Lawrence Technological University
Department of Math and Computer Science
MCS 5623 Machine Learning
Assignment # 04
Michael Giuliani
11/3/2024

I. INTRODUCTION & DOMAIN KNOWLEDGE

In this assignment, a dataset from University of California Irvine (UCI), called “Default of Credit Card Clients” [1], is used to write two different methods of predicting if a given client will default on payment of a credit card bill. The characteristics of the data are analyzed, the preprocessing required to make the data useful is determined, insights are drawn about what data points may or may not be useful in achieving the prediction goals, and visualizations of the data are explored to gain a better understanding of what relationships can be found in the data to make better predictions.

To make the predictions, two methods are explored: the logistic regression classifier, and the random forest ensemble classifier. A logistic regression classifier attempts to identify two classes from each other by calculating the probability that an input matches one of the classes. The probability is based on a linear combination of the input’s features and the sigmoid function [2]. A random forest ensemble classifier builds a group of decision trees based on randomly selected samples and features in the dataset, where the prediction is the averaged result of all trees [3]. These methods are compared and contrasted to see how their performance compares, what they do better or worse, and what can be changed for each to improve performance.

II. DATASET ANALYSIS & UNDERSTANDING

A. Data Characteristics

The “Default of Credit Card Clients” dataset from UCI provides data on thirty thousand individuals, detailing whether they have defaulted or not, six months of credit payment and balance history, and other statistics like age, sex, education level, and marital status. All of the data is numerically encoded and was preprocessed by UCI for any missing values [1].

The data is very imbalanced, with 77.87% not defaulted and 22.12% defaulted. Training techniques need to be used to avoid skewing predictions too much toward the more represented class [4]. When making predictions on this dataset, the performance is expected to be in the 70% - 80% accuracy range at the least, since if the model predicts “Not Defaulted” for every sample, it can be 77.87% accurate.

After cleaning and analysis, some columns were found that were more useful for classification, and some that were not needed at all. There were some rows that could be deleted as well, which were not found until several preprocessing steps were taken.

B. Feature Analysis & Selection

To select features for classification, the best relationship to look at is how the features correlate to the class labels. Figure 1 shows a correlation heatmap between every column in the dataset, where red is a strong positive correlation, white is no correlation, and blue is strong negative correlation. Since the goal is to classify whether someone will default, the row correlating the class labels to each column should be the main focus.

It becomes immediately clear from the correlation values that most of the data in this dataset is not useful for classifying if someone will default. Sex, education level, marital status, age, bill amounts, and payment amounts all have correlation values in the hundredths, if not thousandths, while 1.00 or -1.00 represent perfect correlation.

It also becomes clear which columns are useful. The columns giving data on how many months behind or ahead the person was on payments each month are ranging 0.19 to 0.32. Relatively high, but still low. The limit balance column has a slightly larger negative correlation to the class labels than other columns at -0.15, but it is not high enough to be useful. For the best classification results with the given data, the first payment history column was used, labeled “PAY_1”. This column was chosen because it has the highest correlation of any other column to the class labels at 0.32.

C. Data Cleaning/Preprocessing

The dataset was already made quite clean and ready for use by UCI, but there was more to be done. Every column was labeled by UCI with an ‘X’ followed by a number starting from one. There is also a row in the dataset that gives a more readable label for the column. For example, column “X5” has the label “AGE” in its description row. The more readable column names from that description row were used to replace the column names given by UCI. Doing this results in the description row being duplicated, so after renaming the columns that row was deleted.

After doing this it was found that the first payment history column was labeled “PAY_0”, while the next was “PAY_2”. The payment and balance size history columns begin at one, so “PAY_0” was renamed to “PAY_1”. The dataset also contains a column labeled “ID” that simply numbers each row sequentially, which is not useful for this application, so that column was deleted.

Only after taking the steps outlined above, checking for duplicate values revealed thirty-five duplicate rows. Duplicate training samples can lead to those samples having undue influence on the predictions, so they were deleted to keep each training sample influences balanced. Since the dataset is relatively large at thirty-thousand samples, deleting thirty-five of them is not an issue.

Since UCI did most of the preprocessing themselves, techniques like encoding of categorical columns did not need to be taken. Encoding non-numeric categories allows them to be used in classifiers and other AI algorithms since these algorithms rely on numeric data [5]. Encoding replaces the categorical data with a numeric representation of some sort to represent each unique category, with that numbering technique varying depending on the application. UCI already did this for this dataset.

D. Data Visualization

Figure 1 shows a cross-correlation heatmap of the dataset. The correlation to the "default payment next month" row is what matters for this assignment. Most columns have negligible correlation with it, and are therefore not useful for classification. The payment history columns have the best correlation, with "PAY_1" having the highest.

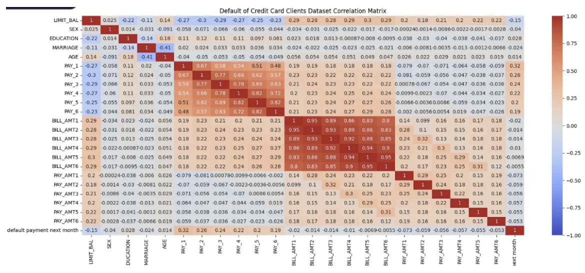


Fig. 1. Cross-Correlation Matrix Heatmap of the "Default of Credit Card Clients" Dataset. [6]

Figure 2 shows the age distribution of the dataset and how many people of each age defaulted or did not default. The ratio of defaulted to not defaulted is consistent and only shrinks a small amount in the older ages. This explains why age had little correlation with whether someone defaulted.

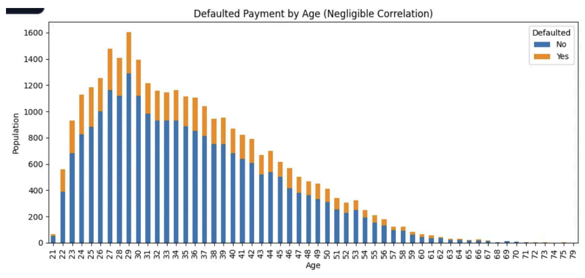


Fig. 2. Population Defaulted (Blue) or Not Defaulted (Orange) by Age. [7]

Figure 3 shows how many months behind or ahead the population was on their payments, and whether the people at each monthly increment defaulted or not. The histories range from two months ahead to eight months behind. As the number of months behind grows, the portion of people that defaulted grows significantly. The columns providing this data had relatively high correlation to whether the person defaulted, so they were most effective for classification training.

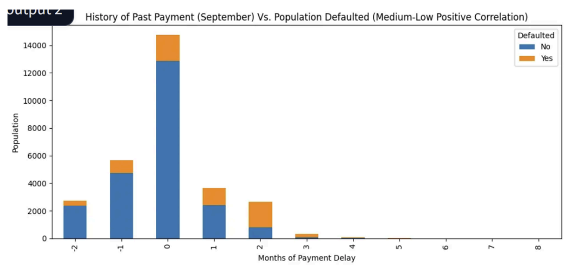


Fig. 3. Population Defaulted (Blue) or Not Defaulted (Orange) Versus Months of Payment Delay. [7]

Figure 4 shows the credit account balance limit distribution of the dataset, and how many people did or did not default at each limit amount. This data had slight negative correlation with whether the person defaulted. It is clear that people with low limits defaulted more often, which explains the correlation.

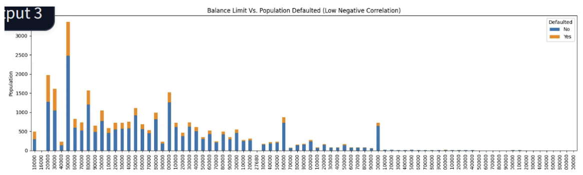


Fig. 4. Population Defaulted (Blue) or Not Defaulted (Orange) Versus Credit Limit Balance. [7]

Figure 5 shows the imbalance of the classes in the dataset. Out of 29,965 samples in the preprocessed dataset, 23,335 samples (77.87%) represent Not Defaulted (0), and 6630 (22.12%) represent Defaulted (1). As a result, the models will be biased towards predicting that someone will not default, since in a worst case, only predicting that someone will not default with this dataset will result in 77.87% accuracy.

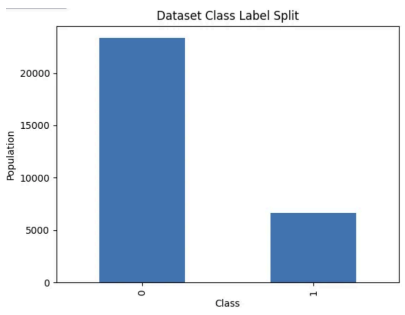


Fig. 5. Population Label Split of Not Defaulted (0) or Defaulted (1). [7]

III. DATA TRANSFORMATION & MODELS USED

A. Feature Scaling

The logistic regression classifier operates with the sigmoid function to calculate its class predictions [2]:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Fig. 6. Sigmoid Function.

The sigmoid function requires its inputs to be scaled between 0 and 1 [2]. Since the dataset is not already scaled this way, a min-max scaler is applied to any data being used as an input to the logistic regression algorithm:

$$X_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Fig. 7. Min-Max Scaler Function.

The random forest ensemble classifier, by nature of being decision-tree-based, is not sensitive to feature scaling [8] and therefore does not require it.

B. One-Hot Encoding for Logistic Regression Classifiers

One-hot encoding is the practice of taking a list of categories in a categorical dataset, assigning an index to each category, then replacing each category with an array of zeros and ones, where every index is a zero except the index that was assigned to the category being replaced [9]. Only the index of the category has a value of one. This is a simple and effective way of making categorical data numeric so it can be used for mathematical operations. For example, if you have categories “High School”, “Bachelors”, and “Grad School”, you would one-hot encode them to [1, 0, 0], [0, 1, 0], and [0, 0, 1] respectively. Since UCI already preprocessed the dataset to be numeric-only for us, doing this with the dataset was not required.

C. Integer Label Encoding for Random Forest Classifiers

Integer label encoding is the practice of simply assigning an integer value to each category in a categorical dataset, then replacing each category with that integer value [10]. This allows the data to be used for mathematical operations. Using the example categories of “High School”, “Bachelors”, and “Grad School” again, you could encode them to 0, 1, and 2 respectively. UCI already preprocessed the dataset in this way for any categorical data, so no additional encoding was required.

D. Logistic Regression Classifier

Logistic regression classification is a supervised learning algorithm, meaning it takes training data with predetermined class labels as training inputs, and uses the sigmoid function (Figure 6) to predict the class of a test sample [2]. Logistic regression is used for binary classification, so it can only be used for two classes, such as “Defaulted”, which was assigned class one, or “Not Defaulted”, which was assigned class zero. The output of the logistic regression algorithm is the probability that the given input is of the class assigned to class one. If the probability is greater than or equal to 0.5, the predicted class is one. Less than 0.5, the predicted class is zero.

Logistic regression starts by scaling the input values to be between zero and one using the min-max scaler function (Figure 7). The argument to the sigmoid function, z, is a linear combination of the input features defined by the following function:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n$$

Fig. 8. Linear Feature Combination Function.

β_0 is the y-intercept, β_1 to β_n are the coefficients to the inputs, and x_1 to x_n are the input features of the test sample. Input z is calculated and given to the sigmoid function to get the probability for the prediction.

Once the predictions are made, they can be used along with truth labels to calculate a loss function, which can then be used to tune the coefficients to achieve better predictions. Tuning a coefficient changes how the feature it applies to influences the predicted probability, so more important features will be tuned to have a higher coefficient and vice versa.

This algorithm benefits from its simplicity, making its implementation easy and calculations light. It suffers most when outliers cause overlap between classes, since the algorithm seeks to cleanly divide the classes

E. Random Forest Ensemble Classifier

Random forest ensemble classifiers create a set of decision trees based on the training input features, then the results of all trees are averaged to form a prediction [3]. The randomness of the algorithm comes from the random selection of the features and data points used to create each tree’s nodes. Multiple trees are generated and trained independently by running training data through and adding nodes until end conditions are met to create prediction leaves [3]. Once trained, the test data is run through each tree based on the respective features they were formed with, and a prediction is made by each tree. The predictions of all trees are then averaged to get one final prediction.

The main benefit of the random forest ensemble classifier is that it works well against overfitting due to its randomization and averaging. This makes it helpful for small datasets where there are few samples to split between training, validation, and testing. The downside is losing the ease of tracing back the predictions relative to a single tree, and the large number of computations required with several trees plus the randomization.

F. Handling of Data Imbalance

77.87% of the samples in the data are of the “Not Defaulted” class, leaving 22.12% as “Defaulted”. To fight against this significant imbalance, the training and test data are divided using the stratified K-fold cross validation algorithm [11]. K-Fold cross validation uses 1/K samples as the testing dataset while the rest are used for training. If K=5, then 20% of the data is used for testing. For each training epoch, a new “fold” is created, using an entirely different portion of data for testing. This allows for a more efficient use of data and prevents overfitting by changing what data is used for training each time.

Stratification being introduced to this method helps to combat the class imbalance. Stratification maintains the percentage of the class representations in each fold to match their representations in the full dataset. Without this there could frequently be too many or too few of one class in the training or test data. If K=5, and stratification is not used, a scenario could easily exist in this dataset where only 2.12% of the “Defaulted” samples are left in the training dataset, leading to an inappropriately large bias toward predicting “Not Defaulted”. Stratification keeps the classes balanced in training and testing so the more represented feature is not overly dominant.

IV. EXPERIMENTS & MODEL RESULTS

A. Logistic Regression Tuning & Evaluation

Logistic regression was performed on the dataset using sklearn’s LogisticRegression class [12]. After thorough experimentation with the hyperparameters available for the class, the maximum average accuracy achieved was 81.96%. The standard deviation of the accuracy across the folds was 0.0036, so it was very consistent in its accuracy. The execution time for this algorithm was as low as 0.059 seconds, while generally ranging from 0.059 to 0.15 seconds.

Several combinations of columns were tested for accuracy, but ultimately the best results were found by classifying based only on the “PAY_1” column. This column has the highest correlation to the class labels overall. Including any other columns with “PAY_1” reduced the performance.

When experimenting with the hyperparameters of the LogisticRegression class, the default parameters did well in achieving the maximum accuracy of 81.96%. Altering the “class_weight” hyperparameter to attempt to further fight the class imbalance resulted in the same or worse performance. The “solver” hyperparameter, which chooses an optimization algorithm to use for training, was tested for each available option: ‘lbfgs’, ‘liblinear’, ‘newton-cg’, ‘newton-cholesky’, ‘sag’, and ‘saga’. None of these achieved higher accuracy than the default (’lbfgs’), but the ‘newton-cholesky’ solver was found to reduce the execution time by about 0.1 seconds while maintaining the accuracy and standard deviation.

B. Random Forest Tuning & Evaluation

Random forest ensemble classification was performed on the dataset using sklearns’s RandomForestClassifier class [12]. The maximum performance achieved with this algorithm was an average accuracy of 81.96%. The standard deviation of the accuracy across the folds was 0.0036, so it was also very consistent. The runtime was as low as 0.049 seconds, ranging from 0.049 to 0.13 seconds.

For this dataset, many different combinations of columns were tested for classification, but the best results were found again using only the “PAY_1” column, which is the column with the highest correlation to the class labels. Once again, any other columns being included with “PAY_1” would reduce the performance.

To achieve the maximum accuracy, many different hyperparameters were experimented with. The performance was the same or worse in most cases when changing the hyperparameters from their defaults. Given the nature of the data, performance was improved by changing two hyperparameters: “max_depth” and “n_estimators”.

The “max_depth” hyperparameter limits the depth of each tree. Limiting this to a depth of only one creates a decision “stump”, which is a tree with only a root node and two leaves. This increased the performance from 81.93% to 81.96%. “PAY_1” simply provides whether someone was behind on payments or not, so it is simple to make a prediction based on that fact alone, which means depth is not required for the trees. The complexity added by deeper trees made performance worse.

The “n_estimators” column limits the number of trees to one. Only one tree is needed because the prediction by the decision tree can be as simple as predicting “Not Defaulted” if payment was up to date or “Defaulted” if the payments were behind when only using the “PAY_1” column as done here. Limiting the tree count to one does not increase accuracy, but it reduces the execution time significantly due to how simple the model becomes. If the original results improved by using more features, then limiting the number of trees in this way would harm performance.

V. CONCLUSION

A. Lessons Learned

In the overall dataset, the majority of the data is not useful for classifying whether someone will default on their next payment. Even the more helpful information in the dataset had low correlation to the classes at a maximum of 0.32. When selecting a dataset, it is very important to consider how the output of the model is determined by *natural* intelligence, meaning how we could make a prediction using all of the data available in the real world. This dataset does not include income or personal expense information, for example, which would be much more useful than anything in this dataset for predicting a default on payment. A different dataset, or adding more information to this dataset, could significantly improve the model's performance.

The importance of cleaning and double-checking the data was seen as well. There were many duplicate rows that were not revealed until redundant or useless information was removed, which would not have been found if duplicates were only checked for at the start of preprocessing.

The models used for this assignment had identical accuracy and standard deviation, with similar execution speeds, but the highest performing implementation of each model made it clear that one is not appropriate for this application. The logistic regression model needed very little to be changed from sklearn's default hyperparameters to achieve its maximum performance. The random forest classifier ended up simpler computationally, and slightly faster, but the algorithm performed best as a single decision stump. Setting the hyperparameters like this defeats the purpose of the random forest algorithm. It is clear that the random forest algorithm is not necessary for classification with this dataset.

B. Mistakes Made, Challenges, & Future Considerations

There are a large number of features, and at first glance many seem like they could be useful. The correlation matrix was critical in quickly and clearly identifying what data was relevant to the goal of the assignment. One look at the correlation matrix made it clear that "PAY_1" alone would give the best results, as was proven by the prediction accuracy data.

Another challenge was identifying how to best set the hyperparameters of the classification algorithms. Weighing the classes seemed like it could help combat the class imbalance, but giving the smaller "Defaulted" class more weight only hurt performance for both algorithms.

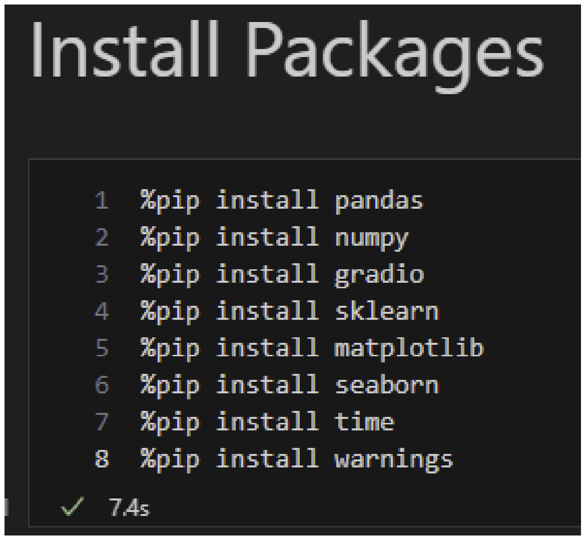
The random forest algorithm seemed to be a good model for this dataset, and it had near maximum performance with default hyperparameter settings, but tuning and exploration revealed the single decision stump to be the best configuration. Without identifying the "PAY_1" column by itself as being best for classification, and without exploring the hyperparameters of the random forest algorithm, it would not have been found that the random forest algorithm is not a good model for this data. Reaching that conclusion had no clear path from the start, and required a good understanding of the relationships in the data, the inner workings of the random forest classifier, and how those relationships in the data affected the random forest classifier's performance.

To improve the prediction performance, more models should be explored to find a true best model for the application. The dataset should also be improved. The given data could only reach less than 82% accuracy, but with more information on income, personal expenses, and other personal social statistics, a much better prediction could be made. With more time, more thorough exploration of hyperparameters could be done to optimize the models. The ranges of each hyperparameter could be fully explored for their effect on performance, and interactions between hyperparameters could be explored.

VI. CODE REVIEW

A. Code Block Summaries

The code for this assignment includes commands to install each package used (Figure 9). They can be removed after being installed if running multiple times.



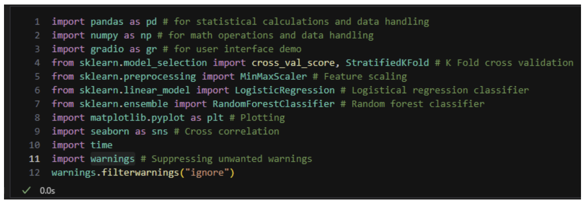
Install Packages

```
%pip install pandas
%pip install numpy
%pip install gradio
%pip install sklearn
%pip install matplotlib
%pip install seaborn
%pip install time
%pip install warnings
```

✓ 7.4s

Fig. 9. "pip" commands to install each Python package used for this assignment.

The next code block imports modules, functions, classes, etc. needed for the assignment (Figure 10). This block also filters out any warnings to remove clutter in any outputs.

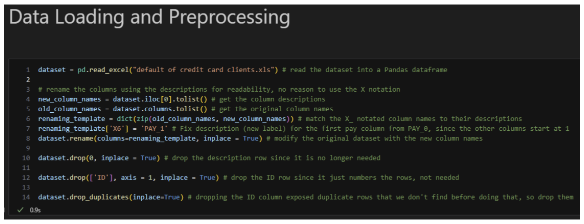


```python
import pandas as pd # for statistical calculations and data handling
import numpy as np # for math operations and data handling
import gradio as gr # for user interface demo
from sklearn.model_selection import cross_val_score, StratifiedKFold # K Fold cross validation
from sklearn.preprocessing import MinMaxScaler # Feature scaling
from sklearn.linear_model import LogisticRegression # Logistical regression classifier
from sklearn.ensemble import RandomForestClassifier # Random forest classifier
import matplotlib.pyplot as plt # Plotting
import seaborn as sns # Cross correlation
import time
import warnings # Suppressing unwanted warnings
warnings.filterwarnings("ignore")
```

```
✓ 0.0s
```

Fig. 10. Code importing all modules, functions, classes, etc.

The "Data Loading and Preprocessing" block (Figure 11) reads the dataset file into the program. It gets the human interpretable column descriptions, matches them to the corresponding "X" labeled column names, then replaces the "X" names with the more easily readable names. The "PAY_0" column is renamed to "PAY_1" to align with the naming scheme of the other columns. After renaming, the description row is dropped since it is a duplicate of the new column names. The "ID" row is dropped since it simply numbers each column sequentially, which is not needed. Lastly, all of these steps revealed duplicate rows of data, so any duplicates are dropped.



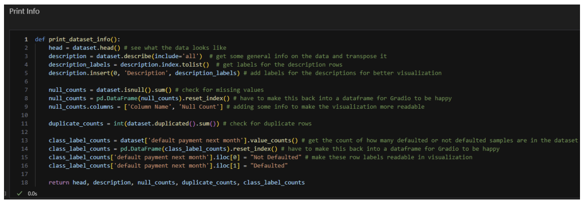
Data Loading and Preprocessing

```python
dataset = pd.read_excel("default of credit card clients.xls") # read the dataset into a Pandas dataframe

# rename the columns using the descriptions for readability, no reason to use the X notation
new_column_names = dataset.iloc[0].tolist() # get the column descriptions
old_column_names = dataset.columns.tolist() # get the original column names
renaming_template = dict(zip(old_column_names, new_column_names)) # match the X_ notated column names to their descriptions
renaming_template['X6'] = 'PAY_1' # Fix description (new label) for the first pay column from PAY_0, since the other columns start at 1
dataset.rename(columns=renaming_template, inplace = True) # modify the original dataset with the new column names

dataset.drop(0, inplace = True) # drop the description row since it is no longer needed

dataset.drop(['ID'], axis = 1, inplace = True) # drop the ID row since it just numbers the rows, not needed

dataset.drop_duplicates(inplace=True) # dropping the ID column exposed duplicate rows that we don't find before doing that, so drop them
```

```
✓ 0.9s
```

Fig. 11. Code to load and preprocess the dataset.

The "Print Info" block (Figure 12) gathers information about the dataset to be displayed as text. The first few rows of data, minimum and maximum values for each column, unique values for each column, a count of the duplicate rows and missing values, and counts of the class label split are gathered. All of this is returned by the function to be displayed by the Gradio application [13].



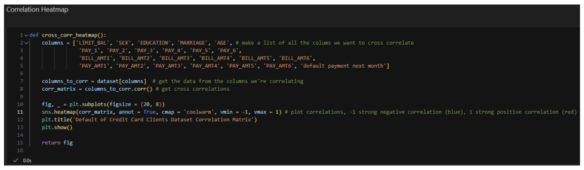
Print Info

```python
def print_dataset_info():
    head = dataset.head() # see what the data looks like
    description = dataset.describe(include='all')  # get some general info on the data and transpose it
    description_labels = description.index.tolist()  # get labels for the description rows
    description.insert(0, 'Description', description_labels) # add labels for the descriptions for better visualization

    null_counts = dataset.isnull().sum() # check for missing values
    null_counts = pd.DataFrame(null_counts).reset_index() # have to make this back into a dataframe for Gradio to be happy
    null_counts.columns = ['Column Name', 'Null Count'] # adding some info to make the visualization more readable

    duplicate_counts = int(dataset.duplicated().sum()) # check for duplicate rows

    class_label_counts = dataset['default payment next month'].value_counts() # get the count of how many defaulted or not defaulted samples are in the dataset
    class_label_counts = pd.DataFrame(class_label_counts).reset_index() # have to make this back into a dataframe for Gradio to be happy
    class_label_counts['default payment next month'].iloc[0] = "Not Defaulted" # make these row labels readable in visualization
    class_label_counts['default payment next month'].iloc[1] = "Defaulted"

    return head, description, null_counts, duplicate_counts, class_label_counts
```

```
✓ 0.0s
```

Fig. 12. Code that extracts text-based information about the dataset.

The "Correlation Heatmap" block (Figure 13) uses seaborn [6] and matplotlib [7] to create a correlation heatmap grid of how each column correlates to each other. All columns are selected, a correlation calculation is run on them, then a figure is made to display the correlations on a range of negative one to one. The figure is returned to be displayed in the Gradio application.



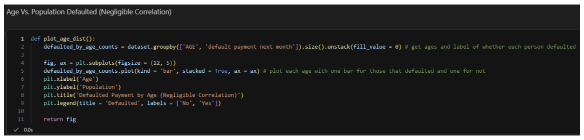
Correlation Heatmap

```python
def cross_corr_heatmap():
    columns = ['LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE', # make a list of all the colums we want to cross correlate
               'PAY_1', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6',
               'BILL_AMT1', 'BILL_AMT2', 'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6',
               'PAY_AMT1', 'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6', 'default payment next month']

    columns_to_corr = dataset[columns]  # get the data from the columns we're correlating
    corr_matrix = columns_to_corr.corr() # get cross correlations

    fig, _ = plt.subplots(figsize = (20, 8))
    sns.heatmap(corr_matrix, annot = True, cmap = 'coolwarm', vmin = -1, vmax = 1) # plot correlations, -1 strong negative correlation (blue), 1 strong positive correlation (red)
    plt.title('Default of Credit Card Clients Dataset Correlation Matrix')
    plt.show()

    return fig

```

```
✓ 0.0s
```

Fig. 13. Code that creates a correlation heatmap between every column of the dataset.

The "Age Vs. Population Defaulted (Negligible Correlation)" block (Figure 14) creates a plot of the age distribution, with bars representing how many people of each age did or did not default. The number of people of each age that did or did not default are counted first. A bar plot is made and the bars for each class label are laid on top of each other for better visualization. The legend labels the classes as "Yes" for the group that defaulted and "No" for those that did not. These labels are used throughout the other plots as well. The figure is returned by the function for display in the Gradio application.



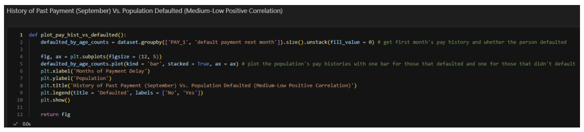
Age Vs. Population Defaulted (Negligible Correlation)

```python
def plot_age_dist():
    defaulted_by_age_counts = dataset.groupby(['AGE', 'default payment next month']).size().unstack(fill_value = 0) # get ages and label of whether each person defaulted

    fig, ax = plt.subplots(figsize = (12, 5))
    defaulted_by_age_counts.plot(kind = 'bar', stacked = True, ax = ax) # plot each age with one bar for those that defaulted and one for not
    plt.xlabel('Age')
    plt.ylabel('Population')
    plt.title('Defaulted Payment by Age (Negligible Correlation)')
    plt.legend(title = 'Defaulted', labels = ['No', 'Yes'])

    return fig
```

```
✓ 0.0s
```

Fig. 14. Code that creates a plot of population defaulted or not defaulted at each represented age.

The "History of Past Payment (September) Vs. Population Defaulted (Medium-Low Positive Correlation)" block (Figure 15) plots how many months behind or ahead each person was on their credit card payments during the first month of collected data and how many people did or did end up defaulting for each increment. We get the counts for each increment of months behind or ahead and how many did or did not default for each. They are also plotted as bars for each class label overlaid onto each other at each monthly increment. The figure is returned for the Gradio application to display.

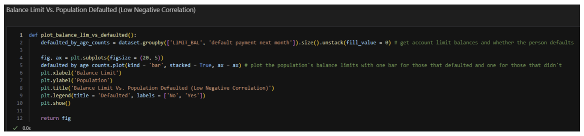


History of Past Payment (September) Vs. Population Defaulted (Medium-Low Positive Correlation)

```python
def plot_pay_hist_vs_defaulted():
    defaulted_by_age_counts = dataset.groupby(['PAY_1', 'default payment next month']).size().unstack(fill_value = 0) # get first month's pay history and whether the person defaulted

    fig, ax = plt.subplots(figsize = (12, 5))
    defaulted_by_age_counts.plot(kind = 'bar', stacked = True, ax = ax) # plot the population's pay histories with one bar for those that defaulted and one for those that didn't default
    plt.xlabel('Months of Payment Delay')
    plt.ylabel('Population')
    plt.title('History of Past Payment (September) Vs. Population Defaulted (Medium-Low Positive Correlation)')
    plt.legend(title = 'Defaulted', labels = ['No', 'Yes'])
    plt.show()

    return fig
```
✓ 0.0s

Fig. 15. Code that creates a plot of the population defaulted or not defaulted for each month ahead or behind on credit card payments.

The "Balance Limit Vs. Population Defaulted (Low Negative Correlation)" block (Figure 16) plots the credit limit of the population against whether they defaulted or not. For each unique credit limit value, the number of people that did or did not default is counted. They are plotted as bar graphs, with the bar for each class label overlaid on each other at each unique credit limit value. The plot is returned to be displayed in the Gradio application.

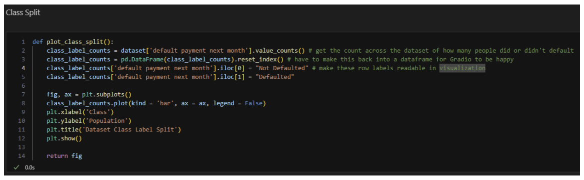


Balance Limit Vs. Population Defaulted (Low Negative Correlation)

```python
def plot_balance_lim_vs_defaulted():
    defaulted_by_age_counts = dataset.groupby(['LIMIT_BAL', 'default payment next month']).size().unstack(fill_value = 0) # get account limit balances and whether the person defaults

    fig, ax = plt.subplots(figsize = (20, 5))
    defaulted_by_age_counts.plot(kind = 'bar', stacked = True, ax = ax) # plot the population's balance limits with one bar for those that defaulted and one for those that didn't
    plt.xlabel('Balance Limit')
    plt.ylabel('Population')
    plt.title('Balance Limit Vs. Population Defaulted (Low Negative Correlation)')
    plt.legend(title = 'Defaulted', labels = ['No', 'Yes'])
    plt.show()

    return fig
```
✓ 0.0s

Fig. 16. Code that creates a plot of the population defaulted or not defaulted for each unique credit limit value.

The "Class Split" block (Figure 17) counts the overall class split of the dataset and plots them as bars side by side. The labels are counted, then the columns are renamed from "0" and "1" to "Not Defaulted" and "Defaulted" respectively for readability. The figure is returned for display by the Gradio application.

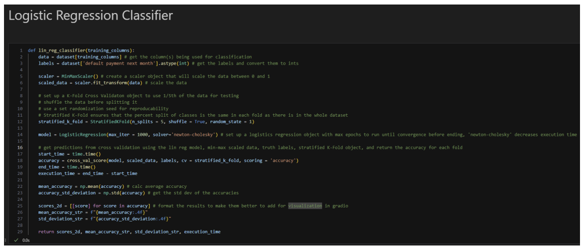


Class Split

```python
def plot_class_split():
    class_label_counts = dataset['default payment next month'].value_counts() # get the count across the dataset of how many people did or didn't default
    class_label_counts = pd.DataFrame(class_label_counts).reset_index() # have to make this back into a dataframe for Gradio to be happy
    class_label_counts['default payment next month'].iloc[0] = "Not Defaulted" # make these row labels readable in visualization
    class_label_counts['default payment next month'].iloc[1] = "Defaulted"

    fig, ax = plt.subplots()
    class_label_counts.plot(kind = 'bar', ax = ax, legend = False)
    plt.xlabel('Class')
    plt.ylabel('Population')
    plt.title('Dataset Class Label Split')
    plt.show()

    return fig
```
✓ 0.0s

Fig. 17. Code that creates a plot of the class label split of the entire dataset.

After the data visualization section the logistic regression model is created and tested using sklearn [12] (Figure 18). The function takes one or more column names as an argument to use for training. The data from those columns are retrieved along with the class labels. A MinMaxScaler object is made and the data is scaled with it. A StratifiedKFold object is configured for five folds, shuffling to randomize the data with each new fold, and a set randomization seed. A LogisticRegression object is configured for a maximum of one-thousand training iterations using Newton-Cholesky decomposition. Cross-validation is run on the model for each fold and the execution time is calculated. The mean and standard deviation of the prediction accuracy are calculated and formatted for easy display in the Gradio application. The accuracies for each fold, mean and standard deviation for the accuracies, and the execution time of the cross-validation are returned for display in the Gradio application.

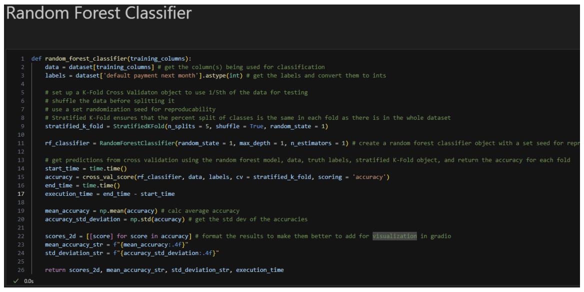


Logistic Regression Classifier

```python
def lin_reg_classifier(training_columns):
    data = dataset[training_columns] # get the column(s) being used for classification
    labels = dataset['default payment next month'].astype(int) # get the labels and convert them to ints

    scaler = MinMaxScaler() # create a scaler object that will scale the data between 0 and 1
    scaled_data = scaler.fit_transform(data) # scale the data

    # set up a K-Fold Cross Validaton object to use 1/5th of the data for testing
    # shuffle the data before splitting it
    # use a set randomization seed for reproducability
    # Stratified K-Fold ensures that the percent split of classes is the same in each fold as there is in the whole dataset
    stratified_k_fold = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 1)

    model = LogisticRegression(max_iter = 1000, solver='newton-cholesky') # set up a logistics regression object with max epochs to run until convergence before ending, 'newton-cholesky' decreases execution time

    # get predictions from cross validation using the lin reg model, min-max scaled data, truth labels, stratified K-Fold object, and return the accuracy for each fold
    start_time = time.time()
    accuracy = cross_val_score(model, scaled_data, labels, cv = stratified_k_fold, scoring = 'accuracy')
    end_time = time.time()
    execution_time = end_time - start_time

    mean_accuracy = np.mean(accuracy) # calc average accuracy
    accuracy_std_deviation = np.std(accuracy) # get the std dev of the accuracies

    scores_2d = [[score] for score in accuracy] # format the results to make them better to add for visualization in gradio
    mean_accuracy_str = f"{mean_accuracy:.4f}"
    std_deviation_str = f"{accuracy_std_deviation:.4f}"

    return scores_2d, mean_accuracy_str, std_deviation_str, execution_time
```
✓ 0.0s

Fig. 18. Code to initialize, run, and validate a logistic regression classification algorithm.

Next, the random forest classifier model is trained and tested using sklearn [12] (Figure 19). The function takes one or more column names as an argument to be used for training and testing. The data is retrieved along with the class labels. A StratifiedKFold object is configured for five folds, shuffling to randomize the data after each new fold, and a set randomization seed. A RandomForestClassifier object is configured with a set randomization seed, a maximum tree depth of one, and the number of trees set at one. This effectively creates a single decision stump, which achieves the highest performance for this algorithm on this dataset. Cross-validation is run on this model with the chosen data and the execution time is calculated. The mean and standard deviation of the prediction accuracies at each fold are calculated. The accuracy of each fold, mean and standard deviation of the accuracies, and the execution time are returned to be displayed in the Gradio application.

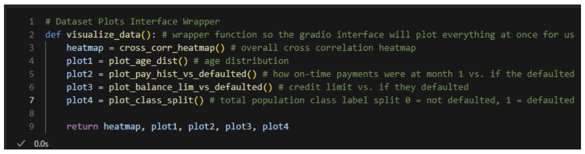


Random Forest Classifier

```python
def random_forest_classifier(training_columns):
    data = dataset[training_columns] # get the column(s) being used for classification
    labels = dataset['default payment next month'].astype(int) # get the labels and convert them to ints

    # set up a K-Fold Cross Validaton object to use 1/5th of the data for testing
    # shuffle the data before splitting it
    # use a set randomization seed for reproducability
    # Stratified K-Fold ensures that the percent split of classes is the same in each fold as there is in the whole dataset
    stratified_k_fold = StratifiedKFold(n_splits = 5, shuffle = True, random_state = 1)

    rf_classifier = RandomForestClassifier(random_state = 1, max_depth = 1, n_estimators = 1) # create a random forest classifier object with a set seed for repr

    # get predictions from cross validation using the random forest model, data, truth labels, stratified K-Fold object, and return the accuracy for each fold
    start_time = time.time()
    accuracy = cross_val_score(rf_classifier, data, labels, cv = stratified_k_fold, scoring = 'accuracy')
    end_time = time.time()
    execution_time = end_time - start_time

    mean_accuracy = np.mean(accuracy) # calc average accuracy
    accuracy_std_deviation = np.std(accuracy) # get the std dev of the accuracies

    scores_2d = [[score] for score in accuracy] # format the results to make them better to add for visualization in gradio
    mean_accuracy_str = f"{mean_accuracy:.4f}"
    std_deviation_str = f"{accuracy_std_deviation:.4f}"

    return scores_2d, mean_accuracy_str, std_deviation_str, execution_time
```
✓ 0.0s

Fig. 19. Code to initialize, run, and validate a random forest classification algorithm.

To display all of this data, a Gradio application was created. First, a wrapper function needed to be made that would retrieve all of the plot and heatmap figures since each tab in the Gradio user interface will only display the returned data of one function call (Figure 20).



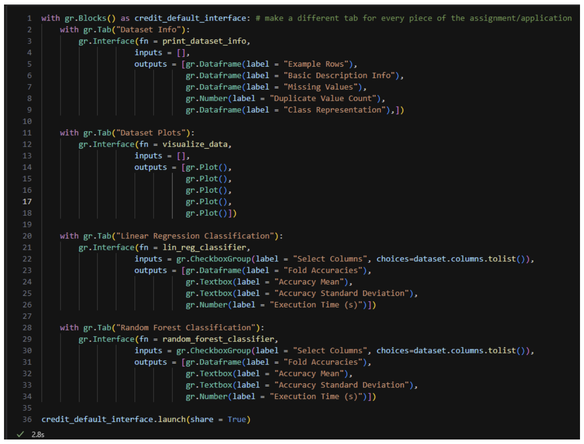
```python
# Dataset Plots Interface Wrapper
def visualize_data(): # wrapper function so the gradio interface will plot everything at once for us
    heatmap = cross_corr_heatmap() # overall cross correlation heatmap
    plot1 = plot_age_dist() # age distribution
    plot2 = plot_pay_hist_vs_defaulted() # how on-time payments were at month 1 vs. if the defaulted
    plot3 = plot_balance_lim_vs_defaulted() # credit limit vs. if they defaulted
    plot4 = plot_class_split() # total population class label split 0 = not defaulted, 1 = defaulted

    return heatmap, plot1, plot2, plot3, plot4
```

```
✓ 0.0s
```

Fig. 20. A wrapper function that calls each plot function and returns all five plots.

Finally, the Gradio application is created (Figure 21). gr.Blocks() is used to create a set of tabs for each element of the assignment. There is a tab for the text-based dataset information, another for the heatmap and all plots, another for the logistic regression classifier, and one for the random forest classifier. The data and plot tabs do not take any inputs and place the returned data of their respective functions in Dataframe, Number, or Plot displays.

The classifier tabs take any combination of column names as an input from a set of checkboxes for each column. Each checked column will be used for prediction. They both output the aforementioned performance data, and that data is shown in Dataframe, Textbox, or Number displays.



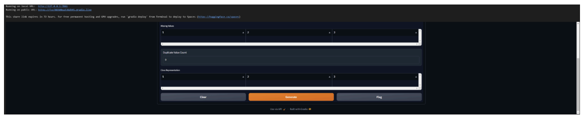
```python
with gr.Blocks() as credit_default_interface: # make a different tab for every piece of the assignment/application
    with gr.Tab("Dataset Info"):
        gr.Interface(fn = print_dataset_info,
                     inputs = [],
                     outputs = [gr.Dataframe(label = "Example Rows"),
                                gr.Dataframe(label = "Basic Description Info"),
                                gr.Dataframe(label = "Missing Values"),
                                gr.Number(label = "Duplicate Value Count"),
                                gr.Dataframe(label = "Class Representation"),])

    with gr.Tab("Dataset Plots"):
        gr.Interface(fn = visualize_data,
                     inputs = [],
                     outputs = [gr.Plot(),
                                gr.Plot(),
                                gr.Plot(),
                                gr.Plot(),
                                gr.Plot()])

    with gr.Tab("Linear Regression Classification"):
        gr.Interface(fn = lin_reg_classifier,
                     inputs = gr.CheckboxGroup(label = "Select Columns", choices=dataset.columns.tolist()),
                     outputs = [gr.Dataframe(label = "Fold Accuracies"),
                                gr.Textbox(label = "Accuracy Mean"),
                                gr.Textbox(label = "Accuracy Standard Deviation"),
                                gr.Number(label = "Execution Time (s)")])

    with gr.Tab("Random Forest Classification"):
        gr.Interface(fn = random_forest_classifier,
                     inputs = gr.CheckboxGroup(label = "Select Columns", choices=dataset.columns.tolist()),
                     outputs = [gr.Dataframe(label = "Fold Accuracies"),
                                gr.Textbox(label = "Accuracy Mean"),
                                gr.Textbox(label = "Accuracy Standard Deviation"),
                                gr.Number(label = "Execution Time (s)")])

credit_default_interface.launch(share = True)
```

```
✓ 2.8s
```

Fig. 21. Code that creates a Gradio application interface with tabs for text info display, plots, and algorithm testing.

B. Gradio Application Use & Evaluation

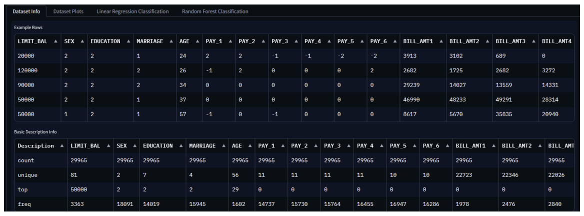
To evaluate the dataset information, visualizations, and performance, the Gradio application is used (Figure 22-29). Jupyter Notebook displays the application, but one of the URL's can be used to open a browser window to interact with the application as well.



```
Running on local URL:  http://127.0.0.1:7861
Running on public URL: [illegible]

This share link expires in 72 hours. For free permanent hosting and GPU upgrades, run `gradio deploy` from Terminal to deploy to Spaces (https://huggingface.co/spaces)
```

Fig. 22. Jupyter Notebook Gradio application.

In the "Dataset Info" and "Dataset Plots" tabs (Figure 23-27), clicking "Generate" will display all of the data described previously. Tables can be scrolled through if needed. Clicking "Clear" will remove the displayed data.



Dataset Info
Dataset Plots
Linear Regression Classification
Random Forest Classification

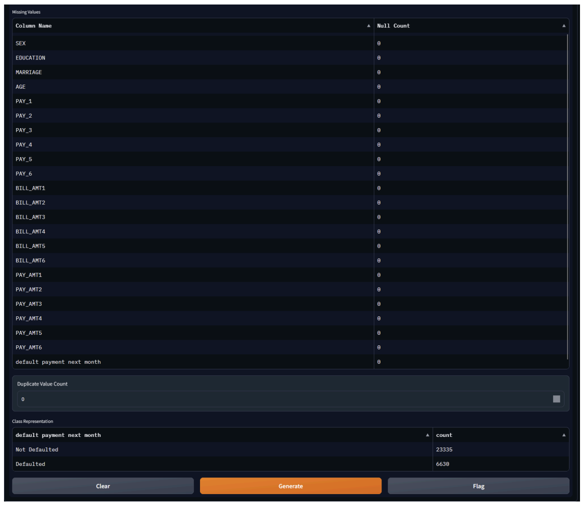
Example Rows

LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_1	PAY_2	PAY_3	PAY_4	PAY_5	PAY_6	BILL_AMT1	BILL_AMT2	BILL_AMT3	BILL_AMT4
20000	2	2	1	24	2	2	-1	-1	-2	-2	3913	3102	689	0
120000	2	2	2	26	-1	2	0	0	0	2	2682	1725	2682	3272
90000	2	2	2	34	0	0	0	0	0	0	29239	14027	13559	14331
50000	2	2	1	37	0	0	0	0	0	0	46990	48233	49291	28314
50000	1	2	1	57	-1	0	-1	0	0	0	8617	5670	35835	20940

Basic Description Info

Description	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_1	PAY_2	PAY_3	PAY_4	PAY_5	PAY_6	BILL_AMT1	BILL_AMT2	BILL_AMT
count	29965	29965	29965	29965	29965	29965	29965	29965	29965	29965	29965	29965	29965	29965
unique	81	2	7	4	56	11	11	11	11	10	10	22723	22346	22026
top	50000	2	2	2	29	0	0	0	0	0	0	0	0	0
freq	3363	18091	14019	15945	1602	14737	15730	15764	16455	16947	16286	1978	2476	2840

Fig. 23. Example dataset rows and statistics about the dataset.



Missing Values

Column Name	Null Count
SEX	0
EDUCATION	0
MARRIAGE	0
AGE	0
PAY_1	0
PAY_2	0
PAY_3	0
PAY_4	0
PAY_5	0
PAY_6	0
BILL_AMT1	0
BILL_AMT2	0
BILL_AMT3	0
BILL_AMT4	0
BILL_AMT5	0
BILL_AMT6	0
PAY_AMT1	0
PAY_AMT2	0
PAY_AMT3	0
PAY_AMT4	0
PAY_AMT5	0
PAY_AMT6	0
default payment next month	0

Duplicate Value Count: 0

Class Representation

default payment next month	count
Not Defaulted	23335
Defaulted	6630

Fig. 24. Counts of missing values for each column, duplicate rows, and the overall class split of the dataset.

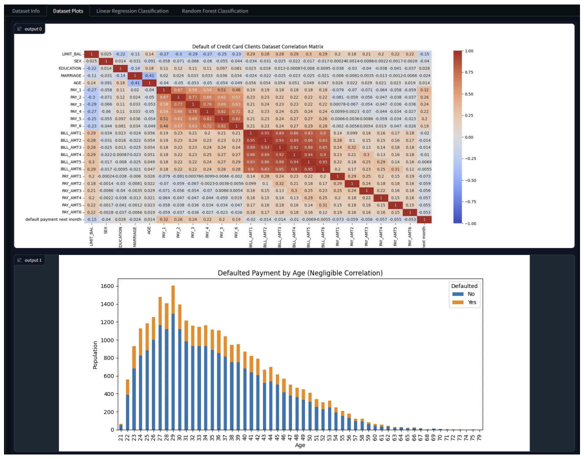


Fig. 25. Cross-correlation heatmap and plot of class labels by age.

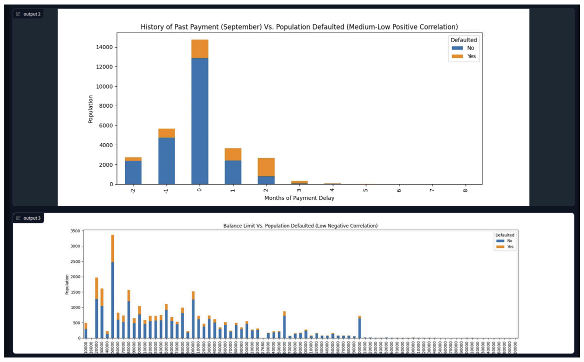


Fig. 26. Plots of class labels versus history of past payment and balance limit, respectively.

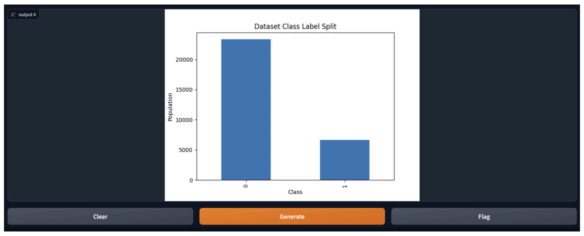
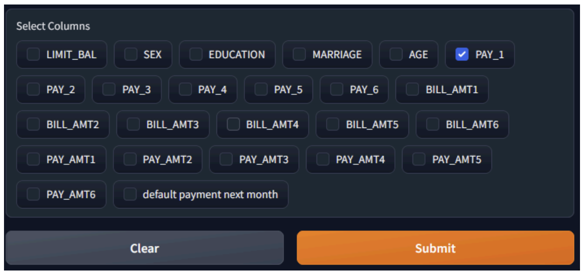


Fig. 27. Plot of the overall class label split of the dataset.

The "Linear Regression Classification" and "Random Forest Classification" tabs (Figure 28-29) work identically. Columns can be selected to be used for predictions by checking boxes next to each column name. Click "Submit" to run the algorithm on the selected columns, or "Clear" to clear your selections (Figure 28). To see optimal performance, select only the "PAY_1" column for both algorithms.

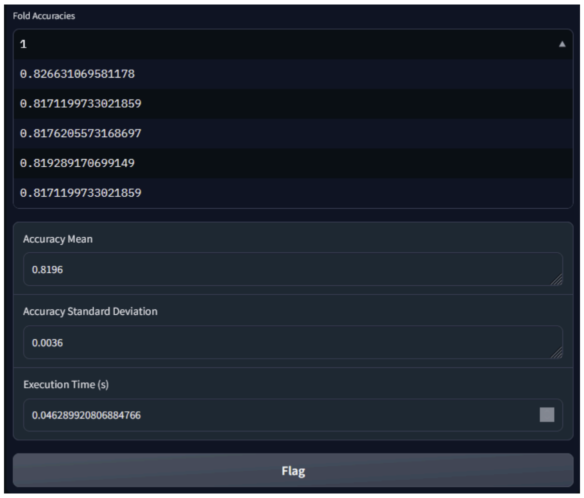


Select Columns

☐ LIMIT_BAL ☐ SEX ☐ EDUCATION ☐ MARRIAGE ☐ AGE ☑ PAY_1
☐ PAY_2 ☐ PAY_3 ☐ PAY_4 ☐ PAY_5 ☐ PAY_6 ☐ BILL_AMT1
☐ BILL_AMT2 ☐ BILL_AMT3 ☐ BILL_AMT4 ☐ BILL_AMT5 ☐ BILL_AMT6
☐ PAY_AMT1 ☐ PAY_AMT2 ☐ PAY_AMT3 ☐ PAY_AMT4 ☐ PAY_AMT5
☐ PAY_AMT6 ☐ default payment next month

Fig. 28. Gradio column checkboxes for selecting data to run predictions with.

After clicking "Submit", the performance results for the given algorithm are displayed (Figure 29). "Fold Accuracies" displays the accuracy of each respective fold, five total. The average accuracy and standard deviation across all folds are displayed as well, along with how long the cross-validation took to execute. For both algorithms, the highest "Accuracy Mean" is 0.8196, the lowest "Accuracy Standard Deviation" is 0.0036, and the execution time varies depending on the machine running the code. Submit only the "PAY_1" column for both algorithms to achieve the maximum performance.



Fold Accuracies

1
0.826631069581178
0.8171199733021859
0.8176205573168697
0.819289170699149
0.8171199733021859

Accuracy Mean
0.8196

Accuracy Standard Deviation
0.0036

Execution Time (s)
0.046289920806884766

Fig. 29. Gradio display of classification algorithm results.

REFERENCES

[1] I. Yeh. "Default of Credit Card Clients," UCI Machine Learning Repository, 2009. [Online]. Available: https://doi.org/10.24432/C55S3H.

[2] GeeksforGeeks, "Understanding Logistic Regression," GeeksforGeeks, May 09, 2024. https://www.geeksforgeeks.org/understanding-logistic-regression/

[3] Wikipedia Contributors, "Random forest," *Wikipedia*, Apr. 09, 2019. https://en.wikipedia.org/wiki/Random_forest

[4] J. Brownlee, "A Gentle Introduction to Imbalanced Classification," *Machine Learning Mastery*, Dec. 22, 2019. https://machinelearningmastery.com/what-is-imbalanced-classification/

[5] K. N. Jarapala, "Categorical Data Encoding Techniques," *AI Skunks*, Mar. 27, 2023. https://medium.com/aiskunks/categorical-data-encoding-techniques-d6296697a40f

[6] M. Waskom, "seaborn: statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, Apr. 2021, Available:

[7] J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.https://joss.theoj.org/papers/10.21105/joss.03021

[8] D. Team, "Decision Trees: An Overview and Practical Guide," DataHeroes, Sep. 20, 2023. https://dataheroes.ai/blog/decision-trees-an-overview-and-practical-guide/

[9] L. Ganji, "One Hot Encoding in Machine Learning," GeeksforGeeks, Jun. 12, 2019. https://www.geeksforgeeks.org/ml-one-hot-encoding/

[10] aakarsha chugh, "ML | Label Encoding of datasets in Python," GeeksforGeeks, Oct. 15, 2018. https://www.geeksforgeeks.org/ml-label-encoding-of-datasets-in-python/

[11] "Stratified K Fold Cross Validation," GeeksforGeeks, Aug. 06, 2020. https://www.geeksforgeeks.org/stratified-k-fold-cross-validation/

[12] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *J. Mach. Learn. Res.*, vol. 12, 2011, doi: https://doi.org/10.5555/1953048.2078195.

[13] G. Team, "Gradio Documentation," *www.gradio.app*. https://www.gradio.app/docs

[14] "3.8.20 Documentation," *Python.org*, 2024. https://docs.python.org/3.8/index.html (accessed Oct. 13, 2024).

[15] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. *Array programming with NumPy*. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2.

[16] "API reference — pandas 1.3.3 documentation," pandas.pydata.org. https://pandas.pydata.org/docs/reference/index.html#api