Credit Limit and Default Prediction in Machine Learning with Business Focus

This report is written on Overleaf
Total word count: 3730 (Shown on the Overleaf)
Total page count: 13 pages

I acknowledge using Copilot (Github, https://github.com/features/copilot) to assist with the part of the code; I have put the reference inside the code comments to make a difference.
I acknowledge using Grammarly (Grammarly, https://app.grammarly.com/) to assist my writing in proofreading.

Credit Limit and Default Prediction in Machine Learning with business focus

I. Introduction

A. Background

Banking is one of the riskiest and most volatile sectors of the national economy, particularly concerning bank loans [1]. According to UK credit statistics, in 2022, 83% of the total UK population used a form of credit or loan product [2]. However, this has been followed by an increase in default rates of 27.2%, the highest recorded figure since Q3 2009 [2]. It becomes crucial for credit companies and banks to identify potential default users to mitigate risks effectively. In addition to managing risks with high default rates, credit companies also face the "cold start" problem, which can be described as the challenge of providing credit limits to individuals who have no credit history.

B. Problem statement

In this study, we aim to address two tasks. The first task involves predicting the likelihood of customers defaulting on their credit card payments. We applied classification algorithms such as Logistic Regression (LR), Support Vector Classifier (SVC), and Random Forest Classifier (RFC).

The second task focuses on predicting credit limits for new users who lack historical credit data. We applied Elastic net linear regression (ElasticNet), Random Forest Regression (RFR), and Artificial Neural Networks (ANN).

We also went beyond traditional performance metrics to introduce new measures that align closely with real-world business values, which are designed to provide a more business-oriented view of model performance.

II. Methodology

Our first step is the data exploration. This includes examining the data distributions, observing data imbalance, and analysing the correlations to discover the relationship between target labels and features.

After gathering a better understanding of the dataset, we did data cleaning to remove outliers, redirected the values, and filled in the missing values based on reasonable assumptions.

Following this, we engaged in feature engineering, which includes creating new features tailored for each task, transforming existing features, scaling data and applying Principal Component Analysis (PCA) for dimensionality reduction. However, given that we are dealing with different tasks with different methodologies and features, we decided to make a more customised approach to each task.

Finally, we trained our predictive models and carefully tuned them to achieve optimal performance. This includes adjusting model hyper-parameters with cross-validation and measuring our models based on performance metrics.

A. Data exploration

The dataset contains 30,000 instances distributed over 25 variables. It contains four features for regression tasks and 23 features suitable for classification. We excluded the 'ID' column as it is too generic. The dataset includes four categorical columns; details have been explained in Table I. One notable observation is data imbalance, particularly in the 'Default payment next month' column, where instances of non-default exceed those of default by 56%. Additionally, the dataset contains missing values in columns like 'Education' and 'Marriage.'

TABLE I: Categorical Column Details

Column Name	Category	Total Instances	Percentage (%)
Sex	1	11,888	40
	2	18,112	60
Education	1	10,585	35
	2	14,030	47
	3	4,917	16
	4	123	0.4
	Out of range	345	1.6
Default next month	0	23,364	78
	1	6,636	22
Marriage	1	13,659	46
	2	15,964	53
	3	323	10
	Out of range	54	1

Below is the code, this belongs to class $Data$:

Checking categories

```
def check_number_of_categories(self, col_name):
    count_category = df.groupby(col_name).size()
    pert_category = df.groupby(col_name).size()/len(
     df)
    return count_category, pert_category
```

We then analyzed the data distribution (Figure 1); it suggests that most of the data is not balanced and does not follow the Gaussian distribution. We then measure distribution when default with education, sex and marital status, from Figure 2; it suggests that younger age groups have a higher likelihood of default. Regarding education, individuals with a university background show a higher likelihood of default than other groups. On the other hand, sex and marital status appear to have a minimal impact on the likelihood of default.

Checking distribution

```
def data_distribution(self, col_name=None,
     target_name="default payment next month"):
    if col_name == None:
        self.data.hist(figsize=(20, 20))
    else:
        df[df["default payment next month"]==1][
     col_name].hist()
    return None
```

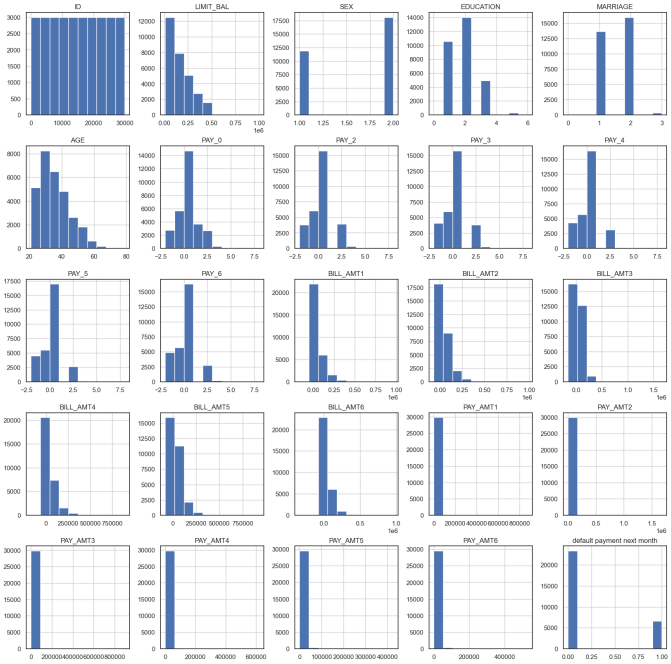


Fig. 1: Distribution of the data

Below is the code for measuring distribution, which belongs to class *Data*:

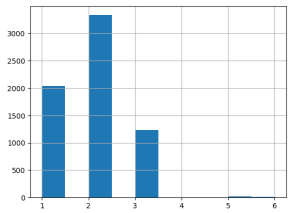
Next, we examined the skewness of the dataset. As indicated in Table II, 19 out of 24 instances show positive skewness, suggesting that most data are right-skewed. It is important to note that there is a significantly high skewness value in the 'PAY AMT' columns. Interestingly, these values first increase and then gradually decrease, showing in Figure 3, implying that most individuals tend to pay smaller amounts initially but larger amounts towards the end.

Below is the code about measuring skewness, which is from class *Data*:

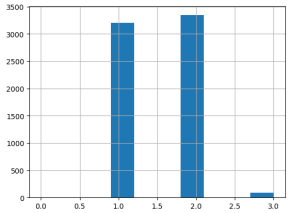
Checking skewness

```
# This code can be called from outside
def check_skewness(self):
    self.data.skew()
    return None
```

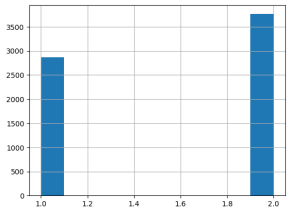
For the regression task, we have examined the 'LIM BAL' label in more detail. The range of values extends from 20,000 to 730,000, with a maximum value of 1,000,000. The mean



(a) Distribution of Education when Default



(b) Distribution of Marriage when Default



(c) Distribution of Sex when Default

Fig. 2: Distribution when Default

TABLE II: Skewness of Each Column in the Dataset

Column Name	Skewness
LIMIT BAL	0.993
SEX	-0.424
EDUCATION	0.971
MARRIAGE	-0.019
AGE	0.732
PAY 0	0.732
PAY 2	0.791
PAY 3	0.841
PAY 4	1.000
PAY 5	1.008
PAY 6	0.948
BILL AMT1	2.664
BILL AMT2	2.705
BILL AMT3	3.088
BILL AMT4	2.822
BILL AMT5	2.876
BILL AMT6	2.847
PAY AMT1	14.668
PAY AMT2	30.454
PAY AMT3	17.217
PAY AMT4	12.905
PAY AMT5	11.127
PAY AMT6	10.641
Default Payment Next Month	1.344

is calculated as 167,484.32, and the standard deviation is 129,747.66, which suggests the presence of some outliers. We observed that some credits have fewer than 30 individuals; these are treated as outliers. Additionally, we also noted that the values in 'LIM BAL' are rounded to two significant figures, indicating that the final results should be presented in the same format for consistency.

Lastly, we analyzed the correlation matrix, as shown in Figure 4. Some features correlate strongly with the target label. However, features in regression tasks do not show such a strong correlation. It is also worth mentioning that features such as 'PAY 0' to 'PAY 6', 'BILL AMT0' to 'BILL AMT6', and 'PAY AMT0' to 'PAY AMT6' are highly correlated with

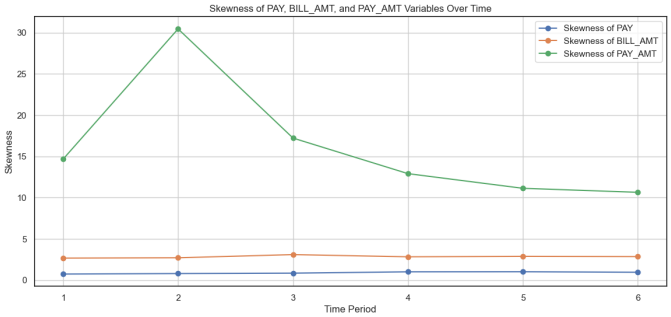


Fig. 3: Skewness over time

Measure skewness when default

```
# This code with assistance from GitHub Copilot
skewness_dict = {
    'PAY': df[['PAY_0', 'PAY_2', 'PAY_3', 'PAY_4', '
     PAY_5', 'PAY_6']].skew().values,
    'BILL_AMT': df[['BILL_AMT1', 'BILL_AMT2', '
     BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6'
     ]].skew().values,
    'PAY_AMT': df[['PAY_AMT1', 'PAY_AMT2', 'PAY_AMT3'
     , 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6']].skew().
     values
}
plt.figure(figsize=(14, 6))
time_periods = [1, 2, 3, 4, 5, 6]
for feature, skewness in skewness_dict.items():
    plt.plot(time_periods, skewness, marker='o',
     label=f'Skewness of {feature}')
plt.legend()
plt.title('Skewness of PAY, BILL_AMT, and PAY_AMT
     Variables Over Time')
plt.xlabel('Time Period')
plt.ylabel('Skewness')
plt.xticks(time_periods)
plt.grid(True)
plt.show()
```

Check outliers

```
# We use this code to identify if there are outliers
     such as "LIMIT_BAL" only having fewer than 30
     individuals. This function can be called from
     outside.
def check_outliers(self,col_name:str):
    data_info = self.data[col_name].describe()
    unique_value = df[col_name].unique()
    count_unique_value = df["LIMIT_BAL"].value_counts
     ().sort_values(ascending=True)
    return data_info, unique_value,
     count_unique_value
```

each other. Such high inter-correlations among these features can lead to multicollinearity, which could cause overfitting [3].

B. Data cleaning

After completing the data analysis and exploration, we cleaned the data for further analysis. Firstly, we renamed columns, particularly those labelled as 'PAY', 'BILL AMT', and 'PAY AMT'. The original and new names of these columns are shown in Table III. Next, we addressed data that fell outside the expected range, as outlined in Table I. We assumed that unclear or out-of-range details provided by users

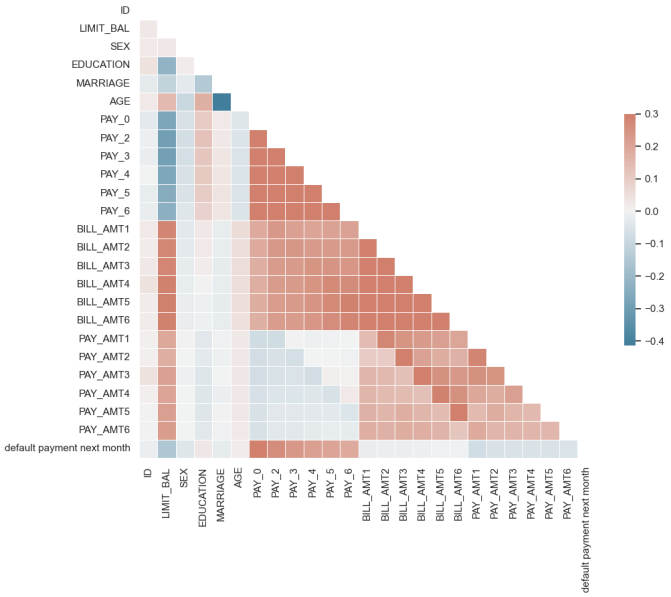


Fig. 4: Correlation Matrix

Measure correlation Matrix

```
# This function can be called from outside:
def correlation(self):
    # reference: Lab resource, CreditCardFraud.ipynb
    sns.set_theme(style="white")
    corr = self.data.corr()
    mask = np.triu(np.ones_like(corr, dtype=bool))
    f, ax = plt.subplots(figsize=(11, 9))
    cmap = sns.diverging_palette(230, 20, as_cmap=
     True)
    sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3,
     center=0,
                square=True, linewidths=.5, cbar_kws
     ={"shrink": .5})
    return None
```

should be treated as the lowest level for prediction purposes. Therefore, we assigned these out-of-range entries a value of -1. After dealing with missing or out-of-range values, we removed outliers in the 'LIM BAL' column. As discussed in the previous section, we removed instances where credit balances had fewer than 30 individuals. We then used the Interquartile Range (IQR) method for general outlier detection and handling, followed by the idea from [4].

The reason for selecting the IQR method is based on our observations from data exploration, where we noted that the features do not follow a Gaussian distribution and IQR can be an effective statistical tool for summarizing data samples that follow non-Gaussian distributions [4].

The IQR method identifies outliers by calculating the interquartile range, shown in Equation 1, where Q1 is defined as the 25th percentile, and Q3 is the 75th percentile [4].

$$\text{IQR} = Q_3 - Q_1 \tag{1}$$

Then, we defined outliers as values below Q_l or above Q_h, which has been shown in equations 2 and 3 [4].

TABLE III: Rename the column

Original column name	New column name
ID	ID
LIMIT_BAL	LIMIT_BAL
SEX	SEX
EDUCATION	EDUCATION
MARRIAGE	MARRIAGE
AGE	AGE
PAY_0	Repayment_Status_Sept
PAY_2	Repayment_Status_Aug
PAY_3	Repayment_Status_Jul
PAY_4	Repayment_Status_Jun
PAY_5	Repayment_Status_May
PAY_6	Repayment_Status_Apr
BILL_AMT1	Bill_Amount_Sept
BILL_AMT2	Bill_Amount_Aug
BILL_AMT3	Bill_Amount_Jul
BILL_AMT4	Bill_Amount_Jun
BILL_AMT5	Bill_Amount_May
BILL_AMT6	Bill_Amount_Apr
PAY_AMT1	Pay_Amount_Sept
PAY_AMT2	Pay_Amount_Aug
PAY_AMT3	Pay_Amount_Jul
PAY_AMT4	Pay_Amount_Jun
PAY_AMT5	Pay_Amount_May
PAY_AMT6	Pay_Amount_Apr
Default Payment Next Month	Default_Payment

$$Q_l = Q1 - 1.5 \times \text{IQR} \tag{2}$$

$$Q_h = Q3 + 1.5 \times \text{IQR} \tag{3}$$

Overall, the data cleaning reduced the data size from 30.000 to 29,791.

Handle outliers

```python
# This function needs to be called from insider the class.
# It connects to the function: _data_cleaning
def _reg_handle_outliers(self, col_name:str) -> pd.DataFrame:
    # Code reference: https://www.analyticsvidhya.com/blog/2022/09/dealing-with-outliers-using-the-iqr-method/
        Q1 = self.data[col_name].quantile(0.25)
        Q3 = self.data[col_name].quantile(0.75)
        IQR = Q3 - Q1
        lower_limit = Q1 - 1.5 * IQR
        upper_limit = Q3 + 1.5 * IQR
        self.data = self.data[(self.data[col_name] > lower_limit) & (self.data[col_name] < upper_limit)]
        return self.data
```

C. Feature engineering

As [5] mentioned, feature engineering should be conducted with an understanding of the problem domain. Therefore, we approached different tasks in various ways.

1) Classification: In the task of classification, we first created three aggregated features by summing 'PAY 0' to 'PAY 6', 'BILL AMT1' to 'BILL AMT6', and 'PAY AMT1' through 'PAY AMT6'. There are two primary reasons for this approach. Firstly, we aimed to avoid multicollinearity, as discussed in the previous section. Secondly, these are time-series data, and aggregating them can give a general view of individuals [5]. After doing this, we removed the original column features and obtained three new features: "Bill Amount Total", "Repayment Status Total", and "Pay Amount Total".

The second approach involves clustering techniques to identify groups within each numerical feature, and it is inspired by [6]. We assumed that different groups could have similar credit limits or default status. Therefore, we employed clustering to identify these similarity groups. There are four features to cluster: age, bill amount, pay amount and repayment amount. For the bill, pay and repayment amount we chose to use the average value instead of the total amount. This is because the average value can reduce the variance of the data, therefore enhancing the clustering performance [5] [1].

Create new features via clustering

```python
# This function needs to be called from inside class
# It connects to function _reg_feature_engineering
def _create_new_feature(self, min_k=5, max_k=10) -> pd.DataFrame:
    # With assistance of Copilot for writing efficient code
    # Define columns to be aggregated
    features = {
        'Bill_Amount': ['Sept', 'Aug', 'Jul', 'Jun', 'May', 'Apr'],
        'Pay_Amount': ['Sept', 'Aug', 'Jul', 'Jun', 'May', 'Apr'],
        'Repayment_Status': ['Sept', 'Aug', 'Jul', 'Jun', 'May', 'Apr']
    }
    # Create new features via aggregation
    for feature, months in features.items():
        self.data[f'{feature}_Total'] = sum(self.data[f'{feature}_{month}'] for month in months)
        self.data[f'{feature}_Average'] = self.data[f'{feature}_Total'] / len(months)
    # Avoid multicollinearity
    columns_to_drop = [f'{feature}_{month}' for feature, months in features.items() for month in months]
    self.data.drop(columns=columns_to_drop, axis=1, inplace=True)
    # Cluster grouping
    cluster_features = [f'{feature}_{agg}' for feature in features.keys() for agg in ['Total', 'Average']]
    cluster_features.append('AGE')
    for feature in cluster_features:
        self.data = self._cluster_group(feature, min_k=min_k, max_k=max_k)
    self.data.to_csv('../data/clustered_data.csv', index=False)
    return self.data
```

We experimented with two clustering algorithms: K-means and Density-Based Spatial Clustering of Applications with Noise (DBSCAN). This decision is based on their distinct approaches to clustering. K-means is valuable for effectively partitioning data into 'k' clusters [7]. On the other hand, DBSCAN offers the advantage of not having to pre-specify the number of clusters [7]. From Figure 5, we can see that

[1] In the approach of [5], they did the log transformation to reduce the variance

the K-means clustering algorithm provided better grouping performance than DBSCAN. Then, we applied the silhouette score to measure the quality of the clusters to find a suitable K for each feature. After carefully tuning the 'K' parameter, we generated four new features: "AGE cluster k", "Bill cluster k", "Repayment cluster k", and "Pay cluster k".

Figure 6 displays the new correlation matrix generated after creating new features. It is observable that the new features correlate fairly with the target label. And the issue of multicollinearity appears to have been effectively addressed.

Clustering: find optional k

```
# This function called inside class
# It connects to  _reg_create_new_feature
def _cluster_group(self, col_name, min_k=5, max_k=5,
    scaler=StandardScaler()) -> pd.DataFrame:
        features_for_clustering = self.data[[col_name
    ]]
        features_for_clustering = scaler.
    fit_transform(features_for_clustering)
        best_score = -1
        best_k = 0
        for k in range(min_k, max_k+1):
            kmeans = KMeans(n_clusters=k,
    random_state=self.random_state)
            kmeans.fit(features_for_clustering)
            score = silhouette_score(
    features_for_clustering, kmeans.labels_)
            if score > best_score:
                best_score = score
                best_k = k
        best_kmeans = KMeans(n_clusters=best_k,
    random_state=self.random_state)
        self.data[f"{col_name}_cluster"] =
    best_kmeans.fit_predict(features_for_clustering)
        return self.data
```

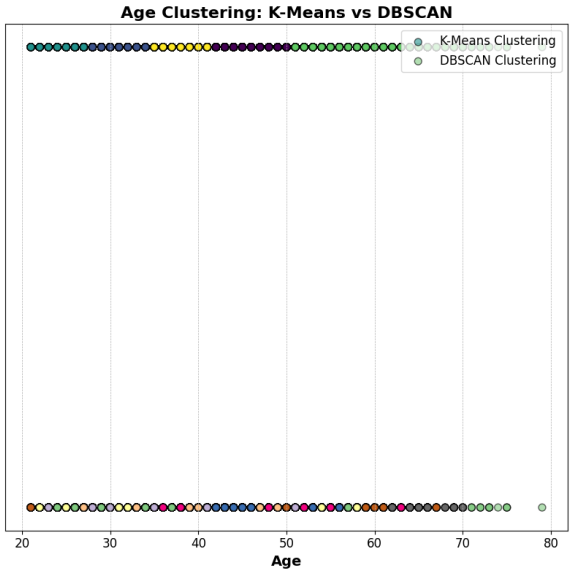


Fig. 5: K-means VS DBSCAN

As for the categorical features in our dataset, we used the one-hot encoding technique to transform these to ensure the model treats each category in the same weight [7]. We observed that the dataset contained many categorical features,

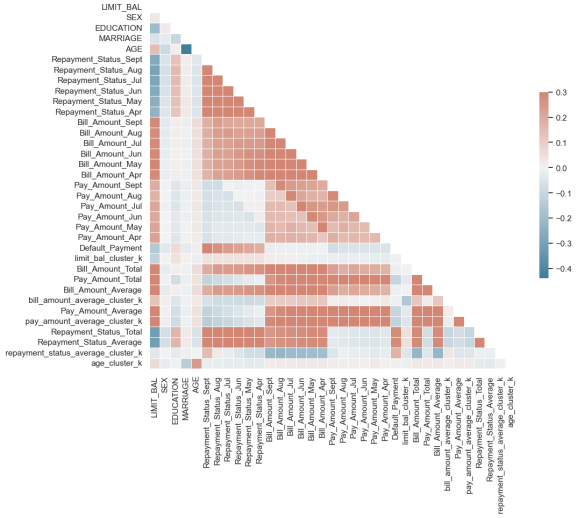


Fig. 6: Classification: Correlation Matrix After Creating New Features

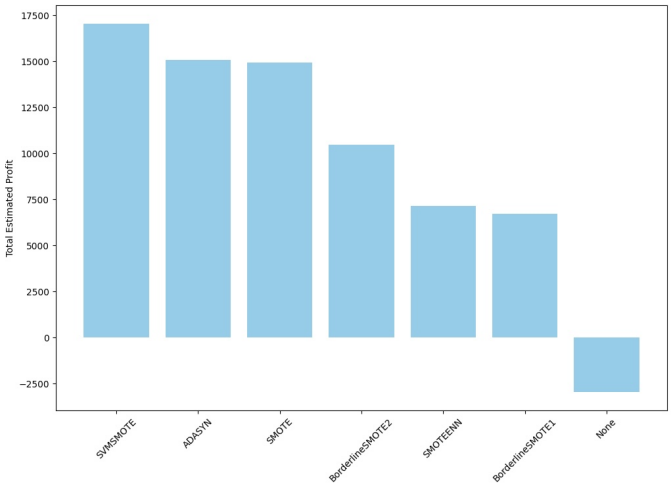
leading to high dimensional complexity after applying one-hot encoding. To mitigate this, we chose binary encoding as a more dimensionally efficient alternative [8]. Next, we

Binary encoder

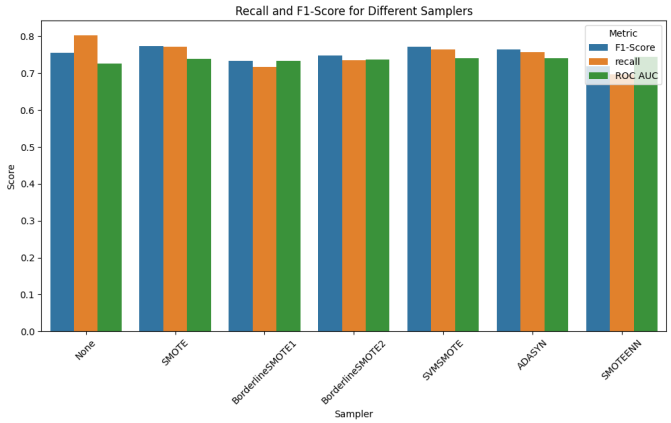
```
def _apply_binary_ohe(self, categorical_columns):
        encoder = ce.BinaryEncoder(cols=
    categorical_columns)
        self.features = encoder.fit_transform(self.
    features)
        return self.features
```

applied a standard scaler to normalize the numerical features as the model, like SVC, is sensitive to the scaling [7]. In the data exploration, we identified an imbalance in the dataset. To address this, we experimented with various resampling techniques, including SMOTE, BorderlineSMOTE, SVMSMOTE, and ADASYN, as suggested from [5]. We also compared them with the baseline, the model without a sampler. For consistency in our experimental setup, we used the same scaler and feature selector and then applied LR for prediction and evaluation. We chose the desired sampler according to several metrics, including ROC AUC score, F1 score, recall and profit chart. Our experiments (Figure 7) found that SVMSMOTE obtained the most desirable outcomes as it achieved the highest profit and with a high value of F1, which means it has a balanced trade-off between recall and precision. Choosing metrics and how profit matric works will be discussed later.

After selecting the appropriate sampler, we applied a similar consistent methodology to choose the best feature selector. Our evaluation included various methods such as VarianceThreshold, feature selection under ANOVA F-value (f_classif), mutual information for a classification task (mutual_info_classif), and FeatureSelector. Using the same process and reasoning as the previous section, we chose f_classif as the desired feature

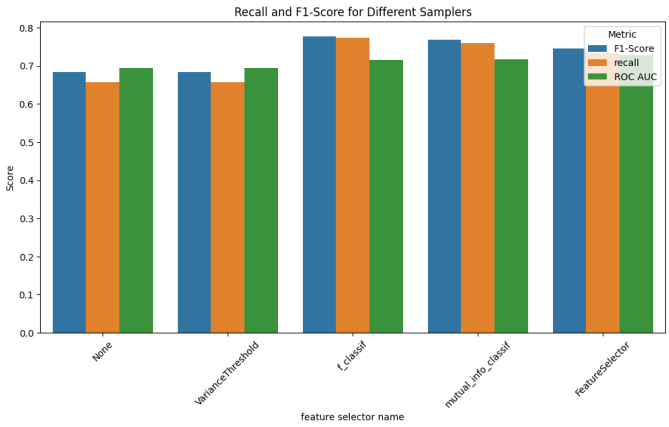


(a) Profit for Sampler Selection

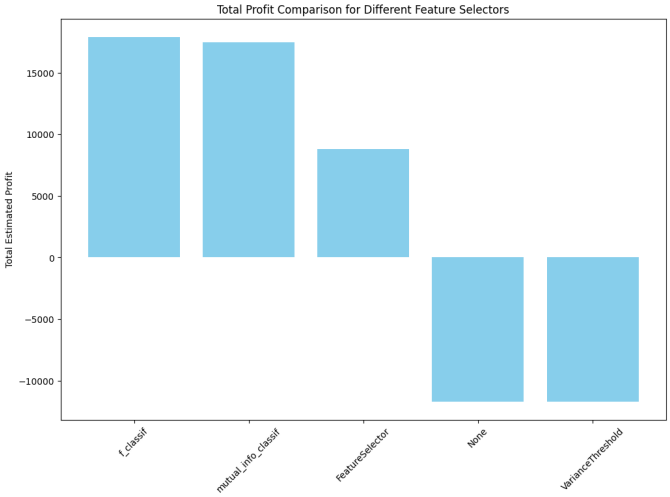


(b) Score for Sampler Selection

Fig. 7: Classification Results for Sampler Selection



(a) Score for Feature Selection



(b) Profit for Feature Selection

Fig. 8: Classification Results for Feature Selection

selector from the result in Figure 8.

Finally, we implemented PCA to reduce the dimensionality of our features. From Figure 9, we determined that using two principal components offers an optimal variance balance, effectively capturing the most significant information.

2) Regression: In the regression task, we first generated polynomial features, inspired by [5], that we can create numerical features from the categorical features. We recognized that combinations of features might influence the target label. Based on this assumption, we created several polynomial features to explore their correlation with the target label. After thorough experimentation and analysis, we selected the most relevant polynomial features. As observed in Figure 10, the new features correlate more strongly with the target label than the original. And it has resulted in an increased number of features. However, we also noticed that multicollinearity appears; we should consider it in model selection. After that, we used the same scaling process as the previous task to deal with our features.

Secondly, we implemented a transformation on the label. This is because we observed a positive skew in data ex-

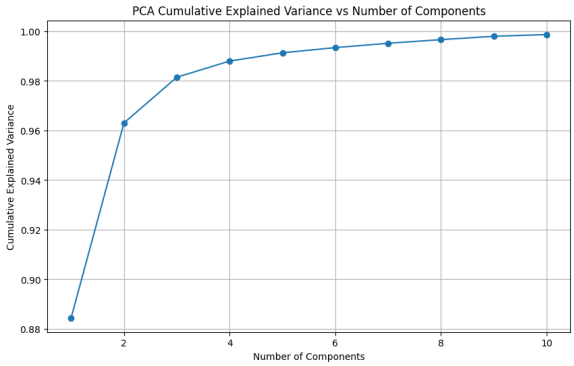


Fig. 9: PCA

Create polynomial features

```
# Code called inside class
# It connects to _reg_feature_engineering
def _reg_create_new_feature(self):
    self.features = self.features.copy()
    self.features['Edu_Marrige'] = self.features['
     EDUCATION'] * self.features['MARRIAGE']
    self.features['Edu_Marrige2'] = self.features['
     EDUCATION'] * self.features['MARRIAGE'] ** 2
    self.features['Edu_Age2'] = self.features['
     EDUCATION'] * self.features['AGE_cluster'] ** 2
    self.features['Edu3_Sex'] = self.features['
     EDUCATION'] ** 3 * self.features['SEX']
    self.features["Edu3"] = self.features["EDUCATION"
     ] ** 3
    self.features["Marriage4"] = self.features["
     MARRIAGE"] ** 4
    self.features["Marriage_Age3"] = self.features["
     MARRIAGE"] * self.features["AGE_cluster"] ** 3
    self.features["Sex_edu3_mar"] = self.features["
     SEX"] * self.features["EDUCATION"] ** 3 * self.
     features["MARRIAGE"]
    self.features["Sex2_mar_age2"] = self.features["
     SEX"] ** 2 * self.features["MARRIAGE"] * self.
     features["AGE_cluster"] ** 2
    self.features['Edu3_mar2_age'] = self.features["
     EDUCATION"]**3 * self.features["MARRIAGE"]**2 *
     self.features["AGE_cluster"]
        return self.features
```

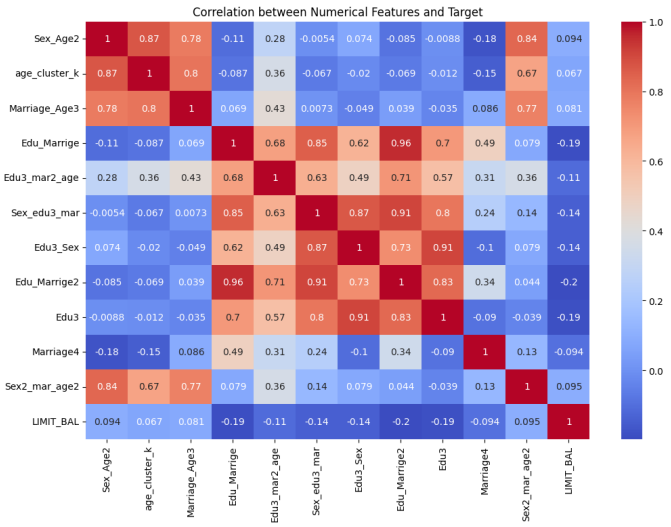


Fig. 10: Correlation matrix after creating features

ploration. We evaluated various techniques, including Yeo-Johnson, Quantile, and Box-Cox transformations, which are inspired by [9] and [10]. To determine the most effective approach, we conducted experiments similar to selecting feature selectors. And the difference was the metrics used for evaluation. We chose mean absolute error (MAE), root mean square error (RMSE), R-squared (R2), and hit rate (HR) as our key metrics. After several experiments, we chose Box-Cox transformations as the desired transformation function as it shows relatively low MAE and also has a higher-value hit rate from Table IV.

Given the code is followed by OOP, we built feature engineering for classification and regression into one part with

TABLE IV: Result of Different Transformation Technique

Transformation name	RMSE	MAE	R2	HR
Box-Cox	118001.91	**88561.67**	0.12	**0.62**
Yeo-Johnson	118001.26	88562.34	0.12	0.62
Quantile	**117799.21**	89242.49	0.12	0.61

several functions followed by SRP:

Prepare the data before spilting

```
# Code called from outside class
# Aim: prepare the features and label for cls and reg
def prepare_data(self, target_name, reg_features=None
     , classify=True):
    # if cleaned_data.csv and clustered_data.csv
     exist, then load them
    if os.path.exists('../data/clustered_data.csv')
     and os.path.exists('../data/cleaned_data.csv'):
        self.data = pd.read_csv('../data/
     clustered_data.csv')
        self.data = pd.DataFrame(self.data)
    else:
        self.data = self._data_cleaning()
        self.data = self.create_new_feature()
    if not classify:
        self.features, self.label_for_train, self.
     label_for_test = self._reg_feature_engineering(
     target_name=target_name, reg_features=
     reg_features, classify=classify)
        return self.features, self.label_for_train,
     self.label_for_test
    else:
        self.features, self.target = self.
     _cls_feature_engineering(target_name,
     reg_features, classify)
        self.target = self.target[target_name]
        return self.features, self.target
```

Feature engineering for regression and classification

```
# Both code called from inside class
# It connects to prepare_data
def _reg_feature_engineering(self, target_name="
     Default_Payment", reg_features=None, classify=
     True):
    # remove outliers
    category_features = ["SEX", "EDUCATION", "
     MARRIAGE"]
    self.features, self.target = self.
     _prepare_feature_target(target_name, reg_features
     , classify=False)
    self.features = self._reg_create_new_feature()
    self.label_for_train = self.target["LIMIT_BAL"]
    self.label_for_test = self.target["
     Default_Payment"]
    self.features = self._apply_binary_ohe(
     category_features)
    return self.features, self.label_for_train, self.
     label_for_test

def _cls_feature_engineering(self, target_name,
     reg_features=None, classify=True):
    self.features, self.target = self.
     _prepare_feature_target(target_name, reg_features
     , classify)
    categorical_columns, numerical_columns = self.
     _different_categories_numerical()
    self.features = self._apply_binary_ohe(
     categorical_columns)
    return self.features, self.target
```

Spilt features and target

```python
# Both functions called from inside class

# Aim: spilt target label and features
def _prepare_feature_target(self, target_name,
     reg_features=None, classify=True):
    if classify:
        self.features = self.data.drop([target_name],
      axis=1)
        self.target = self.data[target_name]
        self.target = pd.DataFrame(self.target)
    else:
        self.features = self.data[reg_features]
        self.target = self.data[target_name]
        self.target = pd.DataFrame(self.target)
    return self.features, self.target

# Aim: different target label and features
def _different_categories_numerical(self):
    categories = []
    numerical = []
    for col in self.features.columns:
        unique_values = self.data[col].unique()
        if len(unique_values) <= 11:
            categories.append(col)
        else:
            numerical.append(col)
    return categories, numerical
```

Feature engineering after spilting data

```python
# This function called from outside class
# Aim:
# 1. spilt train and test set
# 2. continue feature engineering, in order to
     prevent data leak
def split_data(self, test_size=0.2, classify=True):
    if classify:
        features_cls, target_cls = self.prepare_data(
    "Default_Payment", classify=True)
        X_train, X_test, y_train, y_test =
    train_test_split(features_cls, target_cls,
    test_size=test_size, random_state=self.
    random_state)

        X_train_scaled = self.scaler.fit_transform(
    X_train)
        X_test_scaled = self.scaler.transform(X_test)

        X_train_resampled, y_train_resampled = self.
    sampler.fit_resample(X_train_scaled, y_train)

        X_train_selected = self.feature_selector.
    fit_transform(X_train_resampled,
    y_train_resampled)
        X_test_selected = self.feature_selector.
    transform(X_test_scaled)

        n_features = X_train_selected.shape[1]

        # Below part with assistance from GitHub
    Copilot
        # The idea is if feature is too small, will
    skip the PCA
        if n_features > 1:
            pca = PCA(n_components=min(n_features, 2)
    )
            X_train_pca = pca.fit_transform(
    X_train_selected)
            X_test_pca = pca.transform(
    X_test_selected)
        else:
            X_train_pca = X_train_selected
            X_test_pca = X_test_selected
        return X_train_pca, X_test_pca,
    y_train_resampled, y_test
    else:
        feature_reg, label_for_train, label_for_test
    = data.prepare_data(target_name=["LIMIT_BAL", "
    Default_Payment"], reg_features=["SEX", "
    EDUCATION", "MARRIAGE", "AGE_cluster"],classify=
    False)
        train_X, test_X, train_y, test_y, _ ,
    test_y_additional = train_test_split(self.
    features, self.label_for_train, self.
    label_for_test, test_size=test_size, random_state
    =self.random_state)
        train_X_scaled = self.scaler.fit_transform(
    train_X)
        test_X_scaled = self.scaler.transform(test_X)
        train_y_transformed, lambda_ = boxcox(train_y
    )
        return train_X_scaled, test_X_scaled,
    train_y_transformed, test_y, lambda_,
    test_y_additional
```

D. Metrics

In classification tasks, we measured model performance using the F1-score, recall, ROC AUC score, and a custom profit rate metric to assess overall effectiveness. The primary reason for this approach is due to the imbalance dataset. The F1-score is particularly useful as it illustrates the trade-off between recall and precision; recall is valuable for understanding the status of a minority group, such as false negatives. And the ROC AUC score offers insights into the balance between true negatives and false positives. Using these three metrics, we can measure the resulting performance of one model with an imbalanced dataset.

From a business perspective, we assumed banks are particularly concerned with identifying potential defaulters to mitigate risk, as highlighted in the introduction. Therefore, we created a profit metric inspired by [11] to measure the financial impact and match our evaluation process with the business objectives. The profit metric depends on the value in the confusion matrix, which we provide different weights and thus simulate the profit result.

$$\mathrm{P} = \mathrm{TP} * B_{TP} - \mathrm{FP} * C_{FP} - \mathrm{FN} * C_{FN} + \mathrm{TN} * B_{TN} \quad (4)$$

As illustrated in Equation 4, 'P' represents the total profit. 'TP' stands for true positive, 'FN' for false negative, and 'FP' for false positive. The symbol B_{TP} denotes the benefits when an individual will not default and the prediction is correct. B_{TN} represents the benefits when an individual defaults and the prediction is correct. C_{FP} indicates the costs incurred when an individual will not default, but the prediction is wrong, while C_{FN} refers to the costs when an individual will default, but the prediction is incorrect.

We assigned the highest weight to B_{TP}, assuming that the bank earns the most benefit from customers who do not default and continue using their loan or credit card services. Conversely, we gave the least weight to B_{TN}, assuming that while the prediction is correct, it does not significantly contribute to business benefits. The second-highest weight was assigned to C_{FN}, as we assumed that the bank aims to minimize defaults, which can result in costs. However,

this weight is still lower than B_{TP}. Lastly, the weight for C_{FP} is lower than that for C_{FN}, considering that although the prediction is incorrect, the potential loss, such as losing customers, is less compared to C_{FN}. The cost associated with C_{FP} is set at two-thirds of B_{TP}, reflecting the penalty of losing a customer and the impact on the bank's reputation. In this case, we set B_{TP} as 60, C_{FN} as 55, C_{FP} as 40 and B_{TN} as 10.

Profit metric

```
# All functions are called from inside the class
def _cls_profit_value(self, cm):
    tp = cm[0][0]
    fp = cm[0][1]
    fn = cm[1][0]
    tn = cm[1][1]
    total_profit = (tp * self.benefit_tp) - (fp *
     self.cost_fp) - (fn * self.cost_fn) + (tn * self.
     benefit_tn)
    return total_profit
```

In the regression task, we used the RMSE and MAE to determine the precision of the prediction and R2 to provide the metric on the explainability of the model. Beyond that, we introduced a new metric that can tell the model performance from a business perspective, which is HR.

The HR is defined based on the effectiveness of the model's predictions in mitigating risk. A prediction is considered a 'hit' in three scenarios: First, if a user will default and the model's predicted value is lower than the actual, the model has successfully identified a risk. Second, if a user will not default and the model's prediction does not exceed the actual value, the model again accurately assesses the risk. Finally, suppose a user will not default, and the model's prediction is slightly higher than the actual value but still falls within an acceptable boundary. In that case, this is also considered a hit. This last scenario reflects a business's willingness to accept a certain level of risk in real-life operations. In this case, we set the acceptable boundary as 30,000, as we observed that the gap between each credit limit is up to 20,000, and we chose 30,000 to simulate an acceptable risk.

We also used bias and variance as metrics to evaluate our model. Bias determines the accuracy of the prediction. On the other hand, variance measures the variation in the prediction [7].

E. Model training

Before the beginning of the training process, we split the data into training and testing sets into 80% and 20%. The random seed we chose was 33. The specific hardware and system configurations used for running the models are detailed in Table V. We employed LR, SVC, and RFC for the classification tasks. In contrast, for the regression tasks, we applied ElasticNet, RFR, and an ANN implemented using PyTorch.

Since the details of the other models are available in [7], this paper will concentrate more on the architecture of the ANN. Our version ANN consists of four linear layers, three

HR metric

```
# All functions are called from inside the class
def _reg_model_hit_rate(self, y_pred,
     test_y_additional, y_test):
    hits = 0
    total = len(y_test)
    for i in range(total):
        y_pred_value = y_pred[i]
        y_test_value = y_test.iloc[i]
        test_y_additional_value = test_y_additional.
     iloc[i]
        if test_y_additional_value == 1 and
     y_pred_value < y_test_value:
            hits += 1
        elif test_y_additional_value == 0:
            if y_pred_value <= y_test_value or (
     y_pred_value > y_test_value and y_pred_value -
     y_test_value <= 30000):
                hits += 1
    return hits / total if total > 0 else 0
```

Bias and Variance

```
# All functions are called from inside the class
def _calculate_bias(self, y_pred, y_test):
    bias = np.mean(y_pred - y_test)
    return bias
def _calculate_variance(self, y_pred):
    variance = np.var(y_pred)
    return variance
```

Classification model report

```
# All functions are called from outside the class
# Aim: train the model, evaluate the model, generate
     to report for analysis
def cls_model_report(self, X_train, X_test, y_train,
     y_test):
    self._train_cls_model(X_train, X_test, y_train,
     y_test)
    y_pred = self.model.predict(X_test)
    score = self._cls_model_score(X_test, y_test)
    accuracy = self._cls_model_accuracy(y_pred,
     y_test)
    cm = self._cls_model_confusion_matrix(y_test,
     y_pred)
    roc_auc = self._cls_model_roc_auc_score(y_test,
     y_pred)
    total_profit = self._cls_profit_value(cm)
    f1 = self._cls_model_f1_score(y_test, y_pred)
    recall = self._cls_model_recall(y_test, y_pred)
    hashtable = {
        "name": self.model_name,
        "parameters": self.model_params,
        "confusion_matrix": cm,
        "roc_auc_score": roc_auc,
        "total_profit": total_profit,
        "f1_score": f1,
        "recall": recall
    }
    self.model_report.append(hashtable)
    return self.model_report
```

Rectified Linear Unit (ReLU) activation, and two dropout layers. The ReLU activation function is to capture non-linear relationships. The dropout layers are incorporated to prevent overfitting. Meanwhile, we also implemented the learning rate scheduler, which can adjust the learning rate to help with faster convergence.

Regression model report

```
def reg_model_report(self, X_train, X_test, y_train, y_test, test_y_additional, lambda_):
    self._train_reg_model(X_train, y_train)
    y_pred = self.model.predict(X_test)
    y_pred_transformed = inv_boxcox(y_pred, lambda_)
    y_pred_transformed = [round(val, 2) for val in y_pred_transformed]

    rmse = np.sqrt(mean_squared_error(y_test, y_pred_transformed))
    mae = mean_absolute_error(y_test, y_pred_transformed)
    r2 = r2_score(y_test, y_pred_transformed)
    hit_rate = self._reg_model_hit_rate(y_pred_transformed, test_y_additional, y_test)
    bias = self._reg_calculate_bias(y_pred_transformed, y_test)
    variance = self._reg_calculate_variance(y_pred_transformed)

    hashtable = {
        "name": self.model_name,
        "parameters": self.model_params,
        "rmse": round(rmse, 2),
        "mae": round(mae, 2),
        "r2": r2,
        "hit_rate": str(round(hit_rate * 100, 4)) + " %",
        "bias": round(bias, 2),
        "variance": round(variance, 2)
    }
    self.model_report.append(hashtable)
    return self.model_report
```

TABLE V: System setting

Component	Definition
CPU	ARM
GPU	Apple M3 Max
System RAM	48.0GB

Right side is the code about ANN.

F. Hyperparameter tuning

During the hyperparameter tuning process, we selected hyperparameters by considering computational power and the cost-effectiveness of the tuning process. We selected 5-fold for cross-validation to measure whether the model overfitting and training performance.

The chosen hyperparameters are shown below is shown in Table VI.

1) LR in Classification: Regarding LR in classification tasks, given that we have over 25,000 instances, we consider a moderately large dataset. We considered 'sag', 'saga', and 'lbfgs' solvers. And since our feature engineering includes one-hot encoding, which results in the number of samples exceeding the number of features, 'newton-cholesky' is also a choice for a solver. We only use 'l2' as it is the common regularization across all solvers. For the regularization strength, we have selected a range from 0.01 to 100. This range allows us to prevent model overfitting or underfitting. We used grid search to find the optimal solution as LR does not take too much compute power.

2) SVC in Classification: Tuning the SVC requires careful consideration of time complexity and computational power.

ANN

```
# This code with the assistance of GitHub Copilot,
# The architecture inspired by :https://github.com/glingden/Boston-House-Price-Prediction/blob/master/house_price_prediction.ipynb
class ANN(nn.Module):
    def __init__(self, input_size, batch_size=32, learning_rate=0.1, epochs=100, step_size=30, gamma=0.1, random_state=33):
        super(ANN, self).__init__()
        # define layers
        self.fc1 = nn.Linear(input_size, 128)
        self.relu = nn.ReLU()
        self.dropout1 = nn.Dropout(0.5)  # 50% dropout
        self.fc2 = nn.Linear(128, 64)
        self.dropout2 = nn.Dropout(0.5)  # 50% dropout
        self.fc3 = nn.Linear(64, 32)
        self.fc4 = nn.Linear(32, 1)
        # define hyperparameters
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.step_size = step_size
        self.gamma = gamma
        self.random_state = random_state
        # define loss and optimizer
        self.criterion = nn.MSELoss()
        self.optimizer = optim.Adam(self.parameters(), lr=learning_rate)
        self.scheduler = StepLR(self.optimizer, step_size=step_size, gamma=gamma)
        # define input size
        self.input_size = input_size
        # define model report
        self.model_report = []
    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.dropout1(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.dropout2(out)
        out = self.fc3(out)
        out = self.relu(out)
        return self.fc4(out)
    def training_step(self, train_X, train_y):
        train_X = self._convert_data(train_X)
        train_y = self._convert_data(train_y)
        self.train()
        plot_data = []
        for epoch in range(self.epochs):
            total_loss = 0
            for i in range(0, len(train_X), self.batch_size):
                batch_X = train_X[i:i+self.batch_size].view(-1, self.input_size)
                batch_y = train_y[i:i+self.batch_size]
                self.optimizer.zero_grad()
                outputs = self(batch_X)
                loss = self.criterion(outputs, batch_y)
                loss.backward()
                self.optimizer.step()
                total_loss += loss.item()
            avg_loss = total_loss / (len(train_X) / self.batch_size)
            self.scheduler.step()
            current_lr = self.scheduler.get_lr()[0]
            print(f"Epoch: {epoch}. Average Loss: {avg_loss}, Current LR: {current_lr}")
```

We have explored all kernel functions to find a point between overfitting and underfitting. To reduce computational demands, we have selected the polynomial degree to range from 2

Chosen Hyper-parameter

```
best_param = {
    "LR": {'C': 10,'max_iter': 100, 'n_jobs': -1, '
     penalty': 'l2', 'random_state': 33, 'solver': '
     saga'},
    "SVC": {'random_state': 33, 'max_iter': 250, '
     kernel': 'sigmoid', 'gamma': 'auto', 'degree': 5,
      'cache_size': 700, 'C': 50},
    "RFC": {'n_jobs': -1, 'n_estimators': 200, '
     min_samples_split': 4, 'min_samples_leaf': 1, '
     max_features': 'sqrt', 'max_depth': 10, '
     criterion': 'gini'},
    "ElasticNet": {'alpha': 10, 'l1_ratio': 0.9, '
     max_iter': 100, 'selection': 'cyclic'},
    "RFR": {'criterion': 'friedman_mse', 'max_depth':
      1000, 'max_features': 'log2', 'min_samples_split
     ': 4, 'n_estimators': 100, 'n_jobs': -1},
    "ANN": {'gamma': 0.1, 'learning_rate': 0.1, '
     epochs': 100, 'step_size': 30, 'random_state':
     33}
}
```

TABLE VI: Chosen Hyper-parameter

to 5 and limited the maximum iterations up to 270. This decision is based on experimental findings, balancing optimal performance with time efficiency. Lastly, we set the cache size to 700 to reduce computational load. We chose to use a 'random' search and provide 50 interactions to balance the time complexity and efficiency.

3) RFC in Classification: When tuning the RFC, there are numerous options to consider. We have chosen to use both criterion methods, 'gini' and 'entropy'. Additionally, we have allowed up to 300 estimators to limit the number of trees. We have also set the maximum depth to range from 10 to 30, which helps in reducing the height of each tree. This enables the model to deliver results at an earlier stage. For the same reason as SVC, we chose to use a random search and provide 100 interactions.

4) ElasticNet in Regression: We have chosen ElasticNet for linear regression due to its ability for feature selection and effectiveness in handling correlated features, which can address the issues identified in our feature engineering. We have selected the $l1\_ratio$ from 0 to 1, which aims to observe how the combination of L1 and L2 regularization affects our model. Additionally, we set the maximum number of iterations to 500 to reduce the time complexity. We used a grid search as it did not take long to compute.

5) RFR in Regression: In the RFR, we have selected 'mse' and 'friedman_mse' as criteria in our model to reduce the bias. Additionally, we chose 'sqrt' as the option for maximum features to ensure maximal feature utilization. The remaining hyperparameters will be consistent with those used in the RFC. We also took the same random search as in RFC.

6) ANN in Regression: In our ANN model, the primary hyperparameter we focused on is the gamma in the learning rate scheduler. We experimented with a range of gamma values from 0.1 to 0.5.

Below is the tuning process in regression and classification; we used our custom metric to select the most suitable hyper-parameter set: hit rate and profit. These two functions belong to the class $Model$.

Tuning function for classification

```
# All functions are called from outside the class
# It connected to the PipeLine class directly

def tune_parameters_cls(self, X, y, tuning_method='
    grid', param_grid=None, cv=5, n_iter=10):
    # This is our own custom score
    scorer = make_scorer(self.cls_custom_score)
    if tuning_method == 'grid':
        self.tuner = GridSearchCV(self.model,
     param_grid, scoring=scorer, cv=cv, verbose=3)
    elif tuning_method == 'random':
        self.tuner = RandomizedSearchCV(self.model,
     param_grid, scoring=scorer, n_iter=n_iter, cv=cv,
      random_state=self.random_state, verbose=3)
    self.tuner.fit(X, y)
    self.best_params = self.tuner.best_params_
    self.model.set_params(**self.best_params)
    return self.best_params
```

Tuning function for regression

```
def tune_parameters_reg(self, X, y, test_y_additional
    , tuning_method='grid', param_grid=None, cv=5,
    n_iter=10):
    # This is our own custom score
    custom_scorer = CustomScorer(test_y_additional)
    if tuning_method == 'grid':
        self.tuner = GridSearchCV(self.model,
     param_grid, scoring=custom_scorer, cv=cv, verbose
    =3)
    elif tuning_method == 'random':
        self.tuner = RandomizedSearchCV(self.model,
     param_grid, scoring=custom_scorer, n_iter=n_iter,
      cv=cv, random_state=self.random_state, verbose
    =3)
    self.tuner.fit(X, y)
    self.best_params = self.tuner.best_params_
    self.model.set_params(**self.best_params)
    return self.best_params
```

III. Evaluation

In our classification tasks, as illustrated in Figure 11, the SVC model achieved the highest recall and F1 score. This indicates that the SVC model has a lower bias and variance and is more effective at identifying true positives. Additionally, the SVC model also attained the highest ROC AUC score, suggesting a better balance between the false negative rate and the true positive rate. Furthermore, the SVC model yielded the highest profit, underscoring its ability to aid in business decision-making processes.

Regarding the regression tasks, as shown in Figure 12, the ANN achieved the highest HR, followed by the RFR. Meaning the ANN is more effective in mitigating risk for the company. However, regarding model explainability, as reflected by the R2, and precision, as measured by the RMSE and MAE, both the ANN and ElasticNet did not perform as well as the RFR. As shown in Table VII, ElasticNet and ANN predicted lower credit values, whereas the RFR tended to be ambitious. The bias among all three models suggests that they are underfitting

Custom Score

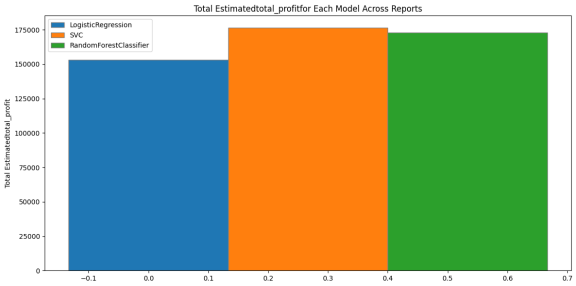
```python
# This is class for tuning purpose -> hit rate
# With assistance of Github Copilot
class CustomScorer:
    def __init__(self, additional_data):
        self.additional_data = additional_data
    def __call__(self, estimator, X, y_true):
        y_pred = estimator.predict(X)
        return self._reg_custom_score(y_pred, y_true,
      self.additional_data)
    def _reg_custom_score(self, y_pred, y_true,
     test_y_additional):
        hits = 0
        total = len(y_true)
        for i in range(total):
            y_pred_value = y_pred[i]
            y_true_value = y_true[i]
            test_y_additional_value =
     test_y_additional.iloc[i]
            if test_y_additional_value == 1 and
     y_pred_value < y_true_value:
                hits += 1
            elif test_y_additional_value == 0:
                if y_pred_value <= y_true_value or (
     y_pred_value > y_true_value and y_pred_value -
     y_true_value <= 30000):
                    hits += 1
        return hits / total if total > 0 else 0
```

Pipeline

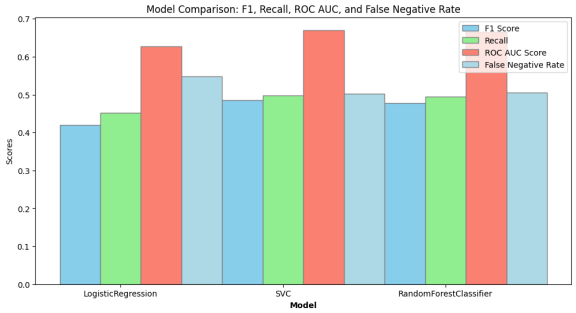
```python
# This is class for Encapsulation all the class we
     discussed before and generate one step to run
     everything.
# Specific for tuning purpose, it is very much
     flexibility
class Pipeline(Model):
    # function to tune and generate the final report,
      called outside
    def start_pipeline_cls(self, X_train, X_test,
     y_train, y_test, tuning_method='grid', param_grid
     =None, cv=5, n_iter=10):
        print("Starting classification pipeline...")
        self._find_best_parameters_cls(X_train,
     y_train, tuning_method, param_grid, cv, n_iter)
        self._find_cls_report(X_train, X_test,
     y_train, y_test)
        print("Classification pipeline completed.")
        return self.model_report

    def start_pipline_reg(self, X_train, X_test,
     y_train,y_test, test_y_additional, lambda_,
     tuning_method='grid', param_grid=None, cv=5,
     n_iter=10):
        print("Starting regression pipeline...")
        self._find_best_parameters_reg(X_train,
     y_train, test_y_additional, tuning_method,
     param_grid, cv, n_iter)
        self._find_reg_report(X_train, X_test,
     y_train, y_test, test_y_additional, lambda_)
        print("Regression pipeline completed.")
        return self.model_report
```

the data. In terms of variance, ElasticNet exhibited the lowest variance, indicating underfitting, while the RFR showed the highest variance, meaning the model is too complex, therefore causing the overfitting. Overall, the ANN appears to be a more balanced trade-off between bias and variance.

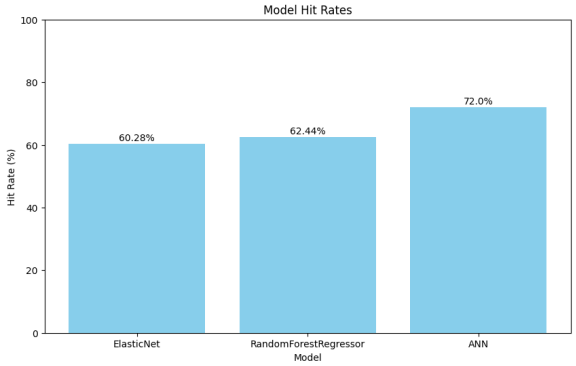


(a) Profit for classification

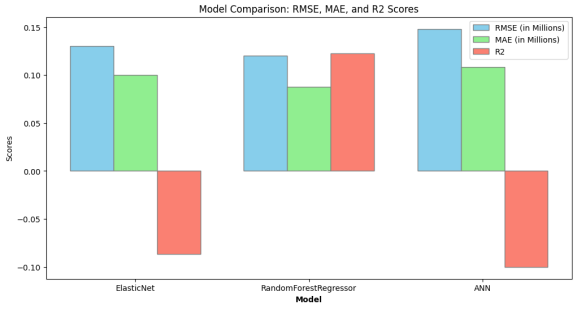


(b) Standard metric for classification

Fig. 11: Classification Results



(a) HR Comparison for Regression



(b) Score Comparison for Regression

Fig. 12: Regression Results

Visulize result

```
# function called outside the class, visualise the result
def visualize_model_reports(self, classfy=True):
    if classfy:
        self._plot_total_profit_bar_chart("total_profit")
        self._confusion_matrix_heatmap()
        self._plot_metric_bar_chart_cls()
    else:
        self._plot_hit_rate_bar_chart()
        self._plot_metric_bar_chart_rg()
        self._plot_bias_variance_bar_chart_rg()
```

Test function

```
# Evaluating the result by adding model report in, easy as one go:
cls_evaluator = Evaluation()
cls_evaluator.add_model_report(cls_lr_report)
# You can add more if you have more
cls_evaluator.visualize_model_reports()
```

TABLE VII: Bias and Variance

Model Name	Bias	Variance
RFR	29000.0	2800000000.0
ElasticNNet	-37000.0	0.0
ANN	-78902.734	453752.72

IV. Conclusion

This paper detailed our approach to classification and regression tasks using various models. In the classification task, the SVC balanced business profit and the bias-variance trade-off. Three models showed the desired result, effectively dealing with imbalanced data. In the regression task, the ANN obtained the highest HR, and it also demonstrated a relatively effective trade-off between bias and variance. However, in the regression task, the model failed to predict the accurate credit limit given the limited information, but it did help the business make decisions to mitigate the risk. In future work, we could collect more relevant features from the new users to prevent the underfitting problem. And we also need to reduce the model complexity to prevent overfitting in RFR.

Additionally, this paper introduced new metrics to evaluate models from a business perspective. However, this is based on many assumptions and would not fit all the business cases. The future work will be collecting more case studies and identifying the business requirements, then evaluating and redesigning the effectiveness of the metrics.

As mentioned in the tuning section, we considered the insufficient time and computational resources allocated, which can result in the model not being as effective as it should be. For example, RFR should obtain lower variance given the bagging process. Future efforts should focus more on careful tuning to enhance model performance.

References

[1] Saurabh Arora, Sushant Bindra, Survesh Singh, and Vinay Kumar Nassa. Prediction of credit card defaults through data analysis and machine learning techniques. *Materials Today: Proceedings*, 51:110–117, 2022.

[2] Salman Haqqi. Credit card statistics 2023 - credit card facts and stats report, 2023.

[3] Songhao Wu. Multi-collinearity in regression. https://towardsdatascience.com/multi-collinearity-in-regression-fe7a2c1467ea, Jan 2023. Accessed: 2023-12-08.

[4] Jason Brownlee. How to use statistics to identify outliers in data. https://machinelearningmastery.com/how-to-use-statistics-to-identify-outliers-in-data, 2020. Accessed: 2023-12-08.

[5] Abhishek Thakur. *Approaching (almost) any machine learning problem*. Abhishek Thakur, 2020.

[6] Dongari Gowtham. How to create new features using clustering!! https://towardsdatascience.com/how-to-create-new-features-using-clustering-4ae772387290, 2017. Accessed: 2023-12-08.

[7] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

[8] Kevin Menear. Comparing label encoding, one-hot encoding, and binary encoding for handling categorical variables in machine learning. https://medium.com/@kevin.menear/comparing-label-encoding-one-hot-encoding-and-binary-encoding-for-handling-categorical-variables-933544ccbd02: :text=Information Accessed: 2023-12-07.

[9] Jason Brownlee. How to use power transforms for machine learning. https://machinelearningmastery.com/power-transforms-with-scikit-learn/, Aug 2020. Accessed: 2023-12-07.

[10] Jason Brownlee. How to use quantile transforms for machine learning. https://machinelearningmastery.com/quantile-transforms-for-machine-learning//, Aug 2020. Accessed: 2023-12-07.

[11] Theos Evgeniou and Spyros Zoumpoulis. Classification for credit card default. https://inseaddataanalytics.github.io/INSEADAnalytics/CourseSessions/ClassificationProcessCreditCardDefault.html, Dec 2023. Accessed: 2023-12-08.