeuralNetwork module handles neural network operations. All these tasks are further broken down into smaller methods with single responsibilities.
	Modules (collection of subroutines)	Subroutines that are commonly utilised across the program are grouped together as static classes. For example, common matrix functions are grouped in the DataMethods static class. There are also a couple of independent subroutines, like ‘clipEdges’ in Loss.
	Defensive programming	The program includes input validation checks to ensure that only valid user input is processed by the neural network model, while also often checking for potential errors during processing.
	Good exception handling	Try-except blocks are used in parts of the code that are vulnerable or prone to errors, mainly during data processing or model training.
Good	Well-designed user interface.	Features a simple, clear, and organised UI layout for ease of navigation, as requested by the client.
	Modularisation of code	Program is organised into modules and packages to promote code organisation and reusability. For instance, all neural network related modules are packaged together, which are made up of subpackages, which are indicated with a __init__.py file.
	Good use of local variable	All variables are encapsulated into classes with different scopes of private, protected and public.
	Minimal use of global variables	There are no instances of global variables in my program.
	Use of constants	Utilises constants for class variables where appropriate, such as default parameters and immutable data, facilitating easier adjustments and maintaining consistency.
	Appropriate indentation	The program uses consistent tab spacing or indentation levels, to improve readability and maintainability.
	Self-documenting	Comments and documentation are provided throughout the codebase, explaining the purpose and functionality of classes, methods, and variables.
Basic	Meaningful identifier names	Variables, functions, and classes in my program have descriptive names that clearly convey their functionality.

	Annotation used effectively where required	The program has been annotated throughout defining all the variables and functions, providing detailed explanations on each class.

File Structure

The following diagram only shows the script folder structure. This doesn't include the data and test folders.



In the next section, the program will be displayed grouped by their folders (indicated '/' after the folder name) and files (indicated by their extension). Rather than how it is displayed above.

Full Code

Scripts/

__init__.py

```
from .DataHandle import DataMethod
from .DataHandle import Preprocess

from .NeuralNetwork import LogisticRegression

__all__ = [
    "DataMethod",
    "Preprocess",
    "LogisticRegression"
]
```

main.py

```
from NeuralNetwork import LogisticRegression as Model
from DataHandle import Preprocess

import tkinter as tk
from tkinter import ttk, simpledialog

'''
User Interface

'''
class GUI:
    def __init__(self):
        self.__model = Model()                      # Neural Network model
        self.__model.addLayer(NoOfInputs=6, NoOfNeurons=1)    # adding Layers
        self.__Preprocessor = Preprocess()          # For preparing data

        # Loading the GUI window
        self.root = tk.Tk()
        self.root.protocol("WM_DELETE_WINDOW", self.__exit)
        self.root.title("Home Loan Eligibility Application")
        self.root.geometry("900x600")

        # Create a notebook (tabs container)
        self.__notebook = ttk.Notebook(self.root)
        self.__notebook.pack(expand=True, fill='both')

        # Setting variables
        self.__Font = ('Arial', 14)     # Default font to use
        self.__training = False         # Pauses user input if model is being trained
```

```
        # Adjustable hyperparameters
        self._updateValsTo = {"newEpoch": tk.IntVar(value = 25),
                              "newRegStr": tk.DoubleVar(value = 0.001),
                              "initialLr": tk.DoubleVar(value = 0.0001),
                              "decay": tk.DoubleVar(value = 0.00005),
                              "momentum": tk.DoubleVar(value = 0.95)}

        # Tkinter variable for outputs and user inputs
        self._resultVal = tk.StringVar(value="...")
        self._saveStatusVal = tk.StringVar(value="Default model loaded")
        self._fileName = tk.StringVar()

        self._CreateTabs()

    # Creates the different tabs (eg. Prediction and hyperparameter adjustment)
    def _CreateTabs(self):
        self._LoadMenu()

        self.__LoadPredictionGUI()
        self.__loadDefault()

        self.__LoadHyperparametersTab()

    # Dropdown menu that allows to change the model used (eg. train new or load default)
    def _LoadMenu(self):
        # Create a menu
        menuBar = tk.Menu(self.root)
        self.root.config(menu=menuBar)

        Menu = tk.Menu(menuBar, tearoff=0)
        menuBar.add_cascade(label="File", menu=Menu)
        Menu.add_command(label="Load Default Model", command=self.__loadDefault)
        Menu.add_command(label="Load other Model", command=self._loadModel)
        Menu.add_command(label="Generate New Model", command=self.__newModel)
        Menu.add_separator()
        Menu.add_command(label="Exit", command=self.__exit)

    # This tab allows users to change hyperparameters and retrain model
    def __LoadHyperparametersTab(self):
        # Create a new frame for the Hyperparameters tab
        hyperparameterFrame = ttk.Frame(self.__notebook)
        self.__notebook.add(hyperparameterFrame, text="Hyperparameters")

        # Add widgets for adjusting hyperparameters
        # Model Parameters
        tk.Label(hyperparameterFrame, text="Epochs:", font=self.__Font).grid(
            row=0, column=0, padx=10, pady=5)
        tk.Entry(hyperparameterFrame, textvariable=self._updateValsTo["newEpoch"], font=self.__Font).grid(
            row=0, column=1, padx=10, pady=5)

        tk.Label(hyperparameterFrame, text="Regularisation Strength:", font=self.__Font).grid(
            row=1, column=0, padx=10, pady=5)
        tk.Entry(hyperparameterFrame, textvariable=self._updateValsTo["newRegStr"], font=self.__Font).grid(
            row=1, column=1, padx=10, pady=5)
```

```
        # Optimiser Parameters
        tk.Label(hyperparameterFrame, text="OPTIMISER PARAMETERS", font=self.__Font).grid(
            row=3, column=0, columnspan=2, padx=10, pady=5)

        tk.Label(hyperparameterFrame, text="Initial Learning Rate:", font=self.__Font).grid(
            row=4, column=0, padx=10, pady=5)
        tk.Entry(hyperparameterFrame, textvariable=self._updateValsTo["initialLr"], font=self.__Font).grid(
            row=4, column=1, padx=10, pady=5)

        tk.Label(hyperparameterFrame, text="Decay:", font=self.__Font).grid(
            row=5, column=0, padx=10, pady=5)
        tk.Entry(hyperparameterFrame, textvariable=self._updateValsTo["decay"], font=self.__Font).grid(
            row=5, column=1, padx=10, pady=5)

        tk.Label(hyperparameterFrame, text="Momentum:", font=self.__Font).grid(
            row=6, column=0, padx=10, pady=5)
        tk.Entry(hyperparameterFrame, textvariable=self._updateValsTo["momentum"], font=self.__Font).grid(
            row=6, column=1, padx=10, pady=5)

        # Retrain Button
        tk.Button(hyperparameterFrame, text="ReTrain Model", command=self.__updateHyperparameters).grid(
            row=8, column=0, columnspan=3, pady=10)

    # Updates the model's hyperparameters and retrains the model with the new hyperparameters
    def __updateHyperparameters(self):
        try:
            optimiserVals = [item.get() for item in list(self._updateValsTo.values())]

            # Apply hyperparameters to the model
            self.__model.updateEpoch(optimiserVals[0])
            self.__model.updateRegStr(optimiserVals[1])
            self.__model.configOptimiser(optimiserVals[2], optimiserVals[3], optimiserVals[4])

            print("Retraining model...")
            self.__newModel()

        except ValueError:
            print("Invalid input for epochs or regularisation strength. Please enter valid values.")

    # Main Prediction Interface
    def __LoadPredictionGUI(self):
        predictFrame = ttk.Frame(self.__notebook)
        self.__notebook.add(predictFrame, text="Prediction")

        # Data that is passed through the model
        DataToGet = {'Applicant monthly income: ': -1,
                     'Coapplicant monthly income: ': -1,
                     'Loan amount (in thousands): ': -1,
                     'Loan amount term (months): ': -1,
                     'Credit history meet guidelines?: ': ["Yes", "No"],
                     'Property area: ': ["Urban", "Semiurban", "Rural"]}
```

```
        # Creates a list of empty Tkinter string variables to user inputs
        self.UserData = [tk.StringVar() for _ in range(len(DataToGet.keys()))]
        for index, (key, data) in enumerate(DataToGet.items()):
            # Data Prompt
            tk.Label(predictFrame, text=key, font=self.__Font).grid(row=index, column=0, padx=5, pady=5)

            # User inputs
            if type(data) == list:  # Multiple choice option
                for col, option in enumerate(data):
                    tk.Radiobutton(predictFrame, text=option, value=option, variable=self.UserData[index],
                                   font=self.__Font).grid(row=index, column=col + 1, padx=5, pady=5)
            else:                    # Integer inputs
                tk.Entry(predictFrame, textvariable=self.UserData[index], font=self.__Font).grid(
                    row=index, column=1, padx=5, pady=5)

        # Button - Processes data
        tk.Button(predictFrame, text="Enter", font=self.__Font, command=self.__ProcessUserData).grid(
            row=len(DataToGet), column=0, columnspan=2, pady=10)

        # Updateable status prompts
        self._ResultLabel = tk.Label(predictFrame, textvariable=self._resultVal, font=self.__Font)
        self._ResultLabel.grid(row=len(DataToGet) + 1, column=1, columnspan=2, pady=10)
        self._saveStatusLabel = tk.Label(predictFrame, textvariable=self._saveStatusVal, font=self.__Font)
        self._saveStatusLabel.grid(row=len(DataToGet)+4, columnspan=4, pady=10, sticky="WE")

        tk.Button(predictFrame, text="Save Model", font=self.__Font, command=self._saveModel).grid(
            row=len(DataToGet)+3, column=0, columnspan=2, pady=10)
        tk.Entry(predictFrame, textvariable=self._fileName, font=self.__Font).grid(
            row=len(DataToGet)+3, column=2,columnspan=2, pady=10)

    # Processes UserData through the model
    def __ProcessUserData(self):
        if not self.__training:
            try:
                # Converts Tkinter variables into normal data to be preprocessed
                self.CollectedData = []
                for data in self.UserData:
                    self.CollectedData.append(data.get())

                # Ensures all data is valid
                if not ('' in self.CollectedData or len(self.CollectedData) != 6):
                    # Encodes data - More info in DataHandle.py file
                    UserData = self.__Preprocessor.encode(self.CollectedData)
                    self.__model.Predict([UserData])
                    result = round(self.__model.Result * 100)

                    status = f"You have a {result}% chance of being Approved"
                else:
                    status = "Missing or incorrect data."
            except ValueError:
                status = "Error in userdata"
        else:
            status = "Model is training. Please wait."
```

```
        # Updates result prompts
        self._resultVal.set(status)
        self._ResultLabel.config(textvariable=self._resultVal)

    # Generates a new Neural network model using default hyperparameters
    def __newModel(self):
        self._saveStatusVal.set("Generating new model...")
        self._saveStatusLabel.config(textvariable=self._saveStatusVal)

        # Restarts already initialised Preprocess object
        self.__Preprocessor.newDataset()
        TrainX, TrainY, TestX, TestY = self.__Preprocessor.getData()

        valid = False
        while not valid:
            # Trains model with new random data
            self.__model.train(TrainX, TrainY)
            self.__model.test(TestX, TestY)
            accuracy = self.__model.Accuracy

            # Due to model limitations the model will be trained again if it is not at an acceptable accuracy.
            # This is to ensure that the model doesn't overfit ('memorise' training data) or converge on one
            # output (eg. always output 1 prediction like 0.643 or 64.3%)
            if accuracy > 0.74:
                status = f"Valid model generated - Accuracy: {accuracy}    |    Unsaved"
                self.__training, valid = False, True

                self._saveStatusVal.set(status)
                self._saveStatusLabel.config(textvariable=self._saveStatusVal)
            else:
                self.__model.resetLayers()
                self.__Preprocessor.newDataset()
                TrainX, TrainY, TestX, TestY = self.__Preprocessor.getData()

    # Saves model data so that it can be loaded later
    def _saveModel(self):
        if self._fileName.get() != "":
            filePath = f"DataSet\\Models\\{self._fileName.get()}.txt"
            status = self.__model.saveModel(filePath, self.__Preprocessor.getScalingData())
            self._saveStatusVal.set(status)
            self._saveStatusLabel.config(textvariable=self._saveStatusVal)

    # Loads saved models
    def _loadModel(self):
        modelName = simpledialog.askstring("Load Another Model", "Enter model name:")
        filePath = f"DataSet\\Models\\{modelName}.txt"
        scalingData = self.__model.loadModel(filePath)

        if scalingData == None:
            print("File not found. Loading default...")
            self.__loadDefault()
        else:
            self.__Preprocessor.setScalingData(scalingData)
```

```
    # Loads the a pretrained model to save time when launching the program
    def __loadDefault(self):
        filePath = "DataSet\\Models\\default.txt"
        self.__model = Model()      # Resets model incase default hyperparameters were changed.
        self.__model.addLayer(NoOfInputs=6, NoOfNeurons=1)
        scalingData = self.__model.loadModel(filePath)

        self.__Preprocessor.setScalingData(scalingData)
        status = "Default model loaded"
        self._saveStatusVal.set(status)
        self._saveStatusLabel.config(textvariable=self._saveStatusVal)

    # Terminates the window
    def __exit(self):
        self.root.destroy()

# Main loop
def main():
    myGUI = GUI()
    myGUI.root.mainloop()
main()
```

Scripts/DataHandle/

__init__.py

```
from .DataUtils import DataMethod
from .Preprocess import Preprocess

__all__ = [
    "DataMethod",
    "Preprocess"
]
```

DataUtils.py

```
import csv
import random

'''
Commonly used maths functions processes by other programs

'''
class DataMethod:
    @staticmethod
    def CsvToArray(path, maxEntries = -1):
        # loads all data ignoring entries with missing data
        table = []
        counter = 0
        with open(path, "r") as file:
            csvreader = csv.reader(file)
            for row in csvreader:
                if '' not in row:
                    table.append(row)
                    counter += 1

                if counter == maxEntries:
                    break
        file.close()
        return table

    # Swaps row and columns (eg. [[2, 4, 3], [5, 6, 7]] --> [[2, 5], [4, 6], [3, 7]] )
    @staticmethod
    def Transpose(array):
        return [[array[x][y] for x in range(len(array))]
                for y in range(len(array[0]))]
```

```python
    # Performs Dot product on two valid matrices
    # valid if b = c for shapes (a, b) (c, d) | (Rows, Columns)
    @staticmethod
    def DotProduct(arr1, arr2):
        # Ensures they are matrices
        if isinstance(arr1[0], list) and isinstance(arr2[0], list):
            # Checks if they are valid
            if len(arr1[0]) != len(arr2):
                arr1Shape = (len(arr1),len(arr1[0]))
                arr2Shape = (len(arr2),len(arr2[0]))

                raise ValueError(f"arr1: ({arr1Shape}) and arr2: ({arr2Shape}) are not valid.")

            # Empty matrix to hold results
            result = [[0 for _ in range(len(arr2[0]))] for _ in range(len(arr1))]

            for i, row in enumerate(arr1):                          # For each row in arr1
                for j, column in enumerate(DataMethod.Transpose(arr2)):# For each column in arr2
                    result[i][j] = sum(DataMethod.Multiply(row, column))

            return result
        else:
            raise ValueError("Parameters aren't matrices")

    # Performs multiplications involving arrays
    @staticmethod
    def Multiply(arr1, arr2): # Limitation: dimensions of arr1 Must be <= dimensions of arr2
        try:
            if not isinstance(arr1, list): # Ensures it is at least 1 dimensional
                if isinstance(arr2[0], list): # uses inside length of a row
                    arr1 = [float(arr1) for _ in range(len(arr2[0]))]
                else:
                    arr1 = [float(arr1) for _ in range(len(arr2))]

            if isinstance(arr2[0], list):  # Check if arr2 is a 2D array
                if isinstance(arr1[0], list):  # For 2d x 2d
                    return [DataMethod.Multiply(row1, row2) for row1, row2 in zip(arr1, arr2)]
                else: # For 1d x 2d
                    return [DataMethod.Multiply(arr1, row) for row in arr2]
            else: # For 1d x 1d
                return [a * b for a, b in zip(arr1, arr2)]
        except Exception as ex:
            print(f"Datamethod.Multipy error: {ex}")

    # shuffles two lists while maintaining their corresponding element
    @staticmethod
    def ShuffleData(X, Y):
        a = list(zip(X, Y))
        random.shuffle(a)
        X, Y = zip(*a)
        return list(X), list(Y)
```

Preprocess.py

```python
from .DataUtils import DataMethod

import random

'''
Raw Data cannot be processed by the model. Therefore both training and userdata have to be adjusted prior to
training or prediction

'''
class Preprocess:
    def __init__(self, path=r"DataSet//HomeLoanTrain.csv"):
        # Initial data holders for training data
        self._TrainX = []
        self._TrainY = []


        # Data specific to the training data used
        self.__featuresToRemove = ["Loan_ID", "Self_Employed", "Gender", "Education", "Married", "Dependents"]
        self.__CategoricalFeatureKeys = {"Y": 1., "Yes": 1., "Male": 1., "Graduate": 1., "Urban": 1.,
                                         "N": 0., "No": 0., "Female": 0., "Not Graduate": 0., "Semiurban": 0.,
                                         "Rural": 2., "3+": 2.}
        self._ScalingData = {'means': [], 'stds': []}
        self.__path = path

    # Generates a new random dataset
    def newDataset(self):
        # Extract data from file and selecting valid entries/features
        Dataset = DataMethod.CsvToArray(self.__path)
        Dataset = self.__RemoveFeatures(Dataset)
        Dataset = self.__AdjustSkew(Dataset)

        # Feature Engineering - Making data usable for the model

        self.__FeatureColumns = DataMethod.Transpose(Dataset)

        self.__ConvertToInteger()

        self._TrainY = self.__FeatureColumns.pop()

        self.__Standardisation()

        self._TrainX = DataMethod.Transpose(self.__FeatureColumns)

        self._TrainX, self._TrainY = DataMethod.ShuffleData(self._TrainX, self._TrainY)

    # Removes specified features (Attribute) from the dataset
    def __RemoveFeatures(self, Dataset):
        features = DataMethod.Transpose(Dataset)
        filteredFeatures = [row[1:] for row in features if row[0] not in self.__featuresToRemove]
        return DataMethod.Transpose(filteredFeatures)
```

```
    # Fixes skew of the dataset by balancing the number of 'Y' and 'N' labels
    def __AdjustSkew(self, dataset):
        Trues = 0           # No. of approved applications
        Falses = 0          # No. of unsuccessful applications
        NewData = []
        size = 250          # Number of entries to utilise

        while len(NewData) != size:
            index = random.randint(0, len(dataset) - 1)
            row = dataset[index]
            if row[-1] == 'Y' and Trues != size // 2:
                NewData.append(dataset.pop(index))
                Trues += 1
            elif row[-1] == 'N' and Falses != size // 2:
                NewData.append(dataset.pop(index))
                Falses += 1
        return NewData

    # Converts categorical values (such as strings) to integers using a predefined mapping
    def __ConvertToInteger(self):
        for ColInd, features in enumerate(self.__FeatureColumns):
            for EleInd, element in enumerate(features):
                try:                        # For data that is already numerical
                    self.__FeatureColumns[ColInd][EleInd] = float(element)
                except ValueError:          # For data that is categorical
                    if element not in self.__CategoricalFeatureKeys.keys():
                        self.__CategoricalFeatureKeys[str(element)] = sum([ord(x) for x in element]) / 16
                    self.__FeatureColumns[ColInd][EleInd] = self.__CategoricalFeatureKeys[str(element)]

    # Z-score normalisation formula: (data - mean) / standard deviation
    # Improves interpretability and model performance by removing scale difference between features
    def __Standardisation(self):
        for ind, feature in enumerate(self.__FeatureColumns):
            mean = sum(feature) / len(feature)
            StandardDeviation = ((sum([x**2 for x in feature]) / len(feature)) - mean**2) ** 0.5

            # To use when scale userdata properly for the dataset
            self._ScalingData['means'].append(mean)
            self._ScalingData['stds'].append(StandardDeviation)

            # Applying the standardisation
            try:
                self.__FeatureColumns[ind] = [(i - mean) / StandardDeviation for i in feature]
            except ZeroDivisionError as EXP:
                print(f"Mean: {mean} \nSTD: {StandardDeviation}")
                print(f"FeatureColumn: {self.__FeatureColumns[ind]} \n {EXP}")

    # Splits the dataset into training and test sets and returns the data
    def getData(self, split=0.2): # default: 80-20 split
        NumOfTrainData = round(len(self._TrainX) * split)
        TestX = [self._TrainX.pop() for _ in range(NumOfTrainData)]
        TestY = [self._TrainY.pop() for _ in range(NumOfTrainData)]
        return self._TrainX, self._TrainY, TestX, TestY
```

```
    # Getters and Setters used when saving and loading a model
    def setScalingData(self, data):
        self._ScalingData = data

    def getScalingData(self):
        return self._ScalingData

    # Encodes userdata by standardising and mapping categorical values
    def encode(self, UserData):
        try:
            for x, val in enumerate(UserData):
                # Converting into numerical data
                if val in self.__CategoricalFeatureKeys.keys():
                    val = float(self.__CategoricalFeatureKeys[val])
                else:
                    val = float(val)

                # Standardising the data
                UserData[x] = (val - self._ScalingData["means"][x]) / self._ScalingData["stds"][x]

            return UserData
        except Exception as ex:
            print(f"Encode Error: {ex}")
```

Scripts/NeuralNetwork/

__init__.py

```
from .Layer import Layer
from .LossFunctions import BinaryCrossEntropy
from .Optimisers import OptimiserSGD
from .Models import LogisticRegression

__all__ = [
    "Layer",
    "BinaryCrossEntropy",
    "OptimiserSGD",
    "LogisticRegression"
]
```

Models.py

```
from .Optimisers import OptimiserSGD
from .LossFunctions import BinaryCrossEntropy
from .Layer import Layer

from Scripts import DataMethod as DM

import matplotlib.pyplot as plt

'''
Model - Logistic Regression

Acts as the brain of the model, collating and controlling all components needed to perform Logistic
Regression.
'''
class LogisticRegression:
    # Creates a blank model
    def __init__(self, Epochs=25, regularisationStrength=0.001):
        # Tracking variables
        self.Accuracy = 0.0
        self.__regStr = regularisationStrength      # How strongly to penalise the model for strong weights
        self.__Epochs = Epochs                      # How many times the model will see the data

        # Layers
        self.__Layers = []

        # For backpass
        self.__LossFunction = BinaryCrossEntropy(self.__regStr)     # Calculates the model's performance
        self.__Optimiser = OptimiserSGD()                           # Improves the model

    # Configuration Modules
    # Adds new layer to the model
    def addLayer(self, NoOfInputs, NoOfNeurons, Activation="Sigmoid", regularisationStrength=0.001):
        self.__Layers.append(Layer(NoOfInputs, NoOfNeurons, Activation, regularisationStrength))
```

```
    # Resets layer with new, random weights
    def resetLayers(self):
        for layer in self.__Layers:
            layer.initialiseNewLayer()

    # Resets Optimiser and update it's hyperparameters
    def configOptimiser(self, InitialLearningRate=1e-4, decay=5e-5, momentum=0.95):
        self.__Optimiser = OptimiserSGD(InitialLearningRate, decay, momentum)

    # Changes epochs for training
    def updateEpoch(self, epoch):
        self.__Epochs = epoch

    # Changes regularisation strength
    def updateRegStr(self, regStr):
        self.__regStr = regStr
        self.__LossFunction.updateRegStr(regStr)

    # Used for forward propagation through layers
    def __forward(self, data):
        for x, layer in enumerate(self.__Layers):
            if x == 0:
                layer.forward(data)
            else:       # Takes the activated output of the prior layer
                layer.forward(self.__Layers[x-1].activation.outputs)

        return self.__Layers[-1].activation.outputs

    # Trains the model based on input data
    def train(self, X, Y, batch=32, show=False, canGraph=False):
        # Data holders to used when plotting the graph and calculating values
        losses = []
        accuracies = []
        lrs = []
        sampleSize = len(Y)

        # Training loop
        for iteration in range(self.__Epochs):
            # data holders for that Epoch (holds output of each batch)
            accHold = []
            lossHold = []
            learningRateHold = []

            DM.ShuffleData(X, Y)                        # Shuffling dataset - Improves generalisation

            # Using batches - Reduces overfitting by passing smaller groups of data to the model at a time
            for i in range(0, sampleSize, batch):
                xBatch = X[i:i+batch]
                yBatch = Y[i:i+batch]

                # Forward Pass
                result = self.__forward(xBatch)
```

```
            # Evaluating the performance of the model
            self.__LossFunction.forward(result, yBatch)
            self.__LossFunction.calcRegularisationLoss(self.__Layers[-1].getWeightsAndBiases()[0])

            accuracy = sum([1 for x,y in zip(result, yBatch) if round(x)==y]) / len(result)

            # Backward Pass

            # Calculating gradients (explained in Layer and Optimiser files)
            self.__LossFunction.backward(result, yBatch)

            for x, layer in enumerate(self.__Layers[::-1]):
                if x == 0:
                    layer.backward(self.__LossFunction.dinputs)
                else:
                    layer.backward(self.__Layers[-x].dinputs)

            # Optimising the layer parameters - changing weights and biases
            self.__Optimiser.adjustLearningRate(iteration)
            for layer in self.__Layers:
                self.__Optimiser.UpdateParameters(layer)

            # Tracking variables
            accHold.append(accuracy)
            lossHold.append(self.__LossFunction.getLoss())
            learningRateHold.append(self.__Optimiser.activeLearningRate)

        accuracies.append(sum(accHold) / (len(accHold)))
        losses.append(sum(lossHold) / (len(lossHold)))
        lrs.append(sum(learningRateHold) / (len(learningRateHold)))

        # Output evaluation of training loop
        if show:
            self.__DisplayResults(iteration, loss=losses[-1], accuracy=accuracies[-1],
                                  learningRate=lrs[-1])

    # Visualises the training outcomes
    if canGraph:
        self.__graph(accuracies, losses, lrs)

# Tests the model using data it has never seen
def test(self, TestX, TestY, showTests=False):
    result = self.__forward(TestX)

    if showTests:
        for x in range(len(result)):
            print(f"True: {TestY[x]} Predicted: {round(result[x])} Output: {result[x]}")

    self.Accuracy = sum([1 for x,y in zip(result, TestY) if round(x)==y]) / len(result)

# Passes Userdata through model
def Predict(self, UserData):
    self.Result = round(self.__forward(UserData)[0], 4)
```

```
    # Displays observable data - used to interpret issues with the model and manually tune hyperparameters
    def __graph(self, accuracies, losses, lrs, sep=True):
        X = [x for x in range(1, self.__Epochs+1)]
        if not sep:     # All data on same graph
            plt.plot(X, losses, label='Loss')
            plt.plot(X, accuracies, label='Accuracy')
            plt.legend()
        else:           # Different data on separate graphs
            _, ax = plt.subplots(3, 1, figsize=(10, 8))
            ax[0].plot(X, losses, label='Loss')
            ax[1].plot(X, accuracies, label='Accuracy')
            ax[2].plot(X, lrs, label='Learning Rate')
            ax[0].legend()
            ax[1].legend()
            ax[2].legend()
        plt.show(block=False)

    # Outputs evaluation for that iteration
    def __DisplayResults(self, iter, loss, accu, Lr):
        print(f"Iteration: {iter} Loss: {round(loss, 5)} Accuracy: {round(accu, 5)} Lr: {Lr}\n\n")

    # Saves the model in a txt file
    def saveModel(self, filePath, ScalingData):
        try:
            # Save preprocess ting needed for encoding
            file = open(filePath,  "w")
            file.write(f"{ScalingData}\n")
            for layer in self.__Layers:
                weights, biases = layer.getWeightsAndBiases()
                file.write(f"{weights}\n")
                file.write(f"{biases}\n")
            return "Model Saved Successfully."
        except FileExistsError:
            return "Filename already used. Try again."

    # Loads a model from a txt file
    def loadModel(self, filePath):
        self.resetLayers()
        try:
            file = open(filePath,  "r")
            scalingData = eval(file.readline().rstrip())
            for layer in self.__Layers:
                weights = eval(file.readline().rstrip())
                biases = eval(file.readline().rstrip())

                if len(layer.getWeightsAndBiases()[1]) != len(biases):
                    print("Layers don't match... cant load.")
                else:
                    layer.setWeightsAndBiases(weights, biases)

            return scalingData
        except FileNotFoundError as ex:
            print(f"Loading error: {ex}")
```

Scripts/NeuralNetwork/Layer/

__init__.py

```
from .Activations import ReLU, Sigmoid
from .Layer import Layer

__all__ = [
    "ReLU",
    "Sigmoid",
    "Layer"
]
```

Layer.py

```
from .Activations import Sigmoid, ReLU

from Scripts import DataMethod as DM

from math import sqrt
from random import gauss

'''
Layer

Houses a collection of Neurons.
This class automatically applies activation to the layer's output.

'''
class Layer:
    def __init__(self, NoOfInputs, NoOfNeurons, activation, regularisationStrength):
        self.__NoOfInputs = NoOfInputs           # Number of neurons/inputs in the previous layer
        self.__NoOfNeurons = NoOfNeurons         # Number of neurons/inputs in this layer

        # L2 regularisation - Adds a penalty to prevent overfitting and improve generalisation
        self.__regStr = regularisationStrength

        # Initialize Activation
        if activation == "Sigmoid":
            self.activation = Sigmoid()
            self.__Numerator = 1
        elif activation == "ReLU":
            self.activation = ReLU()
            self.__Numerator = 2

        self.initialiseNewLayer()

    # Creates new random weights and biases for the layer.
    def initialiseNewLayer(self):
        # Xavier/Glorot weight initialization
        # Creates a dense layer initialised with a small value
```

```
        scale = sqrt(self.__Numerator / (self.__NoOfInputs+self.__NoOfNeurons))
        self._weights = [[gauss(0, scale) for _ in range(self.__NoOfNeurons)]
                         for _ in range(self.__NoOfInputs)]
        self._biases = [0.0 for _ in range(self.__NoOfNeurons)]

        # Velocity for use by Optimiser
        self._weightsVelocity = [[1e-3 for _ in range(self.__NoOfNeurons)] for _ in range(self.__NoOfInputs)]
        self._biasesVelocity = [1e-3 for _ in range(self.__NoOfNeurons)]

    # Formula = DotProduct(input, weights) + bias
    def forward(self, inputs):
        self.inputs = inputs.copy()

        self.output = [[a+b for a,b in zip(sample, self._biases)]
                       for sample in DM.DotProduct(inputs, self._weights)]

        self.activation.forward(self.output)

    # Calculated dvalues, dinputs, dweights, dbiases which are gradients that
    # helps shows how much these attributes impacted the prediction
    def backward(self, dvalues):

        self.activation.backward(dvalues)
        dvalues = self.activation.dinputs.copy() # gradients from the activation function

        # Layer's dvalues (gradients) to be passes to the next layer
        self.dinputs = DM.DotProduct(dvalues, DM.Transpose(self._weights))

        # Used by Optimiser to adjust weights and biases
        self.dweights = DM.DotProduct(DM.Transpose(self.inputs), dvalues)
        self.dbiases = [sum(x) for x in DM.Transpose(dvalues)]

        # L2 regularisation prevents a single weight from getting to large.
        if self.__regStr != 0:
            DweightsSqr = DM.Multiply(self.__regStr, self._weights)
            self.dweights = [[a+(2*b) for a, b in zip(self.dweights[x], DweightsSqr[x])]
                             for x in range(len(self.dweights))]

    # Getters and setters used by the optimiser to retrieve and adjust the weights and biases
    def getVelocities(self):
        return self._weightsVelocity, self._biasesVelocity

    def setVelocities(self, veloWeights, veloBiases):
        self._weightsVelocity, self._biasesVelocity = veloWeights, veloBiases

    def getWeightsAndBiases(self):
        return self._weights, self._biases

    # Also used to load a pretrained model
    def setWeightsAndBiases(self, weights, biases):
        self._weights, self._biases = weights, biases
```

Scripts/NeuralNetwork/Layer/Activations/

__init__.py

```
from .ActivationABC import Activation
from .ReLU import ReLU
from .Sigmoid import Sigmoid

__all__ = [
    "Activation",
    "ReLU",
    "Sigmoid"
]
```

ActivationABC.py

```
from abc import ABC, abstractmethod

'''
Activations

Introduces more flexibility to the network, allowing it to understand non-linear and more complex
relationships and patterns in the data/between features.

'''
class Activation(ABC):
    # Contains formula to pass data through
    @abstractmethod
    def forward(self, inputs):
        pass

    # Uses the derivative to calculate Dvalues (gradients) which are used to minimise the loss.
    @abstractmethod
    def backward(self, dvalues):
        pass
```

ReLU.py

```
from .ActivationABC import Activation

class ReLU(Activation): # Rectified Linear Unit
    # limits inputs between (>= 0)
    def forward(self, inputs):
        self.__inputs = inputs
        self.outputs = [[max(0, element) for element in entry]
                        for entry in inputs]

    # if input value was > 0 then gradient = 1
    def backward(self, _):
        self.dinputs = [[1 if element > 0 else 0 for element in sample] for sample in self.__inputs]
```

Sigmoid.py

```
from .ActivationABC import Activation
from math import exp

class Sigmoid(Activation):
    def __init__(self):
        # So that a new instance is not created each time the forward() is run
        # Prevents overflow errors with the exp() function
        self.__positive = lambda x: 1 / (exp(-x) + 1)
        self.__negative = lambda x: exp(x) / (exp(x) + 1)

    # Squashes data between 0 and 1
    def forward(self, inputs):
        self.outputs = [self.__negative(val[0]) if val[0] < 0 else self.__positive(val[0]) for val in inputs]

    def backward(self, dvalues):
        self.dinputs = [[a*b*(1-b)] for a,b in zip(dvalues, self.outputs)]
```

Scripts/NeuralNetwork/LossFunctions/

__init__.py

```
from .Loss import BinaryCrossEntropy

__all__ = [
    "BinaryCrossEntropy",
]
```

Loss.py

```
from Scripts import DataMethod as DM

from math import log


'''
Loss

Measures how well the model performed by comparing the true and predicted values

The algorithm doesn't utilise the calculated loss value directly. It is use to visualise if the model is
improving when training and identifying what is impacting the model and how much it does.

'''
class BinaryCrossEntropy:
    def __init__(self, regStr=0.0):
        self.__sampleLoss = 0.0                     # Measure of how far the prediction is from the true value
        self.__regLoss = 0.0                        # Additional term to sampleLoss to deter large weights

        self._regStr = regStr                      # How strongly to penalise the model for large weights

    # Calculates how far the predicted values are from the true values
    def forward(self, predictions, TrueVals):
        predictions = clipEdges(predictions)

        # Formula used: -(true * log(Predicted) + (1 - true) * log(1 - Predicted))
        sampleLoss = [-((tVal * log(pVal)) + ((1 - tVal) * log(1 - pVal)))
                      for tVal, pVal in zip(TrueVals, predictions)]

        self.__sampleLoss = sum(sampleLoss) / len(sampleLoss)   # Average of all samples

    # Gradient of what impacted the result the most
    def backward(self, predicted, TrueVals):
        predicted = clipEdges(predicted)

        # Derivative of formula above used: (PredictVal - Tval) / ((1-PredictVal) * PredictVal)
        self.dinputs = [(PredictVal - Tval) / ((1-PredictVal) * PredictVal)
                        for Tval, PredictVal in zip(TrueVals, predicted)]
```

```
    # Used to change the hyperparameter when training a new model
    def updateRegStr(self, regStr):
        self._regStr = regStr

    # L2 regularisation formula: 0.5 * regStr * SumOfSquaredWeights
    def calcRegularisationLoss(self, layerWeights):
        if self._regStr != 0:
            weightSqrSum = sum([sum(x) for x in DM.Multiply(layerWeights, layerWeights)])

            self.__regLoss = 0.5 * self._regStr * weightSqrSum

    # Returns total loss when called.
    def getLoss(self):
        return self.__sampleLoss + self.__regLoss

# Replaces any 0s or 1s to avoid arithmetic errors
def clipEdges(list, scale=1e-7):
    for index, val in enumerate(list):
        if val < scale:
            list[index] = scale
        elif val > 1 - (scale):
            list[index] = 1 - (scale)
    return list
```

Scripts/NeuralNetwork/Optimisers/

__init__.py

```
from .Optimiser import OptimiserSGD

__all__ = [
    "OptimiserSGD"
]
```

Optimiser.py

```
from Scripts import DataMethod as DM

from math import exp

'''
Optimiser

Improve the accuracy of the model.

Does this by adjusting the weights and biases of layers by adding/subtracting a small amount to the weights
depending on their impact on the model and its output which is calculated in the backpass (utilises the
dvalues).

'''
class OptimiserSGD:
    def __init__(self, InitialLearningRate=1e-4, decay=5e-5, momentum=0.95):
        self.__InitialLearningRate = InitialLearningRate          # Starting Learning rate
        self.__minimumLearningRate = InitialLearningRate * 0.001  # Lower bound Learning rate
        self.__decay = decay                                      # Rate at which Learning rate decreases
        self.__momentum = momentum                                # Promotes adjustment in one direction
        self.activeLearningRate = InitialLearningRate             # How much to adjust/step.

    # Gradually decreases the learning rate to avoid overshooting the optimal parameters
    # If it is too high it will overshoot the optimal but if too low the mode won't train properly.
    def adjustLearningRate(self, iter):
        if self.__decay != 0:
            newLearningRate = self.__InitialLearningRate / (1 + self.__decay * iter)

            self.activeLearningRate = max(newLearningRate, self.__minimumLearningRate)


    # Function to update the parameters of a neural network layer using SGD with momentum
    def UpdateParameters(self, layer):
        # Amount to increment the weights and biases
        weightUpdate = DM.Multiply(self.activeLearningRate, layer.dweights)
        biasesUpdate = DM.Multiply(self.activeLearningRate, layer.dbiases)

        weights, biases = layer.getWeightsAndBiases()
```

```
        if self.__momentum != 0:

            # Amount to added to the weights and biases to reduce fluctuations in accuracy and loss
            prevWeightsVelocity, prevBiasesVelocity = layer.getVelocities()

            # New weight velocity = momentum * Velocity - activeLearningRate * dweights
            newWeightsVelocity = [[a - b for a, b in zip(velocityRow, dweightsRow)]
                                  for velocityRow, dweightsRow in zip(
                                      DM.Multiply(self.__momentum, prevWeightsVelocity), weightUpdate)]

            # New bias velocity = momentum * Velocity - activeLearningRate * dbiases
            newBiasesVelocity = [a - b for a, b in zip(
                                  DM.Multiply(self.__momentum, prevBiasesVelocity), biasesUpdate)]

            # Updates velocities for the layer to be used in the next loop
            layer.setVelocities(newWeightsVelocity, newBiasesVelocity)

            # Final (optimising) updates to the weights and biases of the layer for this loop
            weights = [[a + b for a, b in zip(weights[x], newWeightsVelocity[x])]
                       for x in range(len(weights))]
            biases = [a + b for a, b in zip(biases, newBiasesVelocity)]

        else:
            weights = [[a - b for a, b in zip(weights[x], weightUpdate[x])]
                       for x in range(len(weights))]
            biases = [a - b for a, b in zip(biases, biasesUpdate)]

        layer.setWeightsAndBiases(weights, biases)
```

Testing

Testing Table

Due to limited user interaction with the neural network itself, most data selection and processing objectives (tests 1.x.x) which are related directly to the data, will be investigated via the terminal. Most neural network objectives (tests 2.x.x) will be mainly tested through their output on the UI, with the exception of tests 2.1.x which refer to how the model should be developed.

Furthermore, due to the nature of the application, further unit testing will also be conducted in following sections to verify the components used operate as intended, where there is limited user interaction.

Test No.	Objective	Description	Test data / action	Expected result	Actual result
1	1.2	Raw data is retrieved, then cleaned and prepared for training.	Retrieve raw data from .csv file.	A matrix of data that - No missing data - All data are floats - Is Standardised - Balances the target class	Pass
2	1.3.a	Training dataset is randomised after each training epoch.	Output the first 3 entries at different epochs.	The top 3 entries are different each time.	Fail Entries not shuffled
	1.3.a (fixed)	Training dataset is randomised after each training epoch.	Output the first 3 entries at different epochs.	The top 3 entries are different each time.	Pass
3	2.2.b	When provided with valid data, a correct prediction is made.	[Normal data]	A message relaying the probability of being approved.	Pass
4	2.2.c	When invalid data is provided, an appropriate message is conveyed.	[Erroneous data]	A message relaying that invalid data has been supplied.	Pass
5	2.2.d	When provided with unrealistic data a prediction of 0% is produced.	[Erroneous data] / [Boundary data]	A message relaying the probability of being approved is 0%.	Pass

6	2.3.b	When the application is run a default model is automatically loaded.	Run the application.	A message relaying that the default model is loaded.	Pass
7	2.3.c	After generating a new model, a relevant message is relayed.	Select the ‘generate new model’ option in the dropdown menu.	A message relaying the successful generation and loading of a model.	Pass
8	2.4.b	A new model is trained and loaded when the hyperparameters are changed.	[Normal data] Alter entries in the Hyperparameter tab.	A message relaying the successful generation and loading of a model.	Pass
9	2.4.c.i	When invalid hyperparameters are added, the program prevents a new model from being trained.	[Erroneous data]	A message is relayed about trying to set invalid hyperparameters.	Partial Pass Error is caught but not handled as intended.
	2.4.c.i (fixed)	When invalid hyperparameters are added, the program prevents a new model from being trained.	[Erroneous data]	A message is relayed about trying to set invalid hyperparameters.	Pass
10	2.4.c.ii	The program aborts the generation if it takes too many attempts.	Generate a new model.	If a model is unable to be produced, a message is relayed about the process being aborted.	Fail The program doesn’t detect and crashes.
	2.4.c.ii (fixed)	The program aborts the generation if it takes too many attempts.	Generate a new model.	If a model is unable to be produced, a message is relayed about the process being aborted.	Pass
11	2.5.a.i	Unsaved models can be saved using a new unique filename.	[Normal data] Save model using a new filename.	A model generated model that has a ‘-unsaved’ label turns into ‘saved’.	Pass

12	2.5.b	Models are saved in correct sequential order.	View a .txt file that is stored in the model folder.	Only first 3 lines contain data in the following order: 1). A dictionary of the scaling data 2). A 2D array of weights 3). A list of biases	Pass
13	2.5.a.ii	Unsaved models will not be saved if the filename provided already exists	[Erroneous data] Save the model using a used filename (eg. ‘default’)	The filename is rejected and the model remains unsaved.	Fail Saved model file was overwritten.
	2.5.a.ii (Fixed)	Unsaved models will not be saved if the filename provided already exists	[Erroneous data] Save the model using a used filename (eg. ‘default’)	The filename is rejected and the model remains unsaved.	Pass
14	2.5.c.i	A previously trained model is successfully loaded	[Normal data] Load a using a valid model name.	A message relaying the change in model.	Fail No indication or evidence of successful model change.
	2.5.c.i (Fixed)	A previously trained model is successfully loaded	[Normal data] Load a using a valid model name.	A message relaying the change in model.	Pass
15	2.5.c.ii	An non-existent model is attempted to be loaded.	[Erroneous data] Load a model that doesn’t exist.	A message relaying an invalid model name was provided.	Pass

Test Evidence

Test 1

Further evidence, proving that the data is standardised, the target class is balanced and so on can be found in preprocessor unittests, in the following section.

Test 2

The following screenshot shows the initial test results, Where the entry of data is shown followed by the target value for that entry at 3 random epochs.

```
8).
[-0.466242997958814, 1.9881828744823637, 0.6726097990975001, 0.24596389144917227, 0.5434489742052762, 0.06337242505244785] 1.0
[-0.7133857278118853, -0.03586127787798708, -0.4137754029202205, -2.499168825617485, 0.5434489742052762, -1.1553280567253943] 1.0
[-0.6575497777339692, -0.5433557613813924, -1.0381347144246578, 2.0760523694936106, -1.8400991582740054, -1.1553280567253943] 0.0


14).
[-0.466242997958814, 1.9881828744823637, 0.6726097990975001, 0.24596389144917227, 0.5434489742052762, 0.06337242505244785] 1.0
[-0.7133857278118853, -0.03586127787798708, -0.4137754029202205, -2.499168825617485, 0.5434489742052762, -1.1553280567253943] 1.0
[-0.6575497777339692, -0.5433557613813924, -1.0381347144246578, 2.0760523694936106, -1.8400991582740054, -1.1553280567253943] 0.0


19).
[-0.466242997958814, 1.9881828744823637, 0.6726097990975001, 0.24596389144917227, 0.5434489742052762, 0.06337242505244785] 1.0
[-0.7133857278118853, -0.03586127787798708, -0.4137754029202205, -2.499168825617485, 0.5434489742052762, -1.1553280567253943] 1.0
[-0.6575497777339692, -0.5433557613813924, -1.0381347144246578, 2.0760523694936106, -1.8400991582740054, -1.1553280567253943] 0.0
```

Outputs shows test failed.

This is due to a logical error in the program, where in the LogisticRegression class, after the training and result arrays are shuffled, these values are never updated with the new shuffled arrays. The following screenshots shows the changes to the program that were applied to resolve this issue.

Program before

```
        # Training loop
        for iteration in range(self.__Epochs):
            # data holders for that Epoch (holds output of each batch)
            accHold = []
            lossHold = []
            learningRateHold = []

            DM.ShuffleData(X, Y)                    # Shuffling dataset - Improves generalisation

            # Using batches - Reduces overfitting by passing smaller groups of data to the model at a time
            for i in range(0, sampleSize, batch):
                xBatch = X[i:i+batch]
                yBatch = Y[i:i+batch]
```

Program after

```
        # Training loop
        for iteration in range(self.__Epochs):
            # data holders for that Epoch (holds output of each batch)
            accHold = []
            lossHold = []
            learningRateHold = []

            X, Y = DM.ShuffleData(X, Y)                    # Shuffling dataset - Improves generalisation

            # Using batches - Reduces overfitting by passing smaller groups of data to the model at a time
            for i in range(0, sampleSize, batch):
                xBatch = X[i:i+batch]
                yBatch = Y[i:i+batch]
```

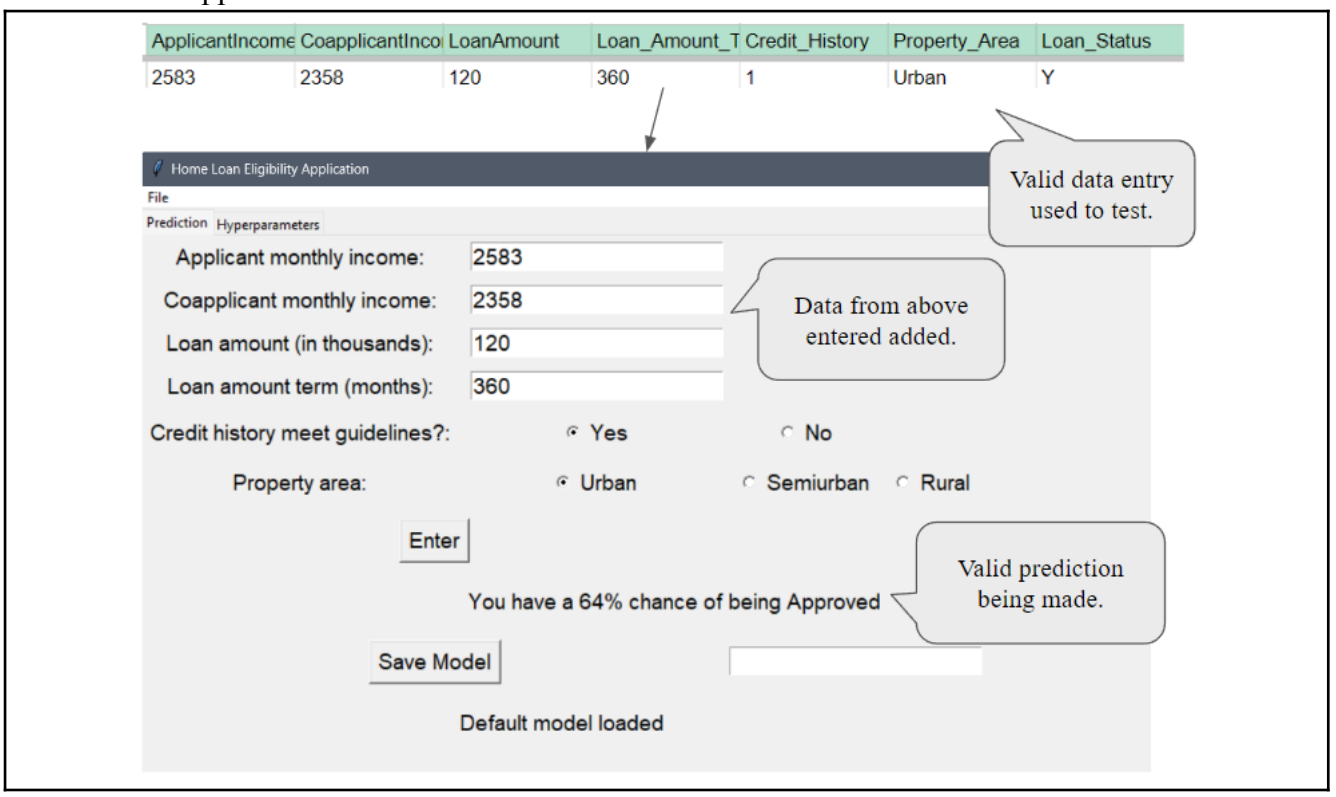
The following screenshot shows the issue has been resolved and the program is working as expected.

```
1).
[0.26987440220401354, -0.514708180977113, 0.585685002738766, 0.2518477613213998, 0.5434489742052762, 1.2394047054562605] 0.0
[0.5840659784753115, -0.514708180977113, 0.5481699500396393, 0.2518477613213998, 0.5434489742052762, -1.1532684711002268] 1.0
[-0.1088161526789056, -0.514708180977113, -0.1771210688101433, 0.2518477613213998, 0.5434489742052762, 1.2394047054562605] 0.0


3).
[-0.19585270542606992, 0.16995032690866733, 0.585685002738766, 0.2518477613213998, 0.5434489742052762, 1.2394047054562605] 1.0
[-0.2774031381534007, 1.1000524510412368, 0.3605946865440059, 0.2518477613213998, 0.5434489742052762, 0.043068117178016806] 0.0
[-0.46274503071551604, -0.514708180977113, -0.6398067187660391, 1.9863364756561113, 0.5434489742052762, -1.1532684711002268] 0.0


4).
[4.185036540686139, -0.514708180977113, 1.4110161621195532, 0.2518477613213998, 0.5434489742052762, -1.1532684711002268] 0.0
[-0.6196184085800904, -0.514708180977113, -1.4026127903149486, 0.2518477613213998, 0.5434489742052762, 0.043068117178016806] 0.0
[0.06362594416089157, 0.9582934650303626, -0.3396862971730256, -2.3498853101806674, 0.5434489742052762, 0.043068117178016806] 0.0
```

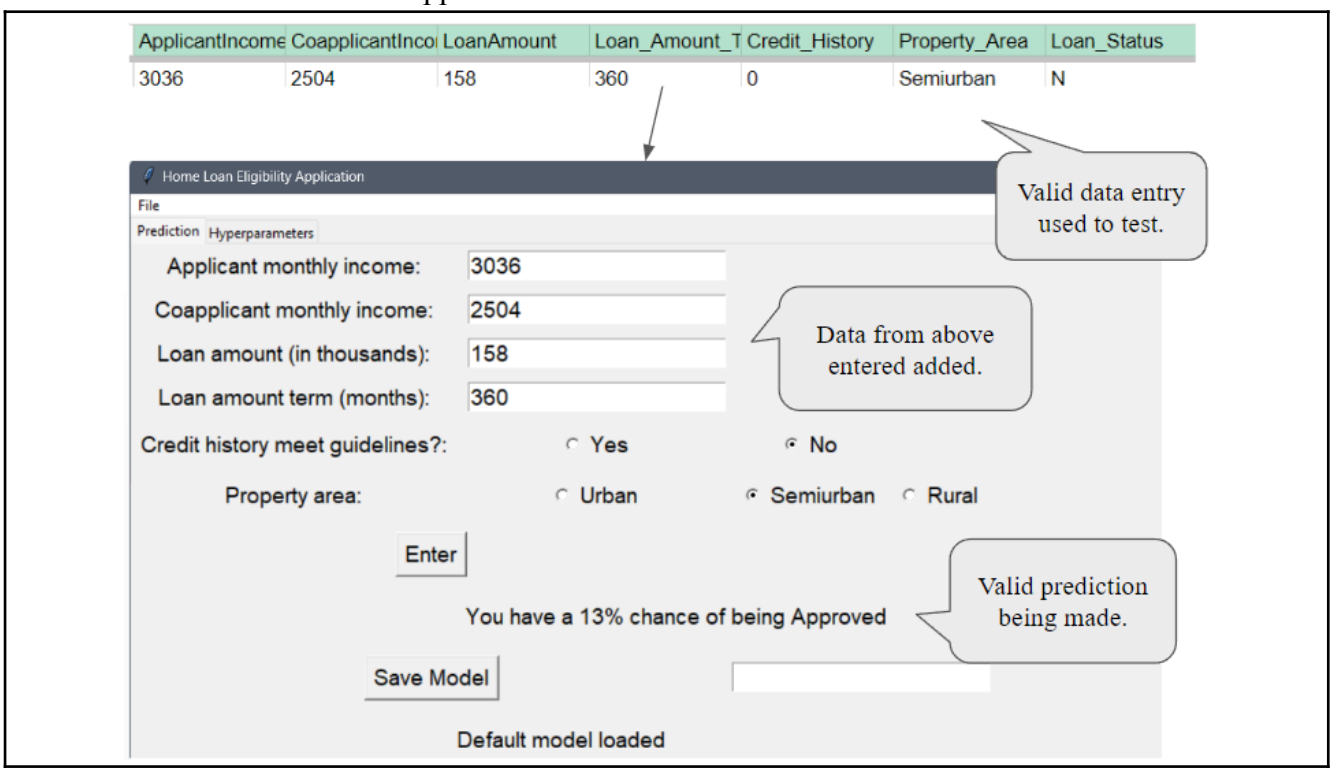
Outputs shows issue resolved.

Test 3

The following screenshot shows a valid prediction being made, when provided with an entry that would be approved.

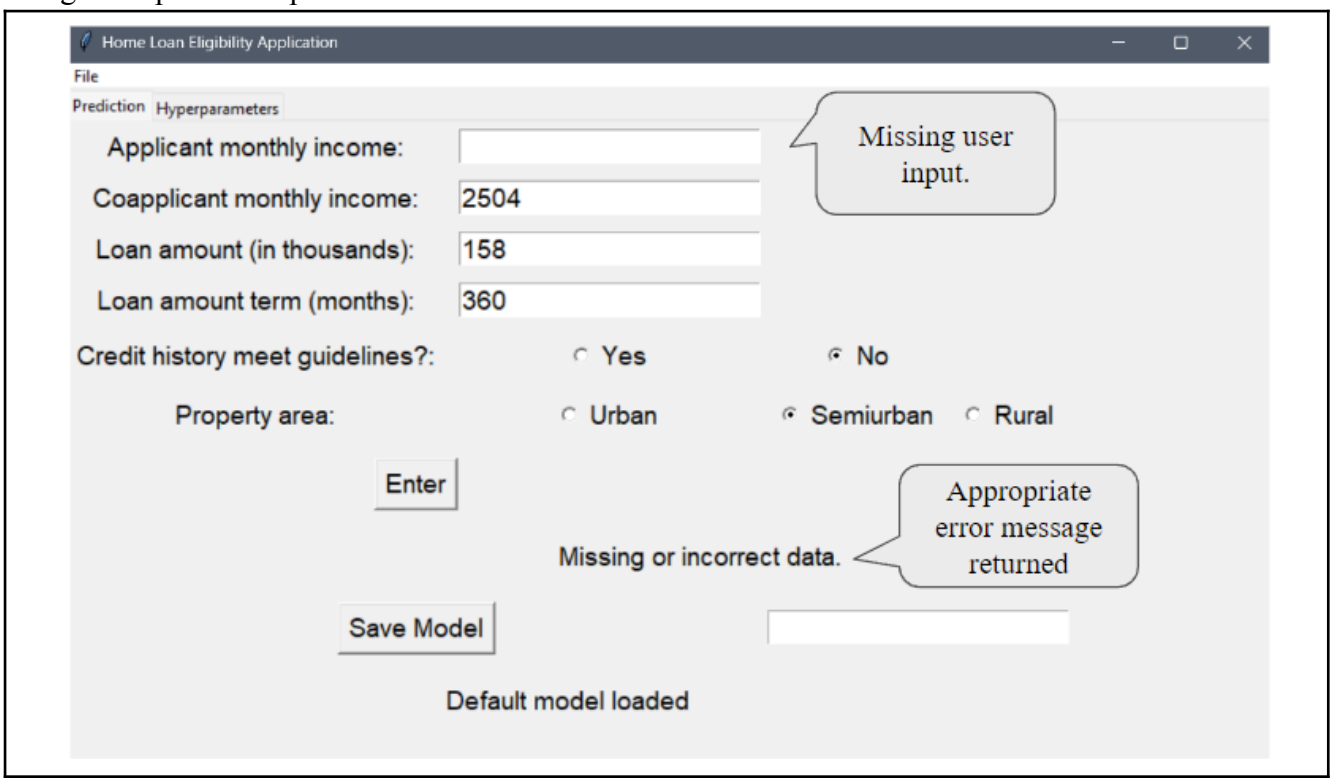


The following screenshot does a valid prediction being made, when provided with an entry that has a low chance / wouldn't be approved.

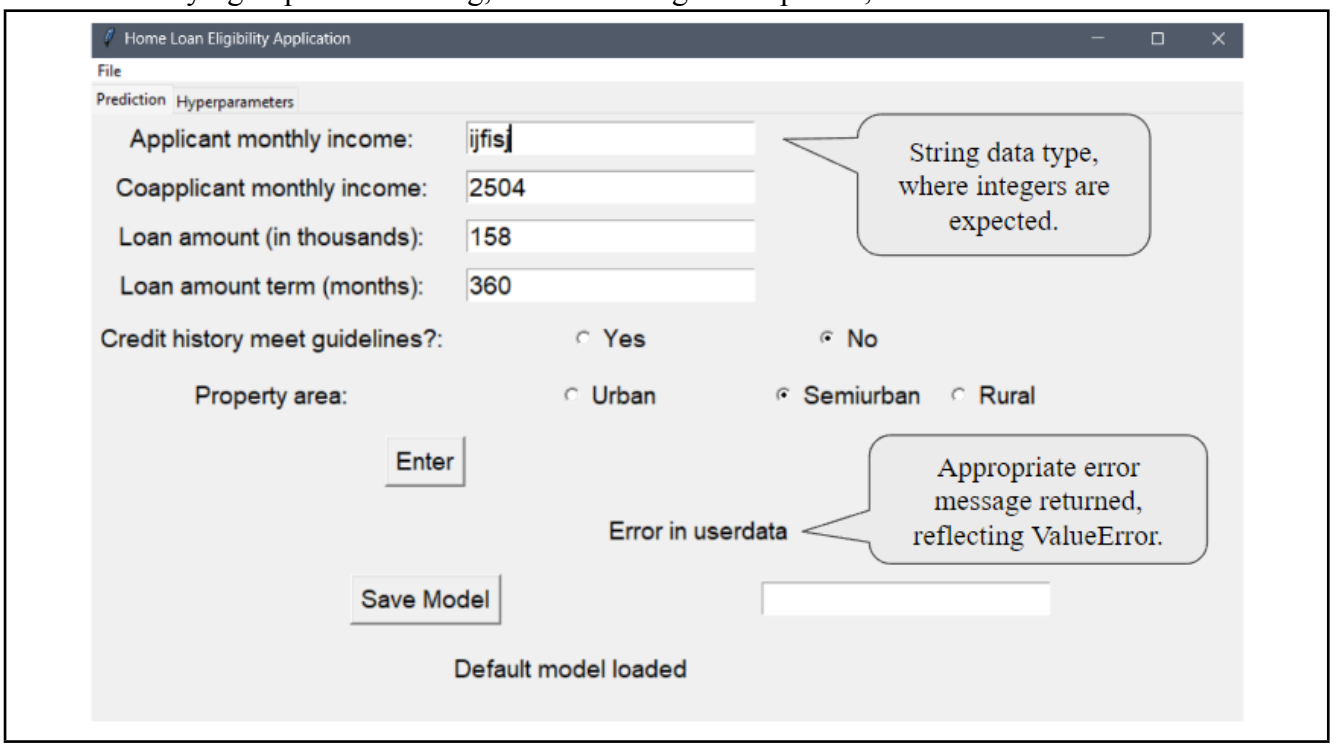


Test 4

The following screenshot, shows how the program correctly handles missing / incorrect data being attempted to be processed.

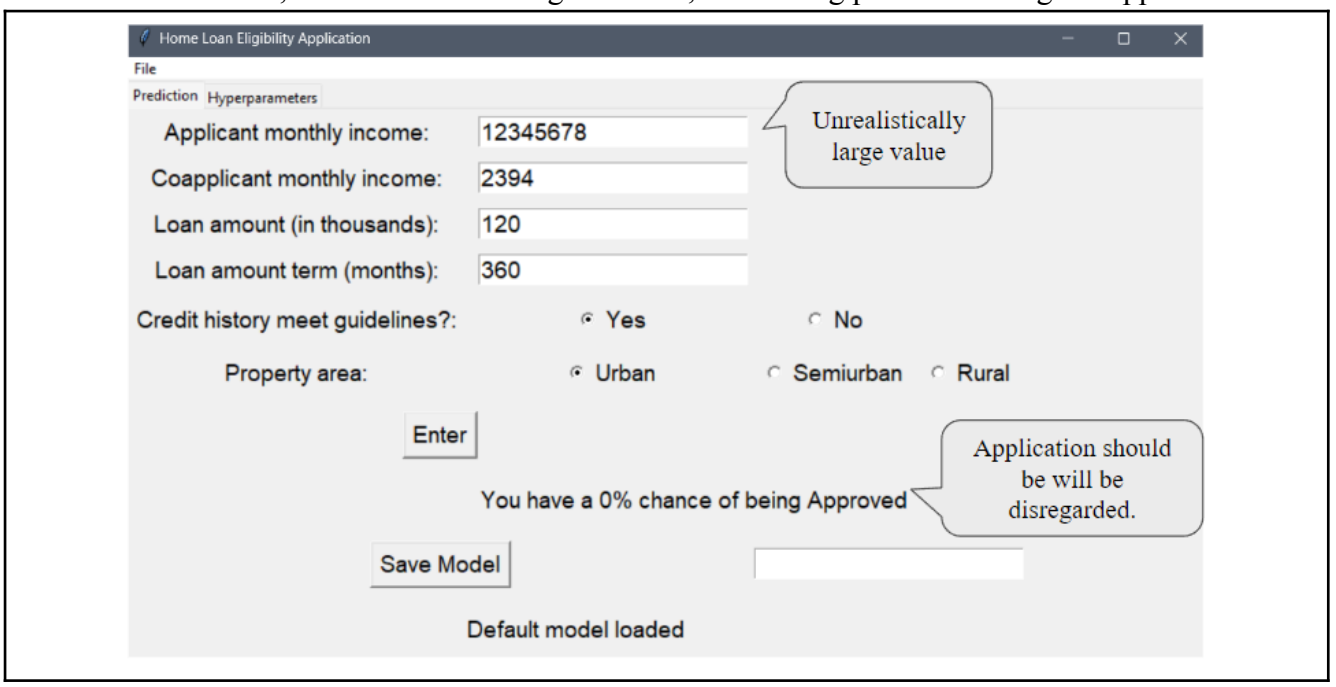


The following screenshot, shows how the program correctly handles a 'ValueError' that will be raised when trying to process a string, where an integer is expected, which cannot be encoded.



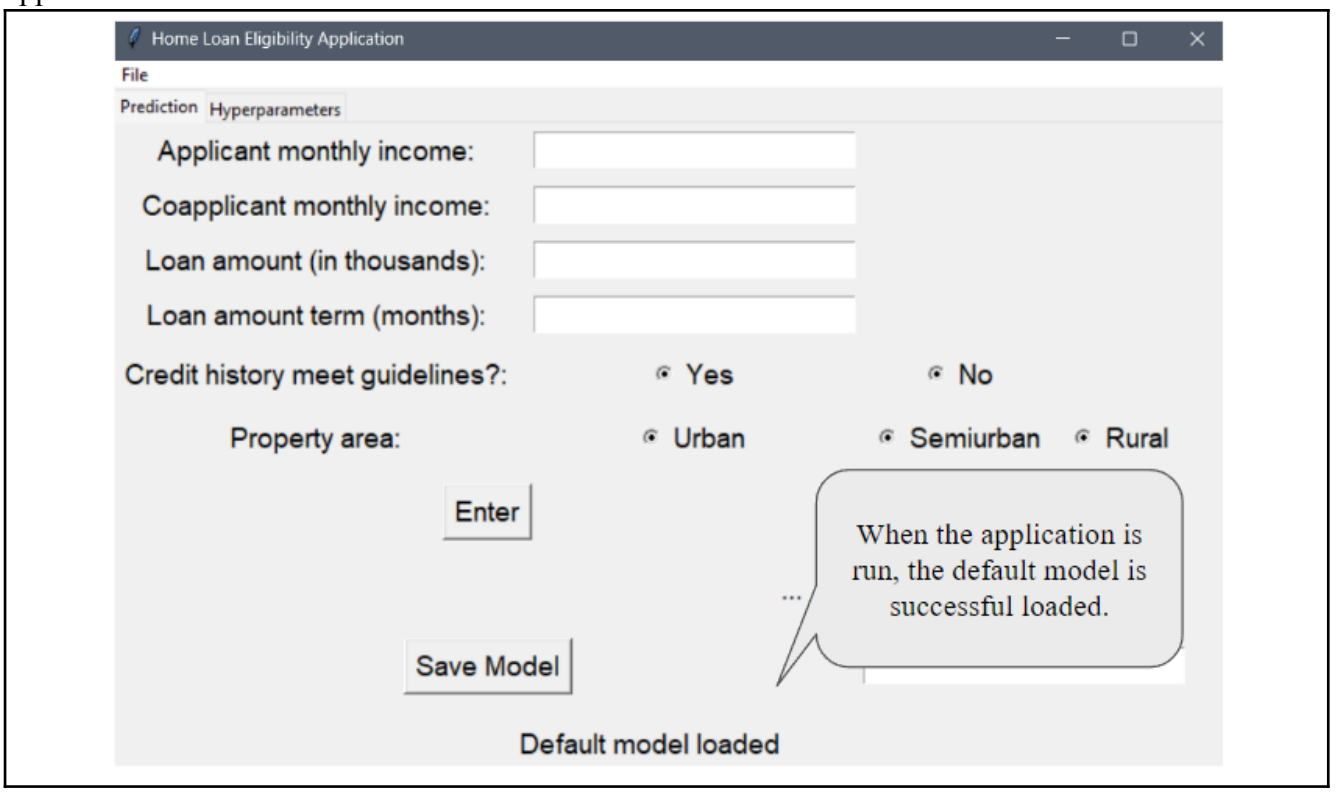
Test 5

When unrealistic data is provided, the system should predict an approval of 0%. This prevents data in a valid format, but doesn't make logical sense, from being predicted as a good application.



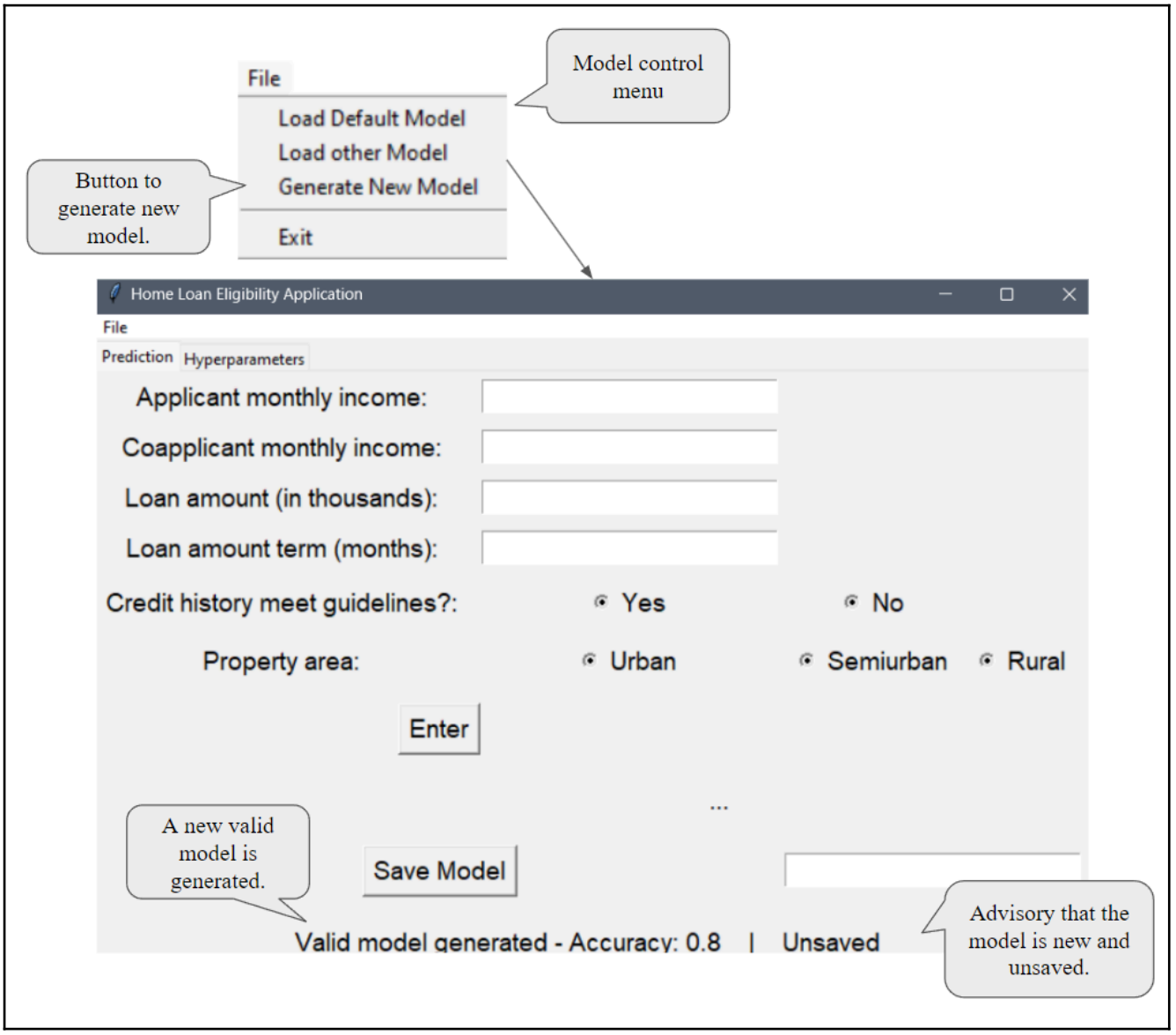
Test 6

The following screenshot shows the default model being successfully loaded when the application is run.



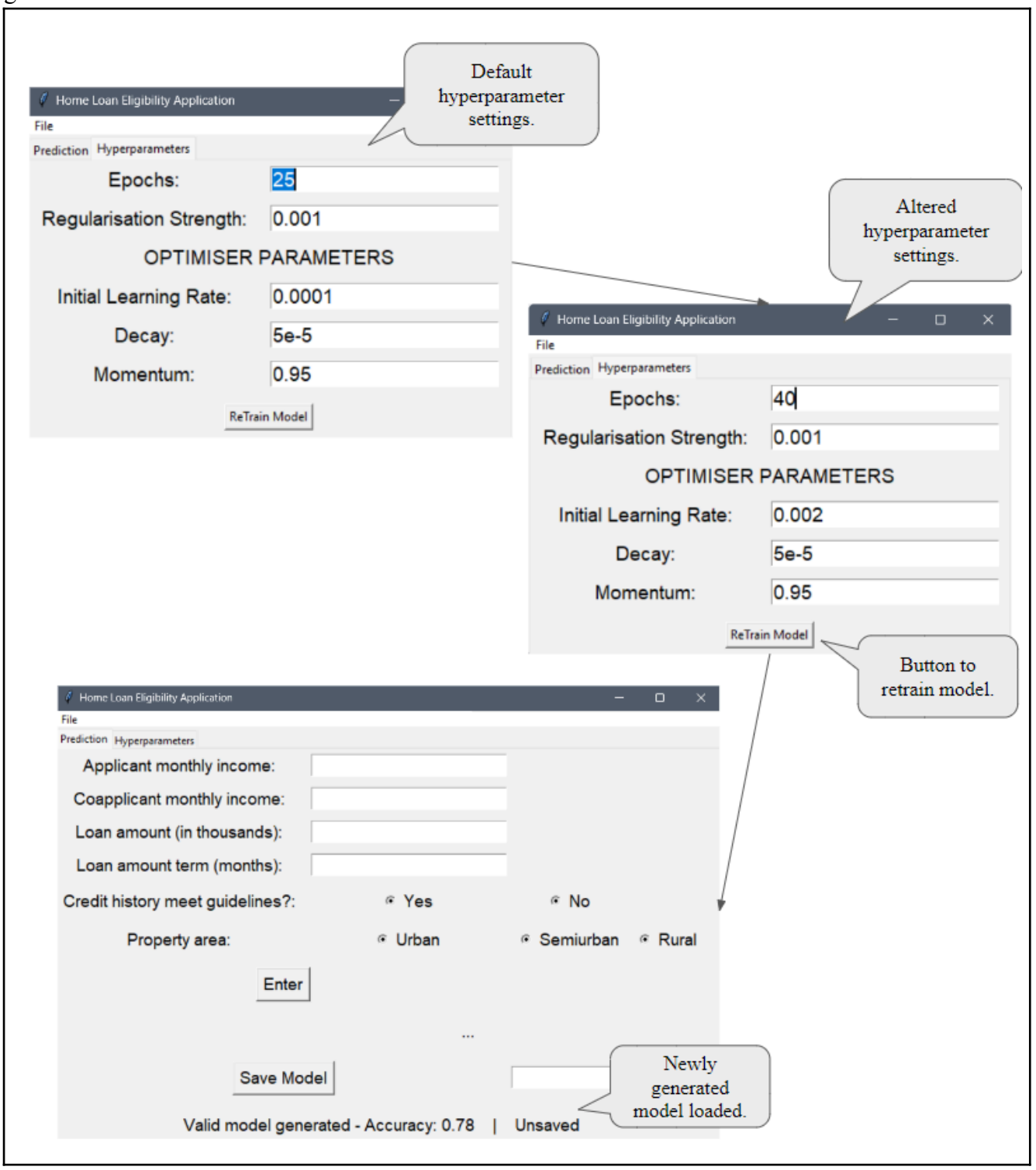
Test 7

The following screenshot shows a new model being successfully generated through the drop down menu.



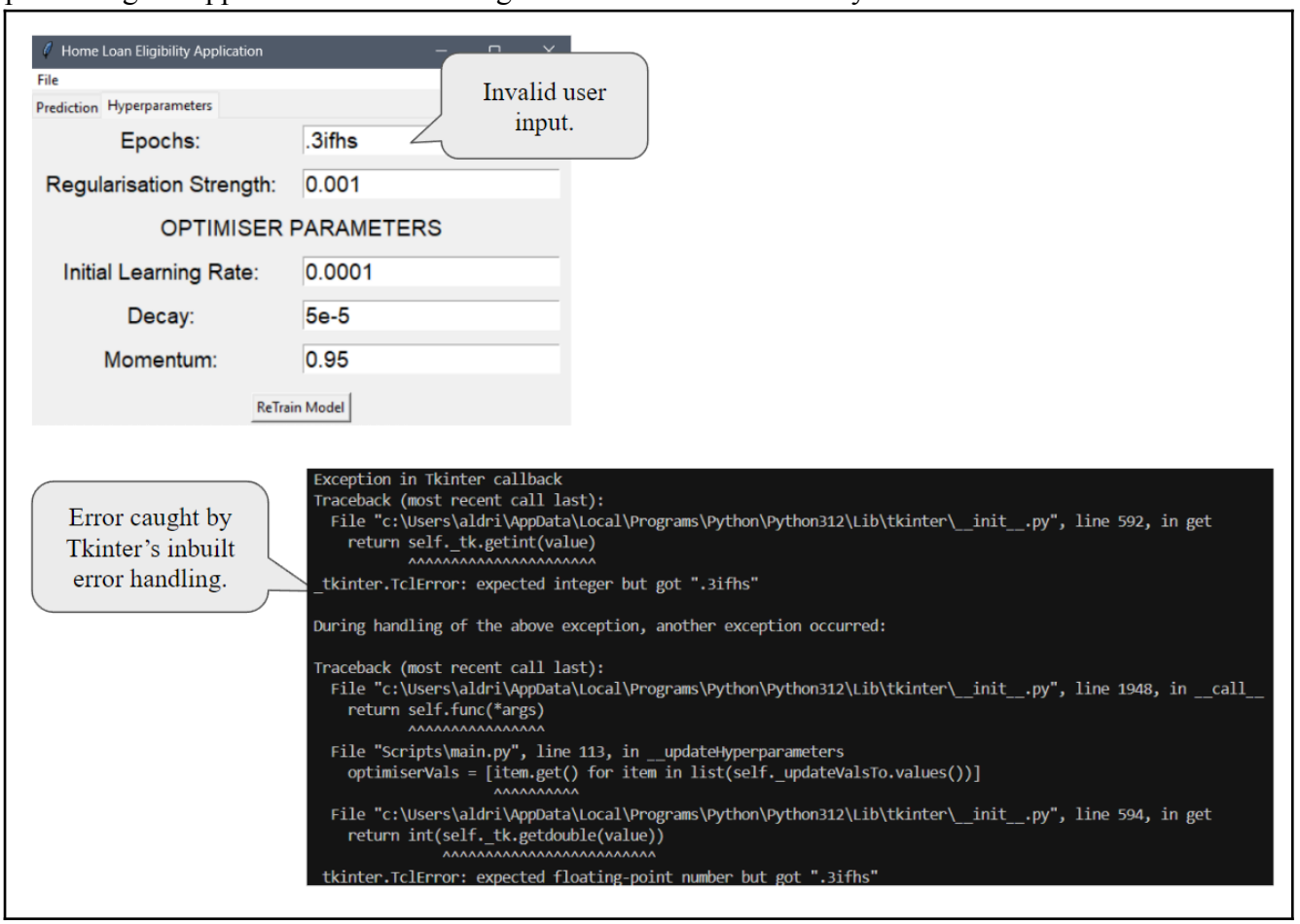
Test 8

The screenshots below show how a model with updated hyperparameters can successfully generate a new model.

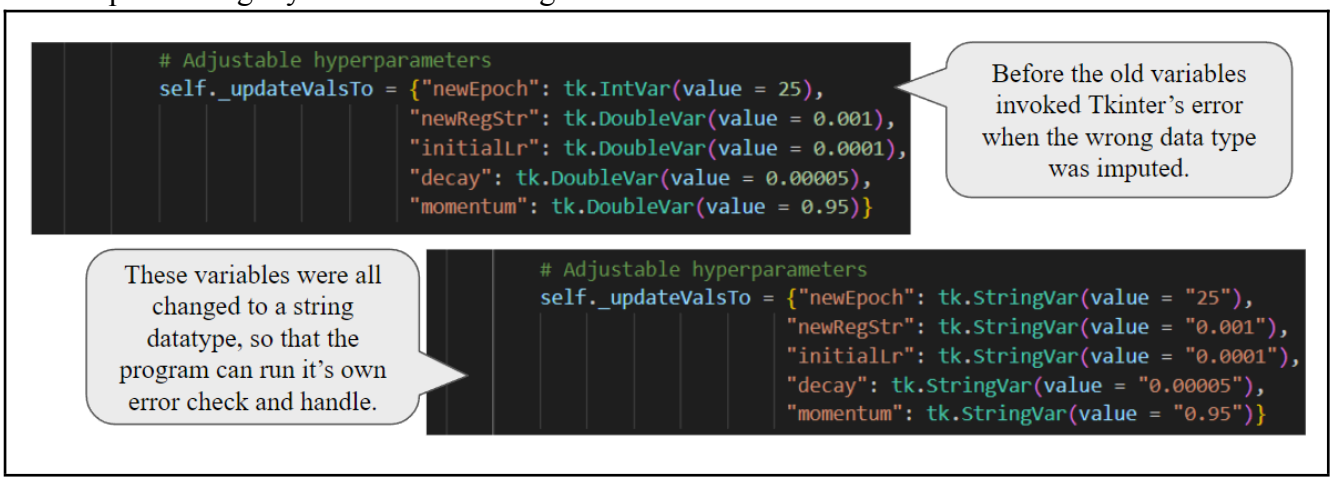


Test 9

The following screenshot, shows an error being caught by Tkinter's inbuilt error handling, preventing the application from crashing. This however doesn't relay to the user the error.



The following changes were made to the GUI class, to let the user know that the retraining process was aborted due to invalid values set. This was previously prevented as when the Tkinter caught an error it would not run the rest of the module and simply exit the method. This prevents me from implementing my own error handling when invalid data is entered.




```
# Adjustable hyperparameters
self._updateValsTo = {"newEpoch": tk.IntVar(value = 25),
                      "newRegStr": tk.DoubleVar(value = 0.001),
                      "initialLr": tk.DoubleVar(value = 0.0001),
                      "decay": tk.DoubleVar(value = 0.00005),
                      "momentum": tk.DoubleVar(value = 0.95)}
```

```
# Adjustable hyperparameters
self._updateValsTo = {"newEpoch": tk.StringVar(value = "25"),
                      "newRegStr": tk.StringVar(value = "0.001"),
                      "initialLr": tk.StringVar(value = "0.0001"),
                      "decay": tk.StringVar(value = "0.00005"),
                      "momentum": tk.StringVar(value = "0.95")}
```

The following changes were also implemented in the program to accommodate these changes.

Old program

```
# Updates the model's hyperparameters and retrains the model with the new hyperparameters
def __updateHyperparameters(self):
    try:
        optimiserVals = [item.get() for item in list(self._updateValsTo.values())]

        # Apply hyperparameters to the model
        self.__model.updateEpoch(optimiserVals[0])
        self.__model.updateRegStr(optimiserVals[1])
        self.__model.configOptimiser(optimiserVals[2], optimiserVals[3], optimiserVals[4])

        print("Retraining model...")
        self.__newModel()

    except ValueError:
        print("Invalid input for epochs or regularisation strength. Please enter valid values.")
```

The program would abort during this line.

New program

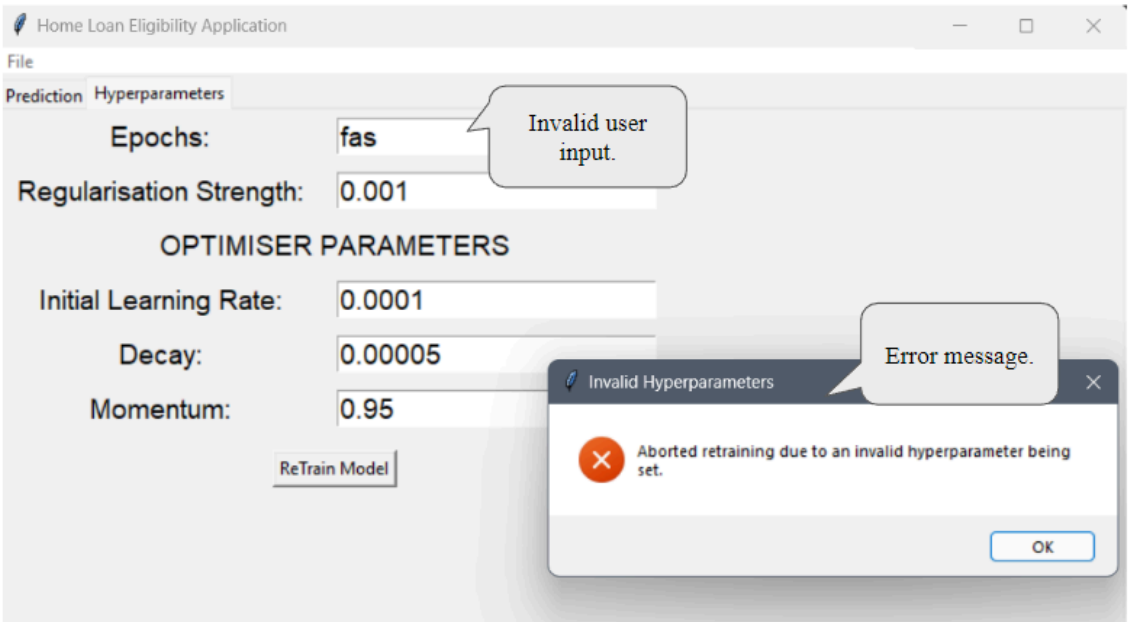
Altered to allow an error to be caught by the try - except statement.

```
# Updates the model's hyperparameters and retrains the model with the new hyperparamet
def __updateHyperparameters(self):
    try:
        optimiserVals = [float(item.get()) if key != "newEpoch" else int(item.get())
                         for key, item in list(self._updateValsTo.items())]

        # Apply hyperparameters to the model
        self.__model.updateEpoch(optimiserVals[0])
        self.__model.updateRegStr(optimiserVals[1])
        self.__model.configOptimiser(optimiserVals[2], optimiserVals[3], optimiserVals[4])

        print("Retraining model...")
        self.__newModel()

    except ValueError:
        messagebox.showerror("Invalid Hyperparameters",
                             "Aborted retraining due to an invalid hyperparameter being set.")
```

Changed error display where previously it would only be displayed in the terminal.

Imported new module.

```
from tkinter import ttk, simpledialog, messagebox
```



Test 10

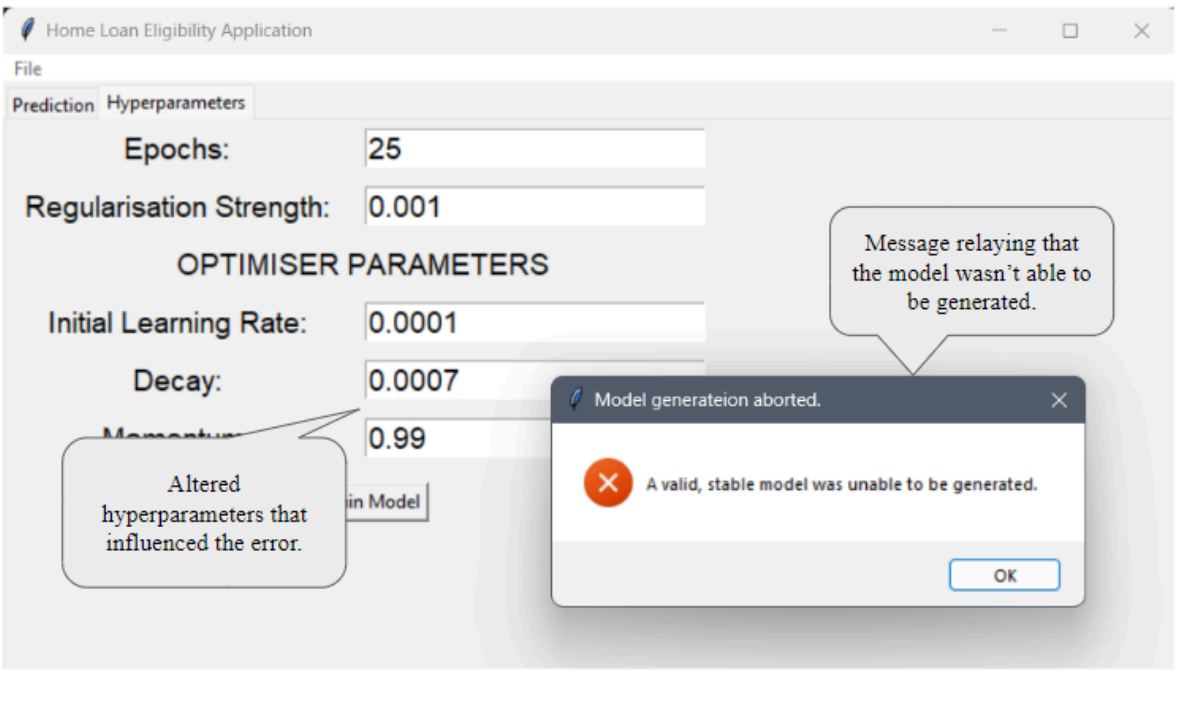
During the previous test, another error was discovered. Occasionally, the program is unable to produce a valid model, it gets stuck in an infinite loop. This shouldn't happen as instead the generation should aborted, and the default model should be loaded. To implement this fix to the program, the newModel method in the GUI class.

```
valid = False
count = 0
while not valid and count != 10:
    # Trains model with new random data
    self.__model.train(TrainX, TrainY)
    self.__model.test(TestX, TestY)
    accuracy = self.__model.Accuracy

    # Due to model limitations the model will be trained again if it is not at an acceptable accuracy.
    # This is to ensure that the model doesn't overfit ('memorise' training data) or converge on one
    # output (eg. always output 1 prediction like 0.643 or 64.3%)
    if accuracy > 0.74:
        status = f"Valid model generated - Accuracy: {accuracy}    |    Unsaved"
        self.__training, valid = False, True

        self._saveStatusVal.set(status)
        self._saveStatusLabel.config(textvariable=self._saveStatusVal)
    else:
        self.__model.resetLayers()
        self.__Preprocessor.newDataset()
        TrainX, TrainY, TestX, TestY = self.__Preprocessor.getData()
        count += 1

if count == 10:
    messagebox.showerror("Model generateion aborted.",
                         "A valid, stable model was unable to be generated.")
    self.__loadDefault()
```

Added limits to number of attempts.

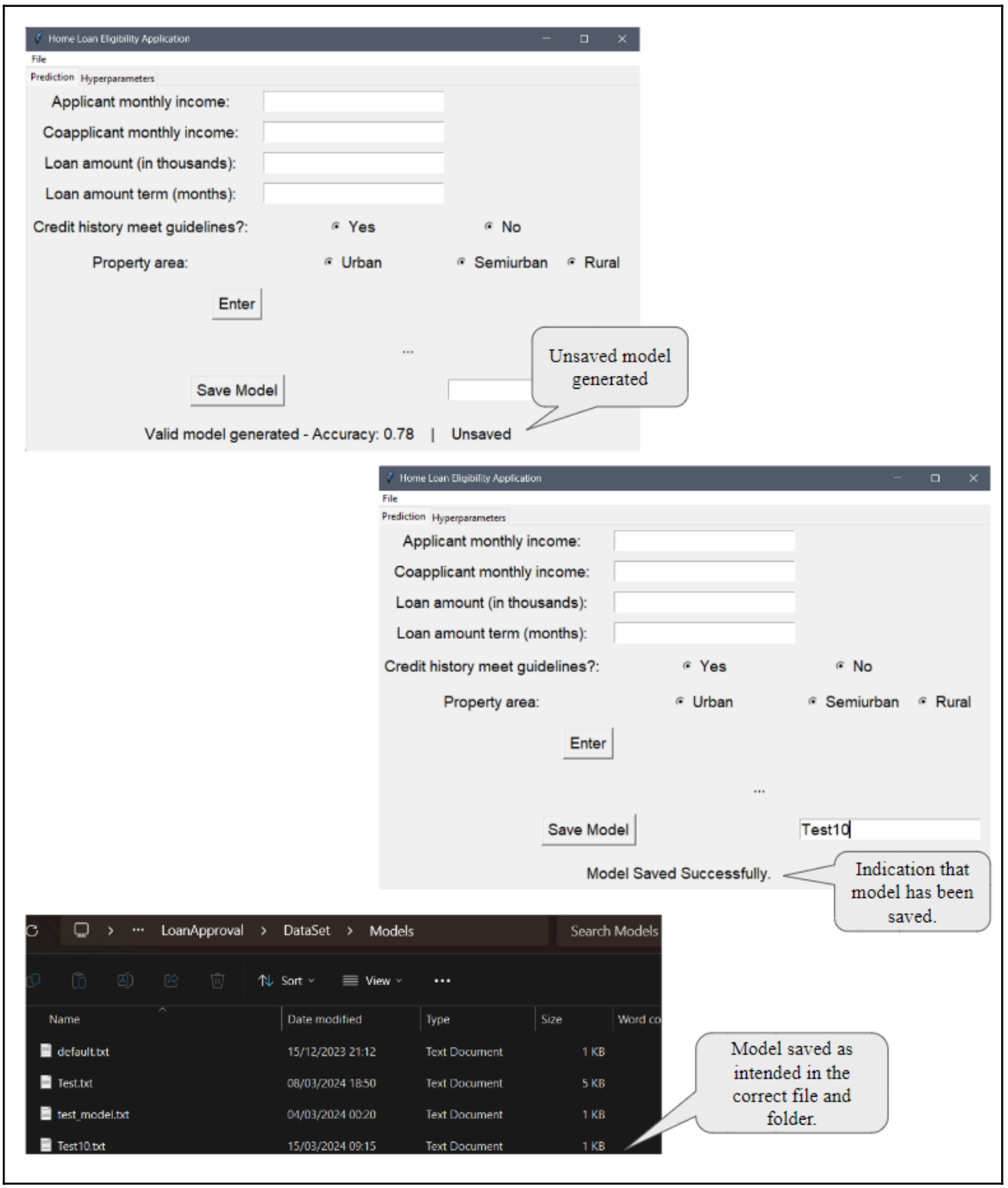
Relays error and loads the default model.

As a result, now when a model is unable to be generated a message is conveyed and the default model is loaded, as shown below.



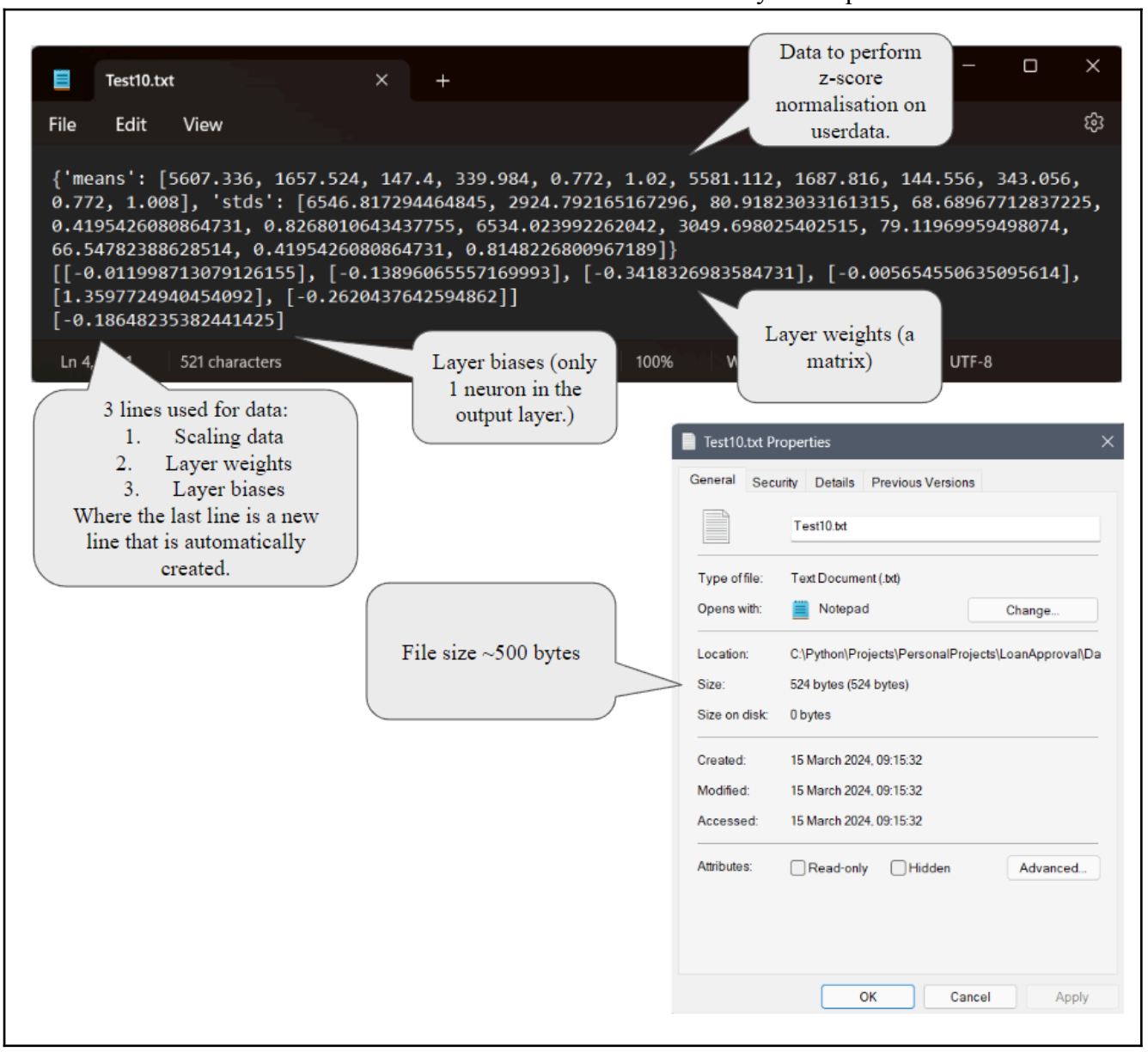
Test 11

The following screenshots show after a new model is generated, when attempted to save with a valid file name, it is saved within the correct folder. The following test will test if the model is saved as intended and in the correct format in the .txt file.



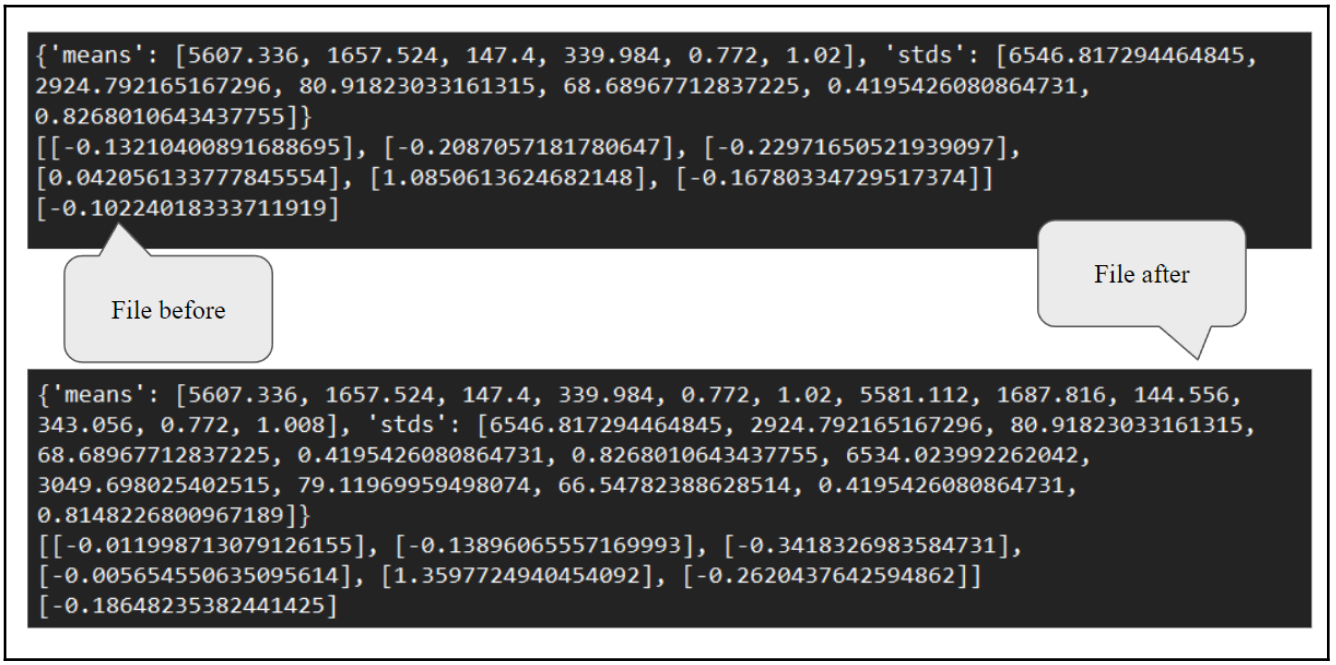
Test 12

The neural network model being saved only consists of 1 output layer. The following screenshot shows inside the test10.txt file that was saved successfully in the previous test.

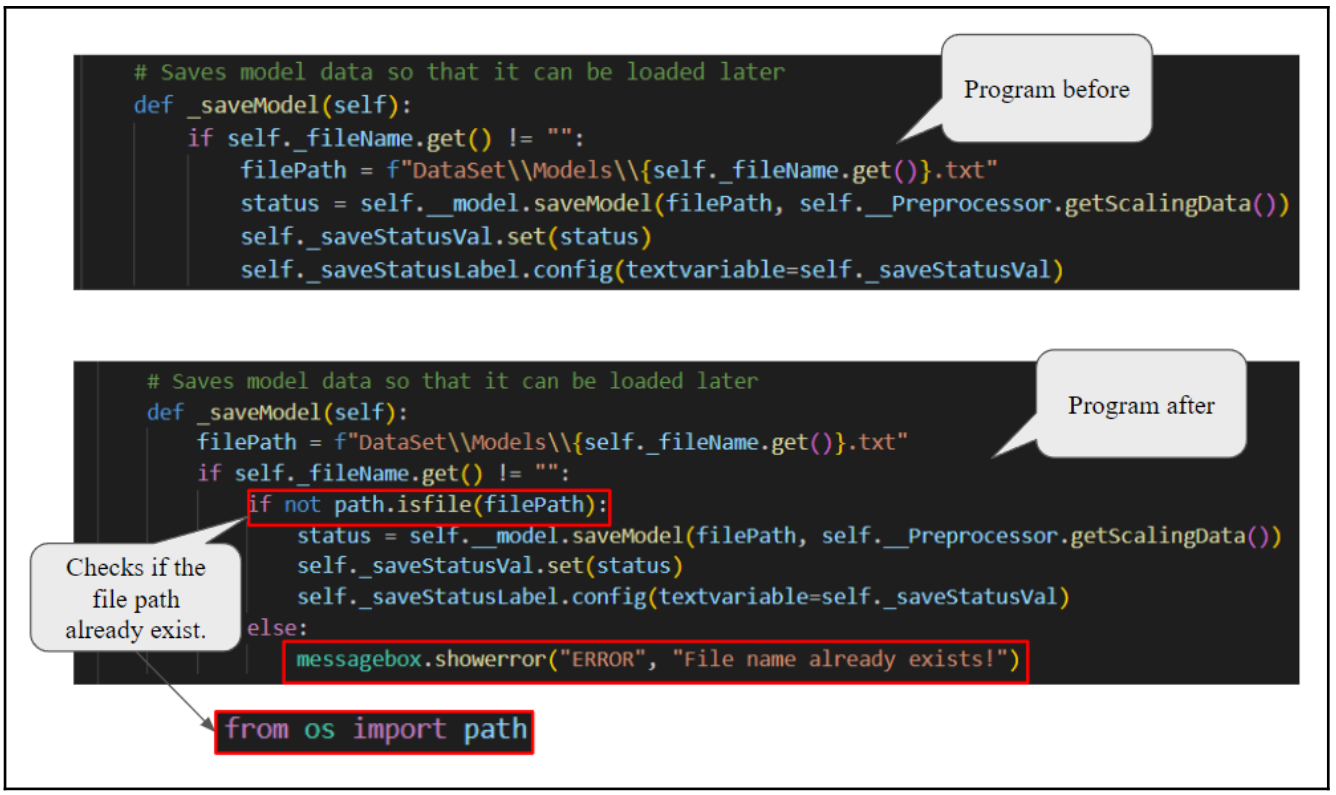


Test 13

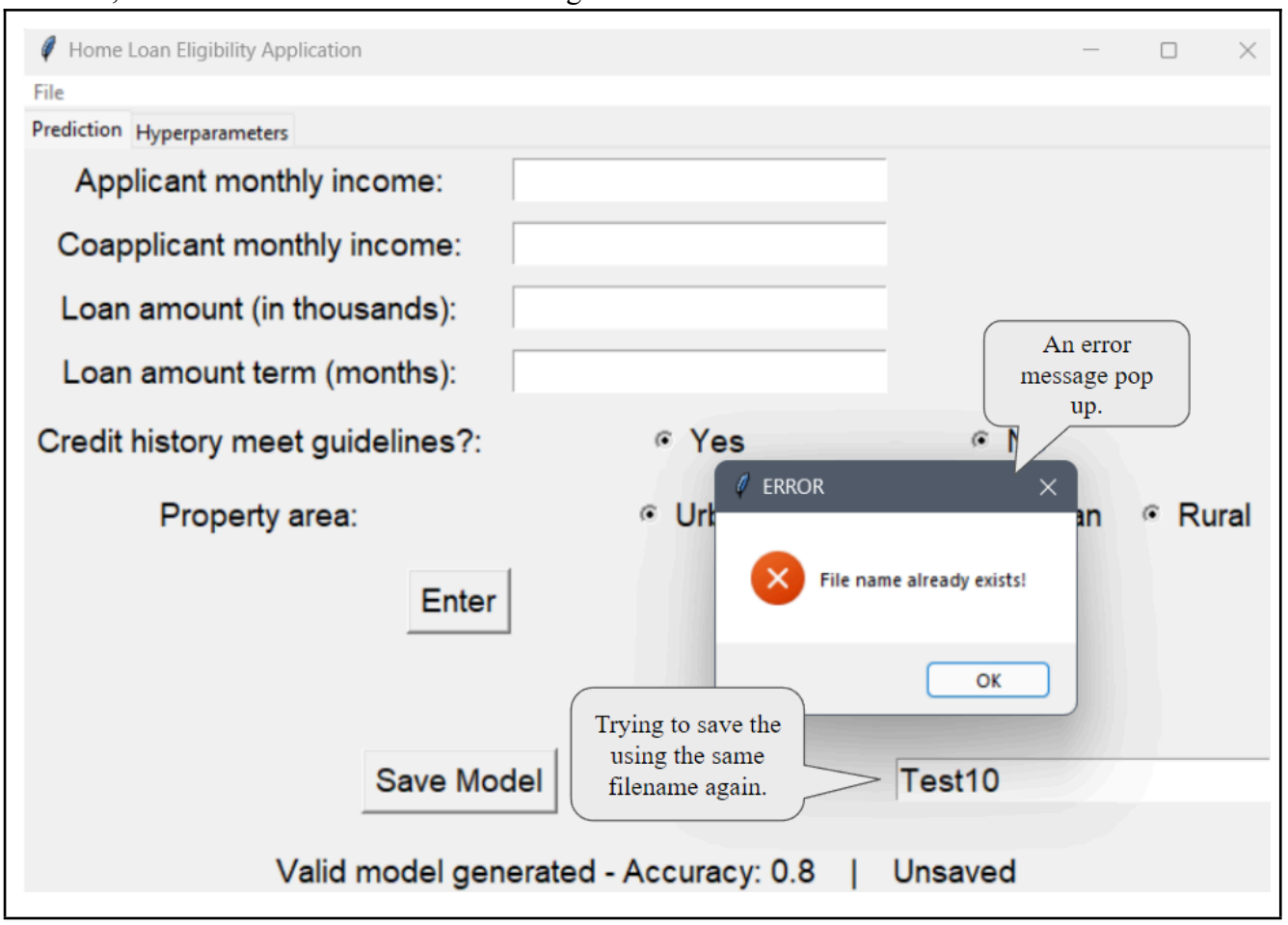
Following from test 10 and 11, I will try and save a file using the same filename ‘Test10’. The screenshots show the file content before and after this.

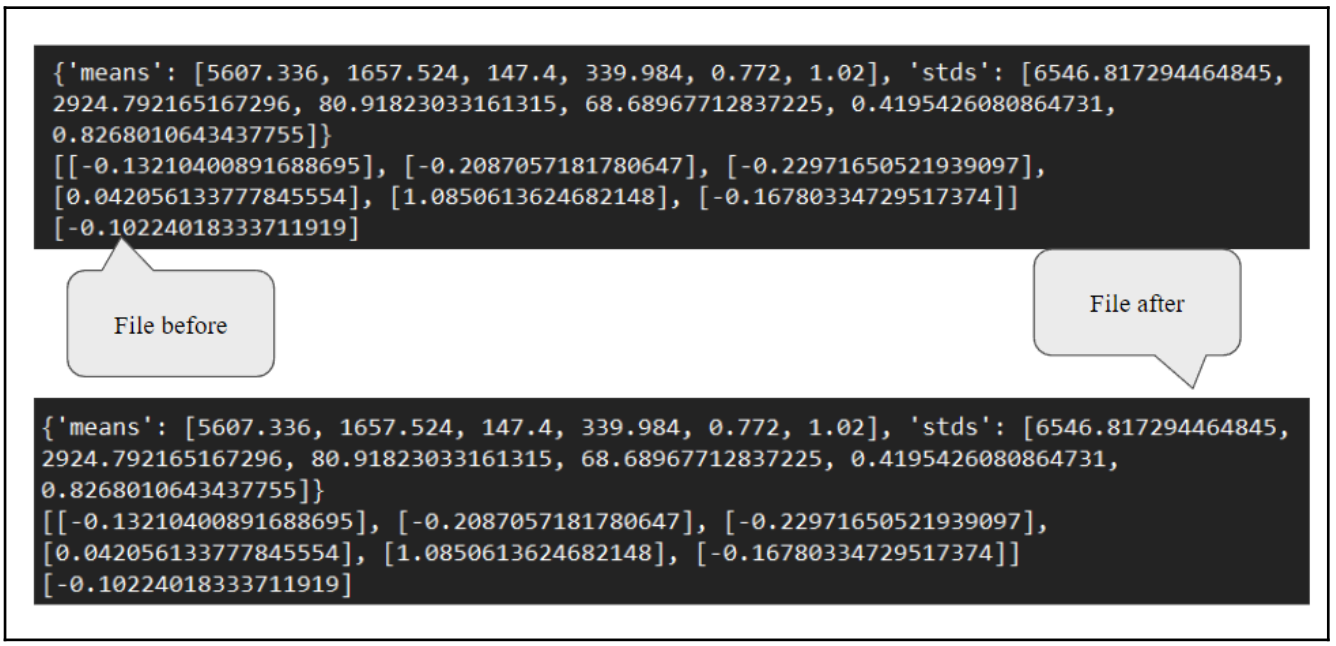


This test has failed as the contents of the file were overwritten, where weights and biases were replaced and new scaling data is added to the old. The following changes were made to the program to fix this issue.



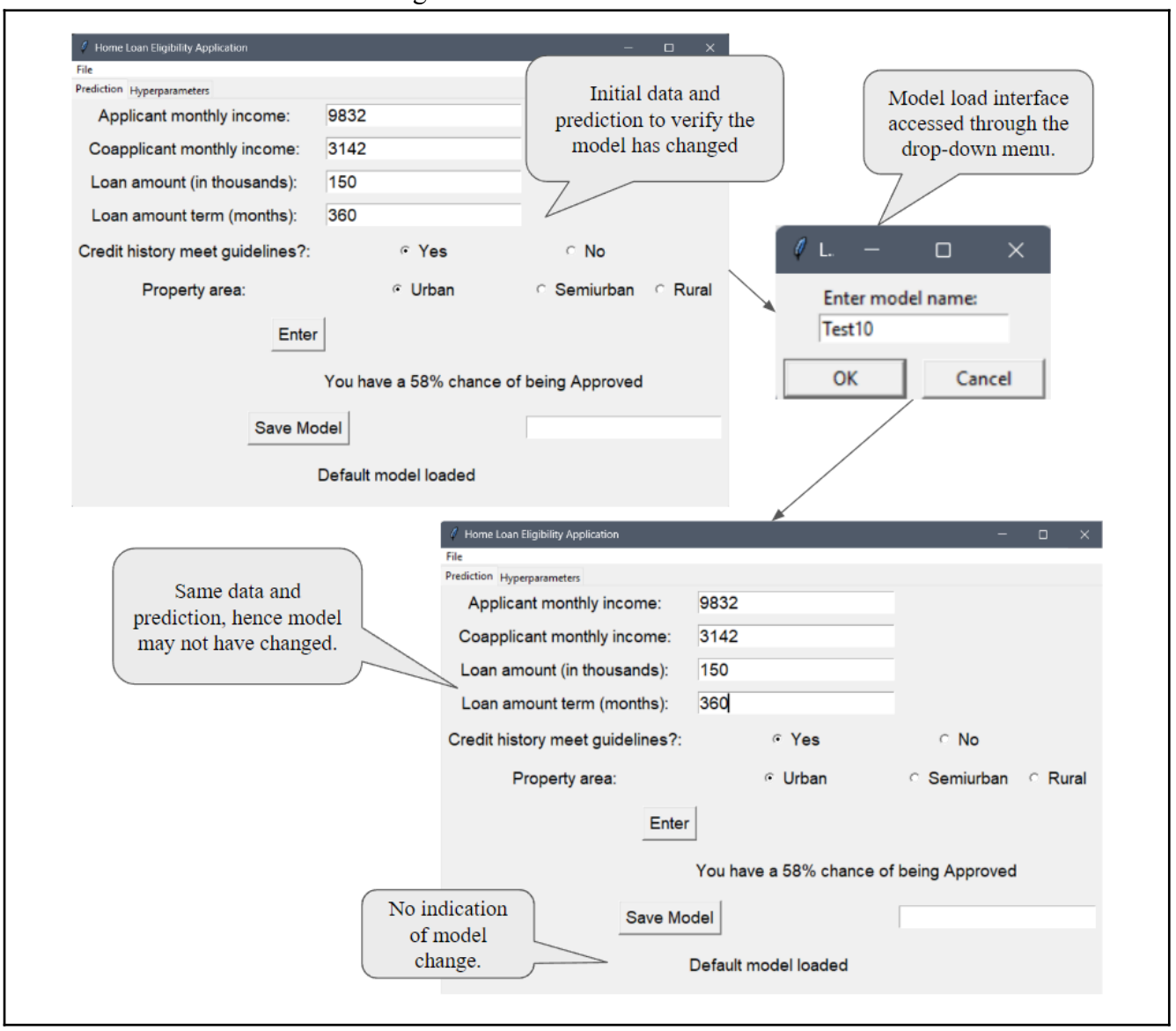
After redoing the same test, a new error message is shown to relay the error as previously intended, and the file contents remain unchanged.





Test 14

For the following test, I entered a set of data and made a prediction. This will allow me to confirm a change in the model, when the accuracy changes. Furthermore, a change in the status will also be observed if the model loading was successful.



The test failed as both expected changes never occurred.

Old program

```
# Loads saved models
def _loadModel(self):
    modelName = simpledialog.askstring("Load Another Model", "Enter model name:")
    filePath = f"DataSet\\Models\\{modelName}.txt"
    scalingData = self.__model.loadModel(filePath)

    if scalingData == None:
        print("File not found. Loading default...")
        self.__loadDefault()
    else:
        self.__Preprocessor.setScalingData(scalingData)
```

Only indication of model rejection is a message is the terminal that the user will not see.

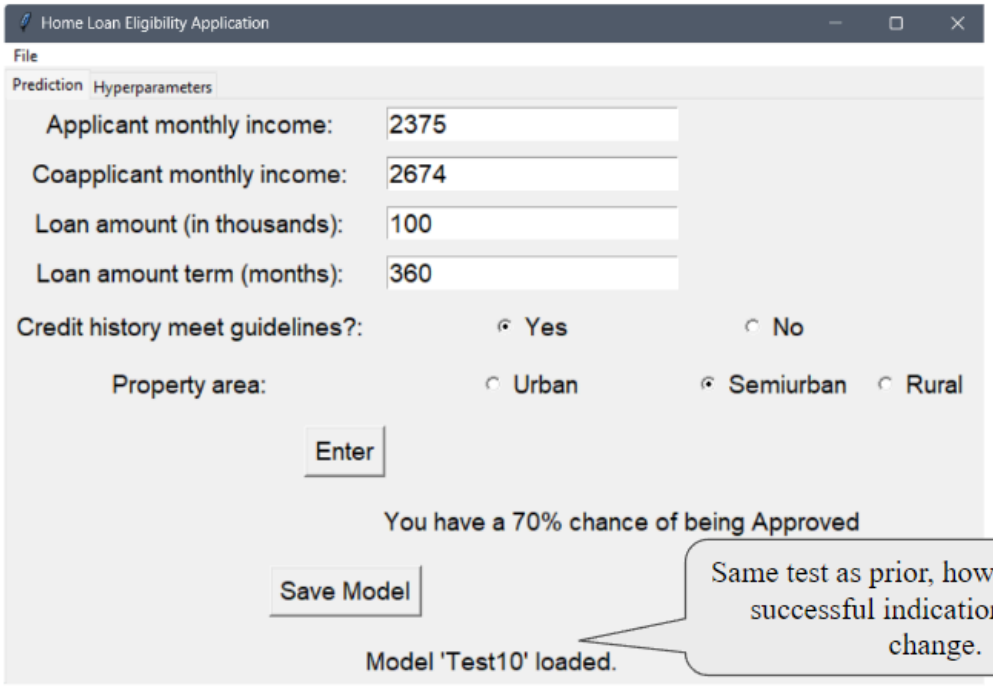
No indication of model change.

New program

```
# Loads saved models
def _loadModel(self):
    modelName = simpledialog.askstring("Load Another Model", "Enter model name:")
    filePath = f"DataSet\\Models\\{modelName}.txt"
    scalingData = self.__model.loadModel(filePath)

    if scalingData == None:
        messagebox.showerror("Error", f"Following file '{modelName}' doesn't exist.")
    else:
        self.__Preprocessor.setScalingData(scalingData)
        self._saveStatusVal.set(f"Model '{modelName}' loaded.")
        self._saveStatusLabel.config(textvariable=self._saveStatusVal)
```

Allows for indication when model have changed.

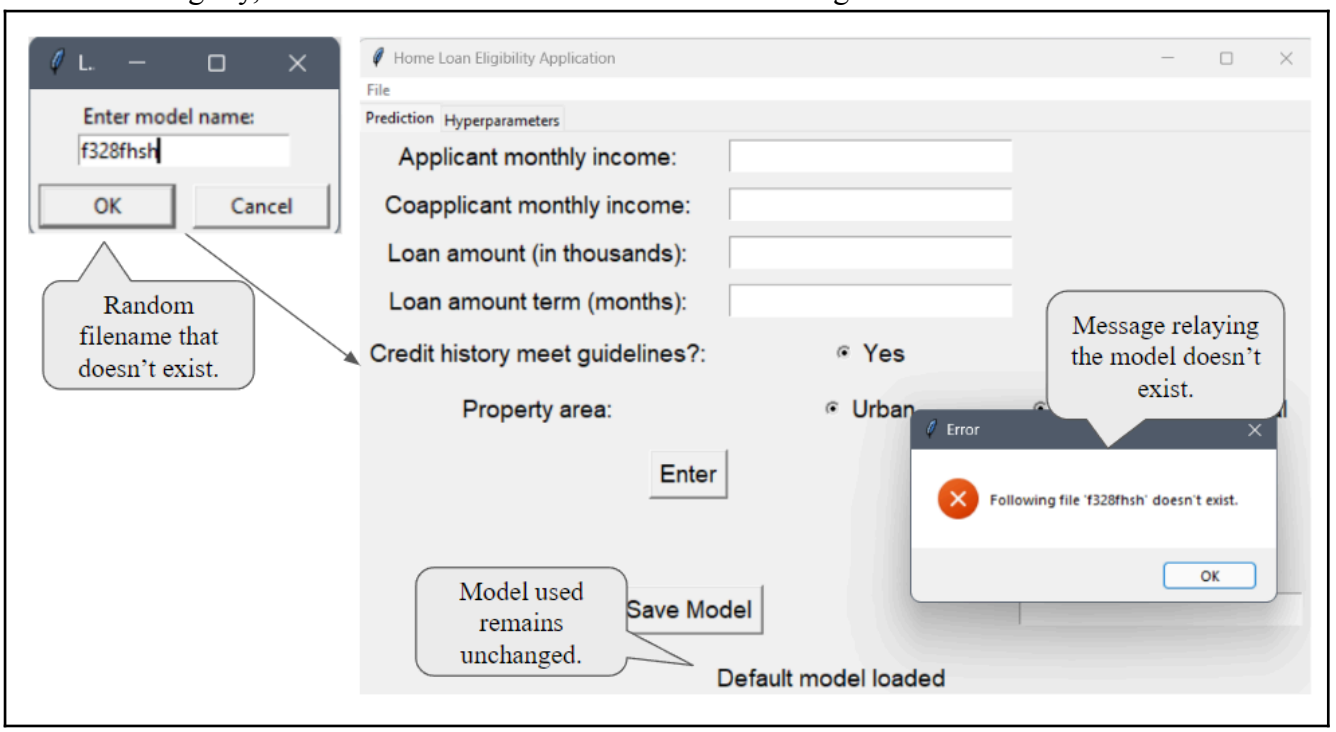
Below is evidence to support that the test has passed.



Same test as prior, however this time successful indication of model change.

Test 15

The screenshot below shows an incorrect filename being rejected as intended. This program was altered slightly, when the error was identified while resolving the test 14 failure.



Unit-testing

In this section, we will employ the `unittest` module in Python to conduct testing across various classes. This is for independent classes such as `DataMethods` and `Activations`, which have limited interactions with the userdata, to undergo individual testing to ensure their functionalities operate as expected. On the other hand, modules such as ‘Optimiser’ and ‘Logistic Regression Model’ will be tested by comparing their direct output similar to the previously conducted tests using a mix of ‘unittests’ and graphs.

Furthermore, tests conducted with `unittest` will primarily focus on asserting that the modules produce the anticipated outputs when supplied with data that adheres to the program's expected input. This approach is designed to streamline the testing process during development, considering that most methods within these modules represent fixed algorithms with minimal user interaction. Additionally, since the program exclusively accepts valid user inputs, we can confidently assume that invalid data scenarios will not occur, further simplifying the testing approach. Exception handling and error prevention mechanisms, including try-except blocks, will also be utilised to handle any unforeseen abnormal scenarios during testing.

The table below outlines the tests to be conducted, detailing the module, aspect, and purpose of each test. Input data and expected outputs are provided within the associated test cases in subsequent sections. These tests serve to verify that the algorithms operate as intended and have been carefully selected to assess the overall system performance comprehensively.

Module Tested	Test	Description
1). DataMethods	A). CsvToArray	Validates the parsing of data stored in a .csv file into a 2D array format, ensuring exclusion of any empty strings or invalid entries.
	B). Transpose	Verifies the proper transposition of the data matrix, confirming correct swapping of rows and columns
	C). DotProduct	Ensures accurate computation of the dot product of two valid arrays, verifying proper alignment and dimensionality.
	D). Multiply (scale)	Checks the correct scaling of a 1D or 2D array with another array of equal or lower dimensions, specifically verifying element-wise multiplication.
2). Preprocessor	A). New Dataset	Tests the generation of a new 2D array with all data stored as floating-point numbers, verifying suitability for training or testing purposes.
	B). Data is standardised	Validates the standardisation process applied to the generated dataset, ensuring that all features have a mean of 0 and a standard deviation of 1.
	C). Data split	Confirms the accurate division of the dataset into training and test sets, ensuring a distribution of sample ratios for both.

	D). Encode	Verifies the conversion of user inputs into a valid format accepted by the algorithm, ensuring proper transformations or mappings are applied.
3). Activation	A). ReLU forward propagation	Verifies that the rectified linear unit (ReLU) function correctly maps all input values less than 0 to 0 and leaves positive values unchanged.
	B). ReLU backpropagation	Ensures that during backpropagation, gradients are computed accurately for the ReLU function. Specifically, it checks whether unchanged inputs during forward propagation result in a gradient of 1, while inputs less than 0 have a gradient of 0.
	C). Sigmoid forward propagation	Confirms that the sigmoid activation function scales all input values to the range [0, 1] as expected.
	D). ReLU backpropagation	Validates the computation of gradients for the sigmoid function, ensuring consistency with the outputs from the forward propagation stage.
4). Loss function	A). Forward propagation	Ensures the accurate evaluation of the model's predictions by computing the loss using a cross-entropy metric.
	B). Backpropagation	Validates the computation of derivatives of the loss function with respect to each prediction, facilitating gradient-based optimization.
	C). L2 regularisation loss	Confirms the correct calculation of regularisation loss, ensuring proper penalization of large model weights to prevent overfitting.
	D). Loss reduction	Tests the behaviour of loss values during training epochs, specifically checking that the loss generally decreases and plateaus before 0.
5). Optimiser	A). Learning rate decrease	Confirms that learning rate has a linear decrease and doesn’t go below the minimum learning rate.
	B). Model accuracy improvement	Validates that the layers are optimised as intended, indicated by a general increase in accuracy.
6). Logistic Regression Model	A). Invalid model rejection	Confirms that if a invalid model is generated it is rejected and a new model is generated.
	B). Valid model training	Show that as training loops are complete, loss decreases and plateaus, accuracy increases and plateaus.
	C). Prediction	Validates the model’s ability to predict the outcome of unseen data.

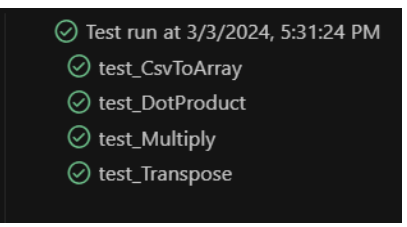
DataMethods

Tests A to D

```python
from Scripts.DataHandle import DataMethod
import unittest

class TestDataMethods(unittest.TestCase):
    def test_CsvToArray(self):
        path = "DataSet\HomeLoanTrain.csv"
        result = DataMethod.CsvToArray(path, maxEntries=10)
        self.assertIsInstance(result, list)
        self.assertNotIn('', result)

    def test_Transpose(self):
        array = [[2, 4, 3], [5, 6, 7]]
        expected = [[2, 5], [4, 6], [3, 7]]
        result = DataMethod.Transpose(array)
        self.assertListEqual(expected, result)

    def test_DotProduct(self):
        arr1 = [[1, 2], [3, 4], [5, 6]]
        arr2 = [[7, 8, 9], [10, 11, 12]]
        expected_result = [[27, 30, 33], [61, 68, 75], [95, 106, 117]]
        result = DataMethod.DotProduct(arr1, arr2)
        self.assertListEqual(result, expected_result)

    def test_Multiply(self):
        tests = {
            "Test1": {"scalar": 2,
                      "array": [1, 2, 3, 4, 5],
                      "expected": [2, 4, 6, 8, 10]},
            "Test2": {"scalar": 2,
                      "array": [[1, 2, 3], [4, 5, 6], [7, 8, 9]],
                      "expected": [[2, 4, 6], [8, 10, 12], [14, 16, 18]]},
            "Test3": {"scalar": [1, 2, 3],
                      "array": [4, 5, 6],
                      "expected": [4, 10, 18]}}

        for test in tests.values():
            result = DataMethod.Multiply(test["scalar"], test["array"])
            self.assertListEqual(result, test["expected"])
```

Results

```
test_CsvToArray (test_DataUtils.TestDataMethods.test_CsvToArray) ... ok
test_DotProduct (test_DataUtils.TestDataMethods.test_DotProduct) ... ok
test_Multiply (test_DataUtils.TestDataMethods.test_Multiply) ... ok
test_Transpose (test_DataUtils.TestDataMethods.test_Transpose) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.003s
```



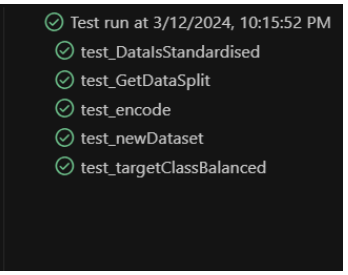
Preprocessor

Test A to D

```
from Scripts.DataHandle import Preprocess

import unittest

class TestPreprocessor(unittest.TestCase):
    def setUp(self):
        self.processor = Preprocess()
        self.processor.newDataset()
        self.TrainX = self.processor._TrainX

    def test_newDataset(self):
        # Check if the data is a 2D array
        self.assertTrue(isinstance(self.TrainX, list))
        for row in self.TrainX:
            self.assertTrue(isinstance(row, list))
            for val in row:         # Check if the data consists of floats
                self.assertTrue(isinstance(val, float))

    def test_targetClassBalanced(self):
        count = {"1.0": 0, "0.0": 0}  # Train Y values converted to numerical form.
        TrainY = self.processor._TrainY
        for val in TrainY:  # Counts number of each target class.
            count[str(val)] += 1
        self.assertEqual(count["1.0"], count["0.0"])

    def test_DataIsStandardised(self):
        # Check if the mean of the dataset is approximately 0
        sumOfFeatures = sum([sum(feature) for feature in self.TrainX])
        totalElements = (len(self.TrainX) * len(self.TrainX[0]))
        mean = sumOfFeatures / totalElements
        self.assertAlmostEqual(mean, 0.0, delta=0.1)

        # Check if the standard deviation of the dataset is approximately 1
        squared_sum = sum([sum([x**2 for x in feature]) for feature in self.TrainX])
        std_dev = ((squared_sum / (len(self.TrainX) * len(self.TrainX[0]))) - mean**2) ** 0.5
        self.assertAlmostEqual(std_dev, 1.0, delta=0.1)

    def test_GetDataSplit(self):
        train_X, train_Y, test_X, test_Y = self.processor.getData()
        # Check if split data has correct dimensions
        self.assertEqual(len(train_X)/ len(train_Y), len(test_X)/ len(test_Y))

    def test_encode(self):
        # Encode sample user data
        encoded_data = self.processor.encode([9732, 6543, 76, 180, "Yes", "Semiurban"])
        # Check if encoded data has correct dimensions
        self.assertEqual(len(encoded_data), len(self.TrainX[0]))
        # Check if encoded data consists of floats
        for val in encoded_data:
            self.assertTrue(isinstance(val, float))
```

Result

```
test_DataIsStandardised (test_PreProcessor.TestPreprocessor.test_DataIsStandardised) ... ok
test_GetDataSplit (test_PreProcessor.TestPreprocessor.test_GetDataSplit) ... ok
test_encode (test_PreProcessor.TestPreprocessor.test_encode) ... ok
test_newDataset (test_PreProcessor.TestPreprocessor.test_newDataset) ... ok
test_targetClassBalanced (test_PreProcessor.TestPreprocessor.test_targetClassBalanced) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.018s

OK
Finished running tests!
```



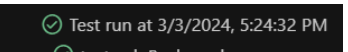
Activations

Tests A and B

```python
from Scripts.NeuralNetwork.Layer import ReLU

import unittest

class TestRectifiedLinearUnit(unittest.TestCase):
    def setUp(self):
        self.relu = ReLU()

    def test_reluForward(self):
        inputs = [[-1, 0, 1], [2, -2, 3]]
        self.relu.forward(inputs)
        expected = [[0, 0, 1], [2, 0, 3]]
        self.assertEqual(self.relu.outputs, expected)

    def test_reluBackward(self):
        inputs = [[-1, 0, 1], [2, -2, 3]]
        self.relu.forward(inputs)
        dvalues = None
        self.relu.backward(dvalues)
        expected = [[0, 0, 1], [1, 0, 1]]
        self.assertEqual(self.relu.dinputs, expected)
```

Result

```
test_reluBackward (test_Activations.TestRectifiedLinearUnit.test_reluBackward) ... ok
test_reluForward (test_Activations.TestRectifiedLinearUnit.test_reluForward) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
Finished running tests!
```



Tests C and D

```python
from Scripts.NeuralNetwork.Layer import ReLU

import unittest

class TestSigmoid(unittest.TestCase):
    def setUp(self):
        self.sigmoid = Sigmoid()

    def test_sigmoidForward(self):
        inputs = [[-1], [0], [1]]
        self.sigmoid.forward(inputs)
        expected = [0.2689414213699951, 0.5, 0.7310585786300049]
        self.assertAlmostEqual(self.sigmoid.outputs, expected)

    def test_sigmoidBackward(self):
        inputs = [[-1], [0], [1]]
        self.sigmoid.forward(inputs)
        dvalues = [-3.7182818284590455, -2.0, -1.3678794411714423]
        self.sigmoid.backward(dvalues)
        expected = [[-0.7310585786300049], [-0.5], [-0.2689414213699951]]
        self.assertAlmostEqual(self.sigmoid.dinputs, expected)
```

Result

```
test_sigmoidBackward (test_Activations.TestSigmoid.test_sigmoidBackward) ... ok
test_sigmoidForward (test_Activations.TestSigmoid.test_sigmoidForward) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
Finished running tests!
```

Test run at 3/3/2024, 5:27:24 PM
test_sigmoidBackward
test_sigmoidForward

Loss

Test A to C

```python
from Scripts.NeuralNetwork.LossFunctions import BinaryCrossEntropy

import unittest

class TestBinaryCrossEntropy(unittest.TestCase):
    def setUp(self):
        self.LossFunction = BinaryCrossEntropy(regStr=0.001)

    def test_forward(self):
        predictions = [0.2, 0.7, 0.9]
        TrueValues = [0, 1, 1]
        self.LossFunction.forward(predictions, TrueValues)
        ExpectedLoss = 0.22839
        self.assertAlmostEqual(self.LossFunction.getLoss(), ExpectedLoss, places=5)

    def test_backward(self):
        predictions = [0.2, 0.7, 0.9]
        TrueValues = [0, 1, 1]
        self.LossFunction.backward(predictions, TrueValues)
        ExpectedGradients = [1.25, -1.42857142857, -1.11111111111]
        for result, expected in zip(self.LossFunction.dinputs, ExpectedGradients):
            self.assertAlmostEqual(result, expected)

    def test_calcRegularisationLoss(self):
        layer_weights = [[0.1, 0.2], [0.3, 0.4]]
        self.LossFunction.calcRegularisationLoss(layer_weights)
        ExpectedLoss = 0.00015
        self.assertAlmostEqual(self.LossFunction.getLoss(), ExpectedLoss, places=5)
```

Result

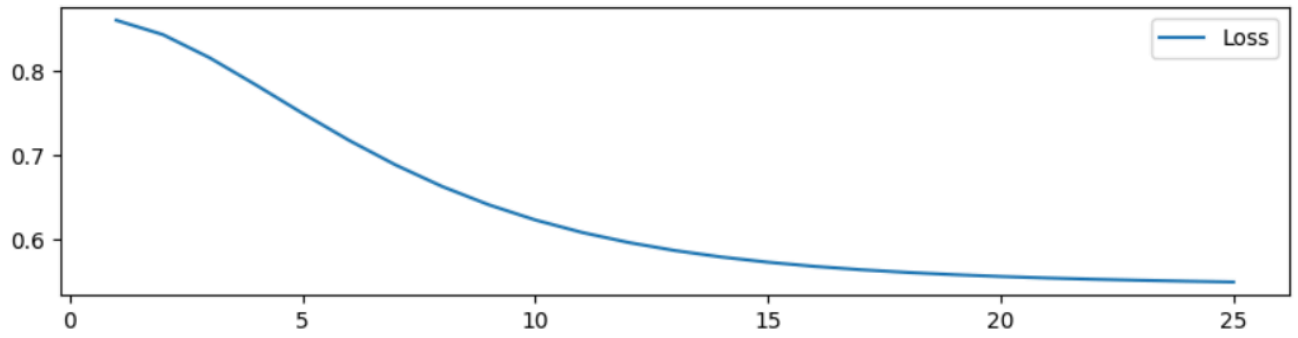
```
test_backward (test_Loss.TestBinaryCrossEntropy.test_backward) ... ok
test_calcRegularisationLoss (test_Loss.TestBinaryCrossEntropy.test_calcRegularisationLoss) ... ok
test_forward (test_Loss.TestBinaryCrossEntropy.test_forward) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
Finished running tests!
```

Test run at 3/3/2024, 6:35:15 PM
test_backward
test_calcRegularisationLoss
test_forward

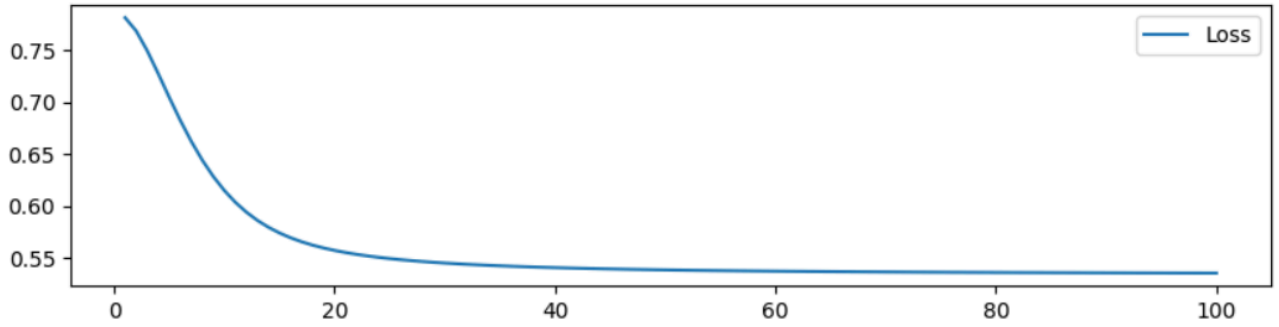
Test D

As mentioned multiple times prior, a good model can be indicated by its decrease in loss, and a smooth curve which plateaus towards the end of the training session. Displayed below are 2 graphs, each displaying the change is loss for 2 separate training sessions, where the first graph uses an epoch (number of training loops) of 25 which is used by default and the next utilises an epoch of 100.

Graph 1



Graph 2



Optimiser

Test A

```
from Scripts.NeuralNetwork.Optimisers import OptimiserSGD

import unittest

class TestOptimiser(unittest.TestCase):
    def test_LearningRateDecrease(self):
        initalLr = 0.1
        optimsier = OptimiserSGD(InitialLearningRate=initalLr, decay=0.01)

        for i in range(20):
            optimsier.adjustLearningRate(i)
            if optimsier.activeLearningRate == 0.0001:
                self.assertLessEqual(optimsier.activeLearningRate, 0.0001)
                break
            self.assertLessEqual(optimsier.activeLearningRate, initalLr)
```

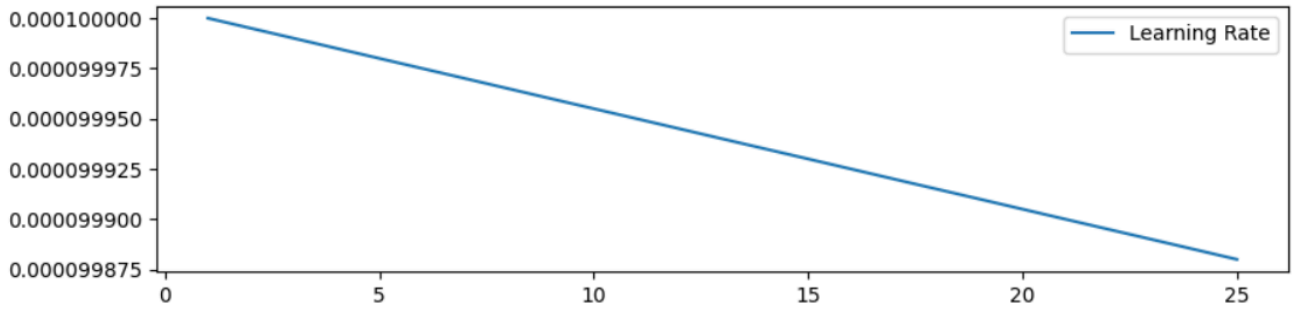
Result

```
test_LearningRateDecrease (test_Optimiser.TestOptimiser.test_LearningRateDecrease) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Finished running tests!
```

Test run at 3/3/2024, 11:50:52 PM
test_LearningRateDecrease

Graph

The image below visualises the linear decrease in learning rate as directly proportional to the iteration of the model (at the end of each training pass).



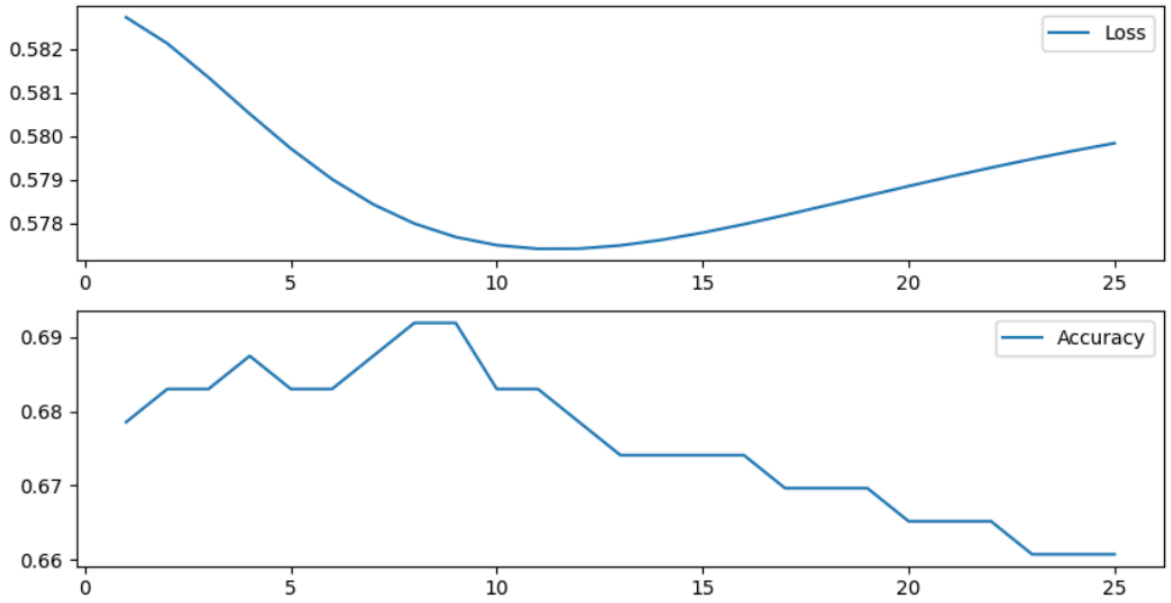
Logistic Regression Model

Test A and B

The following graph shows the difference an invalid and valid model training session.

Invalid Model Graph

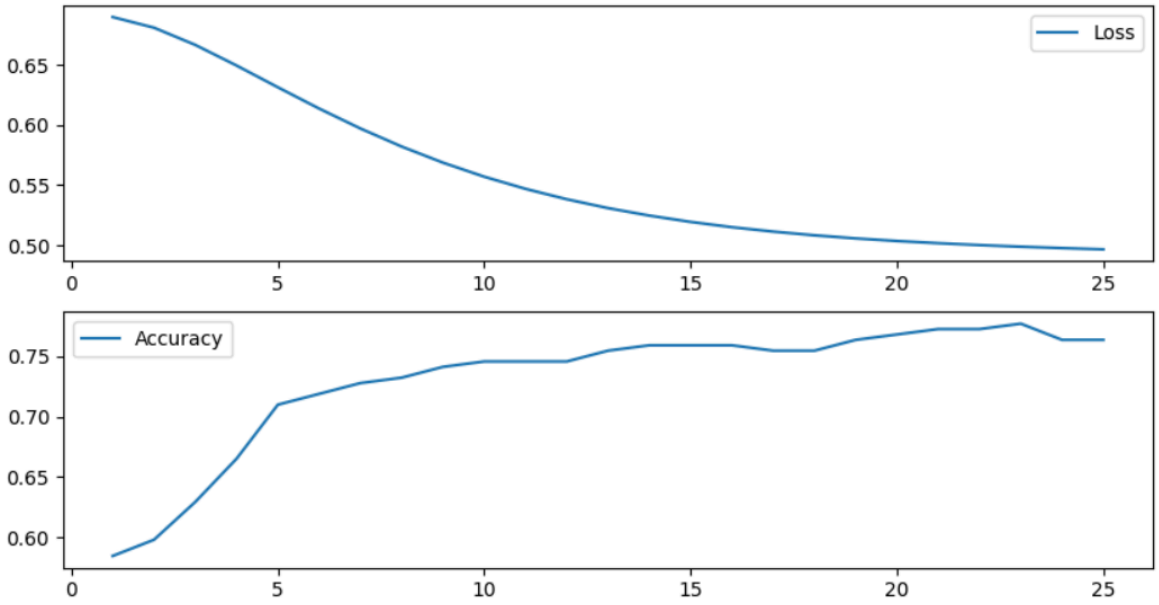
The graph below shows an invalid training cycle, which would be rejected.



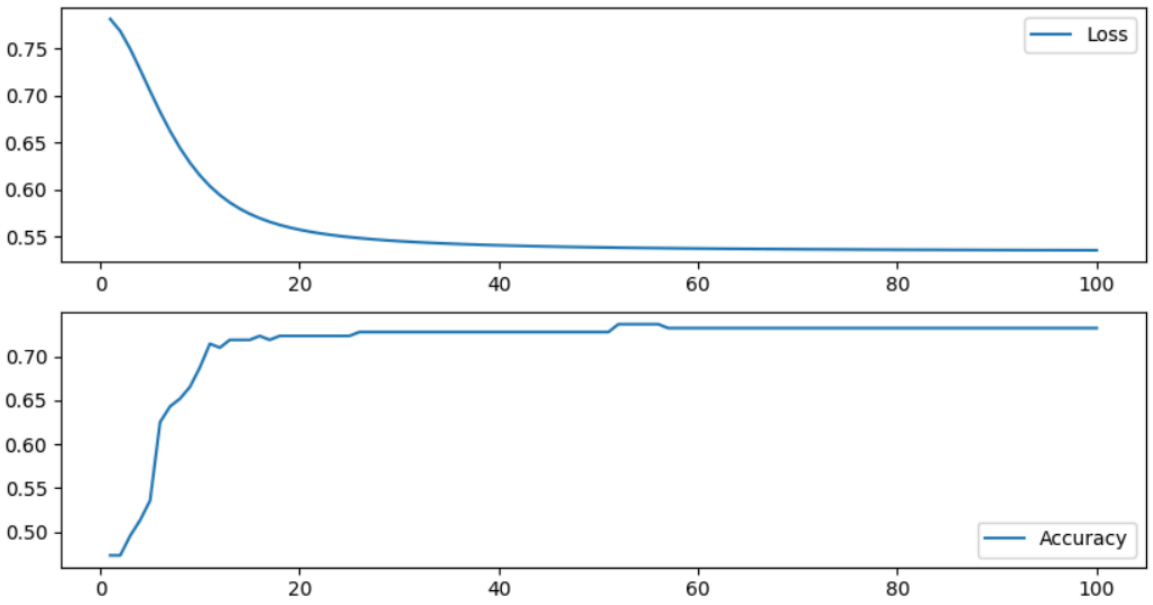
Valid Model Graphs

The following graphs show 2 training instances of valid model training sessions. The first where an epoch of 25 is used and the second where an epoch of 100 is used.

Training session 1:



Training session 2:



Utilising the second graph, the loss starts to plateau around the 25th cycle, hence why the model's default epoch is set to 25.

Test C

```
from Scripts.NeuralNetwork.Models import LogisticRegression
from Scripts.DataHandle import Preprocess

import unittest

class TestLogisticRegression(unittest.TestCase):
    def setUp(self):
        self.model = LogisticRegression()
        self.model.addLayer(6, 1)
        self.Preprocessor = Preprocess()
        self.Preprocessor.newDataset()

    def test_Prediction(self):
        targetAccuracy = 0.70
        TrainX, TrainY, TestX, TestY = self.Preprocessor.getData()

        while True: # Ensures model is valid
            self.model.train(TrainX, TrainY)
            self.model.test(TestX, TestY)

            if self.model.Accuracy > targetAccuracy:
                # Called again to show predicted values
                self.model.test(TestX, TestY)
                self.assertGreaterEqual(self.model.Accuracy, targetAccuracy)
                break
            else:
                self.model.resetLayers()
                self.Preprocessor.newDataset()
                TrainX, TrainY, TestX, TestY = self.Preprocessor.getData()
```

Results

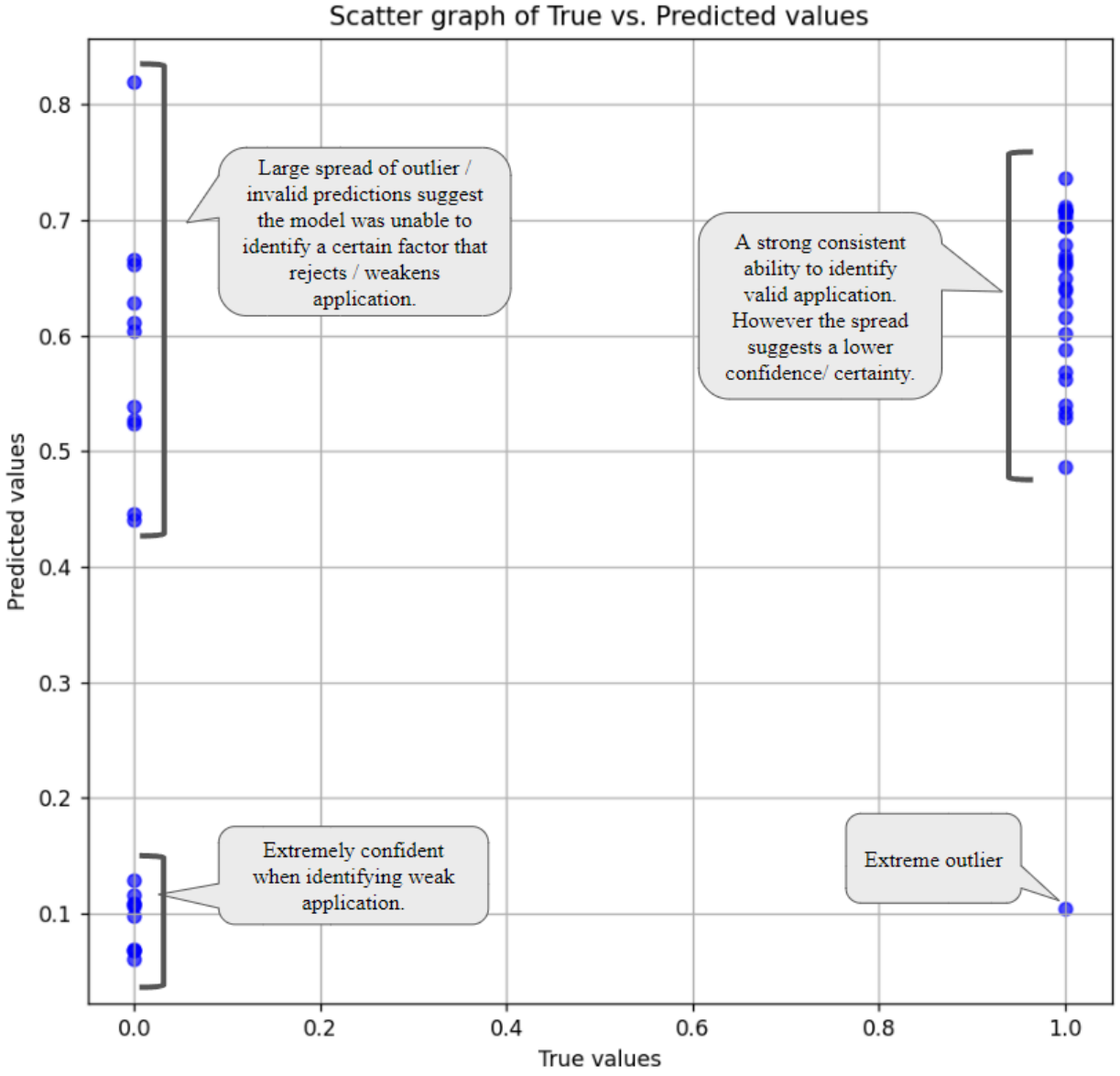
Shown below are examples of different prediction outcomes comparing the true value and the predicted value. The output for this program (between [0, 1]) indicates the likelihood of being approved for a loan, where 0 is being rejected and 1 is being approved.

```
test_Prediction (test_LogistocRegression.TestLogisticRegression.test_Prediction) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.131s

OK
Finished running tests!
```

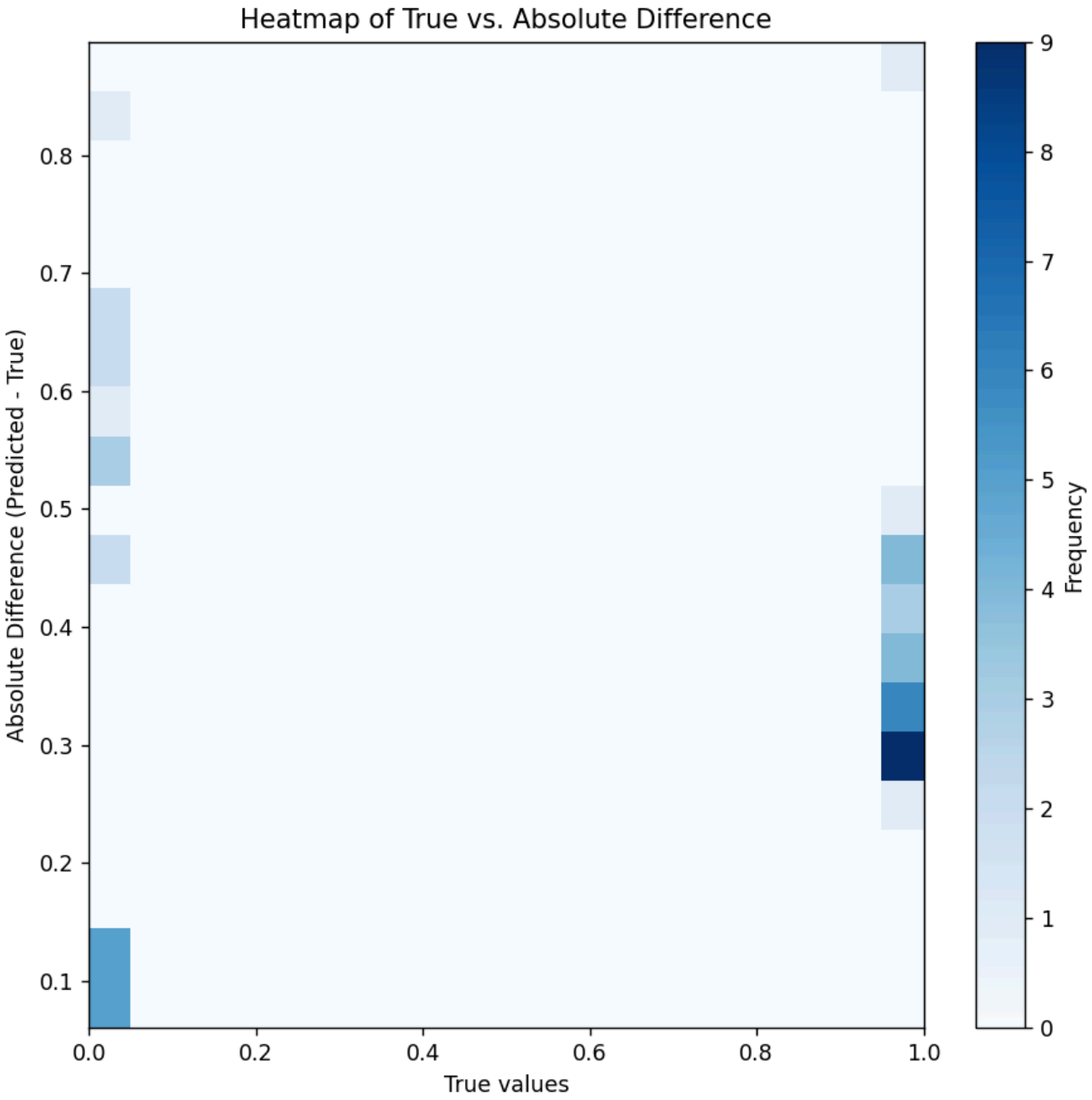
Test run at 3/21/2024, 1:05:46 PM
test_Prediction

Graphs

The following data produced in this test is also shown below as a scatter graph. This graph shows us that overall strong ability to identify when an application is capable of being approved. On the other hand, while it can strongly identify when an application is to be rejected, it is much weaker in its ability to be consistent, also occasionally producing a severely wrong prediction, suggesting that there is a factor that rejects some applications that the model hasn't identified from the training data.



The following heatmap plots the absolute difference between the true and predicted values against the true values. In this graph the closer the values are to the 0 the more confident the prediction was.



Evaluation

Completeness Of Objectives

The project effectively fulfils all objectives set during the analysis phase. Notably, it demonstrates efficiency in processing user data, meeting time-related goals. Moreover, it excels in data handling, resolving issues comprehensively and ensuring data accuracy. The meticulous data selection and processing methodologies employed ensure the model's relevance and reliability, aligning closely with the specified objectives.

The following table provides a breakdown of the objectives and evaluations about their completeness.

Objectives	Criteria	Evaluation
Data Selection and Processing		
1). Obtaining training data.	a). Data is anonymous. b). Data is relevant to the project and contains the outlined key features. c). Data is stored in a valid format (eg .txt or .csv).	The criteria for this objective have been fully satisfied, as the data I have obtained and utilised meets all the set criteria. Utilising reputable data sources such as Kaggle, I was able to source relevant and anonymoused datasets suitable for home loan application analysis. Furthermore, carefully chosen attributes including income, credit history, loan amount, and duration, are selected based on insights provided by Mr. Weiren, ensuring its alignment with project objectives.
2). Data cleaning.	a). Data is correctly cleaned and processed so that it can be used by the model.	This objective has been successfully achieved, as the data is cleaned and prepared to be utilised by the neural network, within the Preprocessor class to engineer features of data that can be effectively utilised for training. This is done in around 0.0036s (0.018s / 5) as identified in the Preprocessor unittest, which runs the ‘setUp’ method before each test method. This effectively addresses common data issues such as null data, feature scaling, and data imbalances, enhancing the quality and reliability of the training data This is shown in Test 1 of the previous section in addition to the preprocessor and data methods unittests.

3). Effectiveness and efficiency.	a). Data is handled as a matrix and entry orders are able to be randomised each epoch.	This objective has been successful, as all the data is handled as matrices in the program. For instance, raw data from the .csv files are extracted and processed as matrices. Furthermore, the DataMethods class provides efficient processing techniques commonly used for matrix manipulation, like transpose and dot product, to assist with this.
Neural Network		
1). Design of model.	a). Program is completely modular. b). Appropriate components used for logistic regression. c). The program is developed using an object-oriented approach.	This objective was successfully achieved. By leveraging an object-oriented approach, the project fosters flexibility, scalability, and code reusability, as outlined in the objectives. Organising neural network components into packages allows for easy integration and extension, further aligning with the objective of developing a modular system. This modular architecture enables diverse customization options, streamlines hyperparameter tuning, and enhances overall performance and user experience. Furthermore, this allows the program to be expanded and debugged easily, as was said to be beneficial previously.
2). User prediction.	a). Easy to use UI. b). An accurate prediction can be made regularly. c). Invalid data should be handled appropriately.	The project successfully achieved this objective. The UI is designed to be simplistic and easy to navigate, with different entries and interactables meaningfully labelled, as requested by the client. The models also regularly provide valid and accurate predictions for entries that are within the bounds of the training data, automatically identifying when data provided is valid but unrealistic for the model. In addition, inbuilt error checking ensures that the rest of the system won't deal with user input related errors, as identified in the tests conducted.
3). Model control.	a). The user is capable of changing the model used. b). A default model is loaded automatically upon application launch.	The project sets a high standard for model accuracy and validation, aligning with the objectives set. Through rigorous testing and iterative adjustments, it ensures the attainment of benchmark accuracy levels. Regularly achieving an average accuracy of around 70-75%, the project maintains consistency in performance, as highlighted in the objectives. In cases of accuracy shortfall, corrective measures such as resetting weights

	c). A new model can be generated.	and biases are implemented, ensuring model integrity and performance. Additionally, the inclusion of a default model meeting accuracy criteria enhances usability, fulfilling project objectives related to model accuracy and validation, overall ensuring that the model is capable of providing accurate predictions which will be imperative for Mr.Weiren's role.
4). Hyperparameter tuning.	a). Separate UI tab for easier navigation. b). A new valid model is generated for altered hyperparameters. c). Program rejects invalid hyperparameter values.	This objective has been achieved to an acceptable extent. Hyperparameter tuning can be conducted by the user in another tab that is clearly and easy to navigate to, as requested by the client. In addition tunable hyperparameters are clearly organised to groups of their respective components and labelled, as to what will be updated. In addition, tests 8 to 10 ensure that the hyperparameter adjustments and retraining works as intended, rejecting invalid inputs, while also aborting training if a viable model is unable to be generated within appropriate amount attempts. The program however doesn't effectively deal with value restriction for certain user entries. For instance, the user can set the decay to a value greater than 1 which shouldn't be accepted but is only rejected after a model is attempted to be generated a certain amount of times.
5). Saving a model.	a). A model can be saved only with a unique filename. b). The model is saved with relevant data only. c). A model is correctly loaded.	The objective is successfully achieved, as shown in tests 11 to 15. The program only allows models to be saved under a new filename and prevents overwriting previously saved models as outlined in the objectives. The project also optimises memory efficiency. With the default model save file occupying a mere 350 bytes of storage, it underscores the project's commitment to maximising resource utilisation and aligns with the specified objective of optimising memory usage.

Final Client Interview

As done before, all responses will be paraphrased slightly for clarity.

What are your thoughts on the application?

“Having had the opportunity to use the application, I must say that it's quite impressive. The concept and approach outlined in the proposal translates well into the actual implementation. The application's functionality aligns closely with our objectives, and I believe it has the potential to significantly improve our efficiency and accuracy in client selection."

Is there anything you would like to see in future iterations of the application?

"In future versions, I see room for enhancements. Given our ever-evolving data and market conditions, the ability to fine-tune the model's performance regularly would be invaluable. Application features that allow for periodic adjustments or recalibrations could ensure sustained optimum performance. Also, the inclusion of real-time data sources could enhance the system's accuracy and real-world applicability.

Additionally, it would be beneficial to have more insights into the data entries, like the typical range of values for each field used in the training data, or even a brief explanation of each hyperparameter for those of us less familiar with these concepts."

How would you rate the overall user interface design in terms of its aesthetics and intuitiveness?

"The overall user interface design exceeded my expectations. The layout is well-organised, making it easy to navigate through various features and functionalities. I particularly appreciate how the interface itself is also simplistic and how each element defines itself and its purpose, which adds to the overall polished look of the application."

Additionally, how would you assess the performance and responsiveness of the application, especially concerning its user interface interactions and load times?

"In terms of performance and responsiveness, the application performs admirably. The user interface interactions are smooth and fluid, with minimal lag or delay, although I would appreciate some form of indication when a model is being generated. From my little knowledge of python, I am impressed by the application’s ability to handle these large datasets, being able to retrieve and prepare data as well as train a model efficiently. Furthermore, the load times are impressively fast, contributing to a seamless user experience. Overall, I'm highly satisfied with the performance of the application."

Improvement / Future Adjustments

While the project has met its foundational objectives, reflecting on Mr. Weiren's feedback, there are a couple of areas where it could be significantly enhanced if revisited:

Enhancing Data Preprocessor

Dynamic Feature Evaluation and Extraction
The current system's inability to autonomously evaluate and select relevant features from diverse datasets significantly limits its adaptability and efficiency. Incorporating machine learning techniques such as automatic feature selection algorithms (e.g. recursive feature elimination) could enable the model to dynamically identify and focus on the most predictive features, thereby improving its accuracy and generalizability.

Automated Categorical Data Handling
The preprocessor's limitation in automatically encoding categorical data into unique numerical keys restricts the model's ability to process diverse datasets without manual intervention. Implementing an automated encoding system, such as one-hot encoding or label encoding that can detect and transform categorical variables, would make the data preparation process more efficient and less prone to errors.

Model Save Data and Configuration Management

Extended Model Save Data
Beyond saving model weights and biases, storing preprocessing configurations, including feature selection criteria and categorical encoding mappings, would allow for a seamless transition from training to deployment phases. This approach ensures consistency in how input data is handled, irrespective of when or where the model is deployed. Furthermore, the current saving and loading process could be updated to separate how the preprocessor and model data are saved. This is because currently the model class handles all the saving and loading, creating an unnecessary dependency on the preprocessor class for scaling data.

Configurable Settings via External Files
Utilising an external configuration or settings file to store and retrieve model parameters and preprocessing settings would enhance the flexibility and usability of the system. It allows users to adjust the model's behaviour without altering the core codebase, facilitating easier experimentation and optimization.

Batch Prediction Interface Development

Interface for Batch Processing
The current limitation of not providing a user-friendly interface for batch predictions constrains the model's practicality in operational environments. Developing a dedicated interface that supports batch inputs and displays predictions in an organised manner would greatly improve the model's usability in real-world scenarios, where bulk processing is often required.

Advanced Training Monitoring and Early Stopping

Early Stopping Implementation
Incorporating an early stopping mechanism would allow the training process to halt once the model ceases to show improvement on a validation set, preventing overfitting and saving computational resources. This feature is particularly important for ensuring that the model remains generalised and effective for unseen data.

Improvement in Model Evaluation
Moving beyond a singular focus on final accuracy to include a more nuanced understanding of model performance during training could involve metrics such as validation loss, precision, recall, and F1 scores. Moreover, developing a system to track and respond to fluctuations in model performance (e.g., through adjustment of hyperparameters or reinitialization of weights) could enhance model reliability and accuracy.

Bibliography

Images and Illustration

Moneysupermarket
‘Our mortgage calculators’
https://www.moneysupermarket.com/mortgages/mortgage-calculator/
(accessed September 20, 2023)

Barclays
‘Mortgage calculators’
https://www.barclays.co.uk/mortgages/mortgage-calculator/
(accessed September 20, 2023)

Rukshan Pramoditha, Medium
‘Overview of a Neural Network’s Learning Process’
https://medium.com/data-science-365/overview-of-a-neural-networks-learning-process-61690a502fa
(accessed October 14, 2023)

Aditya Sharma, LearnOpenCV
‘Activation Function in Deep Learning - A Complete Overview’
https://learnopencv.com/understanding-activation-functions-in-deep-learning/
(accessed October 23, 2023)

ML Glossary
‘Forward Propagation’
https://ml-cheatsheet.readthedocs.io/en/latest/forwardpropagation.html
(accessed October 29, 2023)

Research Sources

freeCodeCamp
'Machine Learning with python'
https://www.freecodecamp.org/learn/machine-learning-with-python/

ML Glossary
'Machine Learning Glossary'
https://ml-cheatsheet.readthedocs.io/en/latest/index.html

Omar Aflak
'Neural Network from scratch in Python'
https://towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65
(accessed November 3, 2023)

sentdex
'Neural Networks from Scratch in Python'
https://youtube.com/playlist?list=PLQVvvaa0QuDcjD5BAw2DxE6OF2tius3V3&si=CecA6KhZODvloT52
(accessed November 20, 2023)

Nick MacCullum, freeCodeCamp
'Deep Learning Neural Networks Explained in Plain English'
https://www.freecodecamp.org/news/deep-learning-neural-networks-explained-in-plain-english/
(accessed November 11, 2023)