FRAUD DETECTION IN FINANCIAL TRANSACTIONS USING MACHINE LEARNING AND DEEP LEARNING

FINAL PROJECT - PHASE 3 (PLACEHOLDER)

UNDER THE GUIDANCE OF PROF. DR. INDIRA VADAPALLY

UNIVERSITY OF WISCONSIN, MILWAUKEE

Group 4

Group Members: Kaivalya Sunil Patkar (kpatkar@uwm.edu)

Gayatri Shankar Shinde (gsshinde@uwm.edu)

Objective: Machine Learning and Deep Learning project designed to spot Credit Card Transaction Fraud using smart algorithms and neural networks to create a system that can catch and stop fake transactions early, making sure every financial move is secure. Building a system that makes credit card users feel totally confident about their money.

Big Data Final Project Plan of Action:

The frauds in credit card transactions are commonplace nowadays and credit card companies have been documenting customer expenditure to identify patterns in a fraudulent transaction. The data has been made open for educational purpose on https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud.

To analyze this data and find valuable insights using different Machine Learning and Deep Learning concepts, we have put up the following plan of action:

- **Step 1** – Data Cleaning
  - This step includes handling any missing values, null values in the dataset, outliers, and anomalies.
  - In case of missing values, imputing data using mean of the entire column

- **Step 2** – Exploratory Data Analysis (EDA)
  - Understanding the distribution and correlation between the datapoints
  - Choosing the insights necessary for featuring engineering

- **Step 3** – Data Preprocessing
  - Cleaning and preprocessing the data that would act as an input to the chosen Machine Learning Model.
  - Further, splitting the data into train & test split to check how well the model has learnt from the input training data.

- **Step 4** – Model Training
  - As the project has an unlabeled dataset, the models we have chosen are Logistic Regression & Random Forest Classifier.
  - Optimizing the hyperparameters to improve the model's performance would be the next step.

- **Step 5** – Model Evaluation

  - We will be employing the Area Under the Precision-Recall Curve to evaluate the performance of the model.
  - Also, using weights we plan to convert the unbalanced dataset to a balanced one to compute the Confusion Matrix.
  - As a part of implementing Deep Learning for performance improvement, we will employ the autoencoders.

- **Step 6** – Model Deployment
  - Deploying the model using Flask.

A. Data used: Kaggle Dataset - CreditCard.csv

B. What platform and language - Jupyter notebooks and using Python Language.

C. Data Collection/Cleaning:

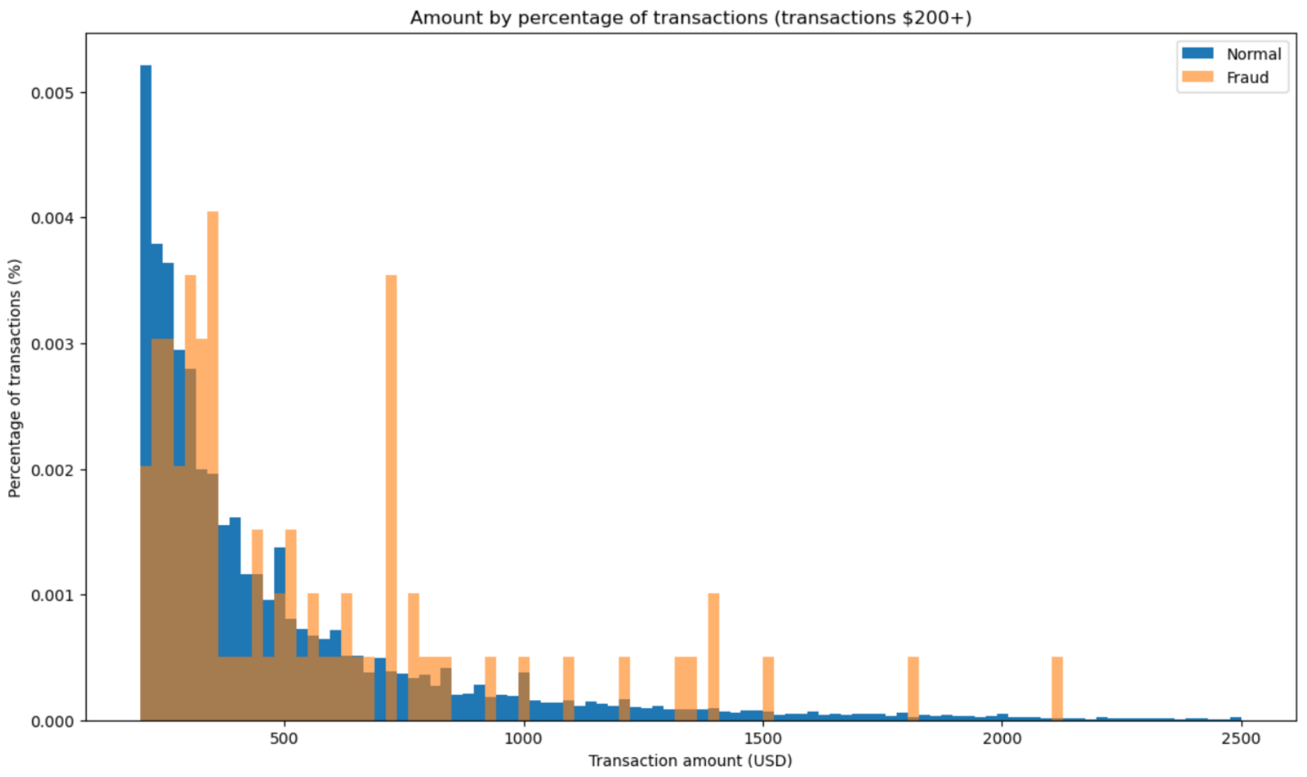
The dataset, which included the predictive class column, was taken from Kaggle's 'Credit Card Fraud Detection' repository. It has 284,807 rows and predicts two classes: '0' for real transactions and '1' for fraudulent transactions. Principal Component Analysis (PCA) was used prior to publication to protect user confidentiality. The purpose of PCA, a statistical method involving feature extraction, was to reduce the dimensionality of the feature space. This method aims to reduce attribute dimensions while keeping as much variance and correlation as possible. Specifically, the attributes labeled 'V1' through 'V28' were subjected to PCA masking, which limited the feature analysis and selection to these 28 attributes. The remaining attributes in the dataset are 'amount' and 'time.'

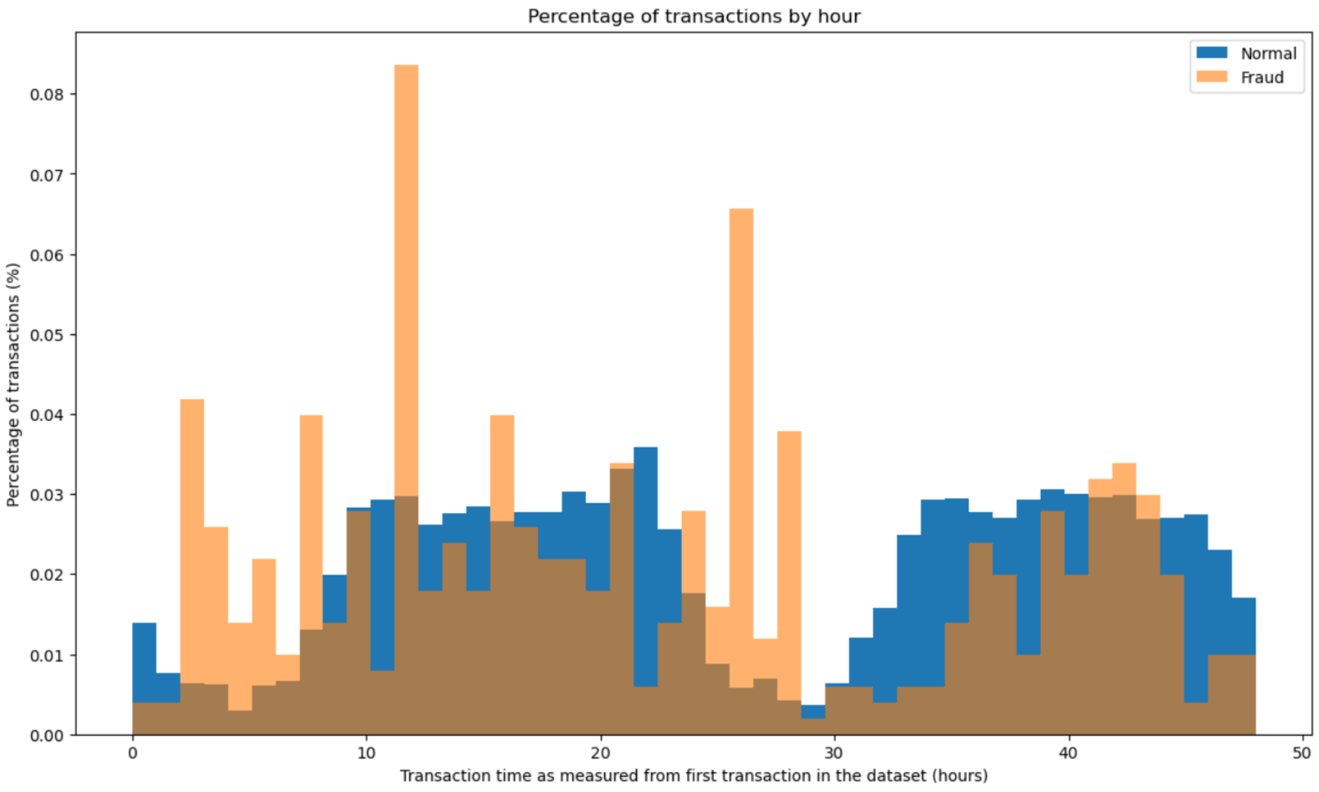
Code Snippet

```
import pandas as pd
# Load the CSV file into a Pandas DataFrame
unscaled_df = pd.read_csv("creditcard.csv")
```

D. EDA & Feature Engineering

No features within the dataset exhibited correlations exceeding 0.5 or less than -0.5, indicating no significant impact on classification. Consequently, no features were removed from V1 to V28 due to this criterion. We tried to visualize the fraudulent transactions withthe2 known features i.e. time and amount to find any specific nature of fraud It challenging to differentiate fraudulent transactions based solely on transaction amounts. The potential time offset between normal and fraud transactions is not enough to make a simple, precise classifier.





Before proceeding with training the model, the 'amount' feature underwent standardization using the 'MinMaxScaler' function from the 'sklearn' library in Python. This process involved scaling values within the range of 0 to 1. The necessity for this step arose from the varied transaction amounts in the dataset, spanning from $0.0 to $1000, with most values concentrated between $1 and $10. Standardization was vital to prevent potential biases during the training of the machine learning model. Following feature selection, 30 attributes, including the class column, remained for further analysis.

Code Snippet

```
# Initialize the StandardScaler
scaler = MinMaxScaler()

# Apply min-max scaling to the entire DataFrame
scaled_values = scaler.fit_transform(unscaled_df)
```

E. Data Preprocessing:

The 'time' feature, starting from t=0 to t=ti, was discarded as it didn't bear any influence on the classification process.

Following feature selection, 30 attributes, including the class column, remained for model development. Addressing the significant imbalance in the dataset is crucial before model creation. With less than 0.5% of transactions being fraudulent among 284,807 transactions, a model incorrectly classifying a fraudulent transaction as legitimate would yield an accuracy exceeding 99%, an unacceptable scenario. To tackle this issue, oversampling via Synthetic Minority Oversampling Technique (SMOTE) is proposed.

SMOTE operates by duplicating instances within the minority class to balance the dataset, mitigating the risk of misleading conclusions. It employs linear interpolation to generate synthetic records by analyzing the nearest neighbors of examples in the minority class. This process efficiently creates new instances proximate to existing examples. Implementing the SMOTE algorithm using the 'SMOTE' function within the Python 'imblearn' library resulted in a total of 398,016 data points after augmentation.

What did we learn in SMOTE algorithm?

- Initially, the algorithm focuses on a set of points belonging to the minority class, denoted as A. For each point x within A, the algorithm identifies its k nearest neighbors by calculating the Euclidean distance between x and other points within A.
- Determining a sampling rate denoted as N, the algorithm adjusts this rate based on the imbalanced proportion present in the dataset. For every point x in A, the algorithm selects N random samples from the k nearest neighbors, forming a new set labeled as A1.
- With the creation of A1, the algorithm proceeds to generate new samples for each point y within A1. Employing a specific formula, $Z = x + \text{rand}(0,1) * |x - y|$, the algorithm derives the coordinates of the new data point Z. Here, x and y denote the original data points' coordinates used as a reference for the generation of Z

  Code Snippet

  ```
  from imblearn.over_sampling import SMOTE
  ```

```
# Apply SMOTE to oversample the minority class
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
```

F. Splitting:

Following the dataset balancing, it underwent division into three subsets: training, validation, and test data. Adhering to the 70/30 rule, 70% of the data was allocated for training, while 15% each was assigned to both the validation and test sets. This ratio was selected due to the dataset's substantial size, eliminating the necessity for additional data points in the training phase. The aim was to prevent variance induced by excessive training data, ultimately avoiding bias in the classification process.

Code Snippet

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

G. Model training:

We have utilized the dataset to train several machine learning models using the following algorithms:

1. Logistic regression.
2. Random forest algorithm.
3. Neural network.

a. Random Forest

The line from sklearn.ensemble import Random ForestClassifier imports the Random ForestClassifier class from scikit-learn's ensemble module

The line clf = Random ForestClassifier(random_state=42) creates an instance of the Random ForestClassifier class. The random_state=42 argument sets a seed for random number generation, ensuring reproducibility of the results.

The clf.fit(X_train_resampled, y_train_resampled) line trains the Random ForestClassifier using the fit() method. Here, X_train_resampled represents the features of the training data, and y_train_resampled represents the corresponding target labels or classes. The model learns patterns and relationships within the data to make predictions.

Code Snippet

```
from sklearn.ensemble import Random ForestClassifier
# Train a classifier (Example: Random ForestClassifier)
clf = Random ForestClassifier(random_state=42)
clf.fit(X_train_resampled, y_train_resampled)
```

b. Logistic regression:

In this code snippet, scikit-learn's Logistic Regression module is employed to train a Logistic Regression classifier. The process involves initializing the logistic regression model with a specified random seed for reproducibility (random_state=42), followed by the training phase.

The model is trained using the resampled training data (X_train_resampled for features and y_train_resampled for corresponding target labels). This training equips the classifier to learn from the provided data and make predictions based on the identified patterns and relationships.

Code Snippet

```
from sklearn.linear_model import Logistic Regression
# Train a Logistic Regression classifier
log_reg = Logistic Regression(random_state=42)
log_reg.fit(X_train_resampled, y_train_resampled)
```

c. ANN

Artificial Neural Network (ANN) using TensorFlow's Keras.

Why ANN?

Complex Relationships: ANNs excel in capturing intricate nonlinear relationships within data, a task challenging for traditional supervised learning algorithms like linear regression or decision trees.

Feature Extraction: Deep neural networks, a type of ANN, autonomously learn pertinent features from raw data, especially beneficial when handling high-dimensional or unstructured data like images, audio, or text, eliminating the need for manual feature engineering.

Scalability: ANNs can effectively scale with increased data and computational resources by adjusting their architecture (layers, neurons), making them adaptable to large datasets and potentially enhancing performance.

Versatility: ANNs can be applied to diverse supervised learning tasks, including classification, regression, image recognition, natural language processing, and even complex tasks like reinforcement learning, showcasing their adaptability across multiple domains.

Adaptability: Trained ANNs can adjust to new and unseen data to a certain extent without requiring significant retraining, proving valuable in dynamic environments where underlying patterns may change over time

The architecture defined consists of multiple layers: the input layer with 64 neurons utilizing the ReLU activation function, followed by a hidden layer with 32 neurons also employing ReLU activation, and finally an output layer with 1 neuron and a sigmoid activation function for binary classification.

The model is compiled using 'adam' as the optimizer, 'binary_crossentropy' as the loss function, and 'accuracy' as the metric for evaluation. The model is trained on the resampled training data for 10 epochs with a batch size of 32. It also uses a validation split of 0.1 (10% of the training data) for validation.

The model predicts probabilities on the test set. A threshold of 0.5 is used to convert probabilities to classes (0 or 1) for classification. Predicted probabilities above the threshold are converted to 1; otherwise, they are set to 0.

Code Snippet

```
model = Sequential([
   Dense(64, input_shape=(X_train_resampled.shape[1],), activation='relu'),
   Dense(32, activation='relu'),
   Dense(1, activation='sigmoid')])
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Train the model on the resampled data
model.fit(X_train_resampled, y_train_resampled, epochs=10, batch_size=32,
validation_split=0.1)
```

Hypertuning theLogistic Regression

We defined a parameter grid (param_grid) containing potential hyperparameters for the Logistic Regression model to explore during tuning. This includes different values for the regularization parameter (C) and penalty norms (penalty).

Using GridSearchCV, the code conducts a cross-validated grid search (grid_search) over the specified parameter grid to find the best combination of hyperparameters. The cv=3 argument denotes a 3-fold cross-validation strategy, and verbose=2 enables detailed logging.

Best Hyperparameters: {'C': 100, 'penalty': 'l2'}

Code Snippet

```
param_grid = {
   'C': [0.001, 0.01, 0.1, 1, 10, 100],  # Regularization parameter
   'penalty': ['l1', 'l2']  # Penalty norm}
lr = Logistic Regression(random_state=42, max_iter=1000)

grid_search = GridSearchCV(lr, param_grid, cv=3, verbose=2)
grid_search.fit(X_train_resampled, y_train_resampled)
```

H. Model Evaluation:

The evaluation utilized the F_1 Score, a robust metric particularly effective for assessing results when dealing with imbalanced classes. Given the prevalence of imbalanced datasets in financial data analysis, the F_1 Score stands out as a superior metric for evaluating model performance. In the realm of fraud detection and within imbalanced datasets, employing evaluation metrics like precision-recall curve, F1-score, or area under the receiver operating characteristic curve (AUC-ROC) tends to offer more nuanced insights compared to traditional accuracy metrics. These specialized metrics better capture the intricacies associated with imbalanced data, providing a clearer understanding of a model's performance in distinguishing between genuine and fraudulent instances. We also visualized it using ROC curve with Matplotlib.

The provided metrics indicate variations among Random Forest, Logistic Regression, and Artificial Neural Network (ANN) models. Differences in metrics could stem from:

1. **Model Complexity**:
   - Random Forest, being an ensemble method, may capture complex relationships more effectively.
   - Logistic Regression might struggle with intricate patterns in the data.
   - ANN's performance depends on architecture; complexity may not align with the problem's needs.

2. **Data Imbalance**:
   - If there's class imbalance, models might favor the majority class, impacting metrics like precision and recall differently.

3. **Feature Importance**:
   - Random Forest assigns importance to features differently; crucial features might affect metrics.
   - Logistic Regression assumes linear relationships; non-linear patterns can be missed.

4. **Hyperparameter Tuning**:
   - Different models may require distinct hyperparameter tuning strategies affecting their performance.

5. **Overfitting/Underfitting**:
   - Overfitting in Random Forest could lead to high accuracy but lower precision and recall.
   - Logistic Regression might underfit complex data, affecting various metrics.

6. **Data Quality**:
   - Noisy or irrelevant features can impact all models differently.

7. **Algorithm Sensitivity**:
   - Random Forest and ANN might be sensitive to outliers, affecting certain metrics more than Logistic Regression.

8. **Cross-Validation**:
   - Differences in cross-validation strategies might impact metric values.

9. **Data Preprocessing**:
   - Varied preprocessing methods might impact how each model handles the data.

10. **Class Separation**:
    - If classes are not well-separated, models might struggle, especially Logistic Regression.

11. **Model Assumptions**:
    - Logistic Regression assumes linearity, and deviations impact its metrics differently than Random Forest or ANN.

12. **Computational Resources**:
    - Limited resources may impact the depth or architecture of the ANN, affecting its performance.

Careful consideration of these factors and further exploration of the data and models can help understand and address the differences in performance metrics.

I. Results analysis

Metrics	Random Forest	Logistic Regression	ANN	Adjusted_Logistic Regression
Accuracy	0.99952	0.975469	0.9852	0.9730
Precision	0.832168	0.056159	0.0908	0.521
Recall	0.875	0.911765	0.9191	0.9265
F1-score	0.853047	0.105802	0.1652	0.0986
ROC	0.94	0.94	0.95	0.94

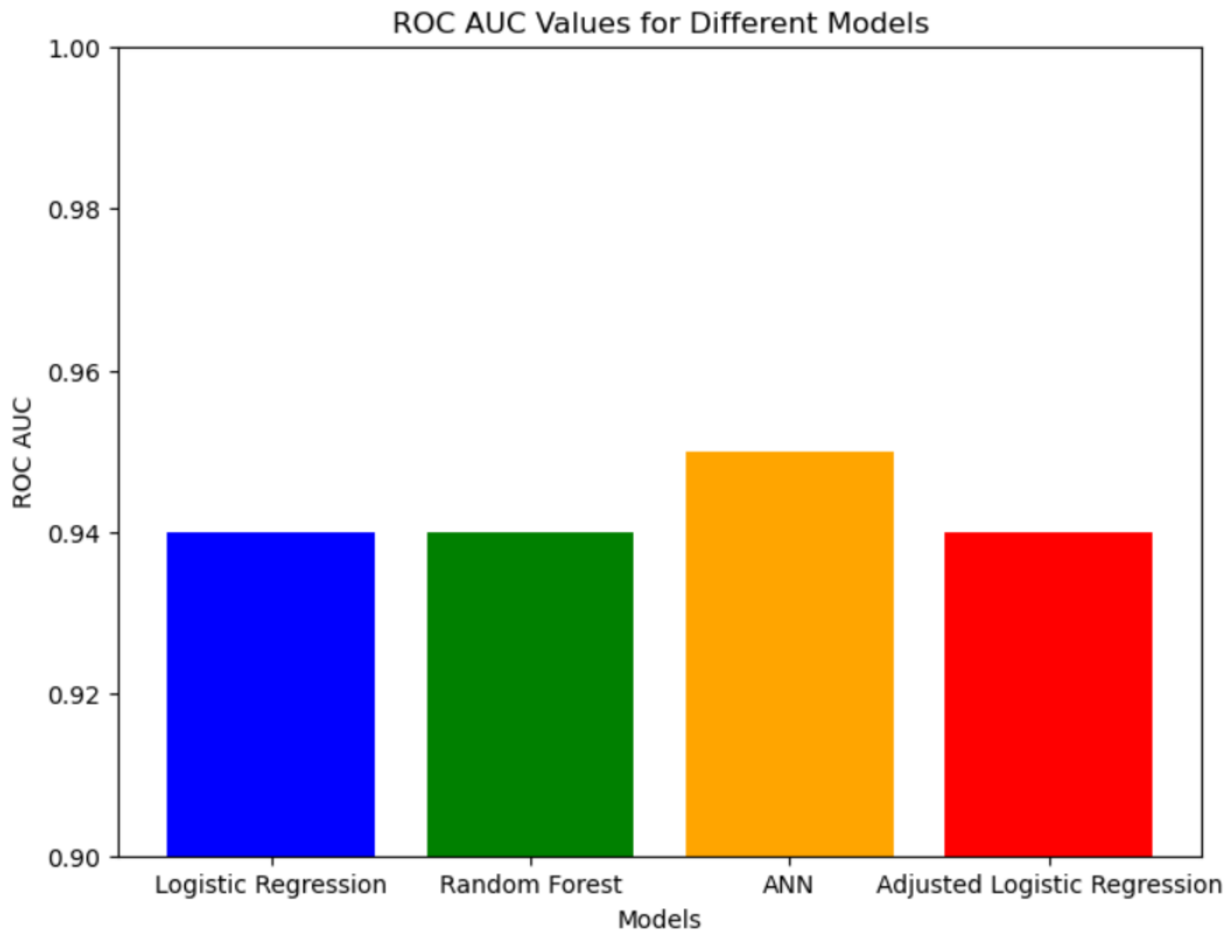
Overall Hypothesis and Model Selection:

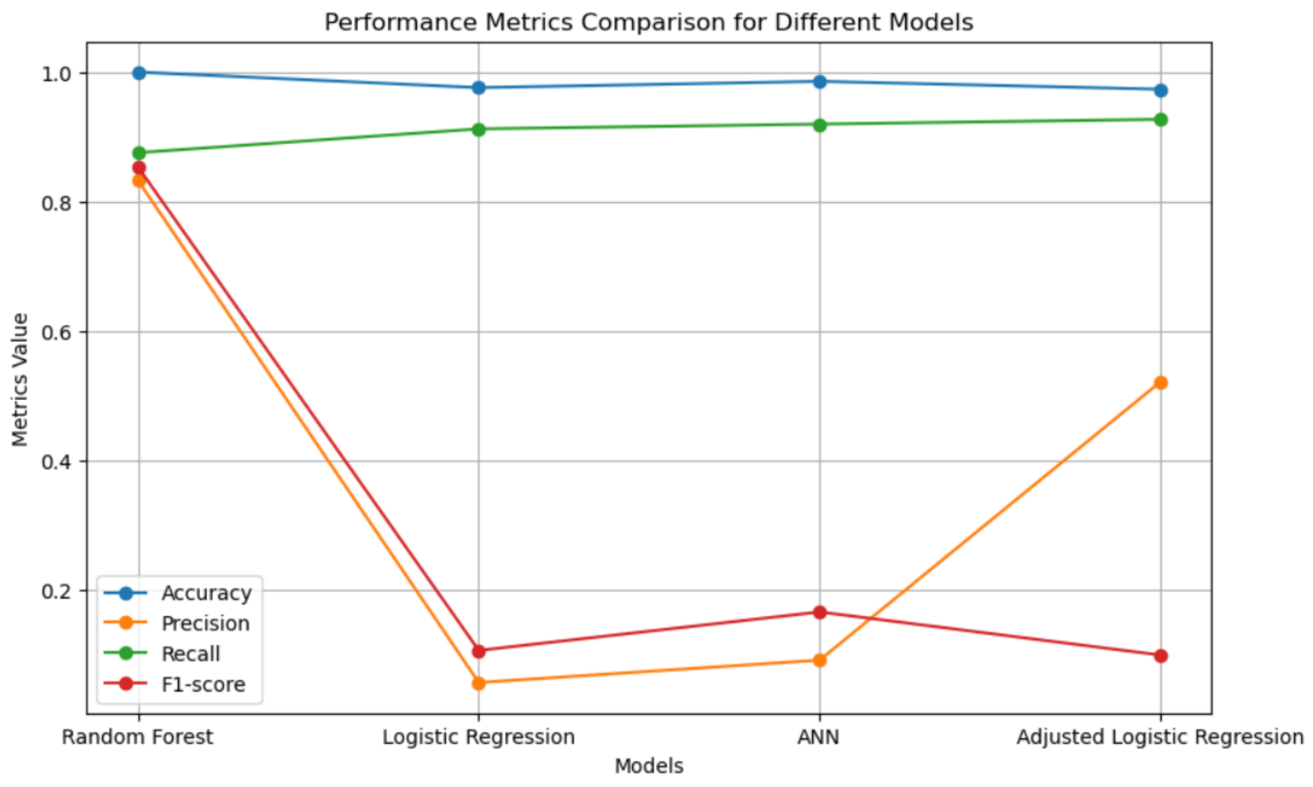
Based on the evaluation metrics and the hypotheses:

- **Random Forest**: It performs consistently well across various metrics, having high accuracy, precision, recall, and a balanced F-1 score. The ROC AUC is also strong.

- **Logistic Regression**: While it has high accuracy and ROC AUC, it has lower precision and F-1 score compared to Random Forest.

- **Artificial Neural Network (ANN)**: It performs well in terms of accuracy, recall, and ROC AUC but has lower precision and F-1 score compared to Random Forest.


ROC AUC Values for Different Models
ROC AUC
1.00
0.98
0.96
0.94
0.92
0.90
Logistic Regression
Random Forest
ANN
Adjusted Logistic Regression
Models


Performance Metrics Comparison for Different Models
Metrics Value
1.0
0.8
0.6
0.4
0.2
Accuracy
Precision
Recall
F1-score
Random Forest
Logistic Regression
ANN
Adjusted Logistic Regression
Models

J. Model Deployment:

Deploying a machine learning (ML) model involves making your trained model accessible for use by others, typically through an API or some other form of integration. Below are general steps to guide you through deploying an ML model:

1. **Train and Save the Model**:
   - Train your machine learning model using your chosen framework (e.g., scikit-learn, TensorFlow, PyTorch).
   - Save the trained model to a file using serialization libraries like `joblib`, `pickle`, or the model-specific saving methods provided by your chosen ML library.

2. **Choose a Deployment Platform**:
   - Select a platform or infrastructure for deploying your model. Common options include cloud platforms (AWS, Google Cloud, Azure), containerization platforms (Docker, Kubernetes), or edge devices.

3. **Create a Prediction Service**:
   - Develop a service or application that will handle incoming requests and use your trained model for predictions. This can be a REST API, a microservice, or another type of server.

4. **Web Framework for API (Optional)**:
   - If deploying a REST API, choose a web framework for your preferred programming language. Flask (Python), Django (Python), FastAPI (Python), Express (Node.js), and Spring Boot (Java) are common choices.

5. **Expose an API Endpoint**:
   - Expose an API endpoint that will receive input data for predictions. Design the API to accept data in a format that your model understands.

6. **Load the Model**:
   - In your prediction service, load the trained model from the saved file during the initialization phase. This ensures the model is ready to make predictions when a request is received.

7. **Handle Requests**:
   - Implement logic to handle incoming requests. Parse the input data, perform any necessary pre-processing, and then use the loaded model to make predictions.

8. **Return Predictions**:
   - Return the model predictions as a response to the incoming request. Ensure the response format is suitable for your use case.

9. **Testing**:
   - Test your deployment thoroughly. Check for any errors, unexpected behavior, or performance issues. Use sample input data to verify that the predictions are accurate.

10. **Scalability and Monitoring**:

- Consider the scalability of your deployment, especially if you expect a large number of requests. Implement monitoring to track the performance of your deployed model over time.

11. **Security**:
 - Implement security measures to protect your deployed model from potential threats. Consider authentication, authorization, and encryption depending on your deployment environment.

12. **Documentation**:
 - Document your API or deployment so that users or developers can understand how to interact with and integrate your model.

13. **Continuous Integration/Continuous Deployment (CI/CD)**:
 - Set up CI/CD pipelines to automate the testing, deployment, and updates of your model. This ensures a streamlined and reliable deployment process.

14**. Versioning (Optional)**:
 - Consider implementing versioning for your model to manage changes and updates without disrupting existing users.

15. **Monitoring and Maintenance**:
 - Implement tools and processes for monitoring the performance of your deployed model. Regularly update the model as needed, considering changes in the data distribution.

The specifics of each step will depend on the technologies and tools used, as well as the requirements of a particular use case.

K. Conclusion:

In Phase 2, we finalized that **Random Forest Model** would be the best option for getting optimum results.

An ML model deployment is generic and all the steps to deploy a model are given .

Although the model could be finalized on the basis of performance of the models, it is practically impossible to deploy the model as the labels of the attributes are hidden for confidentiality reasons. Thus, finalizing the dependent variables which would act as an input to the deployed model is hard to gauge.

As observed in the heatmap in the EDA step, no two variables show a staggering dependence on each other. Hence, using the features without their labels could also make the deployed model not as accurate as expected from user point of view.