1 TABLE OF CONTENTS

2 ABSTRACT

Predicting loan defaults is very important in finance to reduce risks and improve decision-making for lenders. Traditional methods like machine learning have trouble with complex financial data. Deep learning, especially Deep Neural Networks (DNNs), offers a better way to handle large, unbalanced datasets for credit risk prediction. This research aims to create a strong DNN model using the LendingClub dataset to classify loans as either default or non-default. Different DNN architectures were tested, including models with up to four hidden layers, using ReLU activation functions and the Adam optimizer. To handle class imbalances, the Synthetic Minority Over-sampling Technique (SMOTE) was used, which greatly increased the accuracy of predicting minority classes. The final tuned DNN architecture achieved an accuracy of 98.17%, surpassing traditional models and other deep learning approaches in the literature. The results underscore the potential of DNNs to improve loan default prediction, allowing financial institutions to make better lending decisions, minimize defaults, and optimize loan approval processes. This research contributes to the growing body of knowledge in deep learning applications for credit risk analysis, highlighting the efficiency of DNNs when paired with oversampling techniques to handle imbalanced datasets.

Keywords: Loan default prediction, Deep Neural Networks (DNN), Credit risk prediction, Imbalanced dataset, SMOTE, Financial modeling

3 LIST OF FIGURES

4 LIST OF TABLES

CHAPTER 1 INTRODUCTION

1.1 Introduction

Credit risk analysis is very important in the financial world, especially when it comes to figuring out if people might not pay back their loans. Credit risk means the chance that someone won't be able to pay what they owe, which can cause problems for the people who lend them money. Predicting if someone will not pay back a loan is important because it helps find risky borrowers early and prevents financial losses. With lots of data and complicated financial behaviours, older methods like logistic regression and decision trees aren't always good at finding complex patterns. These methods are easy to understand but often don't work well with big, mixed-up data sets from today's financial systems (Adebiyi et al., 2022).

Machine learning is being used more and more to make better predictions in credit risk analysis. One of its methods, deep learning, especially Deep Neural Networks (DNNs), has become a powerful choice compared to older ways. Unlike simple models, DNNs can understand complicated data patterns through many hidden layers, which help find detailed, non-linear patterns. DNNs are very good at handling both organized and unorganized data, making them perfect for tasks like predicting loan defaults (Sifrain, 2023; Jumaa et al., 2023).

The use of deep learning in predicting credit risk has become popular because it can effectively learn from complex data. For example, deep neural networks (DNNs) can handle large amounts of data and provide more accurate predictions than traditional methods like logistic regression, especially when dealing with unbalanced or noisy data (Jumaa et al., 2023). Additionally, the adaptable nature of DNNs, which includes features like ReLU activation functions and Adam optimizers, allows these models to adjust to shifts in financial patterns and trends (Turiel & Aste, 2020).

1.2 Problem Statement

The existing models for predicting credit risk, such as traditional machine learning methods like logistic regression and decision trees, have significant drawbacks. These models frequently struggle to understand the complex, non-linear patterns found in big financial datasets, which results in lower accuracy, particularly when predicting loan defaults (Adebiyi et al., 2022). Although techniques like different machine learning models have been tested, they don't perform well with large, unbalanced datasets where default cases are uncommon (Suliman, 2021). Despite the promise of deep learning, its application in loan default prediction has not been fully optimized. Many studies either underutilize these techniques or fail to handle class imbalances effectively, leading to suboptimal models (Owusu et al., 2023). This research aims to address these gaps by developing a deep learning model capable of learning complex patterns from high-dimensional, imbalanced datasets, thereby enhancing the predictive performance of loan default risk models.

1.3 Research Aim and Objective

The aim of this research is to develop an advanced DNN model for predicting loan defaults, utilizing the LendingClub dataset, to improve the accuracy of credit risk assessment and optimize decision-making for financial institutions.

Research Objectives:

- To design and implement multiple DNN architectures and compare their performance.
- To address the issue of class imbalance in the loan default dataset by applying the SMOTE and evaluating its impact on the model's prediction.
- To optimize the hyperparameters of the DNN models using advanced tuning techniques to achieve the best possible prediction performance.

1.4 The Scope of Research

This study looks at how deep learning (specifically, DNN) can be used, especially in predicting loan defaults using the LendingClub dataset. The focus will be on personal loans and classifying them as either default or non-default. To handle the unequal number of default and non-default cases, the dataset will be adjusted using methods like SMOTE. The study will only focus on improving deep learning models and comparing their results with traditional methods.

1.5 Significance of the Research

This research is important because it helps improve how deep learning is used to predict credit risks. By creating a more accurate prediction model, this study can greatly enhance the risk assessment tools used by banks and other financial organizations. Better models for predicting loan defaults will result in smarter lending practices, lowering the chances of bad loans and improving overall financial stability. Additionally, this research adds to academic knowledge by addressing gaps in current deep learning techniques, providing a new way to manage class imbalances in credit risk data.

2 CHAPTER 2 LITERATURE REVIEW

2.1 INTRODUCTION

The way credit risk is predicted in the financial world has changed a lot with the introduction of deep learning methods. This review looks at why deep learning is important for predicting loans and examines different deep learning models used in this area. In the past, banks and other financial organizations used basic models like logistic regression and decision trees to evaluate creditworthiness and predict loan defaults. Although these methods are simple and easy to understand, they don't always handle the complex, non-linear patterns found in big datasets very well. This limits their ability to make accurate predictions in today's complicated financial markets (Adebiyi et al., 2022; Suliman, 2021).

On the other hand, deep learning has become a strong alternative, greatly improving credit risk prediction. Its strong ability to understand complex patterns has made credit risk models more accurate and dependable. Deep learning methods, like Deep Neural Networks (DNNs), Recurrent Neural Networks (RNNs), and Long Short-Term Memory (LSTM) networks, have shown great success in dealing with both organized and unorganized financial data (Liang et al., 2023). Deep learning models are very good at handling data that changes over time, like financial trends. This makes them perfect for analysing these trends, which is important for predicting loans. To make good loan decisions, it's important to understand past borrower behaviour and economic conditions. As banks deal with more loan applications, deep learning helps find complex patterns that traditional methods might miss. This improves decision-making and reduces financial risks.

This literature review will explore the different deep learning models used for loan prediction, looking at their advantages, drawbacks, and how they compare to each other. By combining recent research results, research hope to give a complete picture of where deep learning stands in credit risk evaluation and suggest future research and practical uses in the finance industry.

2.2 RELATED WORK

Deep learning has greatly improved credit risk prediction by allowing models to understand complex, non-linear patterns in big datasets. Different deep learning methods like DNNs, RNNs, and hybrid models have been studied. These models work well in finance because they can find detailed patterns in large, organized data.

2.2.1 Deep Neural Network (DNN)

DNNs are the backbone of deep learning, able to process structured and unstructured data through layers. These models have input, hidden and output layers, where data is transformed through weighted connections between neurons. The layer structure allows DNNs to capture complex patterns and non-linearity in data, making them perfect for credit risk prediction. In this space researchers have tried DNNs of all shapes and sizes and have shown them to be better than traditional machine learning models.

For example, Sifrain (2023) used a very simple ANN with just one hidden layer of 18 neurons to predict loan defaults. Despite being simple, the model achieved an AUC of 0.936, better than logistic regression and random forest. It had a specificity of 82.5% and accuracy of 89.4%. The ANN was able to capture nonlinearity between variables and showed that even simple models can be powerful in credit risk prediction.

On the other hand, Jumaa et al. (2023) used a more complex DNN architecture for consumer loan default prediction. They expanded the original 10 input features to 26 using one-hot encoding and fed it into a multilayer feed-forward neural network. The hidden layer had 1024 units and ReLU was used to introduce nonlinearity. This architecture achieved 97.45% accuracy on a balanced dataset, showing DNNs can handle high dimensional classification tasks. The complexity of this model shows how DNNs can scale with data dimensionality to improve credit risk prediction.

Turiel & Aste (2020) used a different method by combining logistic regression with a DNN in a two-step model to predict loan defaults on peer-to-peer (P2P) lending platforms. Their model first used logistic regression to predict if a loan would be accepted, and then a DNN with two hidden layers (each having five neurons) was used to predict defaults on those accepted loans. Even though the DNN was small, it performed better than logistic regression and support vector machines, achieving a recall rate of 72%. This research shows that DNNs are very flexible and can work well even when used with other machine learning methods.

2.2.2 Hybrid Approach for Loan Prediction

Combining traditional machine learning with deep learning techniques has become popular for predicting credit risk. These hybrid methods use the best parts of each approach to improve results.

For example, Liu et al. (2022) combined the XGBoost algorithm, which is good with tree-based methods, with a neural network that uses graphs to find important features. In their method, XGBoost changed the data so it could be easily separated, and then this data was used in a deep neural network. This combined method, called forgeNet, effectively found connections between features using graph structures. The hybrid model got an average accuracy of 87.52% and a high F1-score of 93.13%. Using XGBoost for feature selection and the neural network for complex relationships shows how hybrid methods can improve credit risk prediction.

Likewise, Owusu et al. (2023) used a combined method that involved the Adaptive Synthetic (ADASYN) oversampling technique along with DNN to handle class imbalance in predicting loan defaults. ADASYN creates extra samples for the minority class (loan defaults) to even out the dataset. The DNN had three hidden layers and was improved using the Adam optimizer. The model got a recall of 96%, precision of 97.2%, and overall accuracy of 94.1%. This method shows how combing DNNs with oversampling techniques like ADASYN can greatly enhance the detection of minority class events, such as loan defaults, which is a frequent issue in financial data.

2.2.3 Ensemble Methods for Loan Prediction

Combining several models to enhance prediction accuracy has been successful in credit risk assessment. By using the advantages of various models, ensemble methods can perform better than any single model alone.

Zhang et al. (2020) used a Stacking approach that included Artificial Neural Networks (ANNs) along with other models like Random Forest, AdaBoost, and XGBoost as base classifiers. They used loan data from LendingClub, a peer-to-peer lending platform, specifically from the fourth quarter of 2018. After preparing the data to handle missing information and balance the classes, their combined model achieved excellent results: 98.16% accuracy, 97.96% precision, 98.68% recall, and an F1-score of 98.32%. This high performance shows that combining deep learning with ensemble methods can lead to very accurate credit risk predictions.

2.2.4 RNNs and LSTM Networks for Loan Prediction

Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks are especially good for financial modeling, where data that changes over time is very important.

Unlike regular networks that only look at each input once, RNNs and LSTMs remember information from earlier inputs, which makes them perfect for working with data that comes in a sequence and for understanding how things change over time. This makes them very useful for predicting things, like how likely someone is to have credit problems.

Paudel et al. (2023) showed that recurrent models work well by using Gated Recurrent Units (GRUs) and bidirectional Long Short-Term Memory networks (Bi-LSTMs) for predicting credit risk across multiple categories. Their GRU model had one hidden layer with 32 units, while the Bi-LSTM model had a Bi-LSTM layer, followed by a dense layer with 50 units and a dropout layer to avoid overfitting. The Bi-LSTM model did better than the GRU, getting a weighted F1 score of 0.93 and an overall accuracy of 93%. This research shows that LSTM-based models are better at dealing with sequences and improving prediction accuracy in financial data.

In another study by Liang & Cai (2020) used LSTM networks to predict default rates in peer-to-peer (P2P) loans. They used a five-layer LSTM model to understand how default rates change over time. The LSTM model worked better than traditional models like ARIMA and support vector machines (SVMs), with lower errors in its predictions. The model was very good at predicting both how high the default rates would be and whether they would go up or down, with over 80% accuracy for short-term predictions. Even for medium and long-term predictions, the LSTM model stayed accurate around 60%-70%.

Similarly, another study on financial data, conducted by Clements et al. (2020), discovered that a 3-layer LSTM model, which includes two LSTM layers and a dense output layer, performed the best. Techniques to prevent overfitting, like dropout and zone out, were used at low rates, averaging about 0.2 across the model. The model was trained using the Adam optimization method, with a batch size of 512. A decay rate of 0.8, starting from an initial learning rate of 1e-4, was found to be the most effective setup for all models. The model's performance was measured using the Gini coefficient and recall, both showing good results. The Gini coefficient was 95.5% and recall was 81.2%, indicating the model's strong ability to distinguish between default and non-default cases. These results highlight the potential of LSTM models in financial modeling, especially in predicting credit risk with high accuracy.

From basic structures to advanced combined and group models, Deep Neural Networks (DNNs) have repeatedly shown they can find complex patterns in financial information. Whether used with older machine learning models or in mixed forms, DNNs have performed better than traditional methods in predicting credit risk. Architectures like LSTMs, which are part of DNNs, can also help models understand time-related patterns in data that changes over time. As research in this area keeps growing, we can expect more new ideas in DNN designs and their use in credit risk evaluation, giving financial organizations stronger tools for predicting and handling risks.

Table 2.2.1 Comparative Analysis of the above literature on different deep learning architecture

Author	Architecture Used	Dataset Used	Evaluation	Key Findings

Sifrain (2023)	ANN (1 hidden layer, 18 neurons)	LendingClub (2013-2014)	Accuracy = 89.4%
			Specificity = 82.5%
			AUC = 0.936
				Simple ANN models can outperform traditional models like logistic regression and random forest in credit risk prediction.

Jumaa et al. (2023)	Multilayer feed-forward neural network (26 input features, 1 hidden layer with 1024 units, ReLU activation)	UAE Consumer Loan Survey	Accuracy = 97.45%	High accuracy for predicting loan defaults using a balanced dataset and simple feed-forward architecture
Turiel & Aste (2020)	Two-phase model: Logistic Regression for loan acceptance, DNN (2 hidden layers with 5 neurons each) for loan default prediction	LendingClub (Combined dataset of accepted and rejected loans)	Recall = 72%	Combining logistic regression with DNN enhances loan default prediction, outperforming traditional models like SVM.

Liu et al. (2022)	Hybrid XGBoost + graph-based deep neural network (forgeNet)	LendingClub (2007-2016)	Accuracy = 87.52%
			F1-Score = 93.13%
			G-mean = 85.59%
				Combination of XGBoost and graph-based DNN captures feature interactions, improving credit risk prediction

Owusu et al. (2023)	Hybrid Model: ADASYN + DNN (3 hidden layers, Adam optimizer)	LendingClub (2007-2015)	Recall = 96%	DNN models combined with oversampling techniques like ADASYN

			Precision = 97.2%
			Accuracy = 94.1%
				significantly enhance detection of minority class events (e.g., loan defaults) in imbalanced datasets.

Zhang et al. (2020)	Ensemble Model (ANN + Random Forest, AdaBoost, XGBoost, Stacking)	LendingClub peer-to-peer loan dataset	Accuracy = 98.16%
			Precision = 97.96%
			Recall = 98.68%
			F1-Score = 98.32%
				Stacking ensemble models combining deep learning and tree-based methods provide highly accurate predictions for loan defaults.

Paudel et al. (2023)	Recurrent Neural Network (GRU) and Bi-LSTM	LendingClub (multi-class dataset: high, medium, low risk)	F1-Score = 0.93
			Accuracy = 93%
				LSTM-based models outperform GRU in capturing sequential data patterns, leading to better accuracy in credit risk prediction.

Liang & Cai (2020)	LSTM (5-layer LSTM network)	LendingClub (Time-series data for monthly default rates)	Short-term accuracy =80% Medium-term accuracy =60%-70%	LSTM models outperform traditional methods like ARIMA and SVM for predicting loan default rates, especially for short-term predictions.
Clements et al. (2020)	LSTM (3-layer LSTM dropout = 0.2)	Credit Transactional Data	Recall = 81.2% Gini Coefficient = 95.5%	LSTM models perform well in financial modeling, especially in predicting credit risk with high accuracy.

The dataset used in this research is the LendingClub dataset, a widely recognized public dataset often used in credit risk modeling. Many studies, including those by Sifrain (2023) and Turiel & Aste (2020), have used this dataset or similar ones, allowing for direct comparison with the current study. The LendingClub dataset is essential for credit risk modeling studies, providing real-world loan data that helps in analysing borrower behaviour and loan default trends. Its extensive use in various studies serves as a standard for evaluating deep learning models in predicting loan defaults, making it easier to compare with past research. Based on this foundation, this research suggests a new method that combines a Deep Neural Network (DNN) with the Synthetic Minority Over-sampling Technique (SMOTE) to handle the natural class imbalance in the LendingClub data. Previous studies have used under sampling (Sifrain, 2023) or mixed methods (Liu et al., 2022) but SMOTE provides a more refined solution. By creating synthetic samples for the minority class (loan defaults), SMOTE helps create a more balanced dataset without removing important non-default data, which could reduce overfitting and bias problems often found in imbalanced datasets.

The decision to use Deep Neural Networks (DNNs) is based on their proven capability to understand complex, non-linear patterns in both organized and disorganized data, as shown by their successful results in previous research (Sifrain, 2023; Jumaa et al., 2023). DNNs' ability to work with different types of input data and manage large, complex datasets makes them especially good for analysing financial data. To make sure this study can be compared to earlier research, we will use common evaluation methods such as AUC, Accuracy, F1-score, precision, recall, and overall accuracy. Since studies by Liu et al. (2022) and Zhang et al. (2020) showed good results with high F1-scores and recall rates using advanced methods, this study aims to reach or exceed those results with a deep learning model designed for imbalanced data. By implementing a DNN combined with SMOTE on the LendingClub dataset, this study seeks to advance credit risk prediction accuracy and provide valuable insights for financial institutions in managing loan portfolios.

3 CHAPTER 3 RESEARCH METHODOLOY

This chapter uses the CRISP-DM (Cross Industry Standard Process for Data Mining) approach to organize the study on credit risk analysis and loan default prediction. The CRISP-DM process has six steps: Business Understanding, Data Understanding, Data Preparation, Modeling, Evaluation, and Hyperparameter Tuning. These steps create a complete plan for the research, starting with understanding business goals and ending with optimizing deep learning models for the best results.

3.1 Business Understanding

The main goal of this study is to forecast when loans might not be paid back and to lower the risk of lending money for banks and other financial companies. Knowing when loans might fail is very important because it helps find risky borrowers and makes better lending choices. By making loan default predictions more accurate, financial groups can reduce their losses, change interest rates according to risk, and make smarter decisions about who to lend money to.

Traditionally, credit risk prediction used models like logistic regression, decision trees, and random forests. These models are simple to set up but can have trouble with the complex, non-linear patterns found in big datasets. As mentioned in Chapter 2, many researchers have investigated using deep learning methods, such as Deep Neural Networks (DNNs), Recurrent Neural Networks (RNNs), and Long Short-Term Memory (LSTM) models. These advanced models can find detailed connections in financial data that simpler models might miss. However, even with their success, there are still challenges, especially when dealing with imbalanced datasets where non-default cases greatly outnumber defaults. Overcoming these challenges and comparing different model structures are key parts of this research.

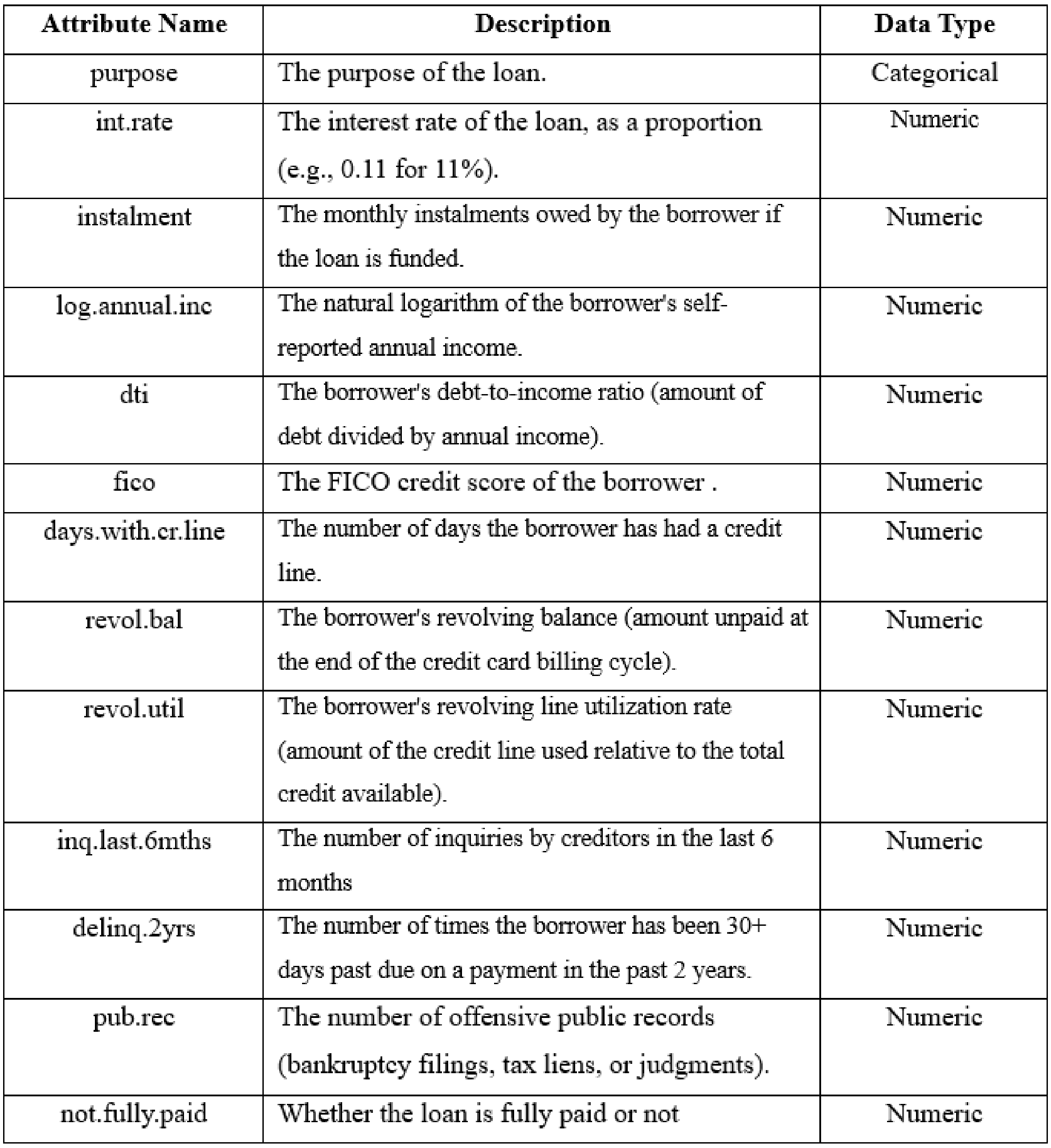
In this research, researcher want to make loan default predictions more accurate by using different Deep Neural Network (DNN) designs. Researcher plan to see how these models compare to older methods and mixed approaches. Our goal is to find the best design and help fill in the gaps found in past studies. Researcher will use the Synthetic Minority Over-sampling Technique (SMOTE) to deal with unequal groups and tune the settings of our models to improve their performance. Previous studies (Zhang et al., 2020; Owusu et al., 2023) have shown that these methods can make a big difference.

3.2 Data Understanding

The data used in this study is from LendingClub, a well-known platform for peer-to-peer lending. The data includes detailed information about loan applications, such as the borrower's FICO score, credit history, reason for the loan, and the status of the loan. As mentioned in Chapter 2, this data has been widely used in similar studies because of its large size and thorough details, which make it good for advanced learning methods.

Meta-Data:

Table 3.2.1 Meta-Data of the Dataset



Attribute Name	Description	Data Type
purpose	The purpose of the loan.	Categorical
int.rate	The interest rate of the loan, as a proportion (e.g., 0.11 for 11%).	Numeric
instalment	The monthly instalments owed by the borrower if the loan is funded.	Numeric
log.annual.inc	The natural logarithm of the borrower's self-reported annual income.	Numeric
dti	The borrower's debt-to-income ratio (amount of debt divided by annual income).	Numeric
fico	The FICO credit score of the borrower .	Numeric
days.with.cr.line	The number of days the borrower has had a credit line.	Numeric
revol.bal	The borrower's revolving balance (amount unpaid at the end of the credit card billing cycle).	Numeric
revol.util	The borrower's revolving line utilization rate (amount of the credit line used relative to the total credit available).	Numeric
inq.last.6mths	The number of inquiries by creditors in the last 6 months	Numeric
delinq.2yrs	The number of times the borrower has been 30+ days past due on a payment in the past 2 years.	Numeric
pub.rec	The number of offensive public records (bankruptcy filings, tax liens, or judgments).	Numeric
not.fully.paid	Whether the loan is fully paid or not	Numeric

The dataset contains 9578 rows and 14 columns, each representing features such as loan amount, interest rate, credit score, and whether the loan was fully paid or not (target variable).

Notably, no missing values were present in the dataset, which simplifies the pre-processing step. Before modeling, an exploratory data analysis (EDA) is to be performed to better understand the dataset's structure, distribution, and relationships between features. The key findings from the EDA might include:

1. **Class Imbalance**: The target variable (not.fully.paid) was found to be highly imbalanced (Owusu et al., 2023), with most loans fully repaid. This imbalance necessitates the use of oversampling techniques like SMOTE to generate synthetic examples of the minority class (loan defaults), ensuring the model could learn patterns from both classes effectively.
2. **Feature Correlations**: A correlation heatmap is to be plotted which reveals significant relationships between variables and the target variable.
3. **FICO Score Analysis**: Analysing FICO scores is very important when predicting loans because it shows how likely someone is to pay back money they borrow.

These insights guide the future data preparation and modeling steps, ensuring that the deep learning models are specific to the dataset's characteristics.

3.3 Data Preparation

The data preparation phase focuses on transforming the raw dataset into a suitable format for deep learning models. Key steps include:

1. **Handling Class Imbalance**: Given the class imbalance in the target variable, SMOTE is to be applied to the training set. SMOTE creates synthetic samples for the minority class by interpolating between existing minority class instances. This oversampling strategy helps mitigate the model's bias toward the majority class (fully paid loans), ensuring that it can better detect defaults.
2. **Feature Engineering**: Features like purpose (a categorical variable) are to one-hot encoded to allow deep learning models to process categorical information. One-hot encoding transforms each category into a new binary column, preventing the model from assuming any ordinal relationship between categories (Jumaa et al., 2023).
3. **Normalization**: Continuous variables, such as loan amount and interest rate, are to be normalized using standardization techniques. Normalization is crucial to ensure that features with different scales did not extremely affect the model's learning process. Each feature is transformed to have a mean of 0 and a standard deviation of 1.

These steps ensures that the dataset is clean, balanced, and ready for modeling, with features formatted correctly for deep learning algorithms.

3.4 Data Modeling

When data modeling a DNN, choosing the right architecture is very important for getting the best results for the task. A DNN usually has many layers of neurons, or units, where each neuron changes the input data in some way. These layers can be split into input, hidden, and output layers. The input layer takes in the raw data, and the output layer gives the model's prediction. The hidden layers help find complex patterns in the data by changing the features.

For many tasks involving predictive modeling, a fully connected feedforward neural network is often the first choice. In this type of network, each neuron in one layer is linked to every neuron in the next layer. The number of layers and the number of neurons in each layer are important decisions. Typically, networks with more layers (deeper networks) can capture more complex patterns (Jumaa et al., 2023).

Every layer in the deep neural network uses an activation function to add non-linearity to the model. Common activation functions are the Rectified Linear Unit (ReLU), which is usually chosen because it's fast and works well in deep networks, and the sigmoid or SoftMax functions, which are often used in the output layer for classification tasks.

Adjusting hyperparameters is crucial for setting up the network structure. These include the number of hidden layers, neurons in each layer, learning rate, batch size, and dropout rate. These need to be carefully adjusted to balance the model's complexity and its ability to work well with new data. Dropout is a method used to randomly turn off neurons during training, which helps prevent the network from focusing too much on any single feature and reduces the risk of overfitting.

In the training process, the Adam optimizer (Adaptive Moment Estimation) is often used to adjust the model's weights. Adam combines the strengths of two well-known optimization methods momentum and RMSProp by changing the learning rate for each parameter based on the first and second moments of the gradients. This helps the model reach its best state more quickly than standard SGD, especially when dealing with gradients that are not smooth or when data is not common. Adam's skill in managing gradients that are not common and adjusting the learning rate makes it very useful for deep learning tasks, where manually setting the learning rate can be difficult (Owusu et al., 2023).

3.5 Model Evaluation

During the Model Evaluation stage in the CRISP-DM process, we check how well DNN model is doing by looking at several important measures: accuracy, precision, recall, and the F1-score. Accuracy tells us how often the model is correct by showing the percentage of correct predictions out of all predictions made. But in cases with uneven data, like predicting loan defaults, accuracy by itself might not give us the full story. That's why researcher also look at precision and recall. Precision tells us how many of the positive predictions were actually correct, while recall shows how many of the actual positives the model correctly identified. To get a balanced view of both precision and recall, we use the F1-score, which is like an average of these two measures, giving us one number that considers both false positives and false negatives.

To evaluate how well the model is learning, charts are created which show the model's accuracy and error over time during training and validation. These charts help see if the model is doing too well on the training data (overfitting) or not well enough (underfitting) by comparing its performance on the training data to its performance on unseen validation data. A well-performing model will show accuracy increasing and error decreasing steadily, with both training and validation lines closely matching, which means the model is doing a good job on new data without focusing too much on the training data.

3.6 Tools and Libraries

For this research, a combination of tools and libraries was used to build, train, and evaluate the deep learning models. **Python** served as the primary programming language, and **TensorFlow** and **Keras** were employed to develop and optimize the Deep Neural Network (DNN) architectures. Data preprocessing tasks, such as handling missing values and transforming categorical features, were carried out using **Pandas** and **NumPy**. The **imlearn** library was particularly crucial for applying **SMOTE** to address class imbalance by generating synthetic samples for the minority class, thereby improving the model's ability to detect loan defaults. Visualization libraries like **Matplotlib** and **Seaborn** were used to generate training and validation plots, helping assess the model's learning process. **Scikit-learn** provided key evaluation metrics such as accuracy, precision, recall, and F1-score, enabling a thorough assessment of the model’s performance. These tools together created a robust framework for developing accurate and efficient models.

ASSIGNMENT 2

4 CHAPTER 4 MODEL IMPLEMENTATION

4.1 EXPLORATORY DATA ANALYSIS (EDA)

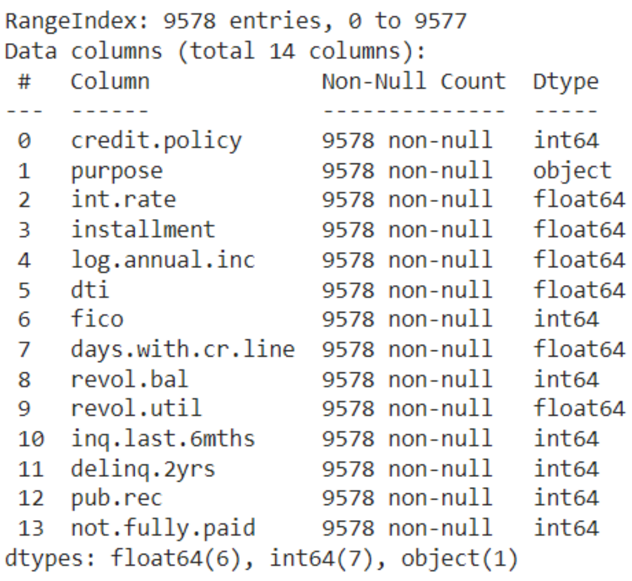
4.1.1 Dataset Overview

The df.info() function in pandas gives a quick summary of a Dataframe. It shows important information like the number of rows and columns, the type of index, column names, how many non-empty entries each column has, and the data type of each column. This function is very helpful for getting a general idea of the dataset's structure, checking data quality, finding missing values, and understanding the types of data researcher is dealing with.

CODE:

```
#Checking for Missing Values
df.info()
```

OUTPUT:



```
RangeIndex: 9578 entries, 0 to 9577
Data columns (total 14 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   credit.policy      9578 non-null   int64
 1   purpose            9578 non-null   object
 2   int.rate           9578 non-null   float64
 3   installment        9578 non-null   float64
 4   log.annual.inc     9578 non-null   float64
 5   dti                9578 non-null   float64
 6   fico               9578 non-null   int64
 7   days.with.cr.line  9578 non-null   float64
 8   revol.bal          9578 non-null   int64
 9   revol.util         9578 non-null   float64
 10  inq.last.6mths     9578 non-null   int64
 11  delinq.2yrs        9578 non-null   int64
 12  pub.rec            9578 non-null   int64
 13  not.fully.paid     9578 non-null   int64
dtypes: float64(6), int64(7), object(1)
```

Figure 4.1.1 Dataset overview

It can be observed that the dataset consists of 9,578 rows and 14 columns. Notably, there are no missing values present in the data. The columns are composed of various data types: 6 columns contain floating-point numbers, 7 columns contain integers, and 1 column contains object data.

4.1.2 Plotting the Target Variable

Visualizing the distribution of the target variable is an important part of exploring data. It shows how balanced or unbalanced the classes are in the 'not.fully.paid' variable. This helps researcher understand the data better and can guide decisions on how to build models, especially when dealing with class imbalance.

CODE:

```
#Plotting Target variable
sns.countplot(x='not.fully.paid', data= df,
              palette= ['blue','Red'],hue = 'not.fully.paid', edgecolor='black')
plt.title('Target Variable Distribution')
plt.show()
```

OUTPUT:

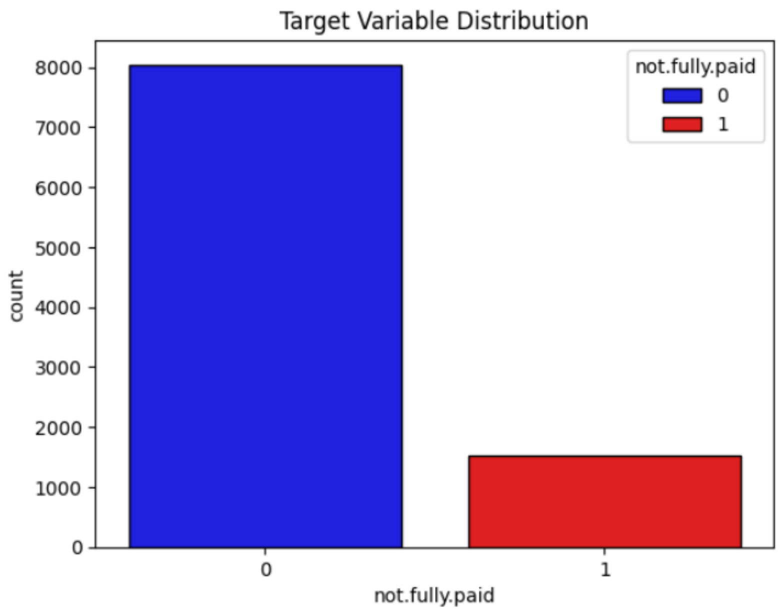


Figure 4.1.2 Target variable distribution

The target variable is not evenly spread out, so there is a need to balance it. This will be done later in the implementation.

4.1.3 Plotting Histogram to Check the Distribution

Plotting histograms for continuous variables is a crucial part of exploring data. These graphs show how each variable is spread out, highlighting important features like the average value, how spread out the data is, if it leans more to one side (skewness), and any unusual points. This helps researcher see the overall pattern of the data, find trends, and decide how to prepare the data and choose the right methods for analysis.

CODE:

```
# Plotting histogram to check the distribution of continous value
# List of colors
colors = ['blue', 'green', 'red', 'purple', 'orange']

for i, col in enumerate(df.columns):
    sns.histplot(df[col], kde=True, color=colors[i % len(colors)])
    plt.title(f'Distribution of {col}')
    plt.show()
```

OUTPUT:

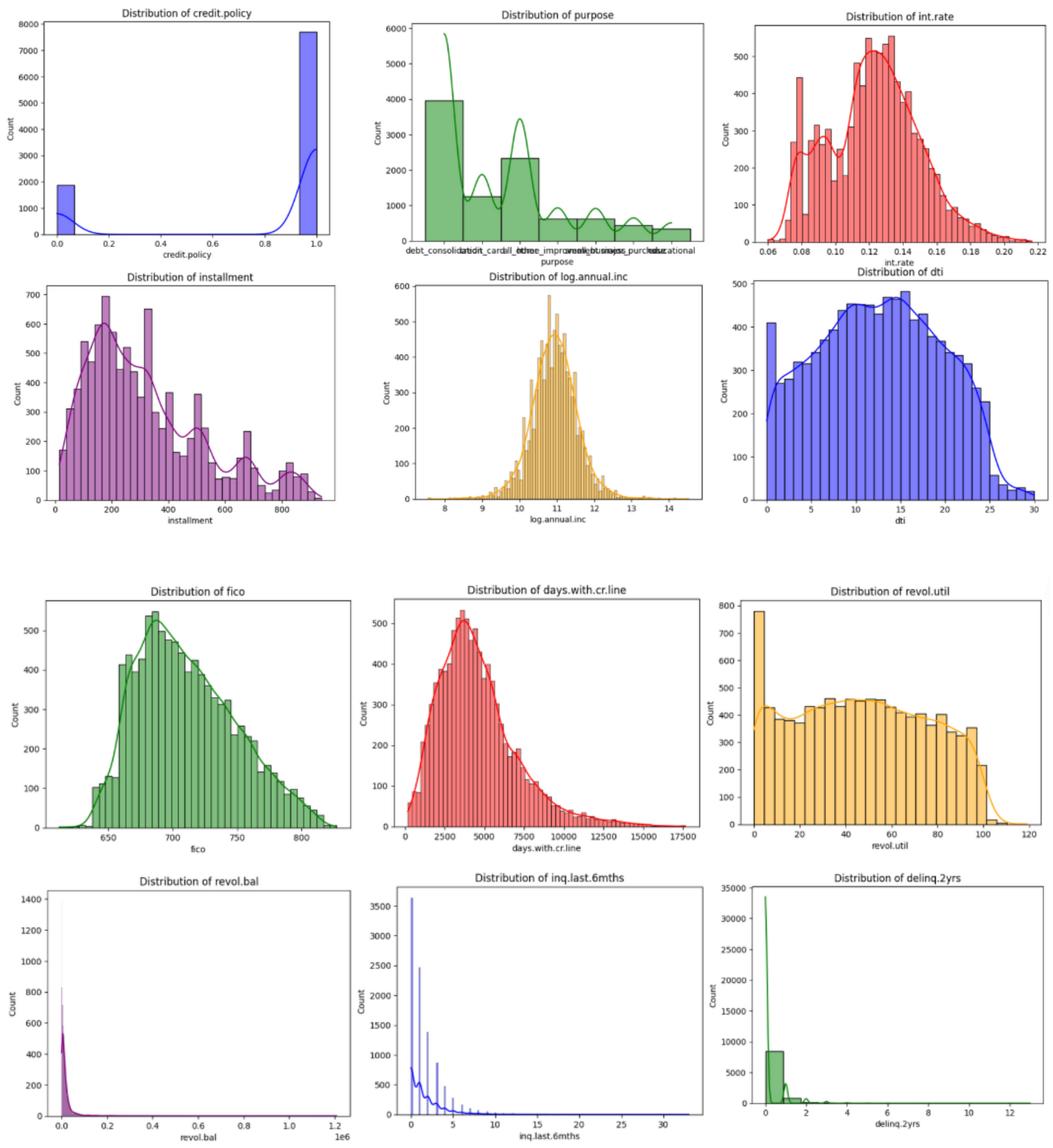


Figure 4.1.3 Distribution of the features

The histograms indicate that the continuous variables are roughly normally distributed. For deep learning models, this distribution is usually fine without any changes, because these models can work well with this kind of data without needing special adjustments.

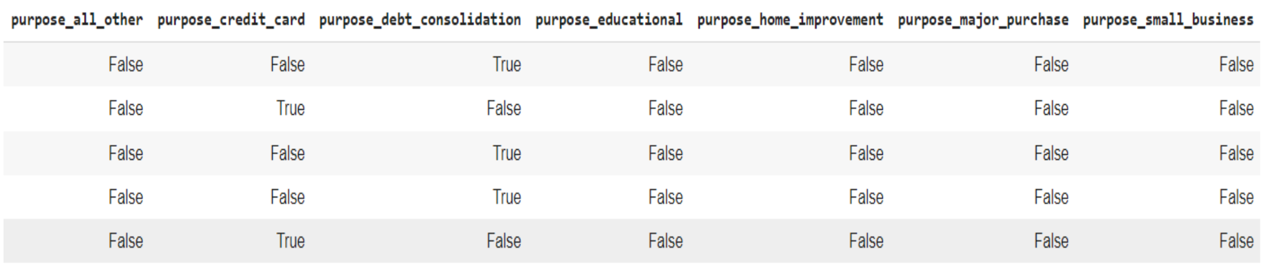
4.1.4 One-hot Encoding the Categorical Variable

One-hot encoding is used on the 'purpose' column to change categorical data into a binary format. This process creates separate columns for each category, with each column containing either a True or False. This helps machine learning models understand and use categorical information without thinking that the categories have a specific order.

CODE:

```
# one-hot encoding the categorical column
df = pd.get_dummies(df, columns=['purpose'])
df.head()
```

OUTPUT:



purpose_all_other	purpose_credit_card	purpose_debt_consolidation	purpose_educational	purpose_home_improvement	purpose_major_purchase	purpose_small_business
False	False	True	False	False	False	False
False	True	False	False	False	False	False
False	False	True	False	False	False	False
False	False	True	False	False	False	False
False	True	False	False	False	False	False

Figure 4.1.4 One-hot encoding of categorical variable

The one-hot encoding process has transformed the 'purpose' column into multiple binary columns. Each new column represents a specific loan purpose, containing True or False values to indicate the loan's intended use.

4.1.5 Correlation Heatmap

A correlation heatmap visually shows how different variables in the dataset are related. It helps researcher see strong positive or negative connections, potential problems with variables being too similar, and how features depend on each other. This information helps researcher choose the right features and make better decisions when creating models during the analysis.

CODE:

```
# Correlation heatmap
plt.figure(figsize = (16,12))
sns.heatmap(data = df.corr(), annot = True)
plt.show()
```

OUTPUT:

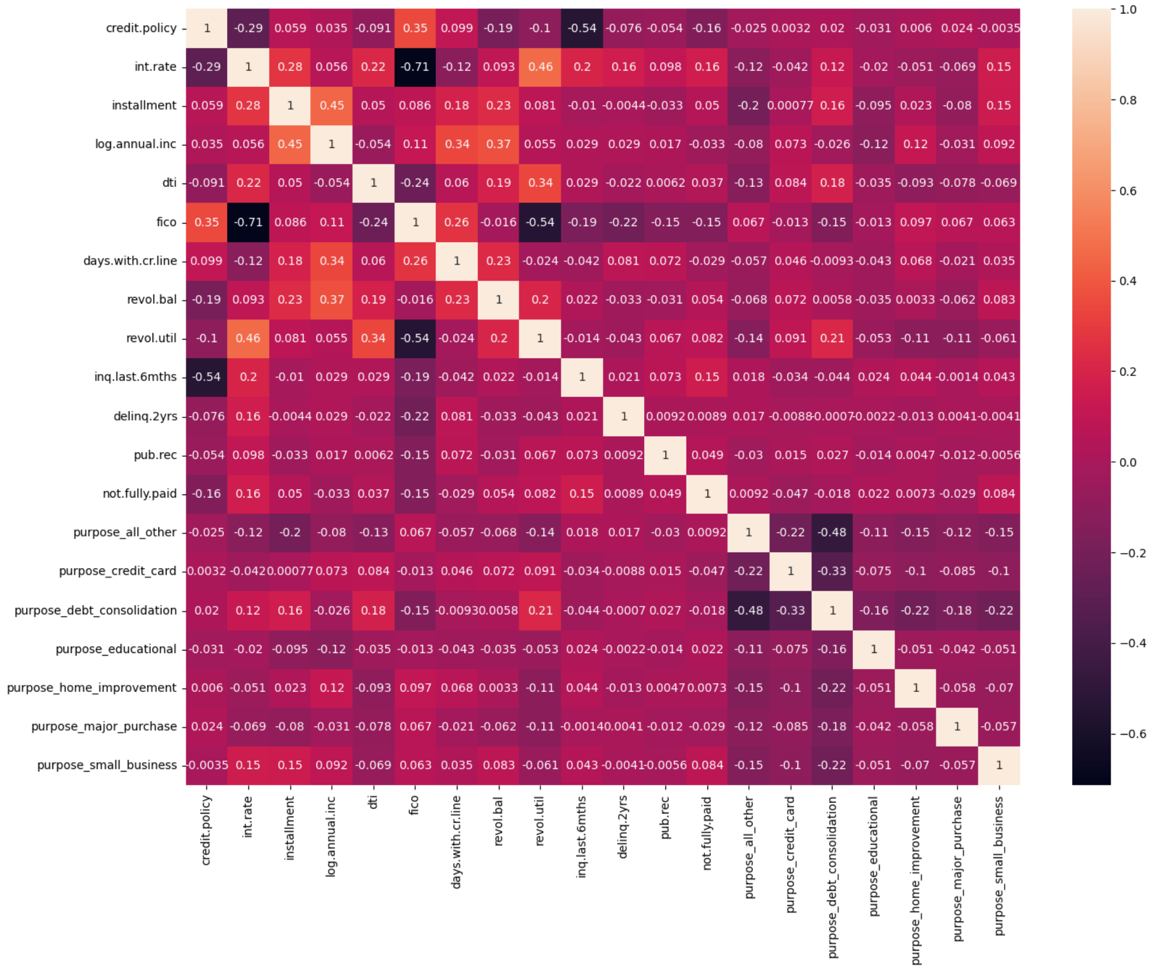


Figure 4.1.5 Correlation heatmap of the variable

The results show significant connections between various factors in the data. Specifically, there's a strong link (-0.715) between the interest rate and the FICO score, suggesting that higher FICO scores lead to lower interest rates. There's also a moderate connection (-0.541) between the FICO score and how much credit is being used. Furthermore, the number of credit checks in the last six months has a moderate link (-0.536) with the credit policy. These connections offer important information about what affects loan conditions and credit reliability.

4.1.6 Analysis of FICO Score

Analysing FICO scores is very important when predicting loans because it shows how likely someone is to pay back money they borrow. This score is based on a person's credit history and helps lenders decide if they should lend money, how much interest to charge, and if the loan is risky. Generally, a higher FICO score means less chance of not paying back the loan, which can lead to better loan conditions and a higher chance of getting approved.

4.1.6.1 FICO Score and Loan Status

This distribution shows how FICO scores are different for loans that were fully paid off and those that defaulted. It helps us find possible credit score limits that might show how likely someone is to repay a loan. This gives us useful information for judging risk and understanding how FICO scores affect loan results.

CODE:

```
# Distribution of FICO Score
plt.figure(figsize=(10,6))

df[df['not.fully.paid']==0]['fico'].hist(bins=35,alpha=0.5,
                                         label='not.fully.paid=1',color ='blue',edgecolor='black')
df[df['not.fully.paid']==1]['fico'].hist(bins=35,alpha=0.7,
                                         label='not.fully.paid=0',color='red',edgecolor='black')
plt.title('FICO Score Distribution')
plt.xlabel('FICO Score')
plt.legend()
```

OUPUT:

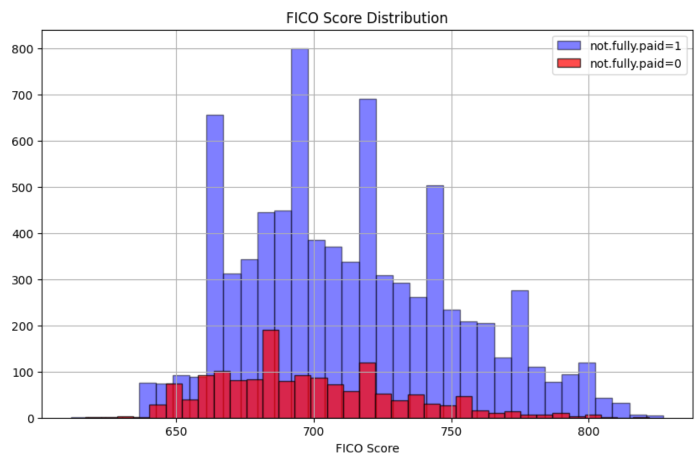


Figure 4.1.6 Fico Score distribution

The chart shows how FICO scores are different for people who paid back their loans and those who didn't. The blue parts show people who didn't pay back their loans, and the red parts show those who did. People with higher FICO scores (about 700–800) are more likely to pay back their loans, as seen by the bigger red parts in this range. On the other hand, people with lower FICO scores (below 700) are more likely to not pay back their loans, which is shown by the taller blue parts.

4.1.6.2 FICO Score and the Credit Policy

Plotting FICO scores against credit policy choices offers useful information about lending methods. This visual representation helps pinpoint the score limits for loan acceptance, uncovers possible biases in lending choices, and shows how credit scores affect loan eligibility for various customer groups.

CODE:

```
# FICO Score and the credit policy
plt.figure(figsize=(10,6))
df[df['credit.policy']==1]['fico'].hist(bins=30,alpha=0.5,
                                        label='credit.policy=1',color ='blue',edgecolor='black')
df[df['credit.policy']==0]['fico'].hist(bins=30,alpha=0.7,
                                        label='credit.policy=0',color ='red',edgecolor='black')
plt.xlabel('FICO Score')
plt.legend()
```

OUTPUT:

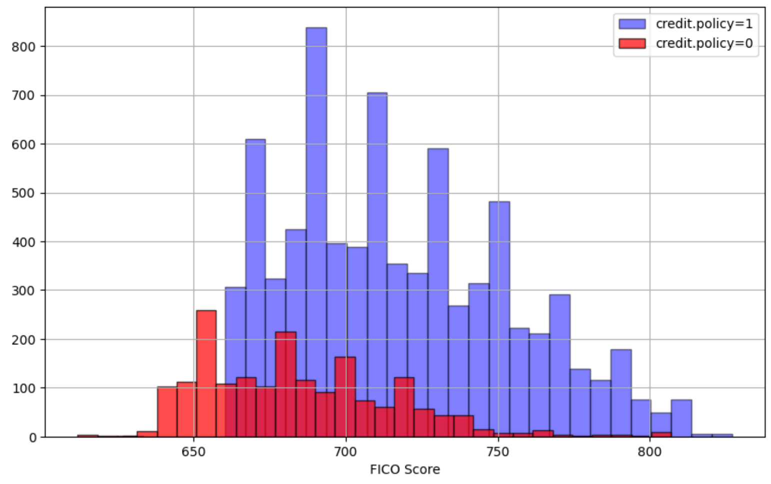


Figure 4.1.7 Fico score and credit policy distribution

The histogram displays how FICO scores are spread out based on the credit policy. Borrowers who meet the credit policy (shown in blue) typically have FICO scores ranging from 650 to 800, with more scores falling between 700 and 750. Borrowers who don't meet the credit policy (shown in red) mostly have lower FICO scores, below 700. This graph indicates that having a higher FICO score makes it more likely to meet the credit policy.

4.1.6.3 FICO Score and Interest Rate

This scatter plot shows the connection between FICO scores and interest rates, with different colours for loan repayment status. It helps us see how credit scores affect interest rates and if these factors are linked to the risk of loan default. This information is important for understanding how loans work and for assessing risk.

CODE:

```
# FICO Score and Interest Rate
sns.scatterplot(x = "fico", y = "int.rate", data = df[["fico", "int.rate", "not.fully.paid"]],
                hue = 'not.fully.paid', palette ='Set1');
```

OUTPUT:

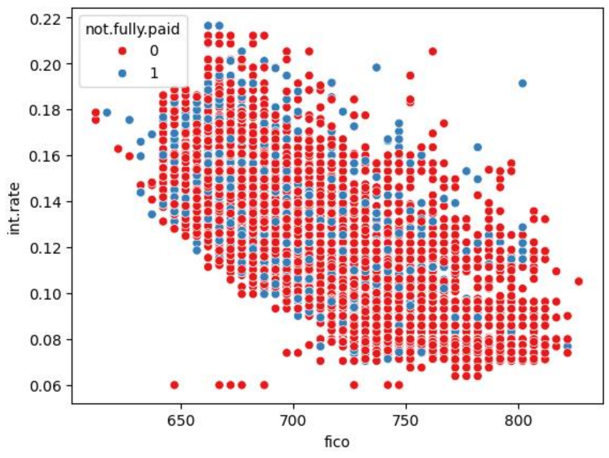


Figure 4.1.8 Scatterplot of Fico score and Interest Rate

The scatterplot above shows that as FICO scores go up, interest rates go down, meaning borrowers with lower risk get better rates. Most of the red dots (loans fully paid back) are on the chart, while blue dots (loans not fully paid back) are mostly found where FICO scores are lower, and interest rates are higher. This suggests that borrowers with lower FICO scores and higher interest rates are more likely to default on their loans.

4.1.6.4 FICO Score and Interest Rate with Credit Policy

This lmplot shows how FICO scores, interest rates, loan status, and credit policy decisions are connected. By dividing the data into parts, it shows how these factors work together, which helps lenders see risk patterns and possibly adjust their rules for approving loans to balance making money with managing risk.

CODE:

```
# FICO Score and Interest with Credit Policy
sns.lmplot(x='fico',y='int.rate',data=df,col='not.fully.paid',
                                hue='credit.policy',palette='Set1')
```

OUTPUT:

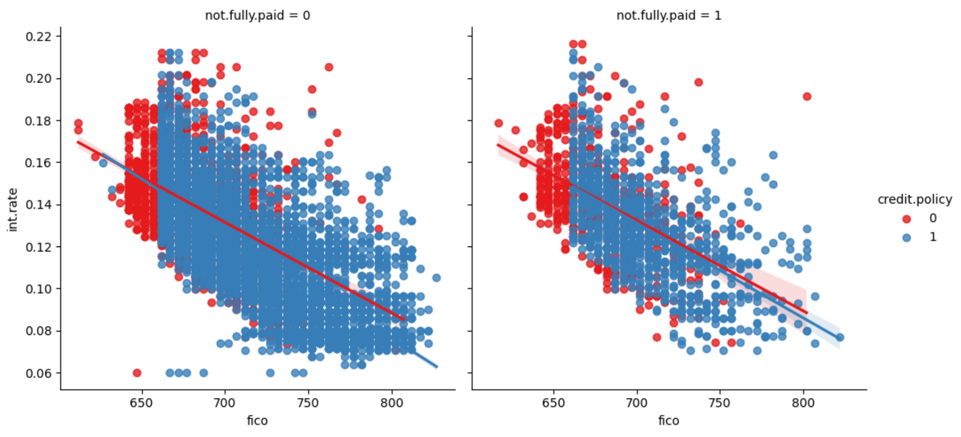


Figure 4.1.9 Fico Score and Interest Rate with Credit Policy distribution

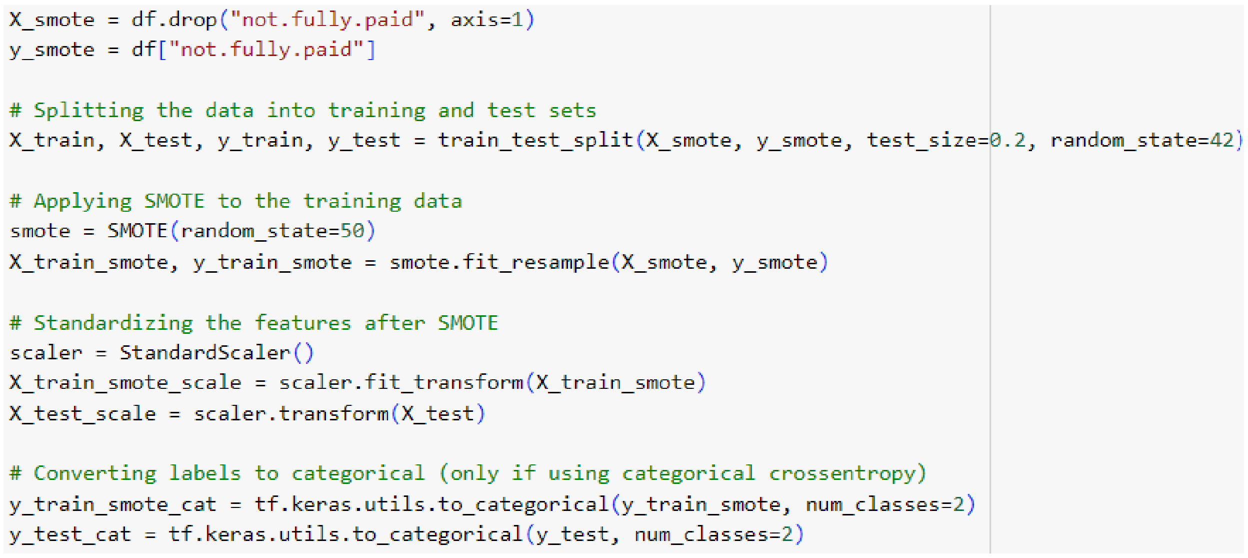
This lmplot shows important information about how loans work. As FICO scores go up, interest rates usually go down, no matter if the loan is paid back or not. Loans that are not fully paid back (right side) have more scattered points and higher interest rates overall. The credit policy (red for not fully paid, blue for fully paid) seems stricter for people with lower FICO scores, especially for fully paid loans. There are many approved loans (blue) at higher FICO scores, which suggests that lenders are careful about who they give loans to.

4.2 MODEL DEVELOPMENT AND TRAINING

4.2.1 Data Splitting and Applying SMOTE

This phase performs several preprocessing steps to address class imbalance and prepare data for machine learning or deep learning models. First, researcher divides the dataset into training and testing sets, with the target being whether a loan is fully paid or not. To handle class imbalance, which can impact model performance, the code uses SMOTE to create synthetic samples for the minority class in the training set, thus balancing the dataset. After applying SMOTE, the code standardizes the features using StandardScaler, ensuring they have a mean of 0 and a standard deviation of 1. This helps improve model performance, especially for models sensitive to feature scales, such as neural networks. Finally, the labels are converted to a categorical format to work with the categorical cross-entropy loss function.

CODE:



```
X_smote = df.drop("not.fully.paid", axis=1)
y_smote = df["not.fully.paid"]

# Splitting the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_smote, y_smote, test_size=0.2, random_state=42)

# Applying SMOTE to the training data
smote = SMOTE(random_state=50)
X_train_smote, y_train_smote = smote.fit_resample(X_smote, y_smote)

# Standardizing the features after SMOTE
scaler = StandardScaler()
X_train_smote_scale = scaler.fit_transform(X_train_smote)
X_test_scale = scaler.transform(X_test)

# Converting labels to categorical (only if using categorical crossentropy)
y_train_smote_cat = tf.keras.utils.to_categorical(y_train_smote, num_classes=2)
y_test_cat = tf.keras.utils.to_categorical(y_test, num_classes=2)
```

The plot shows how the target variable is spread out after using SMOTE. This helps us see that the class imbalance has been fixed, so both classes now have the same number of examples for training.

CODE:

```
#Plotting target variable after applying SMOTE
class_counts = y_train_smote.value_counts()

plt.bar([0, 1], class_counts.values, color=['red', 'blue'], edgecolor='black')
plt.xlabel('Class')
plt.ylabel('Count')
plt.title('Distribution of Target Variable in y_train_smote')
plt.xticks([0, 1])
plt.show()
```

OUTPUT:

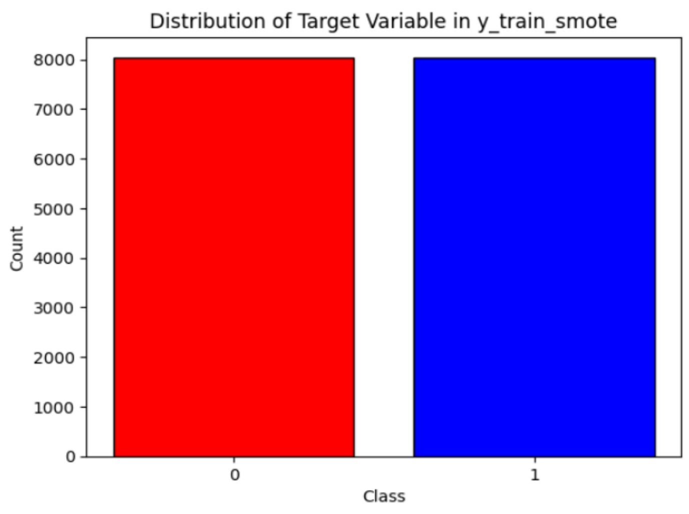


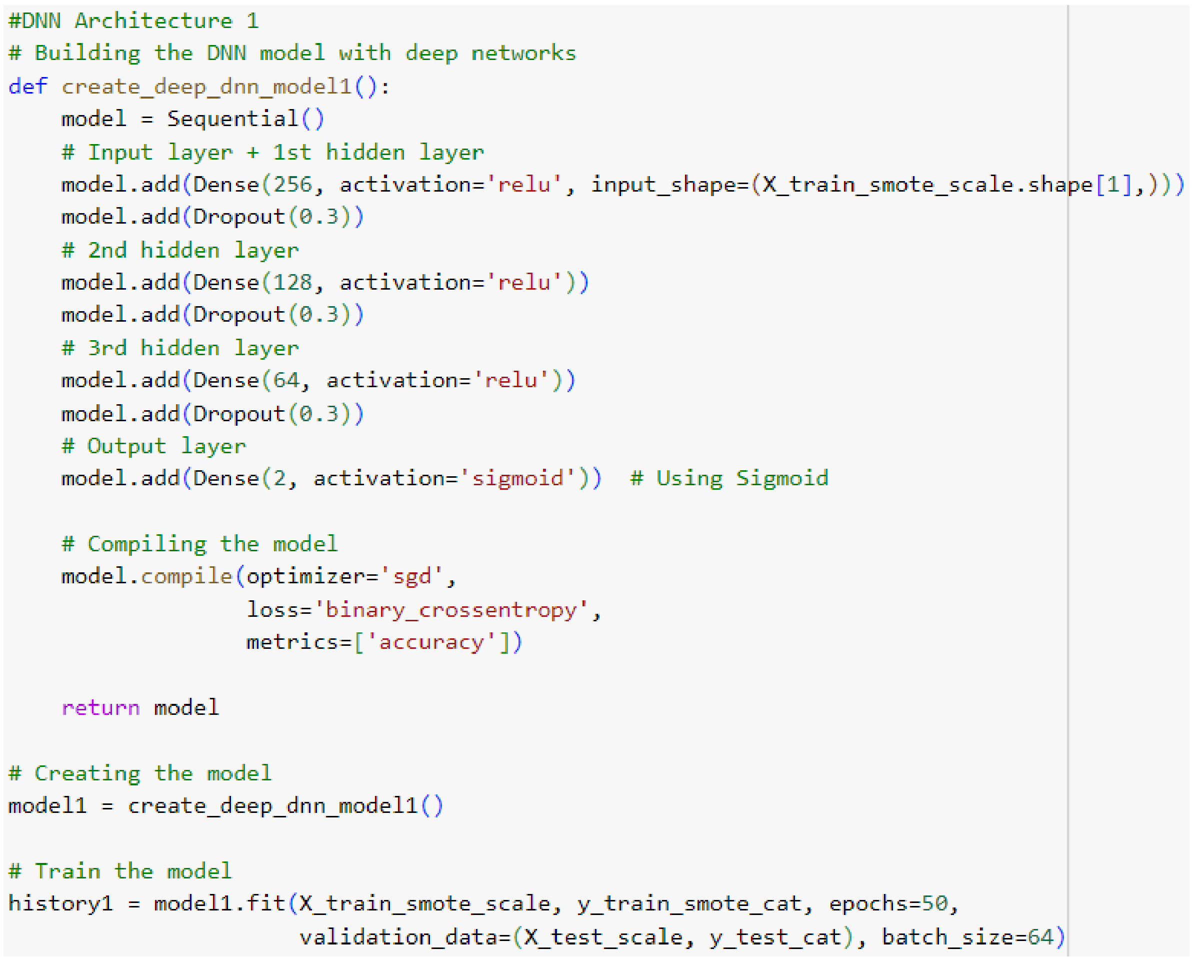
Figure 4.2.1 Distribution of Target Variable after applying SMOTE

4.2.2 DNN Model Architecture 1

The architecture includes several layers that are densely connected, using the ReLU activation function for the middle layers. ReLU is good at handling non-linear changes and stops gradients from vanishing gradient problem. The first layer, which is the input layer, connects to a middle layer with 256 units. After that, there are two more middle layers with 128 and 64 units, respectively. Each of these layers is followed by a Dropout layer that randomly turns off 30% of the units during training to prevent the model from overfitting. The last layer has 2 units with a sigmoid activation function, which is used for binary classification, meaning the model gives probabilities for each of the two possible outcomes.

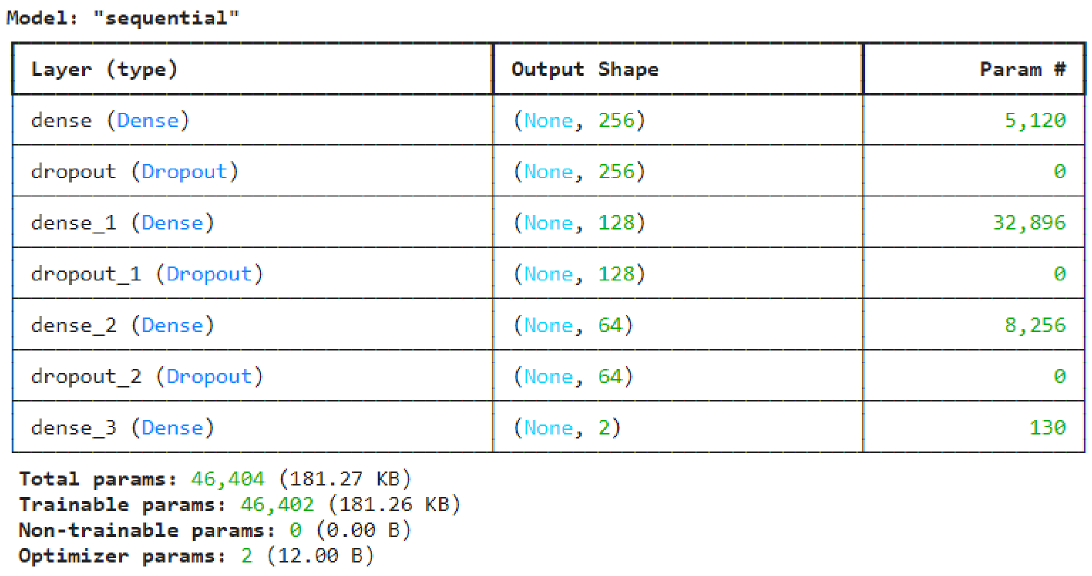
The model is set up using the stochastic gradient descent (SGD) optimizer, which adjusts the weights based on the direction of the loss function. This architecture uses the binary cross-entropy loss function, which works well for problems with two possible outcomes, and measure how well the model is doing using accuracy. The model is trained for 50 epochs, with a batch size of 64. Validation set is used to check how well the model is doing on new data while it's being trained.

CODE:



```python
#DNN Architecture 1
# Building the DNN model with deep networks
def create_deep_dnn_model1():
    model = Sequential()
    # Input layer + 1st hidden layer
    model.add(Dense(256, activation='relu', input_shape=(X_train_smote_scale.shape[1],)))
    model.add(Dropout(0.3))
    # 2nd hidden layer
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.3))
    # 3rd hidden layer
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.3))
    # Output layer
    model.add(Dense(2, activation='sigmoid'))  # Using Sigmoid

    # Compiling the model
    model.compile(optimizer='sgd',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

    return model

# Creating the model
model1 = create_deep_dnn_model1()

# Train the model
history1 = model1.fit(X_train_smote_scale, y_train_smote_cat, epochs=50,
                      validation_data=(X_test_scale, y_test_cat), batch_size=64)
```

OUPUT:



```
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━┓
┃ Layer (type)                        ┃ Output Shape               ┃        Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━┩
│ dense (Dense)                       │ (None, 256)                │          5,120 │
├─────────────────────────────────────┼────────────────────────────┼────────────────┤
│ dropout (Dropout)                   │ (None, 256)                │              0 │
├─────────────────────────────────────┼────────────────────────────┼────────────────┤
│ dense_1 (Dense)                     │ (None, 128)                │         32,896 │
├─────────────────────────────────────┼────────────────────────────┼────────────────┤
│ dropout_1 (Dropout)                 │ (None, 128)                │              0 │
├─────────────────────────────────────┼────────────────────────────┼────────────────┤
│ dense_2 (Dense)                     │ (None, 64)                 │          8,256 │
├─────────────────────────────────────┼────────────────────────────┼────────────────┤
│ dropout_2 (Dropout)                 │ (None, 64)                 │              0 │
├─────────────────────────────────────┼────────────────────────────┼────────────────┤
│ dense_3 (Dense)                     │ (None, 2)                  │            130 │
└─────────────────────────────────────┴────────────────────────────┴────────────────┘
 Total params: 46,404 (181.27 KB)
 Trainable params: 46,402 (181.26 KB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 2 (12.00 B)
```

Figure 4.2.2 Model summary of DNN architecture 1

Above is the model summary for architecture 1 where the model has a total of 46,404 trainable parameters, optimizing for binary classification.

4.2.2.1 Plotting Training and Validation Accuracy Value

Plots are used to track the training and validation accuracy and loss over time to assess how well the model is learning. These graphs help see if the model is overfitting or underfitting by checking if the accuracy keeps getting better and the loss keeps going down on both the training and validation data as the training continues.

CODE:

```
# Plot training & validation accuracy values

plt.plot(history1.history['accuracy'])
plt.plot(history1.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.plot(history1.history['loss'])
plt.plot(history1.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

OUTPUT:

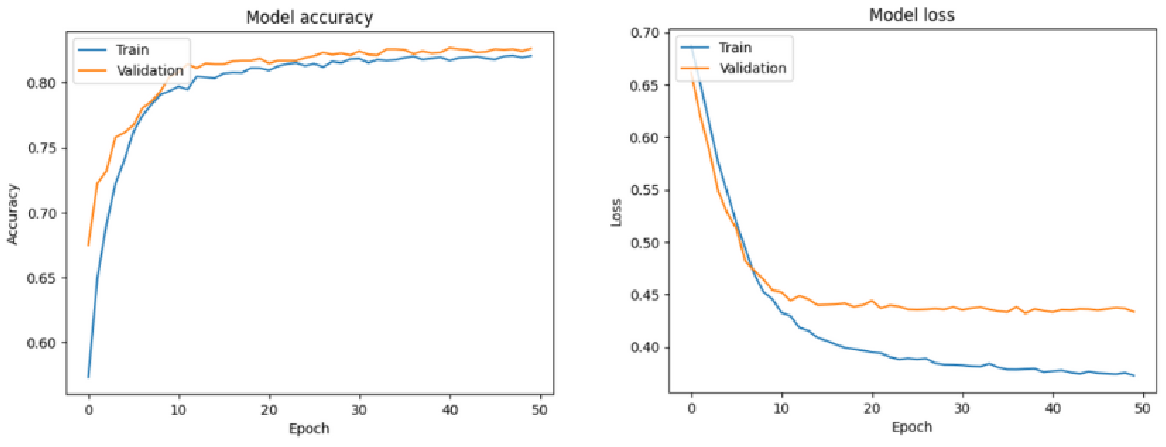


Figure 4.2.3 Training and Validation plot of DNN architecture 1

The plots display the model's accuracy and loss over 50 epochs for both training and validation sets. The accuracy plot shows an increase for both sets, with validation accuracy stabilizing around 80%, indicating good generalization. The loss plot demonstrates a consistent decrease in training loss, while validation loss stabilizes after a few epochs, suggesting that the model is learning effectively without significant overfitting.

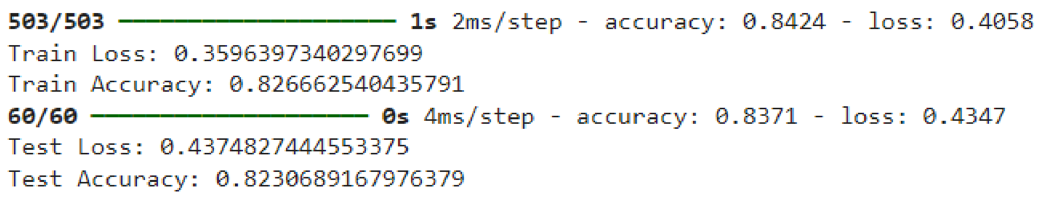
4.2.2.2 Evaluating the Training and Test Set

Here researcher evaluates the deep learning model's performance on both the training and test datasets. It measures the loss and accuracy for each dataset, which helps to understand how well the model has learned from the training data and how well it can apply this learning to new, unseen data in the test set.

CODE:

```
# Evaluating the model on the training set
train_loss, train_accuracy = model1.evaluate(X_train_smote_scale, y_train_smote_cat)
print(f"Train Loss: {train_loss}")
print(f"Train Accuracy: {train_accuracy}")

# Evaluating the model on the test set
test_loss, test_accuracy = model1.evaluate(X_test_scale, y_test_cat)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

OUPUT:



```
503/503 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8424 - loss: 0.4058
Train Loss: 0.3596397340297699
Train Accuracy: 0.826662540435791
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8371 - loss: 0.4347
Test Loss: 0.4374827444553375
Test Accuracy: 0.8230689167976379
```

Figure 4.2.4 Evaluation of Training and Test set of DNN architecture 1

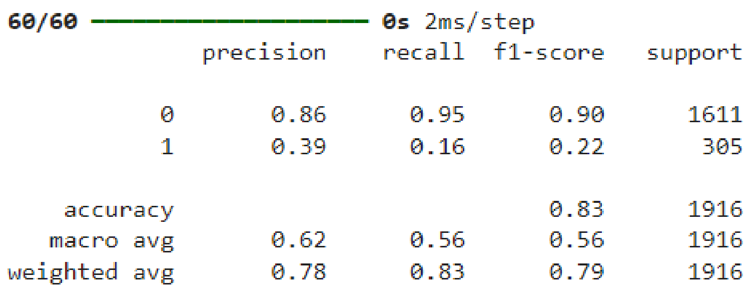
The evaluation shows that the model works well on both the training and test data. The training accuracy is 82.6%, and the loss is 0.36. The test accuracy is 82.2%, with a slightly higher loss of 0.44. This indicates that the model generalizes well, without any major overfitting.

A classification report is created to give a thorough assessment of how well the model is doing. It looks at the predicted categories (y_pred_classes) and compares them to the real test categories (y_test_classes). The report shows important measures such as precision, recall, F1-score, and support for each category. This gives a clear picture of how well the model is at classifying both common and less common categories.

CODE:

```
#Creating Classification report of the model
from sklearn.metrics import classification_report

y_pred = model1.predict(X_test_scale)
y_pred_classes = y_pred.argmax(axis=1)
y_test_classes = y_test_cat.argmax(axis=1)

print(classification_report(y_test_classes, y_pred_classes))
```

OUPUT:



```
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
              precision    recall  f1-score   support

           0       0.86      0.95      0.90      1611
           1       0.39      0.16      0.22       305

    accuracy                           0.83      1916
   macro avg       0.62      0.56      0.56      1916
weighted avg       0.78      0.83      0.79      1916
```

Figure 4.2.5 Classification report of DNN architecture 1

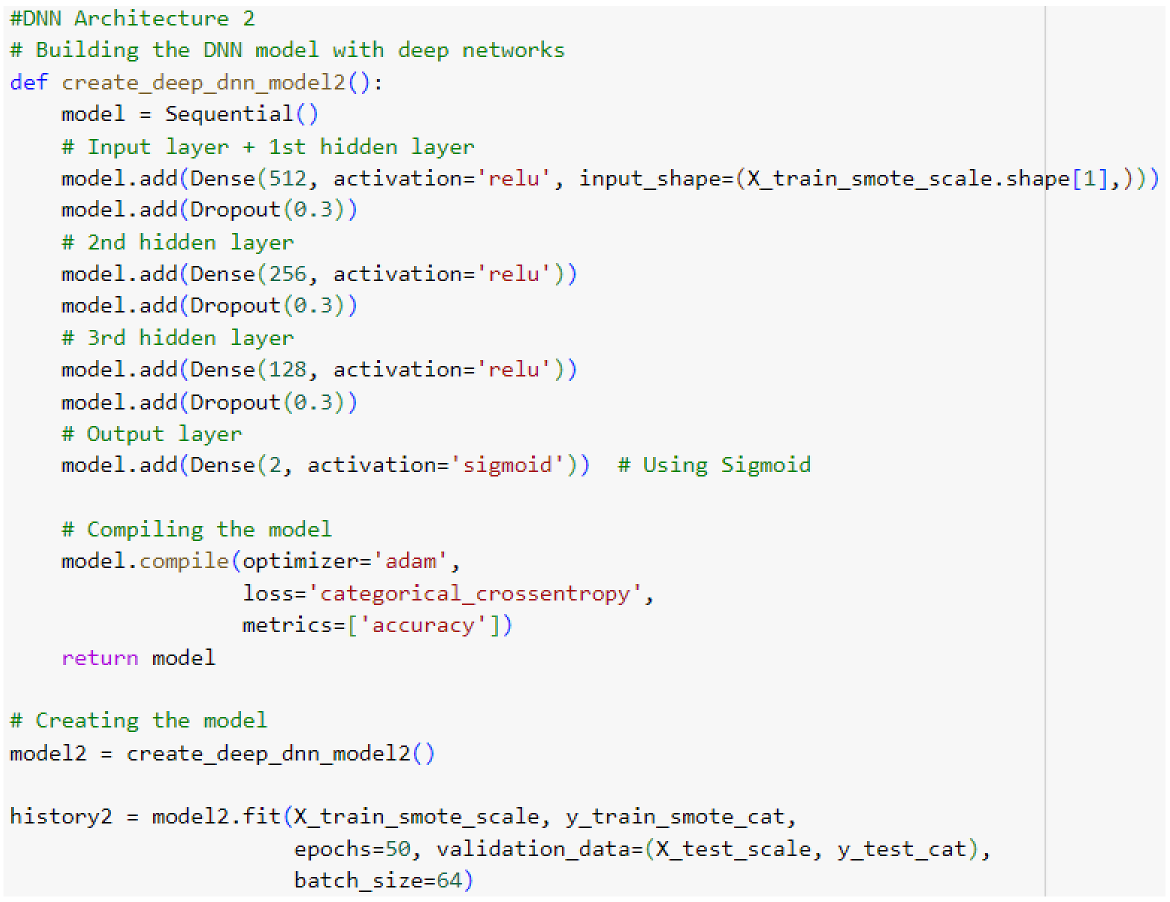
The classification report shows that the model is correct 83% of the time overall. For class 0 (the bigger group), the model does very well, with high scores for precision (0.86), recall (0.95), and F1-score (0.90). But for class 1 (the smaller group), the model struggles, with much lower scores for precision (0.39), recall (0.16), and F1-score (0.22). The macro average shows the model's performance without considering the size of the groups, which is lower due to the imbalance. The weighted average, which does consider the size of the groups, shows better overall performance.

4.2.3 DNN Model Architecture 2

The second deep neural network (DNN) design is more intricate and has more layers than the first. It begins with an input layer linked to the first hidden layer, which has 512 neurons. A Dropout layer is then added, which randomly turns off 30% of the neurons during training to avoid overfitting. The second hidden layer has 256 neurons, and the third has 128 neurons, both using the ReLU activation function for effective non-linear changes. After each hidden layer, Dropout layers are used for better control. The final layer has 2 neurons with a sigmoid activation function, which is ideal for binary classification tasks.

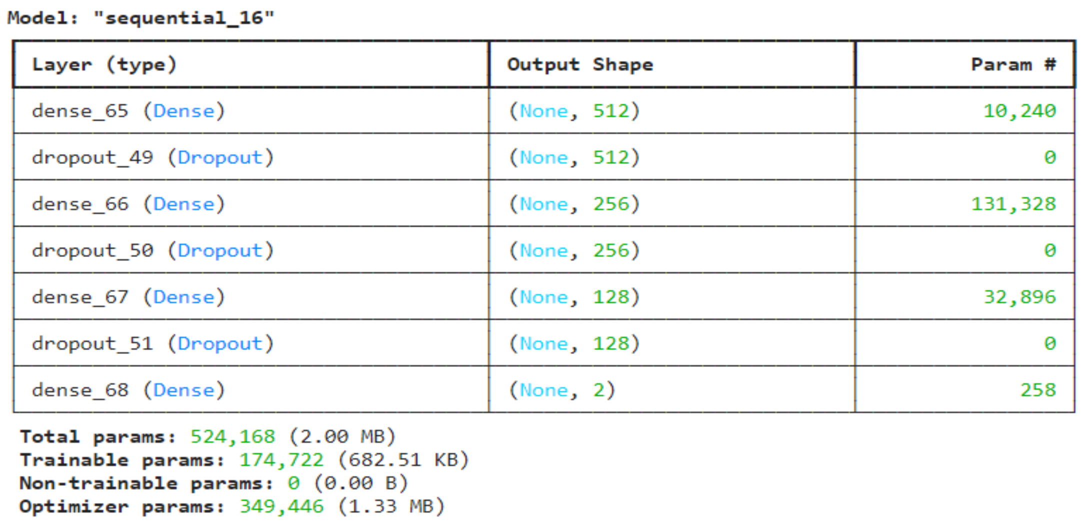
This model is built with the Adam optimizer, which changes learning rates dynamically to speed up the process, and it uses the categorical cross-entropy loss function because of categories for labels. This design should help find more complex patterns because it has more neurons in the middle layers, which makes it better at predicting loan defaults. Same as before the model is trained for 50 epochs, with a batch size of 64. Validation set is used to check how well the model is doing on new data while it's being trained.

CODE:



```
#DNN Architecture 2
# Building the DNN model with deep networks
def create_deep_dnn_model2():
    model = Sequential()
    # Input layer + 1st hidden layer
    model.add(Dense(512, activation='relu', input_shape=(X_train_smote_scale.shape[1],)))
    model.add(Dropout(0.3))
    # 2nd hidden layer
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.3))
    # 3rd hidden layer
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.3))
    # Output layer
    model.add(Dense(2, activation='sigmoid'))  # Using Sigmoid

    # Compiling the model
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# Creating the model
model2 = create_deep_dnn_model2()

history2 = model2.fit(X_train_smote_scale, y_train_smote_cat,
                      epochs=50, validation_data=(X_test_scale, y_test_cat),
                      batch_size=64)
```

OUPUT:



```
Model: "sequential_16"
┌─────────────────────────────────┬──────────────────────────┬───────────────┐
│ Layer (type)                    │ Output Shape             │       Param # │
├─────────────────────────────────┼──────────────────────────┼───────────────┤
│ dense_65 (Dense)                │ (None, 512)              │        10,240 │
├─────────────────────────────────┼──────────────────────────┼───────────────┤
│ dropout_49 (Dropout)            │ (None, 512)              │             0 │
├─────────────────────────────────┼──────────────────────────┼───────────────┤
│ dense_66 (Dense)                │ (None, 256)              │       131,328 │
├─────────────────────────────────┼──────────────────────────┼───────────────┤
│ dropout_50 (Dropout)            │ (None, 256)              │             0 │
├─────────────────────────────────┼──────────────────────────┼───────────────┤
│ dense_67 (Dense)                │ (None, 128)              │        32,896 │
├─────────────────────────────────┼──────────────────────────┼───────────────┤
│ dropout_51 (Dropout)            │ (None, 128)              │             0 │
├─────────────────────────────────┼──────────────────────────┼───────────────┤
│ dense_68 (Dense)                │ (None, 2)                │           258 │
└─────────────────────────────────┴──────────────────────────┴───────────────┘
 Total params: 524,168 (2.00 MB)
 Trainable params: 174,722 (682.51 KB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 349,446 (1.33 MB)
```

Figure 4.2.6 Model summary of DNN architecture 2

Above is the model summary of architecture 2 the model has a total of 524,168 parameters, with 174,722 trainable parameters, meaning these will be adjusted during training. The optimizer manages 349,446 parameters, optimizing the learning process for better performance.

4.2.3.1 Plotting Training and Validation Accuracy Value

Below plot for 2nd architecture illustrates how the model performed over 50 epochs. The plot on the left shows that both the accuracy during training and the accuracy during testing keep getting better. By the end, the accuracy during validation is a bit higher than during training, around 91%. This means the model is learning well and not overfitting too much. The graph on the right shows the model's error rate, which keeps decreasing for both training and testing, meaning the model is making fewer mistakes as it trains. The lower error rate during testing compared to training shows that the model can work well on new data.

OUTPUT:

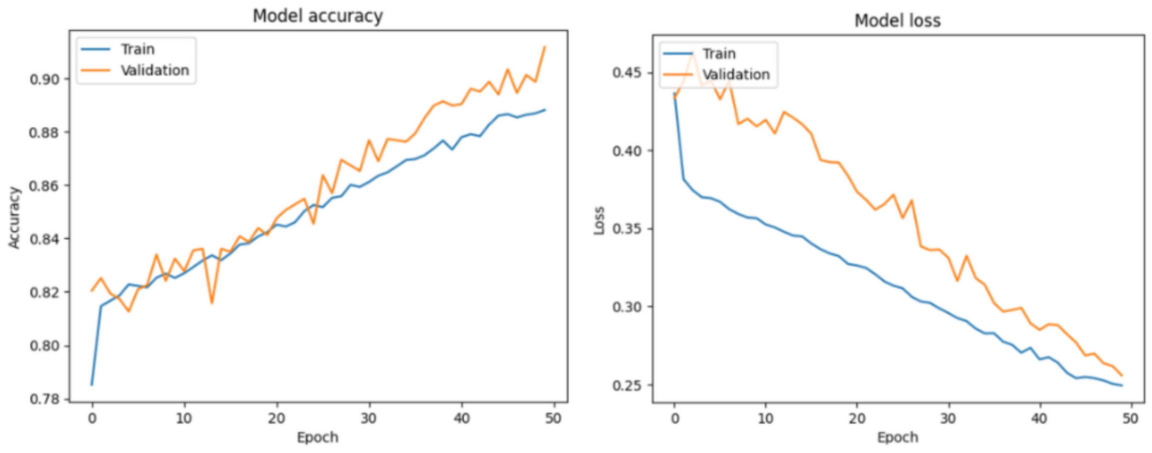
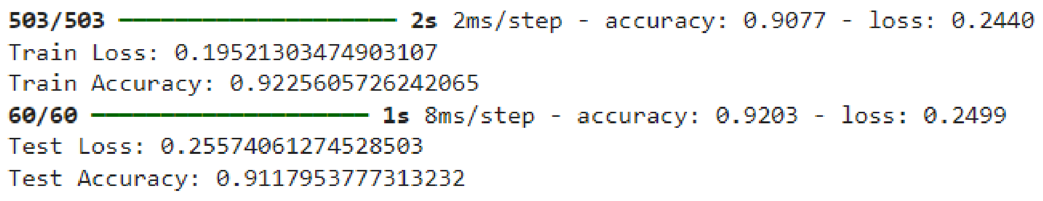


Figure 4.2.7 Training and Validation plot of DNN architecture 2

4.2.3.2 Evaluating Training and Test set

While evaluating the results of 2nd architecture training result demonstrates that the deep learning model performs very well. The accuracy on the training data is about 92.3%, with a small error of 0.1952, showing that the model fits the training data nicely. The accuracy on the test data is also high, at 91.2%, with an error of 0.2557, meaning the model works well on new, unseen data. The similar accuracy and low error on both datasets suggest that the model is not overfitting and has strong predictive ability for both types of data.

OUTPUT:

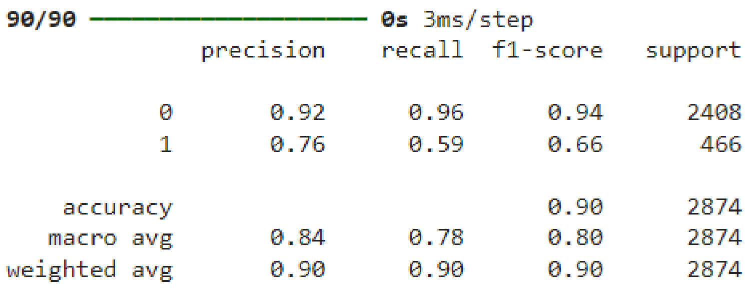


```
503/503 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9077 - loss: 0.2440
Train Loss: 0.19521303474903107
Train Accuracy: 0.9225605726242065
60/60 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.9203 - loss: 0.2499
Test Loss: 0.25574061274528503
Test Accuracy: 0.9117953777313232
```

Figure 4.2.8 Evaluation of Training and Test set of DNN architecture 2

The classification report of 2nd architecture evaluates the model's performance on two categories: 0 (loans fully paid) and 1 (loans not fully paid). For category 0, the model scored a precision of 0.92, recall of 0.96, and an F1-score of 0.94, showing high accuracy in predicting fully paid loans. For category 1, the model's performance is weaker, with a precision of 0.76, recall of 0.59, and an F1-score of 0.66, indicating difficulty in predicting defaulted loans. The overall accuracy is 90%, and the weighted average F1-score is 0.90, showing good overall performance but also suggesting that the model could be improved in detecting defaults.

OUTPUT:



```
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
              precision    recall  f1-score   support

           0       0.92      0.96      0.94      2408
           1       0.76      0.59      0.66       466

    accuracy                           0.90      2874
   macro avg       0.84      0.78      0.80      2874
weighted avg       0.90      0.90      0.90      2874
```

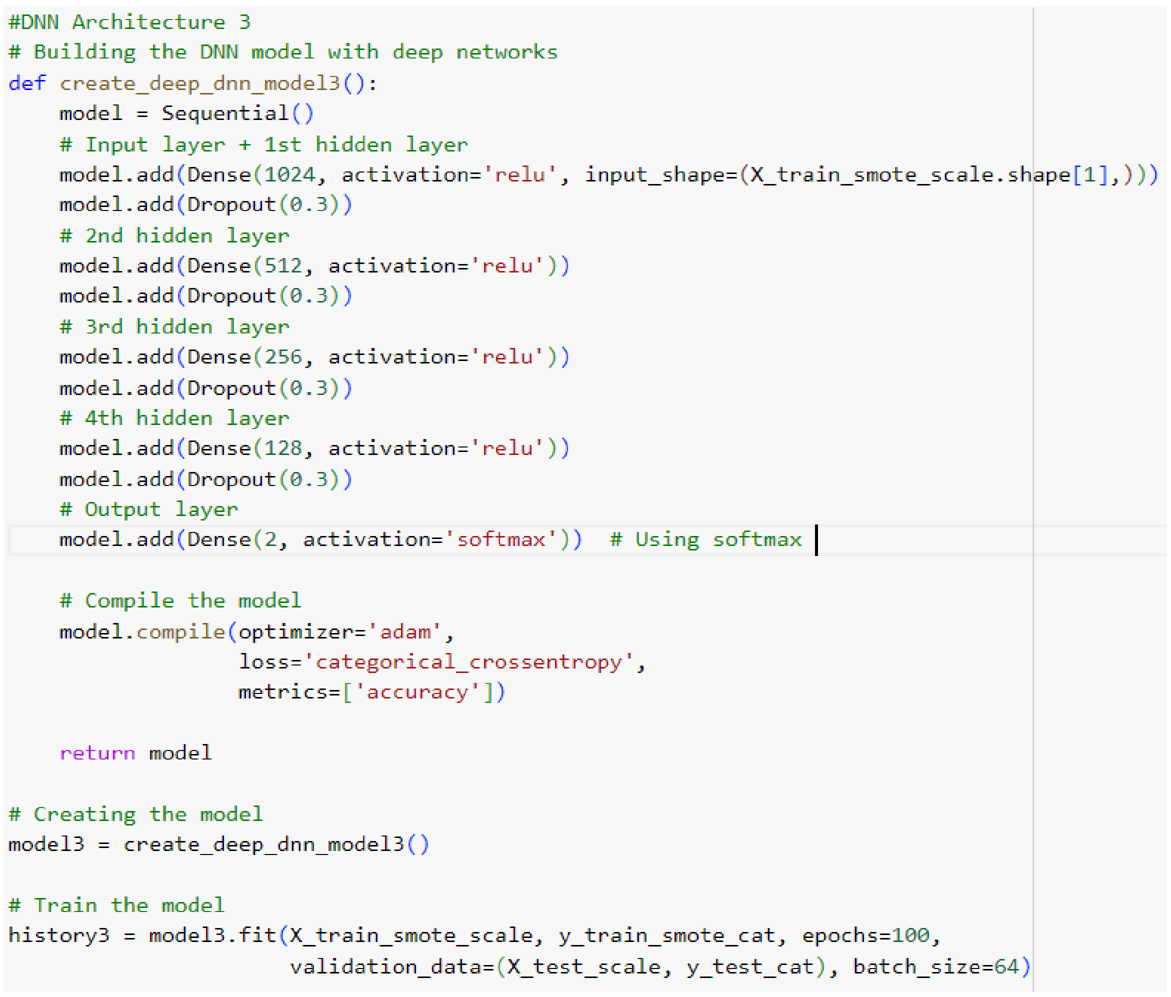
Figure 4.2.9 Classification report of DNN architecture 2

4.2.4 DNN Model Architecture 3

The third DNN architecture is made for binary classification that includes four hidden layers with different numbers of neurons. The input layer connects to the first hidden layer, which has 1024 neurons and uses the ReLU activation function. After that, a dropout layer with a rate of 0.3 is added to help prevent overfitting. The second hidden layer has 512 neurons, also using ReLU and a 0.3 dropout. This pattern continues in the third and fourth hidden layers, which have 256 and 128 neurons, respectively, and include dropout to avoid overfitting. The final output layer of the model has two neurons activated by a SoftMax function, which is suitable for binary classification. The SoftMax function gives probabilities for the two classes.

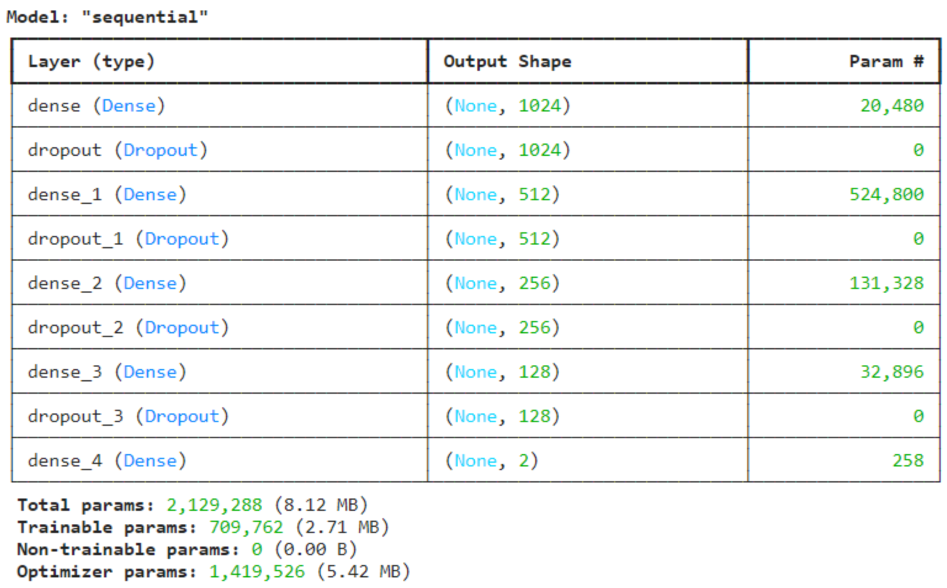
The model is compiled using the Adam optimizer, known for its adaptive learning rate, and categorical cross entropy as the loss function, which is commonly used in multi-class classification, but also works well for binary classification tasks where SoftMax is used. This architecture aims to strike a balance between model complexity and generalization through the layers and dropout to enhance performance. The model is trained for 100 epochs using a batch size of 64, and its performance is validated using the test dataset.

CODE:



```python
#DNN Architecture 3
# Building the DNN model with deep networks
def create_deep_dnn_model3():
    model = Sequential()
    # Input layer + 1st hidden layer
    model.add(Dense(1024, activation='relu', input_shape=(X_train_smote_scale.shape[1],)))
    model.add(Dropout(0.3))
    # 2nd hidden layer
    model.add(Dense(512, activation='relu'))
    model.add(Dropout(0.3))
    # 3rd hidden layer
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.3))
    # 4th hidden layer
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.3))
    # Output layer
    model.add(Dense(2, activation='softmax'))  # Using softmax

    # Compile the model
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    return model

# Creating the model
model3 = create_deep_dnn_model3()

# Train the model
history3 = model3.fit(X_train_smote_scale, y_train_smote_cat, epochs=100,
                      validation_data=(X_test_scale, y_test_cat), batch_size=64)
```

OUTPUT:



```
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                   ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ dense (Dense)                  │ (None, 1024)           │        20,480 │
├────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)              │ (None, 1024)           │             0 │
├────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                │ (None, 512)            │       524,800 │
├────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_1 (Dropout)            │ (None, 512)            │             0 │
├────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                │ (None, 256)            │       131,328 │
├────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_2 (Dropout)            │ (None, 256)            │             0 │
├────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_3 (Dense)                │ (None, 128)            │        32,896 │
├────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_3 (Dropout)            │ (None, 128)            │             0 │
├────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_4 (Dense)                │ (None, 2)              │           258 │
└────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 2,129,288 (8.12 MB)
 Trainable params: 709,762 (2.71 MB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 1,419,526 (5.42 MB)
```

Figure 4.2.10 Model summary of DNN architecture 3

Above is the model summary of architecture 3 the model has a total of 2,129,288 parameters, with 709,762 trainable parameters, meaning these will be adjusted during training.

The optimizer manages 1,419,526 parameters, optimizing the learning process for better performance than before.

4.2.4.1 Plotting the Training and Validation Accuracy Value

The below plot for 3rd architecture shows how the model performed over 100 epoch training and validation. In the accuracy plot, both training and validation accuracy increase steadily. Validation accuracy starts higher than training accuracy and levels off around 97.5%, showing good ability to handle new data. In the loss plot, both training and validation losses decrease over the rounds. Validation loss drops faster and ends lower than training loss, suggesting the model works well with unseen data and isn't too focused on the training data. Overall, the model learns well and performs strongly on both training and validation sets.

OUTPUT

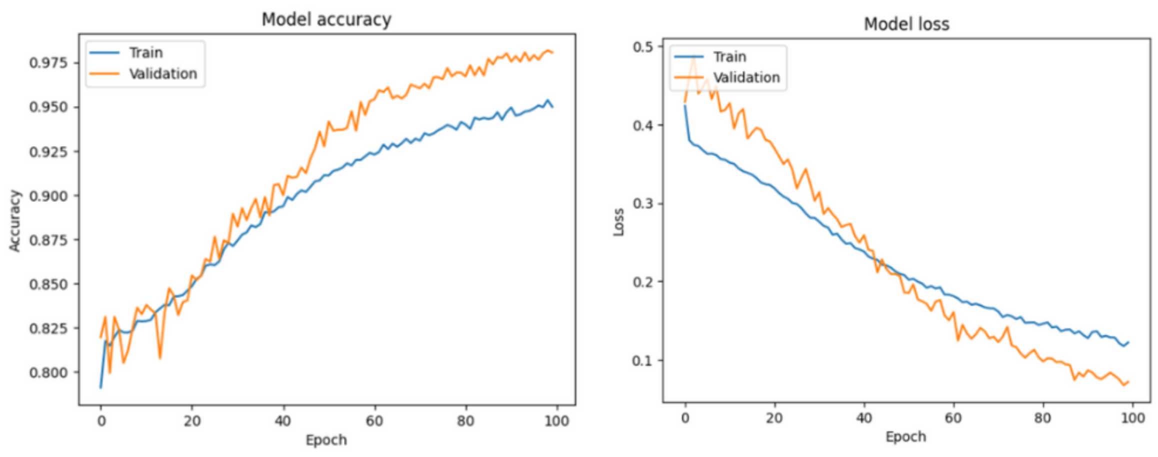
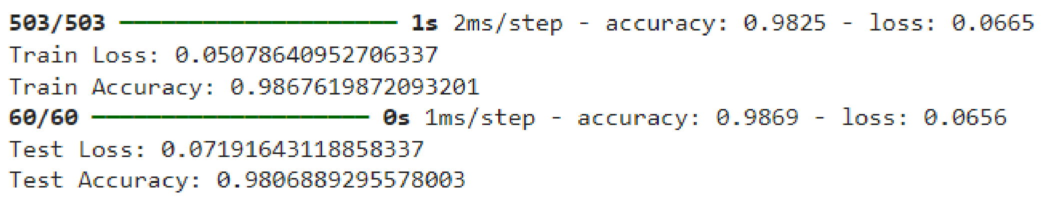


Figure 4.2.11 Training and Validation plot of DNN architecture 3

4.2.4.2 Evaluating the Training and Test Set

While evaluating the 3rd architecture it can be observed that the training results show strong model performance with a high training accuracy of 98.68% and a low training loss of 0.0508, indicating the model fits the training data well. The test results also indicate excellent generalization, with a test accuracy of 98.07% and a slightly higher test loss of 0.0719. Overall, the model performs well on both training and test sets, showing minimal overfitting and strong predictive capabilities.

OUTPUT:

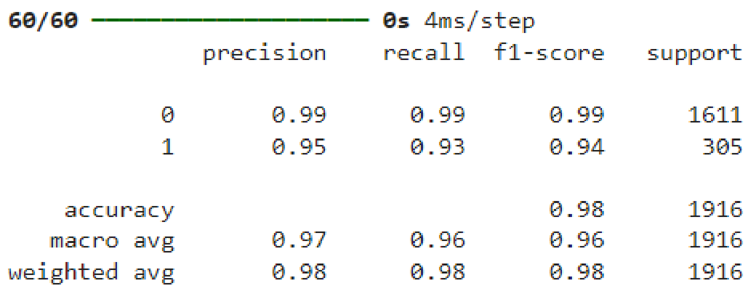


```
503/503 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9825 - loss: 0.0665
Train Loss: 0.05078640952706337
Train Accuracy: 0.9867619872093201
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9869 - loss: 0.0656
Test Loss: 0.07191643118858337
Test Accuracy: 0.9806889295578003
```

Figure 4.2.12 Evaluation of Training and Test set of DNN architecture 3

The classification report demonstrates excellent model performance, with an overall accuracy of 98%. Class 0 has near-perfect precision, recall, and F1-score of 0.99, indicating the model effectively distinguishes the majority class. Class 1 also performs very well, with precision of 0.95, recall of 0.93, and an F1-score of 0.94, showing strong detection of the minority class. The macro average (0.96-0.97) reflects balanced performance across both classes, while the weighted average (0.98) accounts for class distribution, further reinforcing the model's robustness and generalizability.

OUTPUT:



```
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1611
           1       0.95      0.93      0.94       305

    accuracy                           0.98      1916
   macro avg       0.97      0.96      0.96      1916
weighted avg       0.98      0.98      0.98      1916
```

Figure 4.2.13 Classification report of DNN architecture 3

4.2.5 Comparison between different DNN Architectures

This comparison examines three DNN architectures of varying complexity, designed for binary classification. Each model employs different layer configurations, activation functions, and optimization techniques to achieve increasingly accurate predictions in the domain of loan default risk assessment.

Table 4.2.1 Comparison of DNN architectures

Architecture	Optimizer	Epochs	Test Acc.	Precision	Recall	F1-Score
DNN 1	SGD	50	83%	0.78	0.83	0.79
DNN 2	Adam	50	90%	0.90	0.90	0.90
DNN 3	Adam	100	98%	0.98	0.98	0.98

Among the three, Architecture 3 stands out as the best performer. It demonstrates excellent generalization with minimal overfitting, achieving high accuracy on both training (98.68%) and test (98.07%) sets. The classification report shows near-perfect scores for the majority class and very strong performance for the minority class, with an overall F1-score of 0.98. This architecture's superior performance can be attributed to its deeper and wider structure, allowing it to capture more complex patterns in the data, as well as the extended training period of 100 epochs.

4.3 TUNING THE DNN MODEL

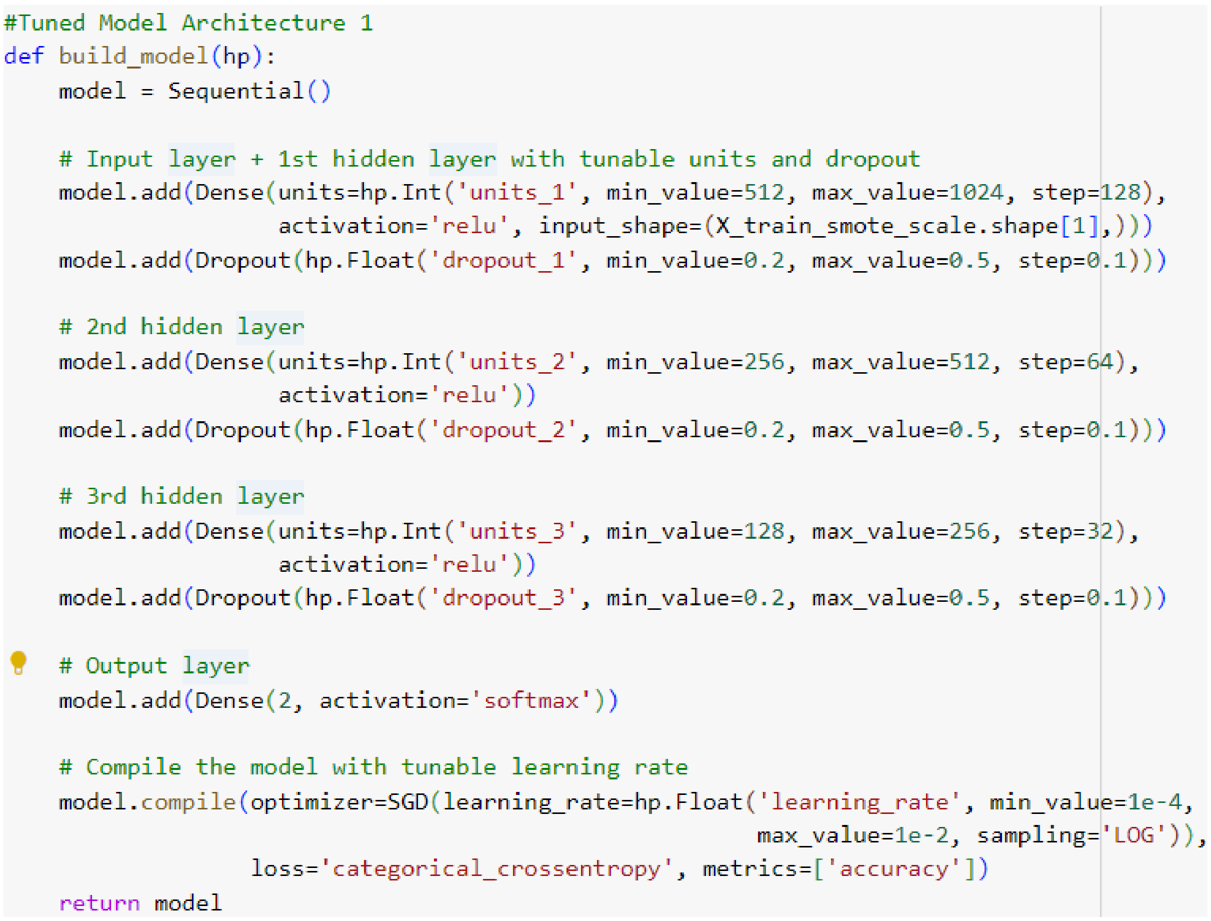
Tuning a deep learning model is a crucial step that improves its performance and ability to generalize. This involves tweaking different settings and parts of the model to find the best setup for a particular task. This step is important because deep learning models are complex with many interconnected parts, and their performance can change a lot based on these settings. Tuning helps avoid issues like overfitting or underfitting, makes learning more efficient, and boosts the model's ability to work well with new data. It may involve changing things like learning rates, batch sizes, the depth and width of the network, activation functions, and methods to prevent overfitting. By carefully tuning the model, researcher can get the most out of it, making sure it accurately captures patterns in the data while staying efficient and avoiding problems like gradients becoming too small or too large.

4.3.1 Tuned DNN Model Architecture 1

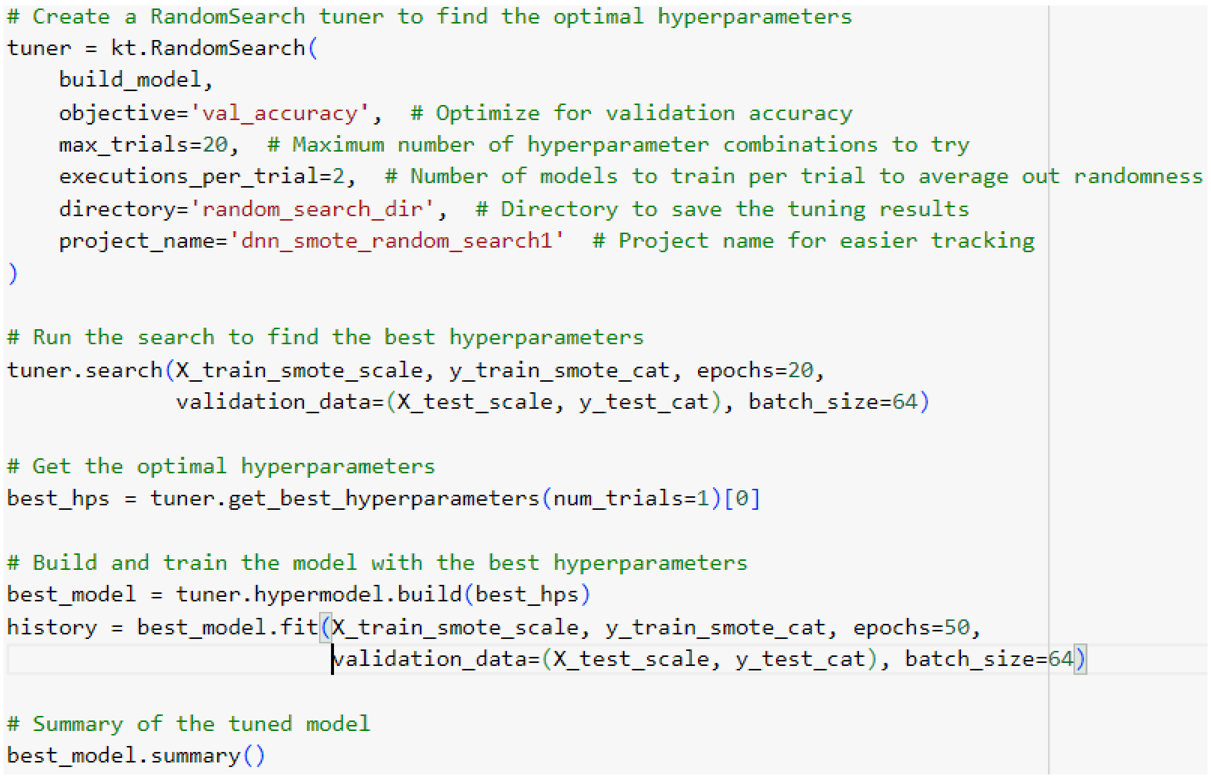
The Tuned DNN architecture 1 is tuned using the Keras Tuner with random search to find the best settings for the model's parameters, which helps improve validation accuracy. The model has three hidden layers, each with adjustable settings like the number of units, dropout rates, and learning rate. In the first hidden layer, the number of units is between 512 and 1024, and the dropout rate is between 0.2 and 0.5. The second hidden layer has units ranging from 256 to 512, with a similar adjustable dropout rate. The third hidden layer has units between 128 and 256, with the same flexibility for dropout. All layers use the ReLU activation function. The model's last layer has two neurons with SoftMax activation for classification.

The model is compiled using Stochastic Gradient Descent (SGD) as the optimizer, with a tuneable learning rate ranging between $1e^{-4}$ and $1e^{-2}$. The search process tries 20 different combinations of hyperparameters, and the best performing model is selected based on validation accuracy. The model is then trained for 50 epochs using the optimal hyperparameters, ensuring efficient training and generalization.

CODE:

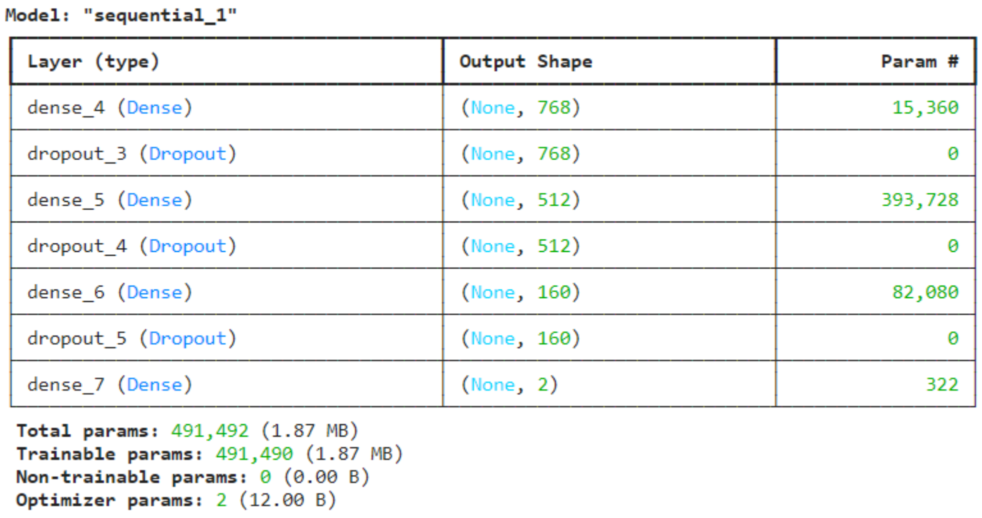


```python
#Tuned Model Architecture 1
def build_model(hp):
    model = Sequential()

    # Input layer + 1st hidden layer with tunable units and dropout
    model.add(Dense(units=hp.Int('units_1', min_value=512, max_value=1024, step=128),
                    activation='relu', input_shape=(X_train_smote_scale.shape[1],)))
    model.add(Dropout(hp.Float('dropout_1', min_value=0.2, max_value=0.5, step=0.1)))

    # 2nd hidden layer
    model.add(Dense(units=hp.Int('units_2', min_value=256, max_value=512, step=64),
                    activation='relu'))
    model.add(Dropout(hp.Float('dropout_2', min_value=0.2, max_value=0.5, step=0.1)))

    # 3rd hidden layer
    model.add(Dense(units=hp.Int('units_3', min_value=128, max_value=256, step=32),
                    activation='relu'))
    model.add(Dropout(hp.Float('dropout_3', min_value=0.2, max_value=0.5, step=0.1)))

    # Output layer
    model.add(Dense(2, activation='softmax'))

    # Compile the model with tunable learning rate
    model.compile(optimizer=SGD(learning_rate=hp.Float('learning_rate', min_value=1e-4,
                                                        max_value=1e-2, sampling='LOG')),
                  loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```



```python
# Create a RandomSearch tuner to find the optimal hyperparameters
tuner = kt.RandomSearch(
    build_model,
    objective='val_accuracy',  # Optimize for validation accuracy
    max_trials=20,  # Maximum number of hyperparameter combinations to try
    executions_per_trial=2,  # Number of models to train per trial to average out randomness
    directory='random_search_dir',  # Directory to save the tuning results
    project_name='dnn_smote_random_search1'  # Project name for easier tracking
)

# Run the search to find the best hyperparameters
tuner.search(X_train_smote_scale, y_train_smote_cat, epochs=20,
             validation_data=(X_test_scale, y_test_cat), batch_size=64)

# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build and train the model with the best hyperparameters
best_model = tuner.hypermodel.build(best_hps)
history = best_model.fit(X_train_smote_scale, y_train_smote_cat, epochs=50,
                         validation_data=(X_test_scale, y_test_cat), batch_size=64)

# Summary of the tuned model
best_model.summary()
```

OUTPUT:



```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 768)	15,360
dropout_3 (Dropout)	(None, 768)	0
dense_5 (Dense)	(None, 512)	393,728
dropout_4 (Dropout)	(None, 512)	0
dense_6 (Dense)	(None, 160)	82,080
dropout_5 (Dropout)	(None, 160)	0
dense_7 (Dense)	(None, 2)	322

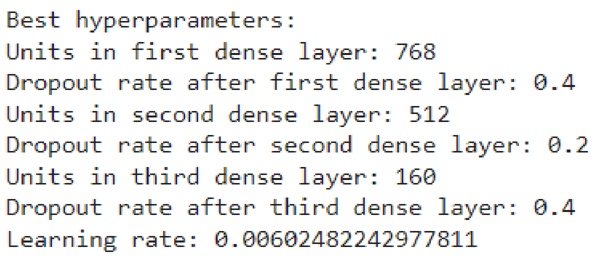
```
Total params: 491,492 (1.87 MB)
Trainable params: 491,490 (1.87 MB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 2 (12.00 B)
```

Figure 4.3.1 Model Summary of Tuned DNN architecture 1

Above is the model summary of tuned architecture 1 the model has a total of 491,492 parameters, with 491,490 trainable parameters, meaning these will be adjusted during training. The optimizer manages 2 parameters, optimizing the learning process for better performance.

The optimal hyperparameters for the tuned DNN architecture 1 model are as follows: 768 units in the first dense layer with a 0.2 dropout rate, 512 units in the second layer with a 0.2 dropout, 128 units in the third layer with a 0.3 dropout, and 96 units in the fourth layer with a 0.3 dropout. The optimal learning rate is 0.00267.

OUTPUT:



```
Best hyperparameters:
Units in first dense layer: 768
Dropout rate after first dense layer: 0.4
Units in second dense layer: 512
Dropout rate after second dense layer: 0.2
Units in third dense layer: 160
Dropout rate after third dense layer: 0.4
Learning rate: 0.006024822429778 11
```

Figure 4.3.2 Best hyperparameter of Tuned DNN architecture 1

4.3.1.1 Plotting the Training and Validation Accuracy Value

The plot displays the training and validation accuracy and loss for a Tuned DNN architecture 1 over 50 epochs. In the accuracy graph, both training and validation accuracy increase rapidly in the first 10 epochs, levelling off at around 80-82%. The validation accuracy shows minor variations but remains close to the training accuracy, indicating no overfitting. In the loss graph, both training and validation losses drop quickly at first, then stabilize at about 0.4 for validation loss and 0.35 for training loss. This shows that the model is learning effectively, with little performance drop during validation.

OUPUT:

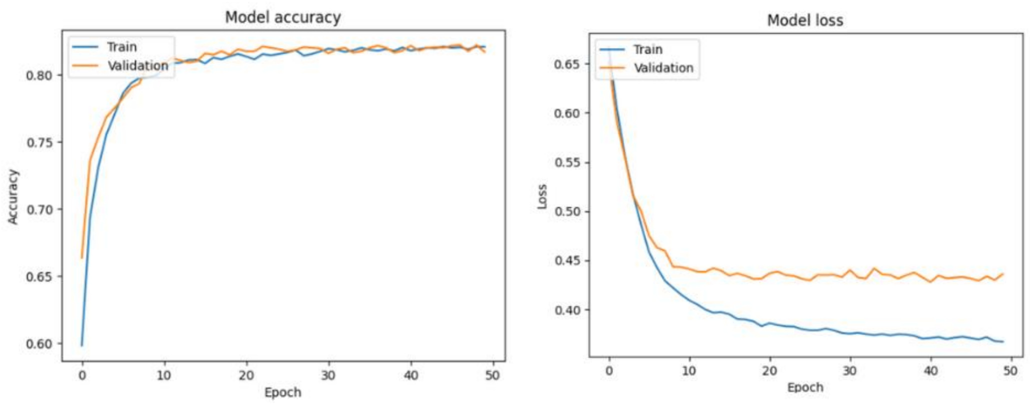
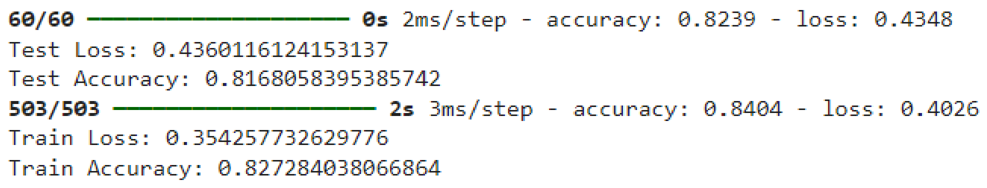


Figure 4.3.3 Training and Validation plot of Tuned DNN architecture 1

4.3.1.2 Evaluating the Training and Test Set

While evaluating the 1st tuned architecture it can be observed that the training results show weaker model performance then the DNN architecture 3 with a training accuracy of 82.72% and a low training loss of 0.354, indicating the model fits the training data decently. The test results also indicate a decent generalization, with a test accuracy of 81.68% and a slightly higher test loss of 0.436. Overall, the model performs well on both train and test set.

OUTPUT:

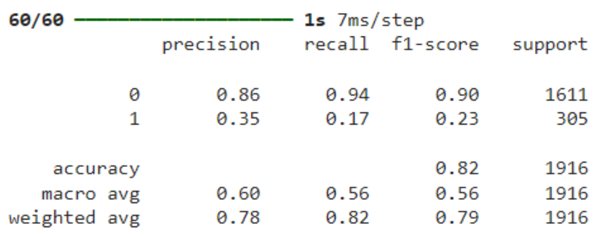


```
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8239 - loss: 0.4348
Test Loss: 0.4360116124153137
Test Accuracy: 0.8168058395385742
503/503 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.8404 - loss: 0.4026
Train Loss: 0.354257732629776
Train Accuracy: 0.827284038066864
```

Figure 4.3.4 Evaluation of Training and Test set of Tuned DNN architecture 1

The classification report shows that the model performs very well for class "0" (non-default), with a precision of 0.86, recall of 0.94, and an F1-score of 0.90. This means the model is very good at recognizing non-default cases. But for class "1" (default), the model doesn't do as well. It has a precision of 0.35, recall of 0.17, and an F1-score of 0.23, which shows that the model has trouble correctly identifying default cases. The overall accuracy is 82%, but the average metrics (precision, recall, and F1-score around 0.56) show that the model is not balanced in dealing with both classes.

OUPUT:



```
60/60 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step
              precision    recall  f1-score   support

           0       0.86      0.94      0.90      1611
           1       0.35      0.17      0.23       305

    accuracy                           0.82      1916
   macro avg       0.60      0.56      0.56      1916
weighted avg       0.78      0.82      0.79      1916
```

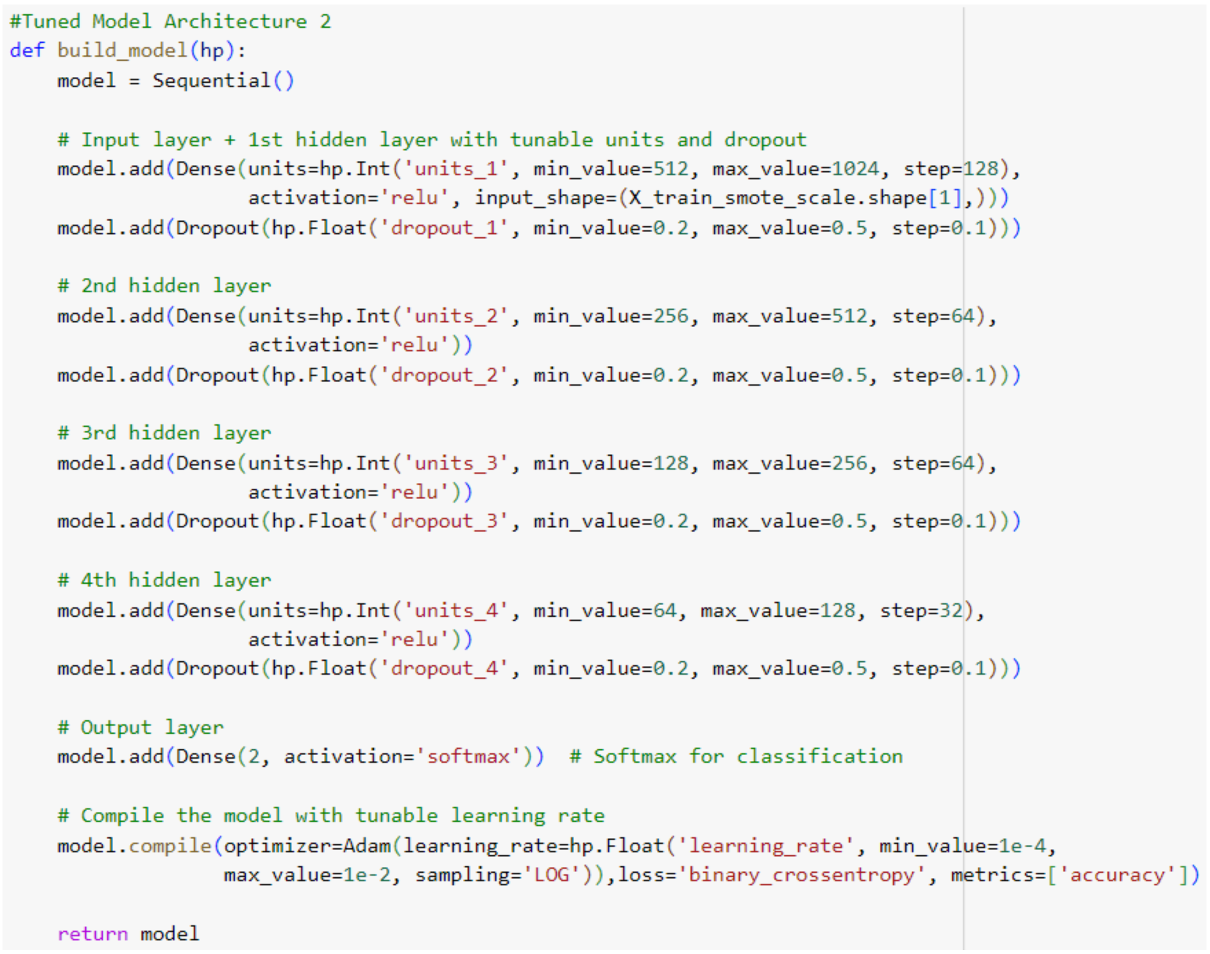
Figure 4.3.5 Classification report of Tuned DNN architecture 1

4.3.2 Tuned DNN Model Architecture 2

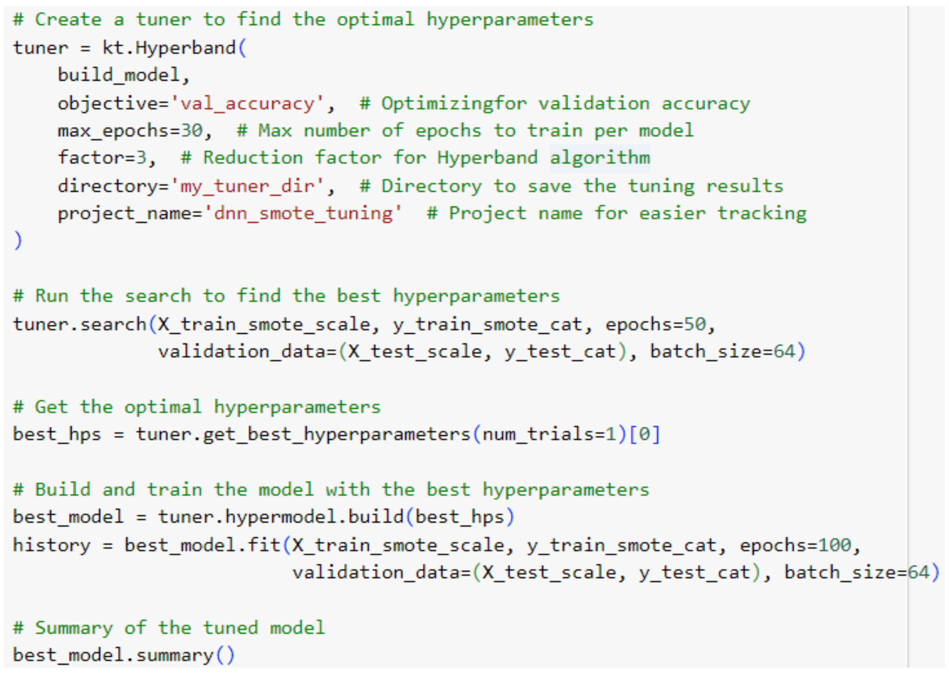
The 2nd DNN architecture includes adjustable settings for fine-tuning with Keras Tuner's Hyperband algorithm. The model begins with an input layer and then has four hidden layers, each with adjustable numbers of units and dropout rates. The number of neurons in each layer is set by the search range for each layer (units_1, units_2, units_3, units_4), with the first hidden layer having between 512 and 1024 neurons and the number decreasing in each subsequent layer. Dropout layers are added after each dense layer to avoid overfitting, with the dropout rate also adjustable between 0.2 and 0.5.

The final layer has two neurons with SoftMax activation for binary classification. The model is set up with a customizable learning rate using the Adam optimizer and binary cross-entropy loss. Hyperband, a smart algorithm for finding the best settings, is used to look for the best combination of settings based on how well the model does on validation data. The tuner.search() function runs this search through many tries and training cycles. After finding the best settings, the model is trained again for 100 epochs with these optimal settings. This design balances flexibility and thorough optimization to improve the model's performance.

CODE:

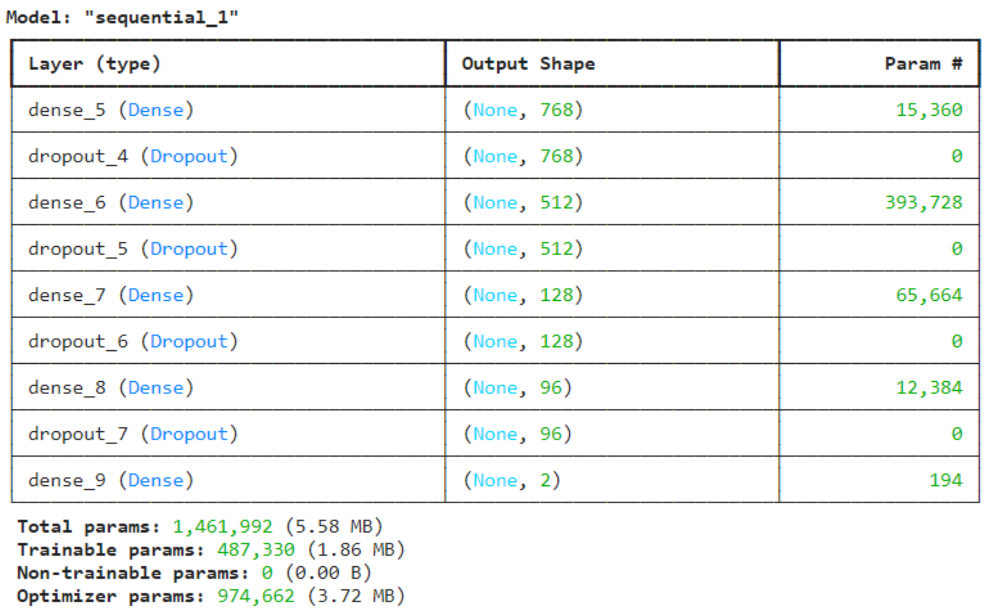


```python
#Tuned Model Architecture 2
def build_model(hp):
    model = Sequential()

    # Input layer + 1st hidden layer with tunable units and dropout
    model.add(Dense(units=hp.Int('units_1', min_value=512, max_value=1024, step=128),
                    activation='relu', input_shape=(X_train_smote_scale.shape[1],)))
    model.add(Dropout(hp.Float('dropout_1', min_value=0.2, max_value=0.5, step=0.1)))

    # 2nd hidden layer
    model.add(Dense(units=hp.Int('units_2', min_value=256, max_value=512, step=64),
                    activation='relu'))
    model.add(Dropout(hp.Float('dropout_2', min_value=0.2, max_value=0.5, step=0.1)))

    # 3rd hidden layer
    model.add(Dense(units=hp.Int('units_3', min_value=128, max_value=256, step=64),
                    activation='relu'))
    model.add(Dropout(hp.Float('dropout_3', min_value=0.2, max_value=0.5, step=0.1)))

    # 4th hidden layer
    model.add(Dense(units=hp.Int('units_4', min_value=64, max_value=128, step=32),
                    activation='relu'))
    model.add(Dropout(hp.Float('dropout_4', min_value=0.2, max_value=0.5, step=0.1)))

    # Output layer
    model.add(Dense(2, activation='softmax'))  # Softmax for classification

    # Compile the model with tunable learning rate
    model.compile(optimizer=Adam(learning_rate=hp.Float('learning_rate', min_value=1e-4,
                  max_value=1e-2, sampling='LOG')),loss='binary_crossentropy', metrics=['accuracy'])

    return model
```



```
# Create a tuner to find the optimal hyperparameters
tuner = kt.Hyperband(
    build_model,
    objective='val_accuracy',  # Optimizingfor validation accuracy
    max_epochs=30,  # Max number of epochs to train per model
    factor=3,  # Reduction factor for Hyperband algorithm
    directory='my_tuner_dir',  # Directory to save the tuning results
    project_name='dnn_smote_tuning'  # Project name for easier tracking
)

# Run the search to find the best hyperparameters
tuner.search(X_train_smote_scale, y_train_smote_cat, epochs=50,
             validation_data=(X_test_scale, y_test_cat), batch_size=64)

# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build and train the model with the best hyperparameters
best_model = tuner.hypermodel.build(best_hps)
history = best_model.fit(X_train_smote_scale, y_train_smote_cat, epochs=100,
                         validation_data=(X_test_scale, y_test_cat), batch_size=64)

# Summary of the tuned model
best_model.summary()
```

OUTPUT:



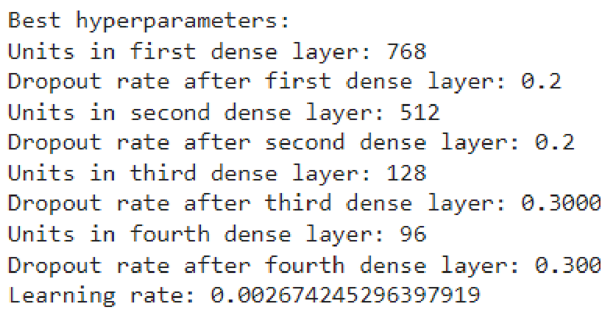
```
Model: "sequential_1"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ dense_5 (Dense)                      │ (None, 768)                 │          15,360 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dropout_4 (Dropout)                  │ (None, 768)                 │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_6 (Dense)                      │ (None, 512)                 │         393,728 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dropout_5 (Dropout)                  │ (None, 512)                 │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_7 (Dense)                      │ (None, 128)                 │          65,664 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dropout_6 (Dropout)                  │ (None, 128)                 │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_8 (Dense)                      │ (None, 96)                  │          12,384 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dropout_7 (Dropout)                  │ (None, 96)                  │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_9 (Dense)                      │ (None, 2)                   │             194 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
 Total params: 1,461,992 (5.58 MB)
 Trainable params: 487,330 (1.86 MB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 974,662 (3.72 MB)
```

Figure 4.3.6 Model summary of Tuned DNN architecture 2

Above is the model summary of tuned architecture 2 the model has a total of 1,461,992 parameters, with 487,330 trainable parameters, meaning these will be adjusted during training. The optimizer manages 974,662 parameters, optimizing the learning process for better performance than before.

The optimal hyperparameters for the tuned DNN architecture 2 model are as follows: 768 units in the first dense layer with a 0.2 dropout rate, 512 units in the second layer with a 0.2 dropout, 128 units in the third layer with a 0.3 dropout, and 96 units in the fourth layer with a 0.3 dropout. The optimal learning rate is 0.00267.

OUTPUT:



```
Best hyperparameters:
Units in first dense layer: 768
Dropout rate after first dense layer: 0.2
Units in second dense layer: 512
Dropout rate after second dense layer: 0.2
Units in third dense layer: 128
Dropout rate after third dense layer: 0.3000
Units in fourth dense layer: 96
Dropout rate after fourth dense layer: 0.300
Learning rate: 0.002674245296397919
```

Figure 4.3.7 Best hyperparameters for Tuned DNN architecture 2

4.3.2.1 Plotting the Training and Validation Accuracy Value

The plot below shows the training and validation accuracy and loss for a tuned DNN model 2 over 100 epochs. In the accuracy plot, both training and validation accuracy increase steadily, with validation accuracy consistently higher than training accuracy, reaching about 97.5%. In the loss plot, both training and validation loss decrease gradually, with validation loss decreasing more quickly than training loss. By the end of training, validation loss is around 0.1, and training loss levels off at about 0.15. These patterns indicate that the model is performing well without major overfitting, balancing training and validation effectively.

OUPUT:

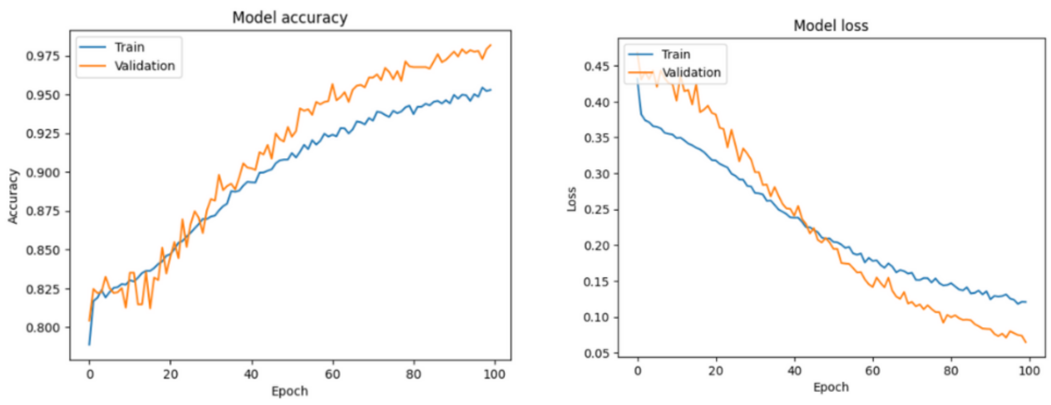
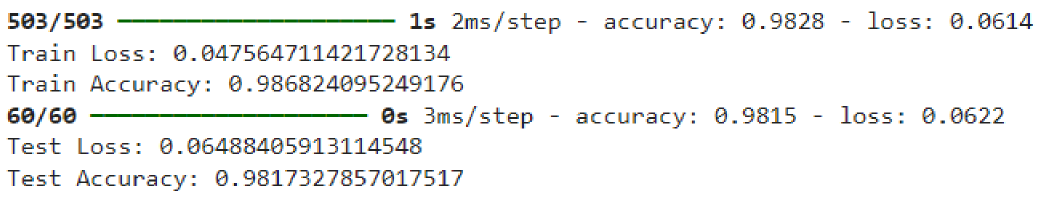


Figure 4.3.8 Training and Validation plot of Tuned DNN architecture 2

4.3.2.2 Evaluating Training and Test set

While evaluating the 2nd tuned architecture it can be observed that the training results show strong model performance with a high training accuracy of 98.68% and a low training loss of 0.047, indicating the model fits the training data well. The test results also indicate excellent generalization, with a test accuracy of 98.17% and a slightly higher test loss of 0.0648. Overall, the model performs well on both training and test sets, showing minimal overfitting and strong predictive capabilities.

OUTPUT:

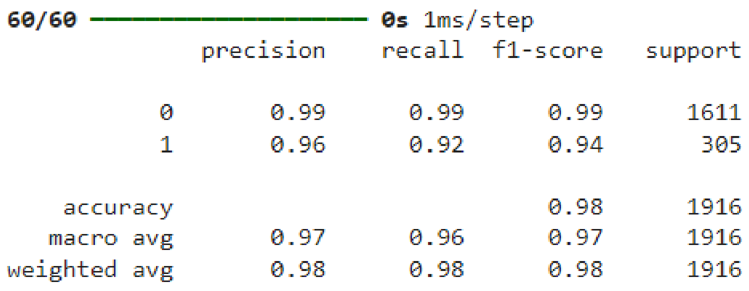


```
503/503 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9828 - loss: 0.0614
Train Loss: 0.047564711421728134
Train Accuracy: 0.986824095249176
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9815 - loss: 0.0622
Test Loss: 0.06488405913114548
Test Accuracy: 0.9817327857017517
```

Figure 4.3.9 Evaluation of Training and Test set of Tuned DNN architecture 2

The classification report shows that the model performs very well, with an overall accuracy of 98%. For class 0, the precision, recall, and F1-score are all 0.99, indicating excellent identification of this class. For class 1, the precision is 0.96, recall is 0.92, and the F1-score is 0.94, which is slightly lower but still strong. The macro average (0.97 for precision, 0.96 for recall, and 0.97 for F1) shows a good balance between the two classes, and the weighted average (0.98 across all metrics) confirms that the model is consistently effective across the entire dataset.

OUPUT:



```
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1611
           1       0.96      0.92      0.94       305

    accuracy                           0.98      1916
   macro avg       0.97      0.96      0.97      1916
weighted avg       0.98      0.98      0.98      1916
```

Figure 4.3.10 Classification report of Tuned DNN architecture 2

4.3.3 Tuned DNN Model Architecture 3

The Tuned DNN architecture 3 uses four hidden layers, each with different numbers of units and dropout rates, which are optimized using a RandomSearch method. The architecture starts with an input layer, followed by the first hidden layer. In this first layer, the number of neurons can be adjusted between 512 and 1024, and the dropout rate can be set between 0.2 and 0.5 to avoid overfitting. The second hidden layer has between 256 and 512 units, with a similar adjustable dropout rate. The third and fourth hidden layers gradually decrease the number of units, from 128–256 in the third layer to 64–128 in the fourth, each with adjustable dropout rates. This gradual reduction in neurons helps the model focus on important features as the data moves through the network. The final output layer uses SoftMax activation to classify the input into one of two categories (default or non-default).

The model is compiled using the Adam optimizer with a learning rate tuneable between 0.0001 and 0.01, and it employs binary cross-entropy loss. RandomSearch optimizes hyperparameters like units, dropout rates, and learning rate, maximizing validation accuracy.

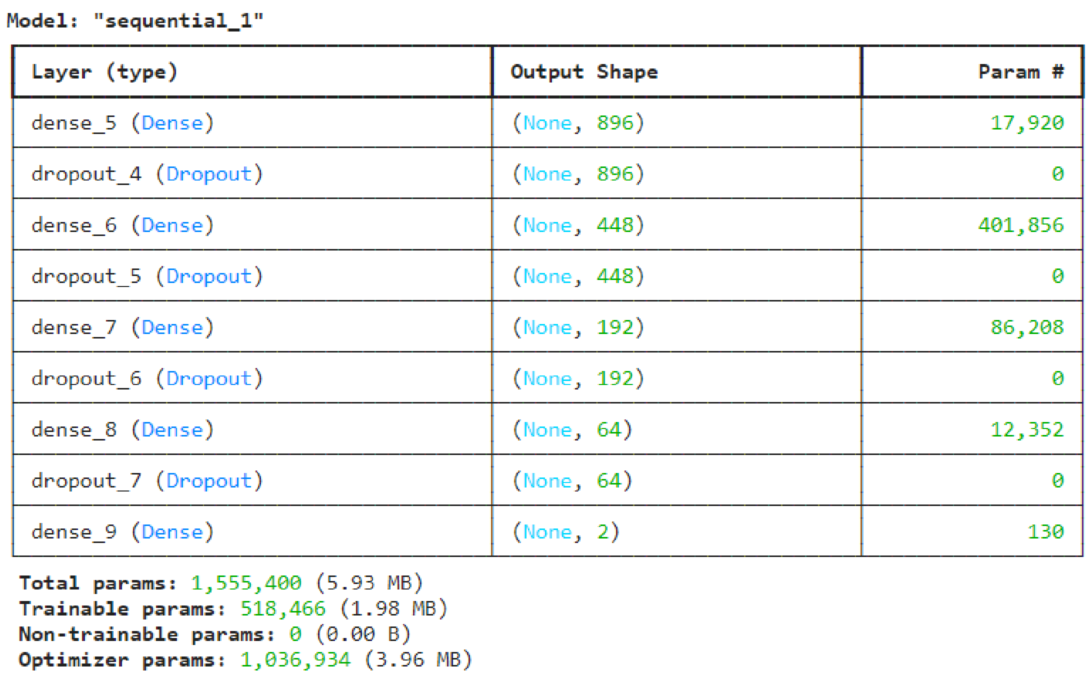
After finding the best settings, the model is trained again for 100 epochs with these optimal settings. This design balances flexibility and thorough optimization to improve the model's performance.

CODE:

```python
#Tuned Model Architecture 3
def build_model(hp):
    model = Sequential()

    # Input layer + 1st hidden layer with tunable units and dropout
    model.add(Dense(units=hp.Int('units_1', min_value=512, max_value=1024, step=128),
                    activation='relu', input_shape=(X_train_smote_scale.shape[1],)))
    model.add(Dropout(hp.Float('dropout_1', min_value=0.2, max_value=0.5, step=0.1)))

    # 2nd hidden layer
    model.add(Dense(units=hp.Int('units_2', min_value=256, max_value=512, step=64),
                    activation='relu'))
    model.add(Dropout(hp.Float('dropout_2', min_value=0.2, max_value=0.5, step=0.1)))

    # 3rd hidden layer
    model.add(Dense(units=hp.Int('units_3', min_value=128, max_value=256, step=64),
                    activation='relu'))
    model.add(Dropout(hp.Float('dropout_3', min_value=0.2, max_value=0.5, step=0.1)))

    # 4th hidden layer
    model.add(Dense(units=hp.Int('units_4', min_value=64, max_value=128, step=32),
                    activation='relu'))
    model.add(Dropout(hp.Float('dropout_4', min_value=0.2, max_value=0.5, step=0.1)))

    # Output layer
    model.add(Dense(2, activation='softmax'))

    # Compile the model with tunable learning rate
    model.compile(optimizer=Adam(learning_rate=hp.Float('learning_rate', min_value=1e-4,
                                                        max_value=1e-2, sampling='LOG')),
                  loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

```python
# Create a RandomSearch tuner to find the optimal hyperparameters
tuner = kt.RandomSearch(
    build_model,
    objective='val_accuracy',  # Optimize for validation accuracy
    max_trials=20,  # Maximum number of hyperparameter combinations to try
    executions_per_trial=2,  # Number of models to train per trial to average out randomness
    directory='random_search_dir',  # Directory to save the tuning results
    project_name='dnn_smote_random_search'  # Project name for easier tracking
)

# Runing the search to find the best hyperparameters
tuner.search(X_train_smote_scale, y_train_smote_cat, epochs=50,
             validation_data=(X_test_scale, y_test_cat), batch_size=64)
# Getting the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Building and train the model with the best hyperparameters
best_model = tuner.hypermodel.build(best_hps)
history = best_model.fit(X_train_smote_scale, y_train_smote_cat, epochs=100,
                        validation_data=(X_test_scale, y_test_cat), batch_size=64)
# Summary of the tuned model
best_model.summary()
```

OUTPUT:



Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 896)	17,920
dropout_4 (Dropout)	(None, 896)	0
dense_6 (Dense)	(None, 448)	401,856
dropout_5 (Dropout)	(None, 448)	0
dense_7 (Dense)	(None, 192)	86,208
dropout_6 (Dropout)	(None, 192)	0
dense_8 (Dense)	(None, 64)	12,352
dropout_7 (Dropout)	(None, 64)	0
dense_9 (Dense)	(None, 2)	130

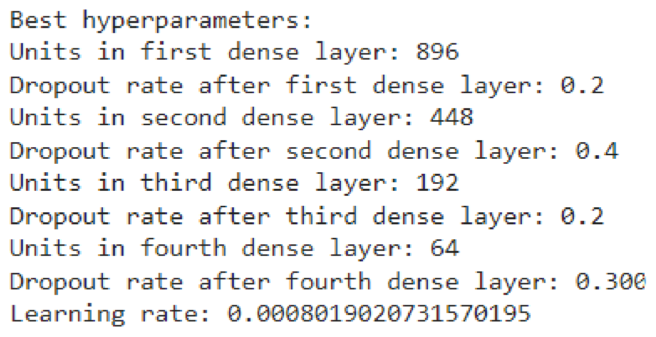
```
Total params: 1,555,400 (5.93 MB)
Trainable params: 518,466 (1.98 MB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 1,036,934 (3.96 MB)
```

Figure 4.3.11 Model summary Tuned DNN architecture 3

Above is the model summary of tuned architecture 3 the model has a total of 1,555,400 parameters, with 518,466 trainable parameters, meaning these will be adjusted during training. The optimizer manages 1,036,934 parameters, optimizing the learning process for better performance than before.

The optimal hyperparameters for the tuned DNN architecture 3 model are as follows: 896 units in the first dense layer with a 0.2 dropout rate, 448 units in the second layer with a 0.4 dropout, 192 units in the third layer with a 0.2 dropout, and 64 units in the fourth layer with a 0.3 dropout. The optimal learning rate is 0.00080.

OUPUT:



```
Best hyperparameters:
Units in first dense layer: 896
Dropout rate after first dense layer: 0.2
Units in second dense layer: 448
Dropout rate after second dense layer: 0.4
Units in third dense layer: 192
Dropout rate after third dense layer: 0.2
Units in fourth dense layer: 64
Dropout rate after fourth dense layer: 0.300
Learning rate: 0.0008019020731570195
```

Figure 4.3.12 Best hyperparameters for Tuned DNN architecture 3

4.3.3.1 Plotting Training and Validation Accuracy Value

The plot below shows the training and validation accuracy and loss for a tuned DNN model 3 over 100 epochs. In the accuracy plot, both training and validation accuracy increase steadily, with validation accuracy consistently higher than training accuracy, reaching about 97.5%. In the loss plot, both training and validation loss decrease gradually, with validation loss decreasing more quickly than training loss. By the end of training, validation loss is around 0.1, and training loss levels off at about 0.15. These patterns indicate that the model is performing well without major overfitting, balancing training and validation effectively and performance is very similar tuned DNN architecture 2.

OUTPUT:

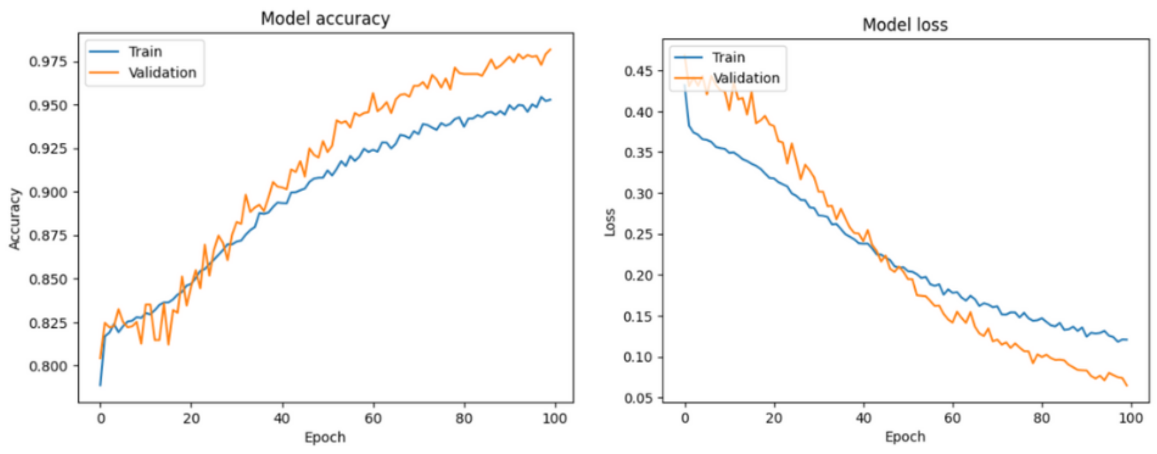
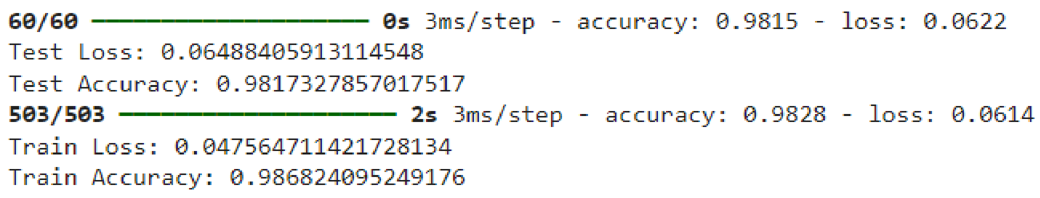


Figure 4.3.13 Training and Validation plot of Tuned DNN architecture 3

4.3.3.2 Evaluation Training and Test set

While evaluating the 3rd tuned architecture it can be observed that the training results show strong model performance with a high training accuracy of 98.68% and a low training loss of 0.047, indicating the model fits the training data well. The test results also indicate excellent generalization, with a test accuracy of 98.17% and a slightly higher test loss of 0.0648. Overall, the model performs well on both training and test sets, showing minimal overfitting and strong predictive capabilities.

OUPUT:

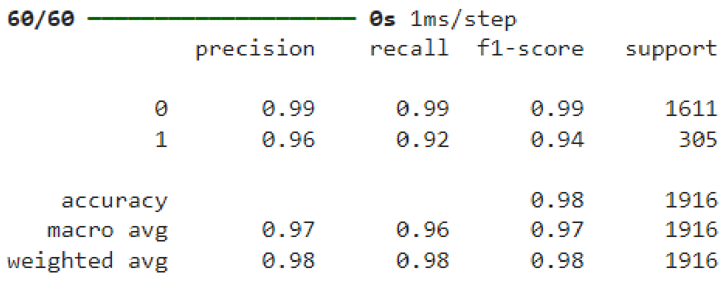


```
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9815 - loss: 0.0622
Test Loss: 0.06488405913114548
Test Accuracy: 0.9817327857017517
503/503 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9828 - loss: 0.0614
Train Loss: 0.047564711421728134
Train Accuracy: 0.986824095249176
```

Figure 4.3.14 Evaluation of Training and Test set of Tuned DNN architecture 3

The classification report shows that the model performs very well and similar to tuned DNN architecture 2, with an overall accuracy of 98%. For class 0, the precision, recall, and F1-score are all 0.99, indicating excellent identification of this class. For class 1, the precision is 0.96, recall is 0.92, and the F1-score is 0.94, which is slightly lower but still strong. The macro average (0.97 for precision, 0.96 for recall, and 0.97 for F1) shows a good balance between the two classes, and the weighted average (0.98 across all metrics) confirms that the model is consistently effective across the entire dataset.

OUPUT:



```
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1611
           1       0.96      0.92      0.94       305

    accuracy                           0.98      1916
   macro avg       0.97      0.96      0.97      1916
weighted avg       0.98      0.98      0.98      1916
```

Figure 4.3.15 Classification report for Tuned DNN Architecture 3

4.3.4 Comparison between different Tuned DNN Architectures

When comparing at the three different architectures of the Tuned DNN, it's clear that architecture 2 and 3 do better than architecture 1 in terms of accuracy, loss, and how well they handle both classes. Architecture 1 got an accuracy of 82%, did well with class 0, but struggled with class 1. Its ability to correctly identify defaults was quite low. On the other hand, architectures 2 and 3 both had an accuracy of 98% and handled both classes more evenly. Their results show high precision, recall, and F1-scores for both classes, especially for class 1, where precision was 0.96, recall was 0.92, and F1 was 0.94 in both models.

Table 4.3.1 Comparison of Tuned DNN Architectures

Architecture	Optimizer	Epochs	Test Acc.	Precision	Recall	F1-Score
Tuned DNN 1	SGD	50	82%	0.78	0.82	0.79
Tuned DNN 2	Adam	50	98%	0.98	0.98	0.98
Tuned DNN 3	Adam	100	98%	0.98	0.98	0.98

Between architectures 2 and 3, the performance metrics are virtually the same, with minor differences in hyperparameters. Both achieved excellent generalization, minimal overfitting, and consistent validation results. Therefore, either architecture 2 or 3 can be considered the best choice, with a slight preference for architecture 3 due to its simpler structure, using fewer trainable parameters, which suggests a more efficient model without compromising performance.

5 CHAPTER 5 DISCUSSION AND RESULT

The DNN model has shown much better results in predicting loan defaults than other models mentioned in previous literature review. The third DNN model in this study, which reached an accuracy of 98.07%, did better than simpler models like the single-layer ANN used by Sifrain (2023), which had an AUC of 0.936 and an accuracy of 89.4%. Also, this DNN model performs as well as more complex models like the stacking model by Zhang et al. (2020), which had an accuracy of 98.16%. The model's success comes from using multiple hidden layers, dropout regularization, and the Adam optimizer, which helped it work well on both training and test data.

The DNN architecture used in this research, particularly the third architecture, greatly improved due to thorough tuning of its hyperparameters. The architecture, which gradually reduced the size of layers and the dropout rates, helped prevent the model from becoming too specialized to the training data and improved its ability to perform well on new data. The Adam optimizer, which adjusts the learning rate automatically, also made the training process more efficient. This well-tuned, deep architecture was better at identifying complex patterns in loan data compared to simpler DNN models mentioned in previous studies.

The study's use of SMOTE to address class imbalance was a crucial factor in the model's improved performance. Previous research, such as Owusu et al. (2023), which also used oversampling methods, demonstrated the importance of handling class imbalance in predicting loan defaults. The final model showed significant improvements in precision, recall, and F1-score for both classes, reaching 0.98 in these metrics. This was a notable improvement compared to models that did not use these techniques, which struggled with class imbalance issues. The tuned DNN architecture not only outperformed traditional deep learning models but also surpassed the hybrid and ensemble models in the literature. Its ability to capture non-linear relationships and address class imbalance made it highly effective for loan default prediction.

6 CHAPTER 6 CONCLUSION

This study shows that deep learning models, especially DNNs, can improve the accuracy of predicting loan defaults better than traditional machine learning methods. By using deep learning's ability to understand complex, non-linear patterns in large datasets, DNNs can greatly improve the accuracy of loan prediction. The research tested and improved several DNN models, with the best one reaching an accuracy of 98.07%, which is better than traditional models. Using SMOTE was important for handling the issue of class imbalance, which is common in financial data where default cases are rare.

This research shows that, when set up correctly, deep learning models can find complex patterns that traditional models might miss. The results from this study show that these models are very useful for evaluating credit risk, giving banks and other financial organizations a more accurate way to predict if someone will not pay back a loan. These better models can help make smarter lending choices, lower financial losses, and make credit markets more stable.

Additionally, this study adds to the existing knowledge in academia by demonstrating how adjusting key settings in deep learning models can improve their performance, especially when dealing with issues like uneven distribution of classes. While using deep learning for predicting credit risks is still developing, future research could investigate combining different models or trying out more complex structures like transformers to make further improvements. Overall, this study has shown that deep learning is a strong method for predicting credit risks, laying the groundwork for future advancements in this area.

REFERENCES

Adebiyi, M. O., Adeoye, O. O., Ogundokun, R. O., Okesola, J. O., & Adebiyi, A. A. (2022). SECURED LOAN PREDICTION SYSTEM USING ARTIFICIAL NEURAL NETWORK. *Journal of Engineering Science and Technology, 17*(2), 854-874. Retrieved from https://jestec.taylors.edu.my/Vol%2017%20Issue%202%20April%20%202022/17_2_03.pdf

Chang, A.-H., Yang, L.-K., Tsaih, R.-H., & Lin1, S.-K. (2022, June). Machine learning and artificial neural networks to construct P2P lending credit-scoring model: A case using Lending Club data. *Quantitative Finance and Economics*, 304-326. doi:10.3934/QFE.2022013

Clements, J. M., Xu, D., Yousef, N., & Efimov, D. (2020, December). Sequential Deep Learning for Credit Risk Monitoring with Tabular Financial Data. Retrieved from https://arxiv.org/pdf/2012.15330

Jumaa, M., Saqib, M., & Attar, A. (2023). Improving Credit Risk Assessment through Deep Learning-based Consumer Loan Default Prediction Model. *INTERNATIONAL JOURNAL OF FINANCE & BANKING STUDIES*, 85-92. Retrieved from https://doi.org/10.20525/ijfbs.v12i1.2579

Kun, Z., Weibing, F., & Jianlin, W. (2020). Default Identification ofP2P Lending Based on Stacking Ensemble Learning. *International Conference on Economic Management and Model Engineering (ICEMME)*, 992-1006. doi:10.1109/ICEMME51517.2020.00203

Liang, L., & Cai, X. (2020, May). Forecasting peer-to-peer platform default rate with LSTM neural network. *Electronic Commerce Research and Applications*. Retrieved from https://doi.org/10.1016/j.elerap.2020.100997

Liu, J., Zhang, S., & Fan, H. (2022, February). A two-stage hybrid credit risk prediction model based on XGBoost and graph-based deep neural network. *Expert Systems With Applications*. Retrieved from https://doi.org/10.1016/j.eswa.2022.116624

Owusu, E., Quainoo, R., Mensah, S., & Appati, J. K. (2023). A Deep Learning Approach for Loan Default Prediction Using Imbalanced Dataset. *International Journal of Intelligent Information Technologies*, 1-16. doi:10.4018/IJIIT.318672

Paudel, S. B., Devkota, B., & Timilsina, S. (2023, November). Multi-Class Credit Risk Analysis Using Deep Learning. *Journal of Engineering and Sciences*, 82-88. Retrieved from https://doi.org/10.3126/jes2.v2i1.60399

Sifrain, R. (2023, March). Predictive Analysis of Default Risk in Peer-to-Peer Lending Platforms: Empirical Evidence from LendingClub. *Journal of Financial Risk Management, 2023, 12, 28-49*, 28-49. Retrieved from https://doi.org/10.4236/jfrm.2023.121003

Suliman, & Fati, M. (2021, October). Machine Learning-Based Prediction Model for Loan Status Approval. *Journal of Hunan University*. Retrieved from http://jonuns.com/index.php/journal/article/view/783/779

Turiel, JD, A., & T. (2020). Peer-to-peer loan acceptance and default prediction with AI. *The Royal Society Open Science*. Retrieved from http://dx.doi.org/10.1098/rsos.191649