REPORT

MULTI-LAYER PERCEPTRON MODEL

This involves pattern classification in an unbalanced dataset to determine the fraudulent transactions.

Dataset

- The dataset used for this research is collected from Kaggle at https://www.kaggle.com/mlg-ulb/creditcardfraud.
- It consists of 284,807 transactions that occurred in 2 days, of which 492 are labelled as Fraud.
- This means that the dataset is highly unbalanced with only 0.172% accounting for the Fraud transactions. It consists of 31 features of which 28 (V1-V28) are the result of PCA transformation, due to confidentiality issues.
- The remaining features that are not transformed are ‘Time’ and ‘Amount’, which represent the seconds elapsed between each transaction and the first transaction in the dataset and, the transaction amount respectively.
- The ‘Class’ feature represents the label of the transaction with ‘1’ for a Fraud transaction and ‘0’ for a ‘Genuine’ transaction.

Model Design:

Algorithm Selection: Firstly, a multi-layer perceptron with 1-hidden layer was trained followed by a multi-layer perceptron with 2 hidden layers.

Most of the parameters were tuned using Gridsearch algorithm followed by Trial and Error and, the rest were used as default provided by the Keras library.

The detailed flow of code is explained on the following pages.

Library Imports and Initial Setup

To set up the environment for data analysis and model development, we imported a range of essential libraries:

1. **Data Manipulation and Processing**:
    - *NumPy* and *Pandas* were used for numerical operations and data handling.
2. **Machine Learning Utilities**:
    - Modules from *scikit-learn*, including `RobustScaler` for feature scaling, `StratifiedShuffleSplit` for stratified dataset splitting, and various performance metrics (e.g., `accuracy_score`, `confusion_matrix`, `roc_auc_score`) were included.
    - *GridSearchCV* was used for hyperparameter tuning, and *PCA* (Principal Component Analysis) for dimensionality reduction.
3. **Visualization**:
    - *Matplotlib* and *Seaborn* were used to visualize data distributions and model performance metrics.
4. **Data Imbalance Handling**:
    - *ADASYN* from *imblearn* provided oversampling to address class imbalance.
5. **Neural Network Implementation**:
    - *Keras* (with *TensorFlow* as backend) enabled building and training neural network models, with modules for setting model architectures (`Sequential`), adding layers (`Dense`, `Dropout`), and optimization (`Adam`).
6. **Utility Modules**:
    - Miscellaneous libraries, including *scipy* and *collections*, were used for statistical analysis and data handling.

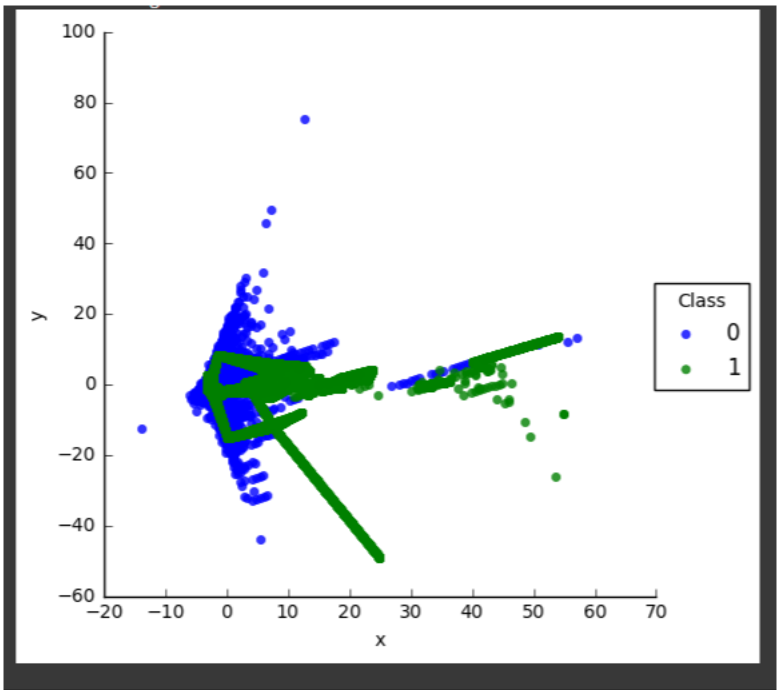
By integrating these libraries, the environment was prepared for streamlined data analysis, visualization, and model training.

Data Oversampling Using ADASYN

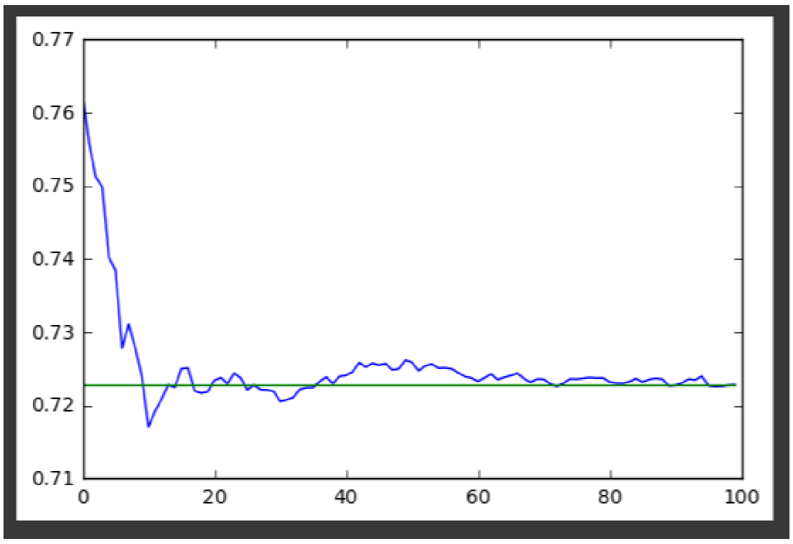
To address class imbalance in the dataset, we applied the **Adaptive Synthetic (ADASYN) Oversampling** technique, which focuses on creating synthetic samples for the minority class. This was implemented only on the training set to avoid data leakage into the test set.

- **Oversampling Process**:
  The ADASYN algorithm was used with a `sampling_strategy` targeting the minority class. A `random_state` was set to ensure reproducibility.
- **Results**:
  - After applying ADASYN, the distribution of classes was balanced, as confirmed by the updated class counts.
  - The resampled training data dimensions were recorded as follows:
    - `X_adasampled` shape: `(number of samples, number of features)`
    - `Y_adasampled` shape: `(number of samples,)`

This oversampling step helped enhance the model's ability to generalize across both classes.



The above plot gives a visual representation of the balanced dataset. As seen, the data is not linearly separable. Hence, a Multi-Layer Perceptron is trained on this data, in order to predict the data as 'fraud' or 'genuine'. For the training of the Multi-layer perceptron, many parameters need to be tuned, thus a Grid-Search is used for this purpose. The Grid-Search is performed on the subset of the data, as it takes a lot of time to run. The following blocks create a subset of the dataset using random undersampling used only for this purpose.



As seen in the plot above, the average of F-score is getting stabilized after 60 iterations. Thus, the sample size sufficiency is taken as 60, which means that the model will be trained for 60 repetitions and the overall average of the score from these iterations will determine the skill of the model.

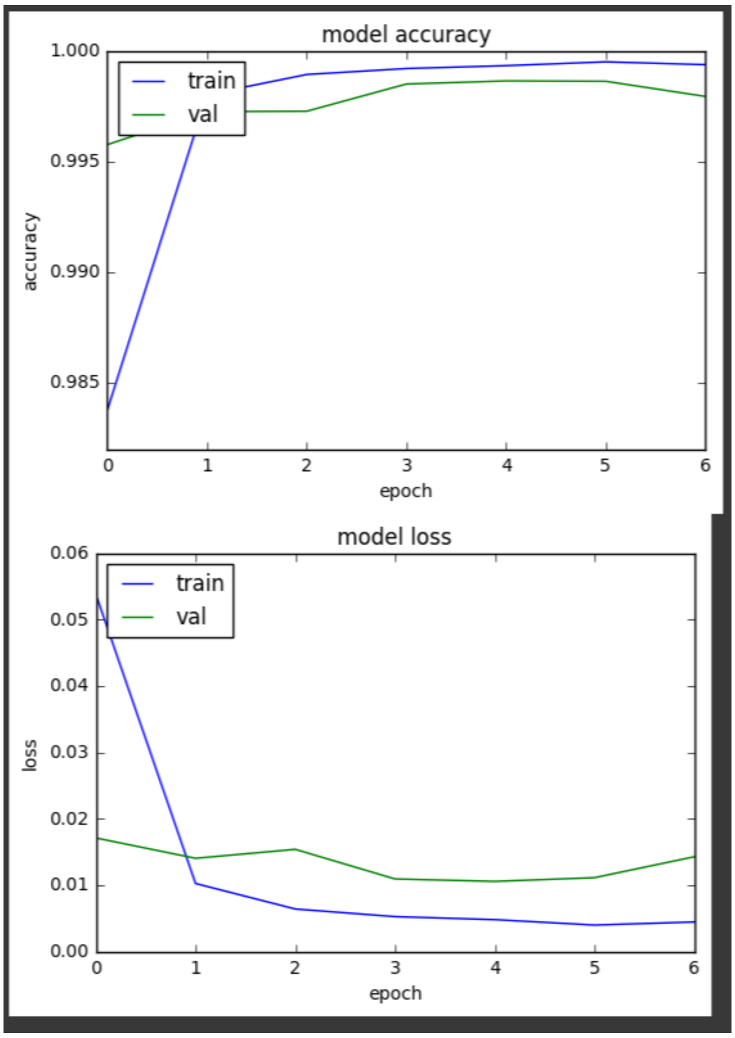
Training a Multi-Layer Perceptron (MLP) on the Oversampled Dataset

To classify the oversampled dataset, we trained a **Multi-Layer Perceptron (MLP)** model using Keras with a single hidden layer and no dropout, incorporating parameters optimized through previous tuning steps.

- **Model Architecture**:
    - Input Layer: Configured with `n_inputs` nodes, matching the feature count of the oversampled data.
    - Hidden Layer: Contains 65 nodes, uses the ReLU activation function, and the He normal initializer.
    - Output Layer: A single node with a sigmoid activation function to output binary class predictions.
- **Model Compilation**:
    - Optimizer: The Adam optimizer with a learning rate of 0.01 was used to train the model.
    - Loss Function: Binary cross-entropy was chosen as the loss function, suitable for binary classification.
    - Metrics: Accuracy was tracked as the primary performance metric.

- **Training Process**:
    - An **Early Stopping** callback was implemented, monitoring validation loss and configured to restore the best weights if no improvement was seen after 2 epochs.
    - The model was trained over 30 epochs, with a batch size of 700, on the oversampled training data (`X_adasampled`, `Y_adasampled`) and validated on a separate validation set (`Xval_arr`, `Yval_arr`).
    - Training history details, including accuracy and loss values, were recorded for both training and validation sets.

This setup aimed to achieve effective classification on the balanced dataset while avoiding overfitting through early stopping.



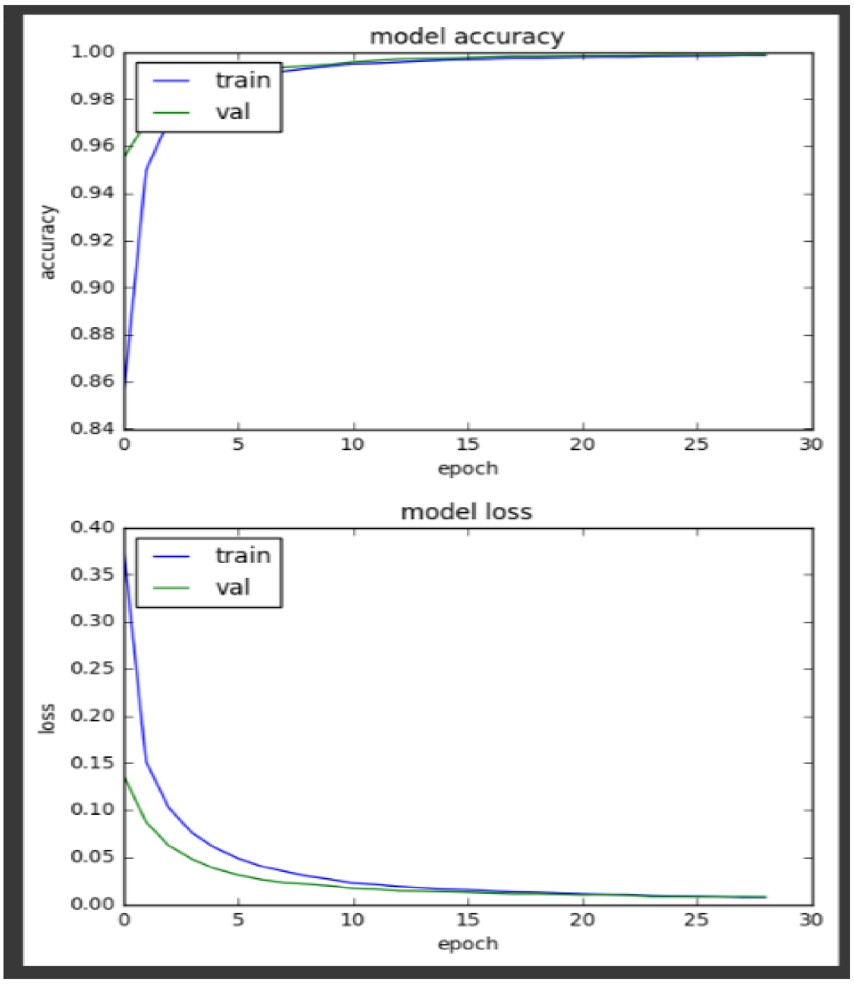
The variation in the performance of the train and validation data shows that it has been overlearned on the train dataset. So, let's try adding Dropout and see it's effect on the performance of the model.

Training a Multi-Layer Perceptron (MLP) with Dropout on the Oversampled Dataset

In this iteration, we trained a **Multi-Layer Perceptron (MLP)** model similar to the previous setup, but with the addition of a dropout layer to help prevent overfitting on the oversampled dataset.

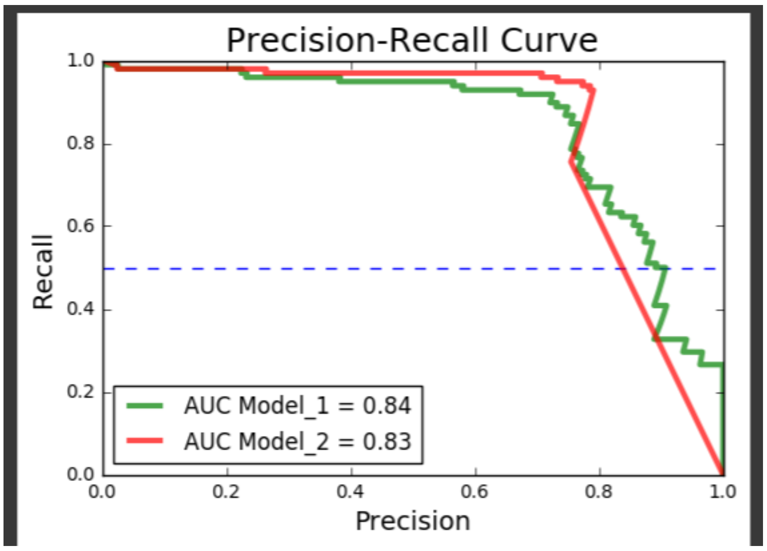
- **Model Architecture**:
  - Input Layer: Set to `n_inputs`, matching the number of features in the oversampled data.
  - Hidden Layer: Composed of 65 nodes, using the ReLU activation function and He normal initializer.
  - **Dropout Layer**: A dropout rate of 0.5 was added after the hidden layer to randomly disable 50% of the nodes during training, which improves generalization.
  - Output Layer: A single node with a sigmoid activation function for binary classification.
- **Model Compilation**:
  - Optimizer: The Adam optimizer with a reduced learning rate of 0.001 was used to adapt the learning speed given the added dropout.
  - Loss Function: Binary cross-entropy remained the chosen loss function, suitable for binary classification tasks.
  - Metrics: Accuracy was tracked as the performance metric.
- **Training Process**:
  - An **Early Stopping** callback was used, monitoring validation loss with a patience of 2 epochs to prevent overfitting and to restore the best weights.
  - The model was trained for a maximum of 40 epochs with a batch size of 700 on the oversampled training data (`X_adasampled`, `Y_adasampled`) and validated on `Xval_arr` and `Yval_arr`.
  - Training history, including loss and accuracy for both training and validation sets, was recorded.

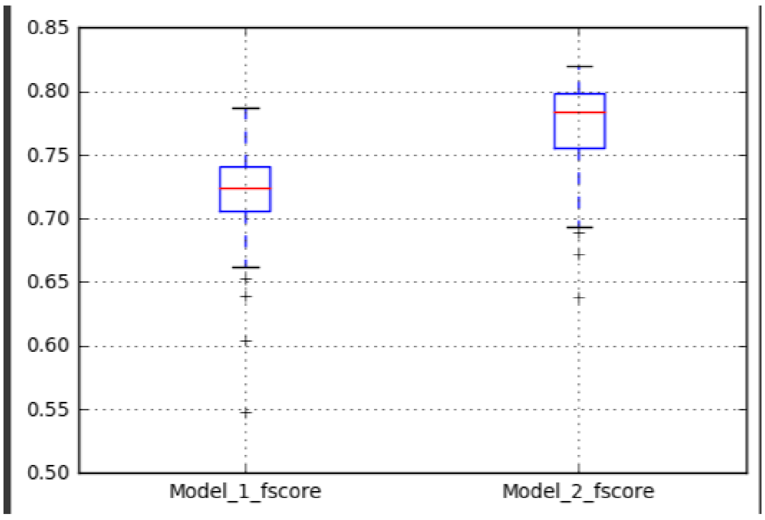
By incorporating dropout, this model aimed to improve robustness and generalization compared to the previous model without dropout.

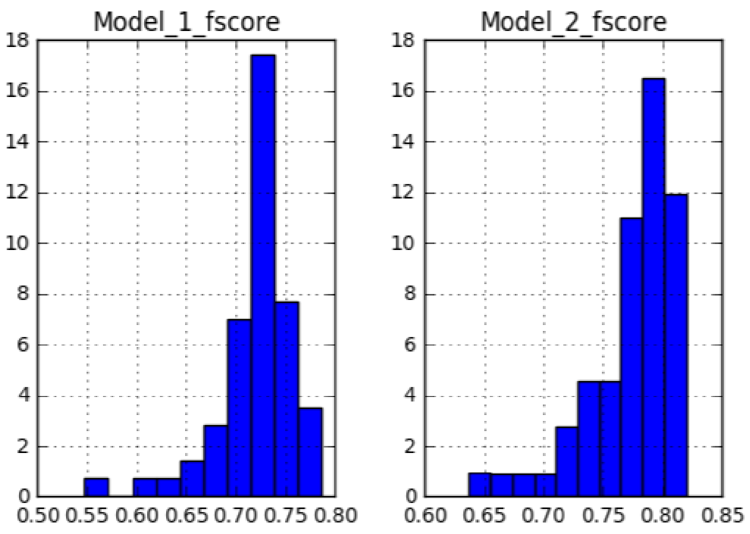
As 

observed, with the addition of Dropout, the validation dataset performs better. And the train and vaidation set performance is comparable.

Final Model Accuracy Comparison


Precision-Recall Curve
Recall
Precision
AUC Model_1 = 0.84
AUC Model_2 = 0.83


Model_1_fscore
Model_2_fscore


Model_1_fscore
Model_2_fscore