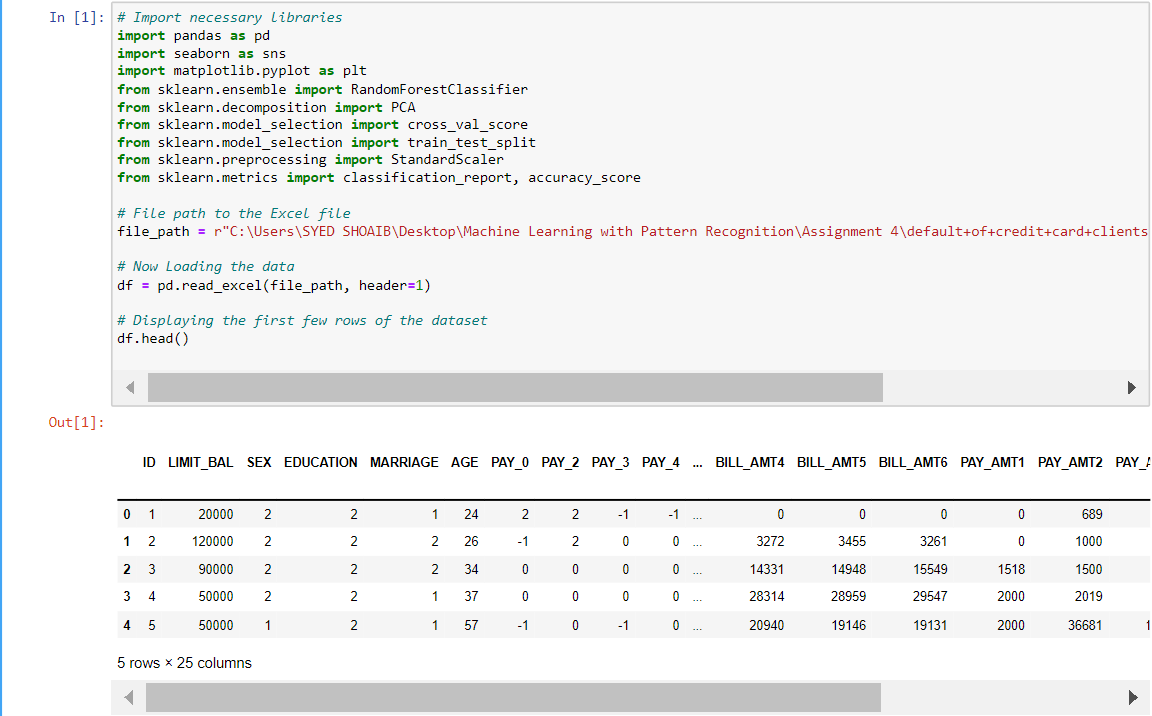
Credit Card Default Prediction

Shoaib Akhter Syed and Junaid Khan, *graduate Students*,

Lawrence Tech University,

Fall 2024

***Step 1: Loading and displaying the data set***

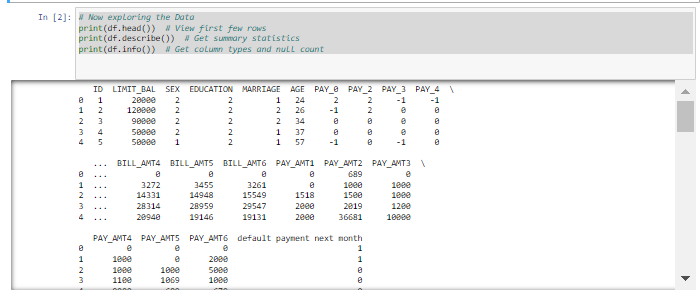


This code begins by importing necessary libraries for data handling, visualization, machine learning, and evaluation. pandas is used to manage and manipulate the dataset, **seaborn** and **matplotlib.pyplot** enable data visualization, and **sklearn** libraries provide tools for model training, evaluation, and data preprocessing. The file path to the Excel dataset is defined, specifying the exact location on the user's computer. The dataset is then loaded into a pandas **DataFrame** using **pd.read\_excel**, where header=1 ensures the appropriate row is used for column names. Finally, **df.head()** displays the first few rows of data, giving an initial glimpse of the dataset's structure and contents for verification .

***Step 2: Initial Data Exploration***

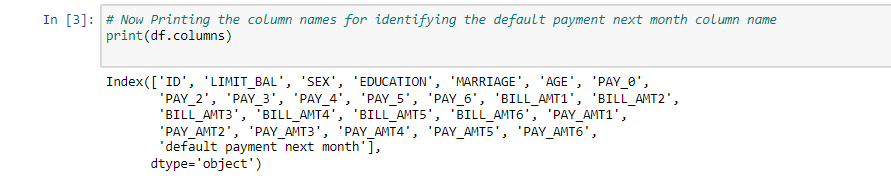
The code performs an initial exploration of the dataset to understand its structure and contents:

* print(df.head()): Displays the first few rows of the dataset, providing a quick view of the data and its columns to confirm it loaded correctly.
* print(df.describe()): Provides summary statistics for numerical columns, such as mean, standard deviation, minimum, and maximum values. This helps identify basic characteristics of the data and potential outliers.
* print(df.info()): Outputs information about the dataset, including column names, data types, and the count of non-null values in each column. This summary is essential for identifying data types and checking for any missing values in the dataset.

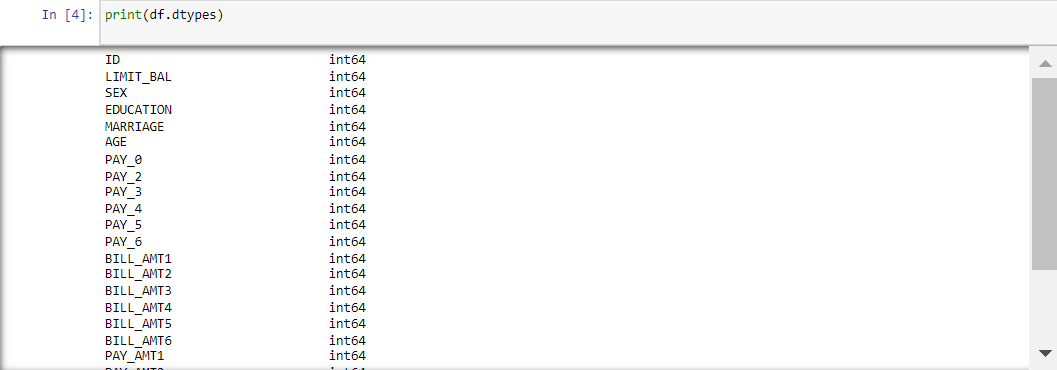


***Step 3: Identifying the Column Names***

Here we are identifying the column names to identify the column of default payment next month which is our target variable too.



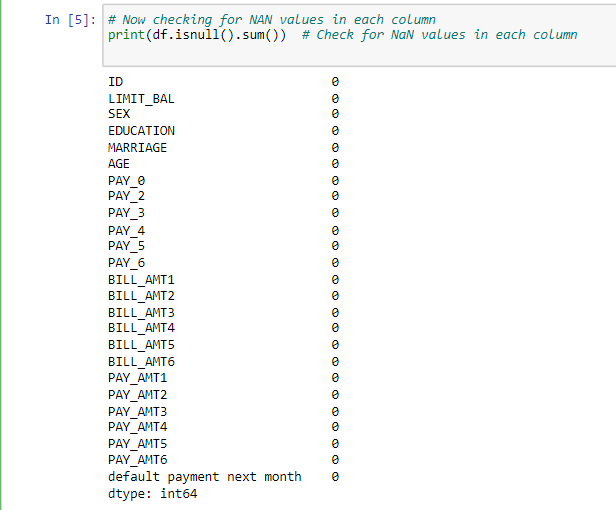
***Step 4: Checking and ensuring the data types of the feature variables***



***Step 5: Checking and ensuring that there are no NAN or missing values in columns.***

This below code block checks for any missing values (NaNs) in the dataset:

* print(df.isnull().sum()): Calculates and displays the count of missing (NaN) values in each column. If any column shows a non-zero count, it indicates missing data in that column. This step is essential for deciding on data cleaning strategies, such as filling or dropping missing values, before further analysis or modeling.



***Step 6: Dropping the missing values found***

We didn’t find any missing values still running the below function to make code efficient.



***Step 7: Normalizing the featured data***

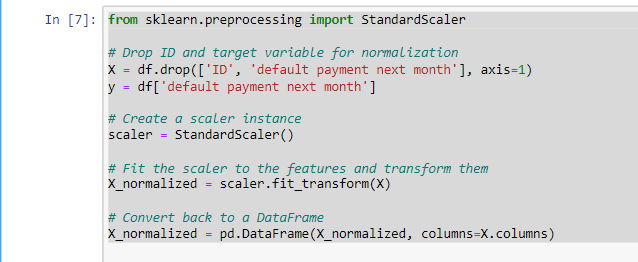
This code normalizes the feature data, preparing it for machine learning models that perform better with standardized input.

X = df.drop(['ID', 'default payment next month'], axis=1): Removes the ID and target columns from df, as these aren’t features for model training. X now contains only the input features, while y holds the target variable (default payment next month).

scaler=StandardScaler(): Initializes a StandardScaler instance, which standardizes features by removing the mean and scaling to unit variance.

X\_normalized = scaler.fit\_transform(X): Fits the scaler to X and transforms it, producing normalized feature values.

X\_normalized=pd.DataFrame(X\_normalized,columns=X.columns): Converts the normalized data back into a DataFrame, preserving the original column names, for easier reference in the next steps of analysis.



***Step 8: Writing functions to detect and remove outliers in the data.***

This code defines and uses a function to detect and remove outliers from the normalized dataset.

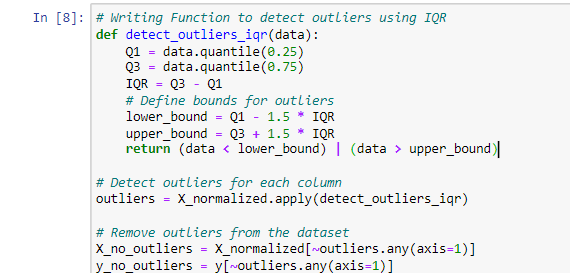
detect\_outliers\_iqr(data): Defines a function that detects outliers using the Interquartile Range (IQR) method.

Q1 and Q3 are the first and third quartiles of the data, respectively. IQR is the difference between Q3 and Q1. Outliers are identified as values that fall below Q1 - 1.5 \* IQR or above Q3 + 1.5 \* IQR. The function returns a Boolean mask, where True indicates an outlier.

outliers = X\_normalized.apply(detect\_outliers\_iqr): Applies the detect\_outliers\_iqr function to each column of X\_normalized, generating a DataFrame with True values for outliers.

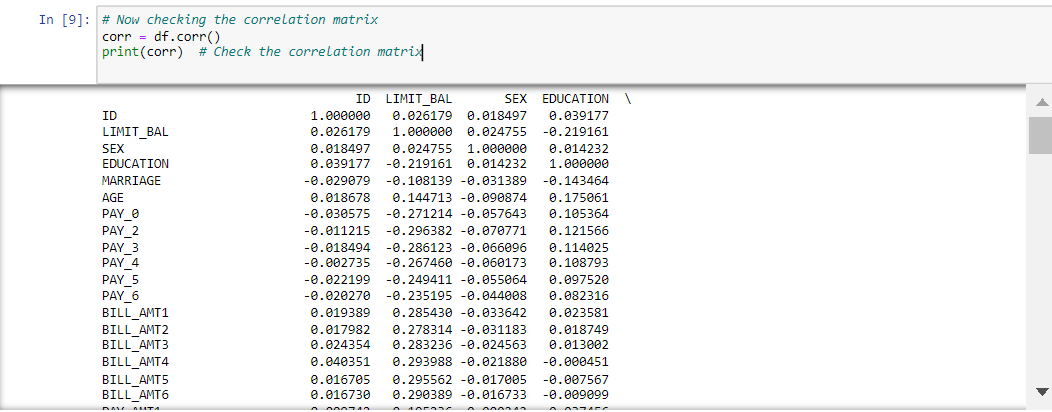
X\_no\_outliers X\_normalized[~outliers.any(axis=1)]: Removes rows containing any outliers from X\_normalized, creating a cleaned dataset (X\_no\_outliers).

y\_no\_outliers = y[~outliers.any(axis=1)]: Removes corresponding rows in the target variable y to keep it aligned with X\_no\_outliers.

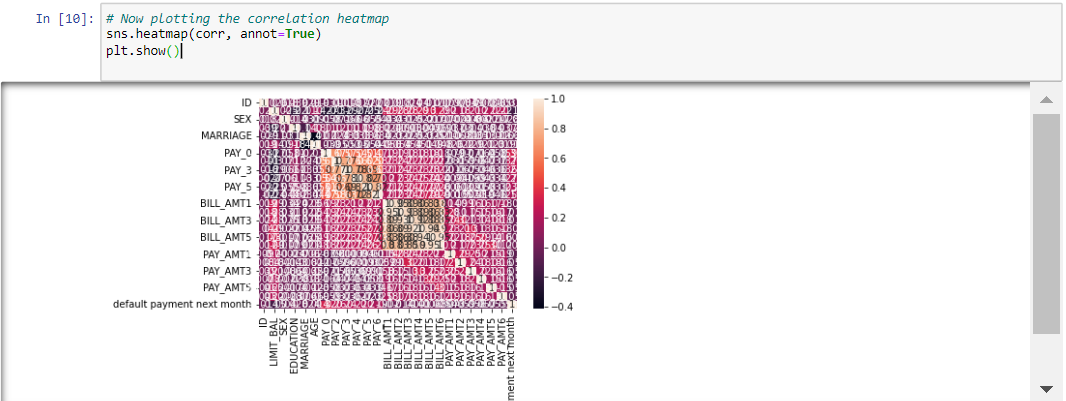


***Step 9: Checking and calculating correlation matrix for data***

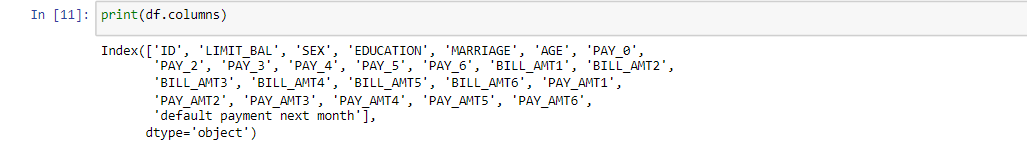
This code calculates and displays the correlation matrix for all numerical columns in the dataset, providing insight into the relationships between features. By using df.corr(), a correlation matrix is generated, where each value represents the strength and direction of the relationship between two features, ranging from -1 (strong negative correlation) to 1 (strong positive correlation). The printed matrix helps identify patterns and dependencies within the data, such as features that may be highly correlated. This information is valuable for detecting multicollinearity, as well as for making informed decisions on feature selection and further data preprocessing steps.



***Step 10: Now plotting the correlation heatmap for the defined values***

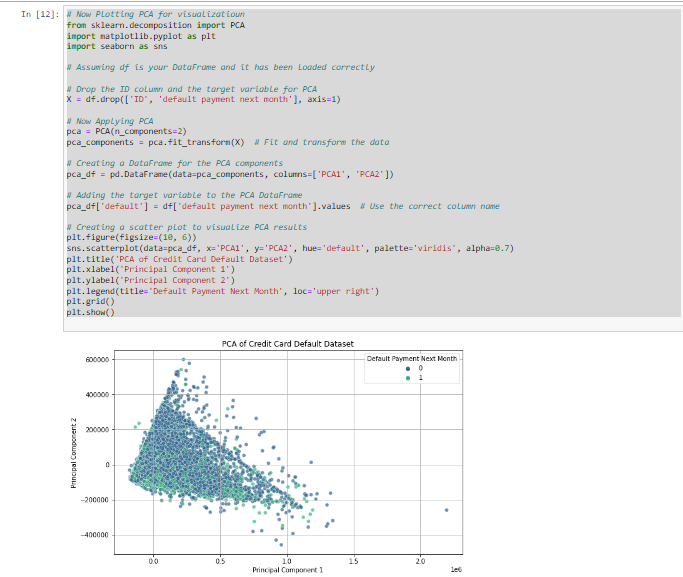
****

***Step 11: Now again printing the column names of this data set for PCA***



***Step 12: Plotting PCA for Visualization***

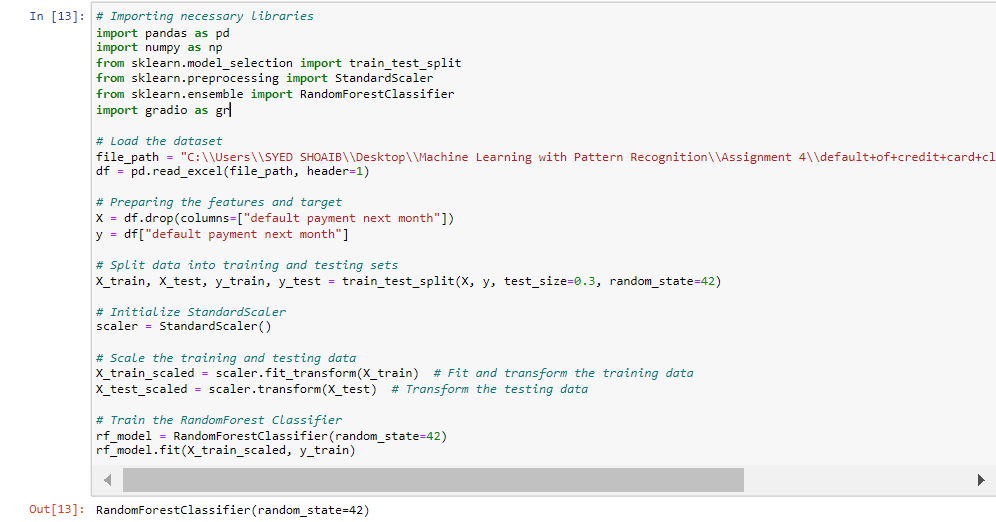
Using Principal Component Analysis (PCA) to reduce the dataset's dimensions and plot the outcome, this code visualizes the data. To make sure that only feature columns are utilized in the PCA transformation, the ID and target columns are first removed from **df**. The data is then reduced to two main components, which represent the most important variation in the data, using a PCA instance with **n\_components=2**. A new **DataFrame** called **pca\_df** contains the altered data, with columns PCA1 and PCA2 standing for the two principal components. Each plot point is labeled by adding the goal variable (default payment next month) to **pca\_df**. PCA1 and PCA2 are shown on the x and y axes of a scatter plot made by Seaborn, and points are color-coded based on the target variable. This plot provides a visual representation of the data’s structure, showing how different classes (default or not) are distributed in the transformed two-dimensional space.



***Step 13: Splitting the data into training and testing sets***

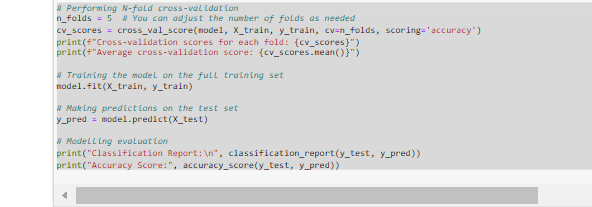
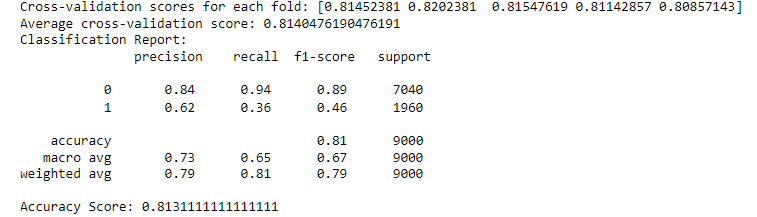
This code divides the data into training and testing sets after separating the target variable and features to get the dataset ready for modeling. The target variable itself is stored in y, and the target column (default payment next month) is first dropped from the **DataFrame** to generate feature set X. The data is then split into training and testing sets using the **train\_test\_split** function, with 30% set aside for testing and a random seed of 42 for repeatability.

The feature data is then standardized by initializing the **StandardScaler**, which guarantees that every feature makes an equal contribution to the model's performance. While the testing data is just converted to preserve the same scaling parameters, the training data is both fitted and transformed using the scaler. Ultimately, a **RandomForestClassifier** model is created and fitted using the training data **(X\_train\_scaled and y\_train)**. This gets the model ready to predict on data that hasn't been seen yet.



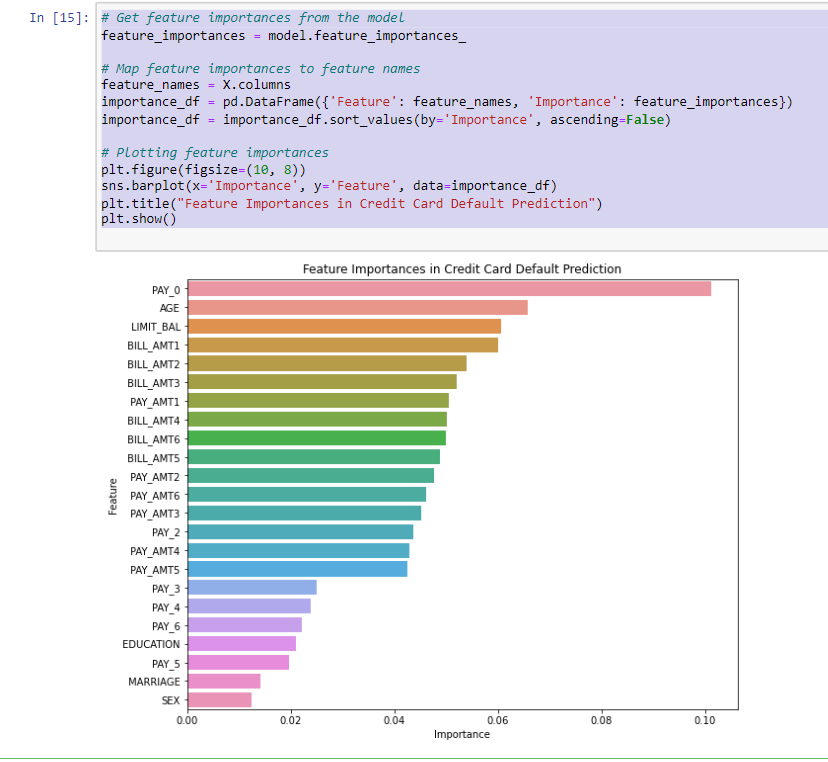
***Step 14: Performing N-Cross Validation***

This code provides predictions on the test set after evaluating the Random Forest model's performance using N-fold cross-validation. Initially, n\_folds = 5 is used to set the number of folds for cross-validation to 5. The accuracy of the model is then evaluated across these folds using the cross\_val\_score function and the training data (X\_train and y\_train).   
  
Following cross-validation, model.fit(X\_train, y\_train) is used to train the model on the whole training dataset. The model can now learn from all of the training data that is accessible. Next, y\_pred = model is used to make predictions on the test set (X\_test).forecast (X\_test).

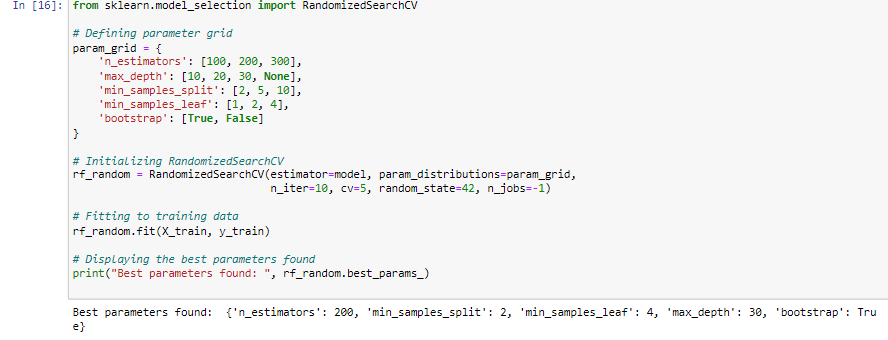
Lastly, two measures are used to assess the model's performance: For every class in the target variable, the classification\_report function produces a comprehensive report that includes precision, recall, and F1-score. The overall accuracy of the model's predictions in relation to the test set's actual values is determined by the accuracy\_score function. To evaluate the prediction power of the model, the outcomes of both tests are printed.  

***Step 15: Extracting feature importances***

In order to show which features are most important for forecasting credit card default, this code extracts and visualizes the feature importances from the trained Random Forest model. Model.feature\_importances\_ is used to extract the importance scores, which are then mapped to the feature names in a new DataFrame and sorted in descending order. Seaborn is used to construct a horizontal bar plot that shows the significance of each feature, making it simple to see which elements have the most influence on the model's predictions. Decisions about feature selection are informed by this representation, which also helps to understand model behavior.

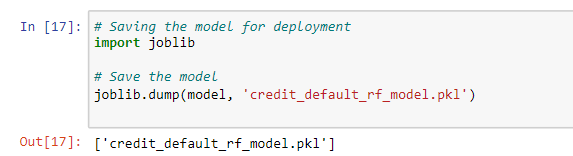


***Step 16: Now Displaying the best parameters found***



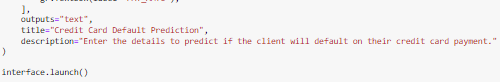
***Step 17: Now Saving the Model for Deployment***

This code saves the trained Random Forest model to a file for future use or deployment. The joblib library is imported, which is efficient for serializing large Python objects. The joblib.dump() function is then used to save the model (model) as a pickle file named credit\_default\_rf\_model.pkl. This allows the model to be easily loaded and reused later without needing to retrain it, facilitating deployment in applications that require credit default predictions.



***Step18: Implementation of Credit Card Default Prediction using Gradio***

This code block creates a Gradio web application that uses a trained machine learning model to forecast credit card default. User input for a number of credit card client characteristics, including LIMIT\_BAL, SEX, EDUCATION, MARRIAGE, and payment history, is received by the predict\_default function. The ID column is left out since it is not required for predictions, and these inputs are arranged into a dictionary before being transformed into a pandas DataFrame. The same StandardScaler that was fitted to the training data is then used to scale the input data. Based on the scaled input, the model then predicts and returns "Default" or "No Default" according to the result.



Output: 