CM3015 Machine Learning and Neural Networks

Midterm Paper

Student ID: 220516639
Name: Teo Jia En
Date of Submission: 5th January 2025

Abstract

This report is written in fulfilment of CM3015 (Machine Learning and Neural Networks) midterm coursework.

The purpose of this project is to find the best Machine Learning (ML) model for the classification task of determining whether a customer will default on the payment of credit card.

To this end, the paper will include a comprehensive analysis of how different ML algorithms perform on the same dataset.

Specific algorithms used are:

- K-Nearest Neighbours
- Naive Bayes
- Logistic Regression
- Decision Tree

The performances of the models are then evaluated based on their precision, accuracy and recall.

1. Introduction

Aim and objectives

The aim of the project is to find the best Classification Machine Learning model to predict the probability of default of credit card clients. We will compare the performance of K-Nearest Neighbours (KNN), Naive Bayes, Logistic Regression and Decision Trees. Three algorithms - KNN, Naive Bayes annd Logistic Regression will be implemented from scratch.

A model will predict whether a default will occur in the following month, with Yes = 1 and No = 0.

The project will include cross-validation and hyperparameter tuning. By the end of this project, we will have the optimal model parameters and the best classifier.

This project will contribute to the discourse by demystifying foundational algorithms and basic techniques that underpins modern credit default prediction systems. This knowledge will lay the groundwork for exploration into more complex prediction methods.

Dataset

The dataset used, named 'Default of Credit Card Clients' is acquired from the UC Irvine Machine Learning Repository. It was donated by Yeh and Lien in January 2016. [1] It contains 30,000 instances of Taiwanese credit card clients and 23 features. It is licensed under Creative Commons Attribution 4.0 International (CC BY 4.0).

This dataset is chosen for its high usability. The multivariate dataset reflects real-world conditions where relationships between variables are complex and diverse. The features selected has also proven effective in its predictive accuracy.

Ethical considerations

Algorithm bias occurs when the models produce biased results that perpetuate human biases and systemic inequality. This is especially prevelant in training datasets about bank loans, where overrepresentation or underrepresentation of disadvantaged groups result in discriminatory outcomes. [2] This project will not examinine or detect such bias, as it falls outside the scope of the coursework.

As such, ML models produced by this project should not be used in place of an authority on the subject.

In addition, any future works have to comply with the updated Conditions of Use by the owner of the dataset.

2. Dataset pre-processing

In this section, we will prepare the data by handling any missing values. We will perform scaling to ensure optimal performance of distance-based algorithms, used by classifiers such as KNN.

In [1]:
```
"""
LIBRARIES USED FOR THIS PROJECT
"""
import numpy as np # for data manipulation
import pandas as pd # for data exploration and processing


# to download the dataset
!pip install ucimlrepo

# to import the method used to import datasets
from ucimlrepo import fetch_ucirepo

# to fetch the dataset, which has an id of 350
default_of_credit_card_clients = fetch_ucirepo(id=350)

import seaborn as sns
import matplotlib.pyplot as plt

# data (loaded as pandas dataframes)
X = default_of_credit_card_clients.data.features
y = default_of_credit_card_clients.data.targets

import warnings
warnings.filterwarnings('ignore')
```

```
Requirement already satisfied: ucimlrepo in /opt/anaconda3/lib/python3.12/site-packages (0.0.7)
Requirement already satisfied: pandas>=1.0.0 in /opt/anaconda3/lib/python3.12/site-packages (from ucimlrepo) (2.2.2)
Requirement already satisfied: certifi>=2020.12.5 in /opt/anaconda3/lib/python3.12/site-packages (from ucimlrepo) (2
024.8.30)
Requirement already satisfied: numpy>=1.26.0 in /opt/anaconda3/lib/python3.12/site-packages (from pandas>=1.0.0->uci
mlrepo) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /opt/anaconda3/lib/python3.12/site-packages (from pandas>=
1.0.0->ucimlrepo) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /opt/anaconda3/lib/python3.12/site-packages (from pandas>=1.0.0->ucim
lrepo) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in /opt/anaconda3/lib/python3.12/site-packages (from pandas>=1.0.0->uc
imlrepo) (2023.3)
Requirement already satisfied: six>=1.5 in /opt/anaconda3/lib/python3.12/site-packages (from python-dateutil>=2.8.2-
>pandas>=1.0.0->ucimlrepo) (1.16.0)
```

2.1 Cursory look at dataset

```
In [2]: X.describe()
```

Out[2]:

	X1	X2	X3	X4	X5	X6	X7	X8
count	30000.000000	30000.000000	30000.000000	30000.000000	30000.000000	30000.000000	30000.000000	30000.000000
mean	167484.322667	1.603733	1.853133	1.551867	35.485500	-0.016700	-0.133767	-0.166200
std	129747.661567	0.489129	0.790349	0.521970	9.217904	1.123802	1.197186	1.196868
min	10000.000000	1.000000	0.000000	0.000000	21.000000	-2.000000	-2.000000	-2.000000
25%	50000.000000	1.000000	1.000000	1.000000	28.000000	-1.000000	-1.000000	-1.000000
50%	140000.000000	2.000000	2.000000	2.000000	34.000000	0.000000	0.000000	0.000000
75%	240000.000000	2.000000	2.000000	2.000000	41.000000	0.000000	0.000000	0.000000
max	1000000.000000	2.000000	6.000000	3.000000	79.000000	8.000000	8.000000	8.000000

8 rows × 23 columns

In the code block above, we call the describe() method on the Pandas DataFrame to generate descriptive statistics that summarise central tendency and distribution of the features. We can see that there are 30000 data points for each of the features. Notably, we can see that the features are of different scales - the mean for X1 is 167,484 while the mean for X2 is 1.6. This implies a need for scaling as part of the pre-processing step.

In [3]:
```
# Check whether the features has any missing values
X.isnull().values.any()
```

Out[3]:
```
False
```

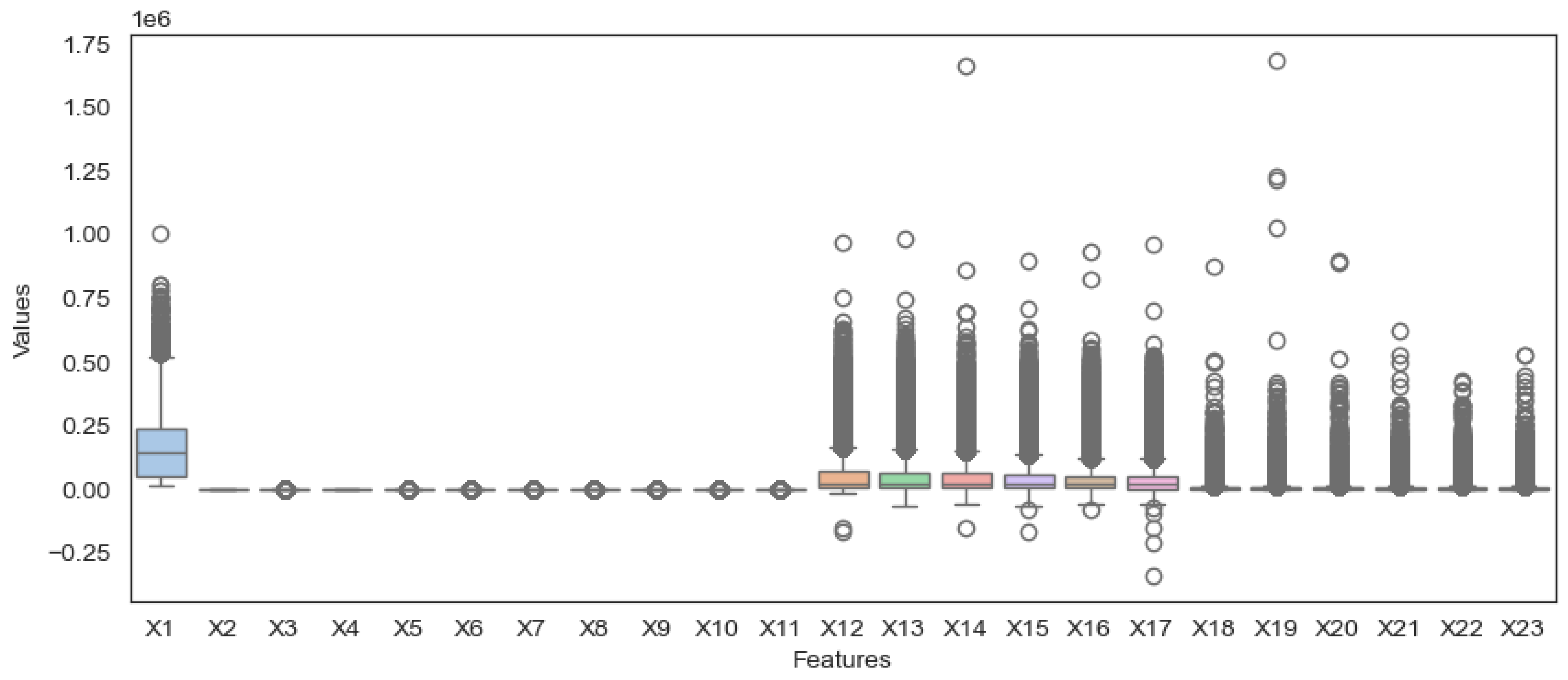
In the code above, we check if there are any missing values. Missing values have to be removed to ensure the integrity of the dataset, and to prevent skewed predictions by our machine learning models. Fortunately, there are no missing values in this dataset.

2.3 Removing outliers with Inter Quartile Range (IQR)

Classification models such as NaiveBayes and KNN are sensitive to outliers - an anamalous data point that deviates significantly from the rest. A single outlier will dramatically change class boundaries, which leads to inaccurate models. Hence, to ensure that the ML models are not leveraged towards outlier data points, outliers in the dataset must be removed. [3]

In [4]:
```
import seaborn as sns
import numpy as np
sns.set_style("white")
```

In [5]:
```
fig, ax =  plt.subplots(figsize=(10,4))
# sns.set_theme(rc={'figure.figsize':(10,4)})
sns.boxplot(data = X, orient = "v", palette = "pastel", ax = ax)
ax.set(xlabel='Features', ylabel='Values');
```



In order to identify outliers, we use the IQR (Inter-quartile Range) method to identify the data that falls outside our intended scope of observations. We first find the midpoint of the dataset and define Quartile 1 and Quartile 3. We then determine the extreme, caculated by adding and subtracting 1.5 of the iqr. [4]

In this project, we use the 15th and 85th percetile as it gives us a higher accuracy of the model. The calculation is facilitated using the percentile method from the NumPy library. [5]

In [6]:
```
## Reference: https://www.geeksforgeeks.org/detect-and-remove-the-outliers-using-python/

pd.options.mode.chained_assignment = None

# for each feature in the dataset (23 features)
for feature in X.columns:
    # calculate the 15th percentile
    quartile1 = np.percentile(X[feature],15, method = 'midpoint')
```

```python
    # calculate the 85th percentile
    quartile3 = np.percentile(X[feature],85, method = 'midpoint')

    # calculate the IQR
    iqr = quartile3 - quartile1

    # calculate the lower bound (extreme below which data will be discarded)
    lower_bound = quartile1 - 1.5 * iqr

    # calculate the uper bound (extreme above which data will be discarded)
    upper_bound = quartile3 + 1.5 * iqr

    # identify the row index where there exists an outlier, determined by upper and lower bound
    lower_array = np.where(X[feature] <= lower_bound)[0] # index of bottom outliers
    upper_array = np.where(X[feature] >= upper_bound)[0] # index of top outliers


    # using try and except to
    try:
        # Create a combined DataFrame for easier indexing
        df = pd.concat([X, y], axis=1)

        # Identify rows to drop based on conditions
        rows_to_drop = df.index[(df < bottom_bound) | (df > top_bound)].tolist()

        # Drop the identified rows from both X and y
        X = X.drop(index=rows_to_drop)
        y = y.drop(index=rows_to_drop)


    except:
        pass

print(X.shape, y.shape) # to check that the number of rows are equal
```

```
(30000, 23) (30000, 1)
```

2.4 Scaling

Scaling of features is an important pre-processing step that transforms features to a common scale. This normalises the distribution and magnitude of features and ensure that each feature contribute equally to the model.

Feature scaling is especially important in ML models like KNN classifiers, which utilises distance-based calculations for its predictions

Reference: https://medium.com/@shivanipickl/what-is-feature-scaling-and-why-does-machine-learning-need-it-104eedebb1c9

In [7]:
```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import pandas as pd

# Split the data into training and test

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42, stratify=y)
# X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)

# scaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = pd.DataFrame(scaler.transform(X_train))
X_test_scaled = pd.DataFrame(scaler.transform(X_test))
```

In [8]:
```python
print(f"Shape of training data is {y_train.shape}")
print(f"Shape of testing data is {y_test.shape}")
```

```
Shape of training data is (18000, 1)
Shape of testing data is (12000, 1)
```

The code snippet below demonstrates the importance of using scaled data. The accuracy score increases from 0.738 to 0.994 after scaling.

In [9]:
```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Classifier without scaling
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train.values, y_train.values.ravel())
predictions = knn.predict(X_test.values)
accuracy_nonscaled = accuracy_score(y_test.values.ravel(), predictions)
```

```
print(f"Accuracy without scaling: {accuracy_nonscaled:.3f}")

# Classifier with scaling
knn_scaled = KNeighborsClassifier(n_neighbors=3)
knn_scaled.fit(X_train_scaled.values, y_train.values.ravel())
scaled_predictions = knn_scaled.predict(X_test_scaled.values)
accuracy_scaled = accuracy_score(y_test.values.ravel(), scaled_predictions)
print(f"Accuracy with scaling: {accuracy_scaled:.3f}")
```

```
Accuracy without scaling: 0.738
Accuracy with scaling: 0.778
```

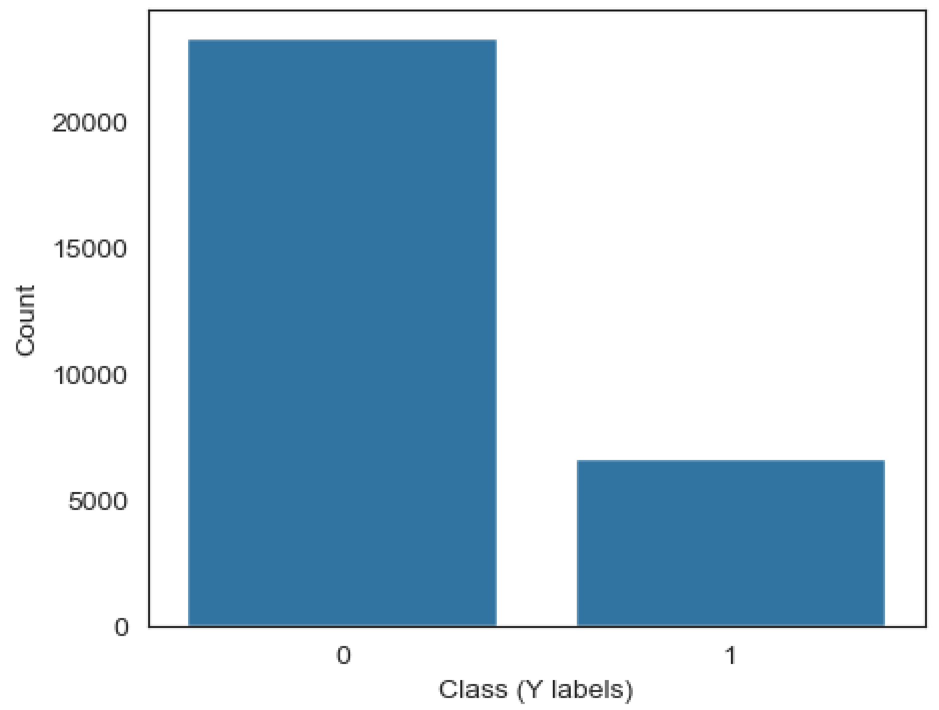
3. Exploratory Data Analysis

In this section, we will do a cursory statistical analysis of the dataset to analyse the balance of the clasess, and determine the evaluation metric we should use for the classifiers. If the dataset is balanced, the accuracy score will b used. If the dataset is imbalanced. The F1 score (or AUC) will be used.

We will then use pair plots, scatter plots and histograms to find how each feature would contribute to the prediction.

3.1 Class imbalance

In [10]:
```
fig, ax =  plt.subplots(figsize=(5,4))
s = y.value_counts().to_frame()
sns.barplot(data = s, x = 'Y', y = 'count', orient = 'v');
ax.set(xlabel='Class (Y labels)', ylabel='Count');
```



```
In [11]: y.value_counts()
```

```
Out[11]: Y
         0    23364
         1     6636
         Name: count, dtype: int64
```

https://developers.google.com/machine-learning/crash-course/overfitting/imbalanced-datasets#:~:text=Imbalanced%20datasets%20sometimes%20don%27t,learn%20enough%20about%20positive%20labels.

We can see that the dataset is imbalanced - there are significantly more data classified as class 0 than class 1. This may potentially pose a problem in our machine learning model as the model will train exclusively on the predominant label '1', and will not learn enough about the minority class '0'.

Since around 20% of the dataset belong to the minority class, this is a mild degree of imbalance. We will train on the original dataset without downsampling or upweighting.

3.2 Correlation matrix for features and label

In the following section, we will explore the correlation coefficients between the features and the label. This will show us potential relationships amongst the dataset.

In [12]:
```
# Convert labels into a dataframe object
y = pd.DataFrame(y, columns = ['Y'])

# Join features and labels to find any correlation
full_df = X
X['Y'] = y

 # plot the correlation matrix between the features and the label
cor_mat = full_df.corr()
cor_mat.style.background_gradient(cmap = 'GnBu')
```

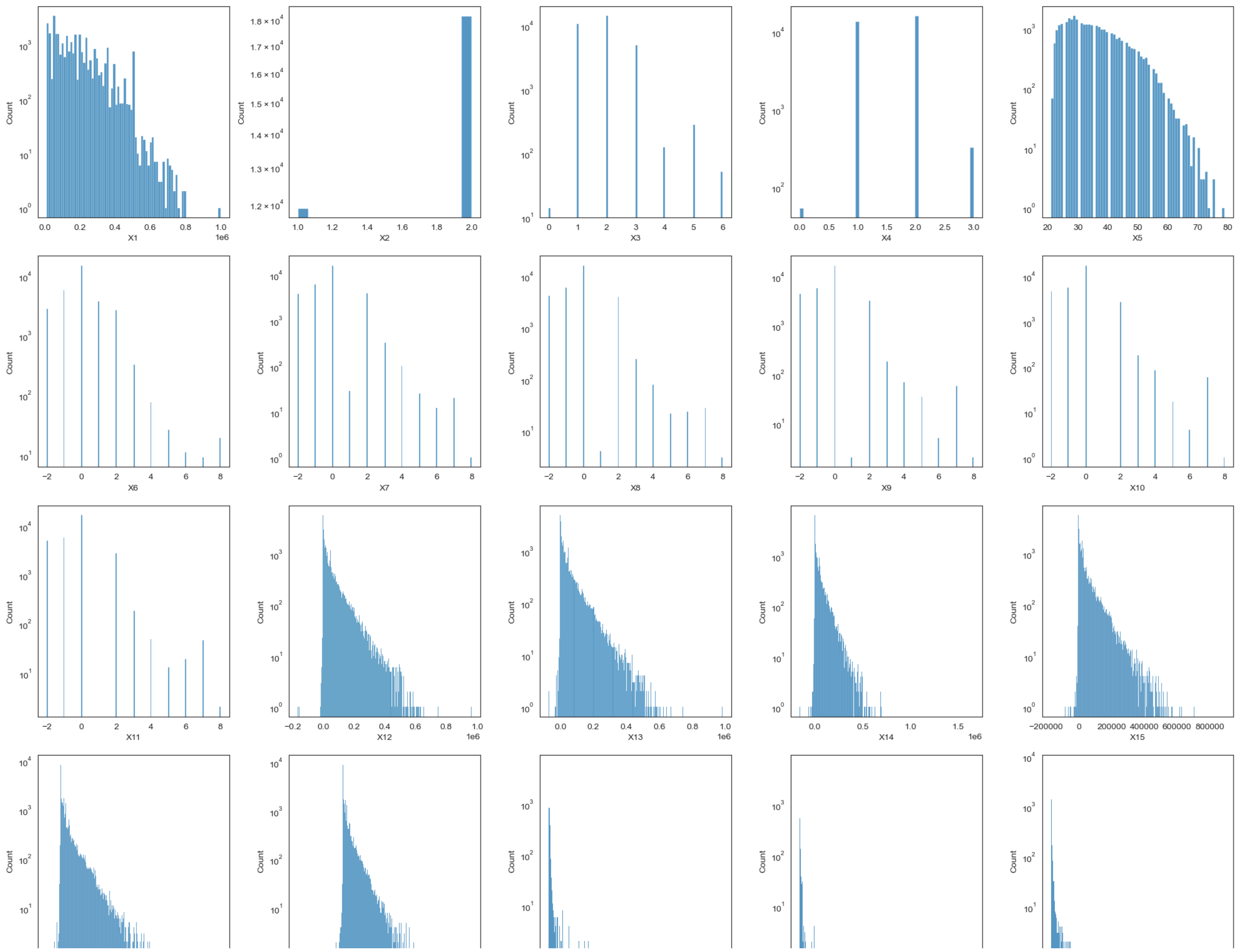
Out[12]:

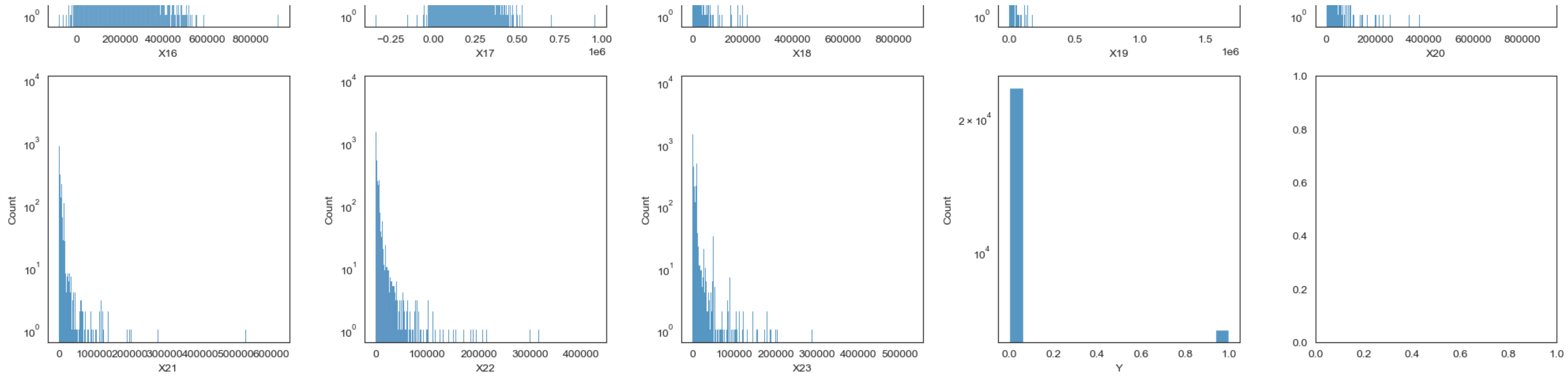
	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11
X1	1.000000	0.024755	-0.219161	-0.108139	0.144713	-0.271214	-0.296382	-0.286123	-0.267460	-0.249411	-0.235195
X2	0.024755	1.000000	0.014232	-0.031389	-0.090874	-0.057643	-0.070771	-0.066096	-0.060173	-0.055064	-0.044008
X3	-0.219161	0.014232	1.000000	-0.143464	0.175061	0.105364	0.121566	0.114025	0.108793	0.097520	0.082316
X4	-0.108139	-0.031389	-0.143464	1.000000	-0.414170	0.019917	0.024199	0.032688	0.033122	0.035629	0.034345
X5	0.144713	-0.090874	0.175061	-0.414170	1.000000	-0.039447	-0.050148	-0.053048	-0.049722	-0.053826	-0.048773
X6	-0.271214	-0.057643	0.105364	0.019917	-0.039447	1.000000	0.672164	0.574245	0.538841	0.509426	0.474553
X7	-0.296382	-0.070771	0.121566	0.024199	-0.050148	0.672164	1.000000	0.766552	0.662067	0.622780	0.575501
X8	-0.286123	-0.066096	0.114025	0.032688	-0.053048	0.574245	0.766552	1.000000	0.777359	0.686775	0.632684
X9	-0.267460	-0.060173	0.108793	0.033122	-0.049722	0.538841	0.662067	0.777359	1.000000	0.819835	0.716449
X10	-0.249411	-0.055064	0.097520	0.035629	-0.053826	0.509426	0.622780	0.686775	0.819835	1.000000	0.816900
X11	-0.235195	-0.044008	0.082316	0.034345	-0.048773	0.474553	0.575501	0.632684	0.716449	0.816900	1.000000
X12	0.285430	-0.033642	0.023581	-0.023472	0.056239	0.187068	0.234887	0.208473	0.202812	0.206684	0.207373
X13	0.278314	-0.031183	0.018749	-0.021602	0.054283	0.189859	0.235257	0.237295	0.225816	0.226913	0.226924
X14	0.283236	-0.024563	0.013002	-0.024909	0.053710	0.179785	0.224146	0.227494	0.244983	0.243335	0.241181
X15	0.293988	-0.021880	-0.000451	-0.023344	0.051353	0.179125	0.222237	0.227202	0.245917	0.271915	0.266356
X16	0.295562	-0.017005	-0.007567	-0.025393	0.049345	0.180635	0.221348	0.225145	0.242902	0.269783	0.290894
X17	0.290389	-0.016733	-0.009099	-0.021207	0.047613	0.176980	0.219403	0.222327	0.239154	0.262509	0.285091
X18	0.195236	-0.000242	-0.037456	-0.005979	0.026147	-0.079269	-0.080701	0.001295	-0.009362	-0.006089	-0.001496
X19	0.178408	-0.001391	-0.030038	-0.008093	0.021785	-0.070101	-0.058990	-0.066793	-0.001944	-0.003191	-0.005223
X20	0.210167	-0.008597	-0.039943	-0.003541	0.029247	-0.070561	-0.055901	-0.053311	-0.069235	0.009062	0.005834
X21	0.203242	-0.002229	-0.038218	-0.012659	0.021379	-0.064005	-0.046858	-0.046067	-0.043461	-0.058299	0.019018
X22	0.217202	-0.001667	-0.040358	-0.001205	0.022850	-0.058190	-0.037093	-0.035863	-0.033590	-0.033337	-0.046434
X23	0.219595	-0.002766	-0.037200	-0.006641	0.019478	-0.058673	-0.036500	-0.035861	-0.026565	-0.023027	-0.025299

	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11
Y	-0.153520	-0.039961	0.028006	-0.024339	0.013890	0.324794	0.263551	0.235253	0.216614	0.204149	0.186866

The last row shows the correlation between the features and the label (Y). As shown in the table above, X6 is the most corelation with Y, with a value of 0.324884, while x7 has the least correlation with Y, with a value of -0.005763.

In [13]:
```
fig, axs = plt.subplots(5,5, layout = "tight", figsize = (20,20))
import math

for i, column in enumerate(full_df.columns):
    sns.histplot(x = full_df[column], ax = axs[math.floor(i/5), i%5], log = True)
```


Count
X1
1e6
X2
X3
X4
X5
X6
X7
X8
X9
X10
X11
X12
X13
X14
X15



4. Implementation of Algorithm

In this section, I will be implementing three algorithms from scratch, and one algorithm from Scikit-learn. The three classification algorithms crafted from scratch are - k-NN, Naïve Bayes and Logistic Regression. The final algorithm from Scikit-learn is a Decision Tree Classifier.

In [14]:
```python
"""
Libraries used to build and evaluate the Machine Learning algorithms.
"""
import plotly.express as px  # plotly express to create interactive charts

from sklearn.metrics import f1_score, accuracy_score, precision_score, recall_score, roc_auc_score
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.model_selection import GridSearchCV, KFold, cross_validate, cross_val_score
from sklearn.utils.multiclass import unique_labels

from scipy.stats import mode # mode method returns an array of the most common values
from sklearn import tree # tree class for Decision Tree Classifier
from collections import Counter # supports convenient and rapid talles

# BaseEstimator is the base class and inheritance provides default implementations of estimators in scikit-learn
# ClassifierMixin provides the score method and enforces fit to require y
```

```
from sklearn.base import BaseEstimator, ClassifierMixin


from sklearn.naive_bayes import GaussianNB
from sklearn.decomposition import PCA # to perform principal component analysis
from sklearn.manifold import TSNE # to perform TSNE
from sklearn.preprocessing import PolynomialFeatures
```

In [15]:
```
y.shape # find the shape of the labels
```

Out[15]:
```
(30000, 1)
```

In order to judge the performance of our algorithms, we must establish a baseline model to set the standard for evaluation. The baseline model would provide a reference point, and help us measure how much value our custom models bring. For the purpose of this project, the baseline would be the fraction of the majority class.

As seen in Section 3, we have 23364 data points belonging to class 0 and 6636 belong to class 1. We calculate the baseline accuracy as the fraction of the count of the majority class over the total count of the data points.

In [16]:
```
# Baseline
baseline_accuracy = y.value_counts()[0] / y.count()

print(baseline_accuracy)
```

```
Y    0.7788
dtype: float64
```

Hence, the baseline accuracy will be 0.7788. For an algorithm to be meaningful, it must perform better than this baseline.

4.1 K-Nearest Neighbours (From Scratch)

In the code below, I will implement the K-Nearest Neighbours from scratch. KNN predicts the class of a data point based on the majority class among the 'k' nearest neighbours. It is distance-based, and can learn complex decision boundaries. The performance of this model depends on the 'k' chosen.

In [17]:
```python
class KNN_Scratch(BaseEstimator, ClassifierMixin):
    """
    This is the KNN classifier implemented from scratch. Inheriting from BaseEstimator and ClassifierMixin,
    we define the methods init, fit, pairwise and predict in the format that allows us to use
    Scikit-learn's libraries and helper functions.
    """
    def __init__(self, k = 3, distance_metric='euclidean'):
        """
        Initialises the model with a default k value of 3. The default distance metric used is euclidean.
        This class does not support other distance metrics.
        """
        self.classes_ = None
        self.k = k # get the number of neighbours initialised with the KNN instance
        self.distance_metric = distance_metric # get the distance metric

    def fit(self, X, y):
        """
        Fits the model with the training dataset. Initialises the X, y, classes and shape of the dataset.
        """
        self.X_train_ = X # fit the training data set
        self.y_train_ = y # fit the labels for the training dataset
        self.classes_ = unique_labels(y) # get the classes as y
        self.m, self.n = X_train.shape # get the shape of the data
        return self

    def pairwise(self, X, point):
        """
        Calculates the function between two points.
        """
        # using Euclidenan formula
        return np.sqrt(np.sum(np.square(point-X))) # this is the Euclidiean formula


    def predict(self, X):
        """
        Predict the class
        """
        # array to store predictions
        y_predictions = []
        m, n = X.shape # get the shape of the test dataset

        # for as many data points
        for i in range(m):

            # get the pairwise distances
            distances = self.pairwise(self.X_train_, X.iloc[i]).ravel()

            # get the indices of the first k closest distances
            nearest_indx = np.argsort(distances)[:self.k]

            # get the label of the nearest distances (x correspond to y)
            nearest_labels = (self.y_train_.iloc[nearest_indx])

            # compute majority using value counts - the majority class will be on the first index
            prediction = nearest_labels.value_counts().index[0]

            # if the prediction class is 1, store 1 as the predicted value. Otherwise, store 0.
            predicted_val = 1 if (prediction == 1) else 0
            y_predictions.append(predicted_val)

        # return the predictions as an array, to use during the testing stage
        return np.array(y_predictions)
```

In [123…

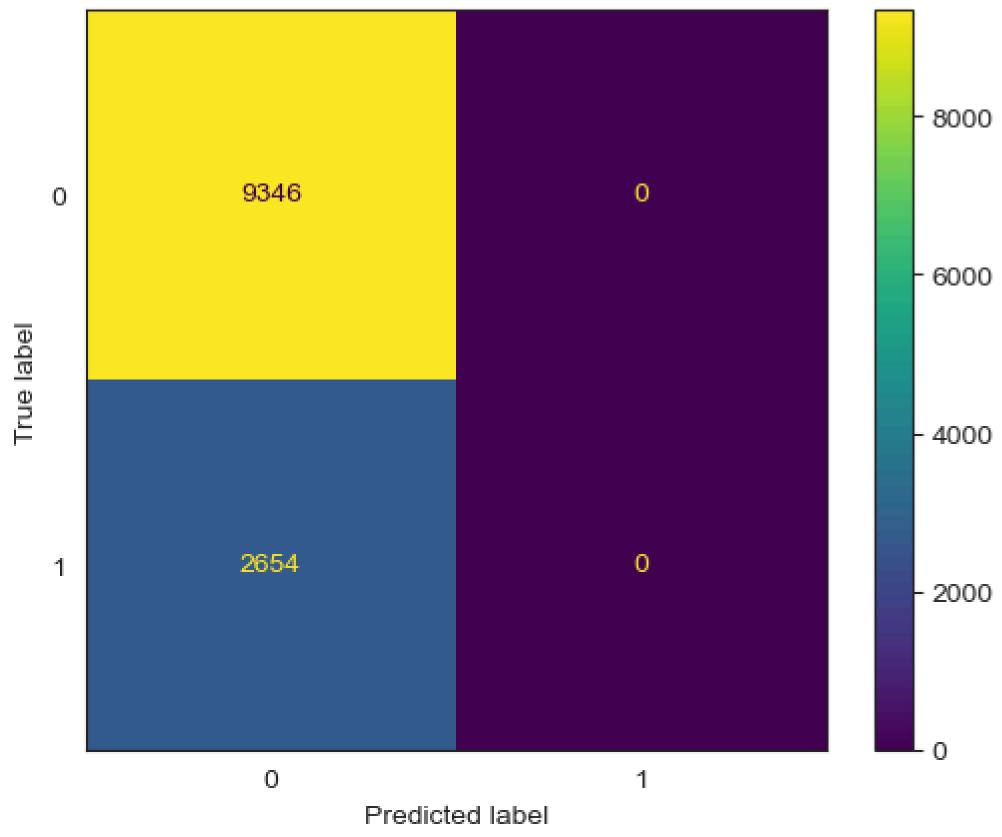
```
knn = KNN_Scratch(k=3, distance_metric = 'euclidean')
knn.fit(X_train_scaled, y_train)
predictions = knn.predict(X_test_scaled)
predictions_trained = knn.predict(X_train_scaled)

# F1 score, Accuracy score and ROC AUC score for the custom KNN on training data
custom_knn_f1_train = f1_score(y_train, predictions_trained, average='weighted', labels=np.unique(predictions_train
custom_knn_accuracy_train = accuracy_score(y_train, predictions_trained)
custom_knn_auc_train = roc_auc_score(y_train, predictions_trained)

# F1 score for the custom KNN
custom_knn_f1 = f1_score(y_test, predictions, average='weighted', labels=np.unique(predictions))

# Accuracy score for the custom KNN
custom_knn_accuracy = accuracy_score(y_test, predictions)
```

```python
# ROC AUC score for the custom KNN
custom_knn_auc = roc_auc_score(y_test, predictions)

# print statements for training scores
print(f"F1 score for KNN from scratch (Training): {custom_knn_f1_train:.5f}")
print(f"Accuracy score for KNN from scratch(Training): {custom_knn_accuracy_train:.5f}")
print(f"ROC AUC score for KNN from scratch(Training): {custom_knn_auc_train:.5f}\n")

# print statements for testing scores
print(f"F1 score for KNN from scratch: {custom_knn_f1:.5f}")
print(f"Accuracy score for KNN from scratch: {custom_knn_accuracy:.5f}")
print(f"ROC AUC score for KNN from scratch: {custom_knn_auc:.5f}")
```

```
F1 score for KNN from scratch (Training): 0.87563
Accuracy score for KNN from scratch(Training): 0.77878
ROC AUC score for KNN from scratch(Training): 0.50000

F1 score for KNN from scratch: 0.87567
Accuracy score for KNN from scratch: 0.77883
ROC AUC score for KNN from scratch: 0.50000
```

In [19]:
```python
matrix = confusion_matrix(y_test, predictions, labels=knn.classes_)
cm_display = ConfusionMatrixDisplay(confusion_matrix=matrix, display_labels=knn.classes_)
cm_display.plot()
```

Out[19]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x3174c1a00>
```



There are 0 false positives (predict that the client will default) and 0 true negatives (predict that the client will not default and it is true). This means that the model is unable to make a correct positive prediction.

Summary of Custom KNN classifier

In the code block below, we implemented the KNN from scratch, with the number of k-neighbours set as 3. With this algorithm, we achieved a F1 score of 0.87566 and an Accuracy score of 0.77883. We can perform hyperparameter tuning to find the best k value using GridSearchCV.

A F1 score of 0.87 tells us that we achieved a good balance between precision and recall. While our Accuracy Score of 0.77 is lower, it is to be expected as the dataset is imbalanced.

Fine-tuning for k-neighbours and distance metric with Grid Search

Reference: https://machinelearningknowledge.ai/knn-classifier-in-sklearn-using-gridsearchcv-with-example/

In the code block below, we will perform GridSearchCV on Scikit-learn's KNieghboursClassifer to tune find the best hyper-parameters.

In [20]:
```
params = {
    'n_neighbors': [1,3,5,7,9],
    'metric' : ['euclidean', 'manhatten']
}
grid_knn = GridSearchCV(KNeighborsClassifier(), params, cv = 5, scoring = 'f1_weighted',return_train_score = True)

grid_knn.fit(X, y)

# best model
best = grid_knn.best_estimator_
best_params = grid_knn.best_params_
grid_knn_f1 = grid_knn.best_score_
```

In [21]:
```
print(f'Best parameters for KNN: {grid_knn.best_params_['metric'], grid_knn.best_params_['n_neighbors']}')
print(f'Best score with GridSearchCV: {grid_knn.best_score_:.5f}')
```

```
Best parameters for KNN: ('euclidean', 5)
Best score with GridSearchCV: 0.72059
```

Result of hyper-parameters tuning

As seen above, the best parameters for Scikit-learn's KNN Classifier is a k value of 5 and euclidean as the distance metric. This gives us a F1 score of 0.72.

4.2 Naïve Bayes (From Scratch)

In the code below, I will implement the Naïve Bayes algorithm from scratch. Naïve Bayes uses Bayes' theorem with the assumption that features are independent.

```
In [22]: class NaiveBayes_Scratch(BaseEstimator, ClassifierMixin):
             """
             This is the Naive Bayes classifier implemented from scratch. Inheriting from BaseEstimator and ClassifierMixin,
             we define the methods init, fit, pairwise and predict in the format that allows us to use
             Scikit-learn's libraries and helper functions.
             """
             def __init__(self):
                 """
                 Initialises the Naive Bayes Classifier instance.
                 """
                 self.classes_ = None
                 # priors refer to the probability of a class occuring before other information is taken into account
                 self.priors = {}
                 # stats dictionary will store means and variances for the label
                 self.stats = {}

             def fit(self, X, y):
                 """
                 Fits the model with the training data, getting the prior probability, mean and variance.
                 """
                 # To find the unique classes - in this dataset, we expect two classes - 0 and 1
                 self.classes_ = unique_labels(y)

                 # Get the shape of the dataset
                 Xm, Xn = X.shape

                 # For each class, we have to calculate the mean, variances and priors
                 # This allows us to get the bell curve for each class.

                 for label in self.classes_:

                     # filter for the data points that belong to the class (samples)
                     X_class = X[y==label]

                     # get the shape of the samples
                     m, n = X_class.shape

                     # get prior probability (samples in class divided by samples in total)
                     self.priors[label] = m / Xm
```

```
            # get mean of samples
            means = np.mean(X_class, axis = 0)

            # get variance of samples
            variances = np.var(X_class, axis = 0)

            # store means as a dictionary
            self.stats[label] = {'means': means, 'variances': variances}

        return self

    # private function
    def _calculate_likelihood(self, x, mean, variance):
        """
        Calculate the log likelihood
        """
        exp = np.exp(-((x-mean)**2)/(2*variance))
        return (1/np.sqrt(2 * np.pi * variance)) * exp

    # get the list of probabilities
    def predict_probability(self, X):
        """
        Generate an array of the probabilities
        """
        # get probabilities as a list
        probabilities = []

        m, n = X.shape

        # for each instance
        for i in range(m):
            posteriors = {}

            # for each of the class
            for label in self.classes_:

                # get the log prior

                posterior = np.log(self.priors[label]) # initialise log likelihood

                # add log likelihood to each feature
                for j in range(n): # go through the features
                    mean = self.stats[label]['means'][j] # get the mean
                    variance = self.stats[label]['variances'][j] # get the variancec
                    # with mean and variance, get the posterior
                    posterior += np.log(self._calculate_likelihood(X.iloc[i][j], mean, variance))

                # the posterior for class 0 or class 1
                posteriors[label] = posterior

            # the code below is to normalise
            max_log = max(posteriors.values())
            # the exponent for the posteriors
            exponent_posteriors = {k: np.exp(v - max_log) for k, v in posteriors.items()}
            # get the sum of the exponents
            total = sum(exponent_posteriors.values())
            # get the log probabilities
            probabilities.append([exponent_posteriors[label]/ total for label in self.classes_])

        return np.array(probabilities) # return probabilities as an array

    def predict(self, X):
        """
        Get the maximum probability, based on the array of probabilities returned by predict_probability
        """
        probabilities = self.predict_probability(X)
        return np.argmax(probabilities, axis = 1) # get the class
```

In [122…

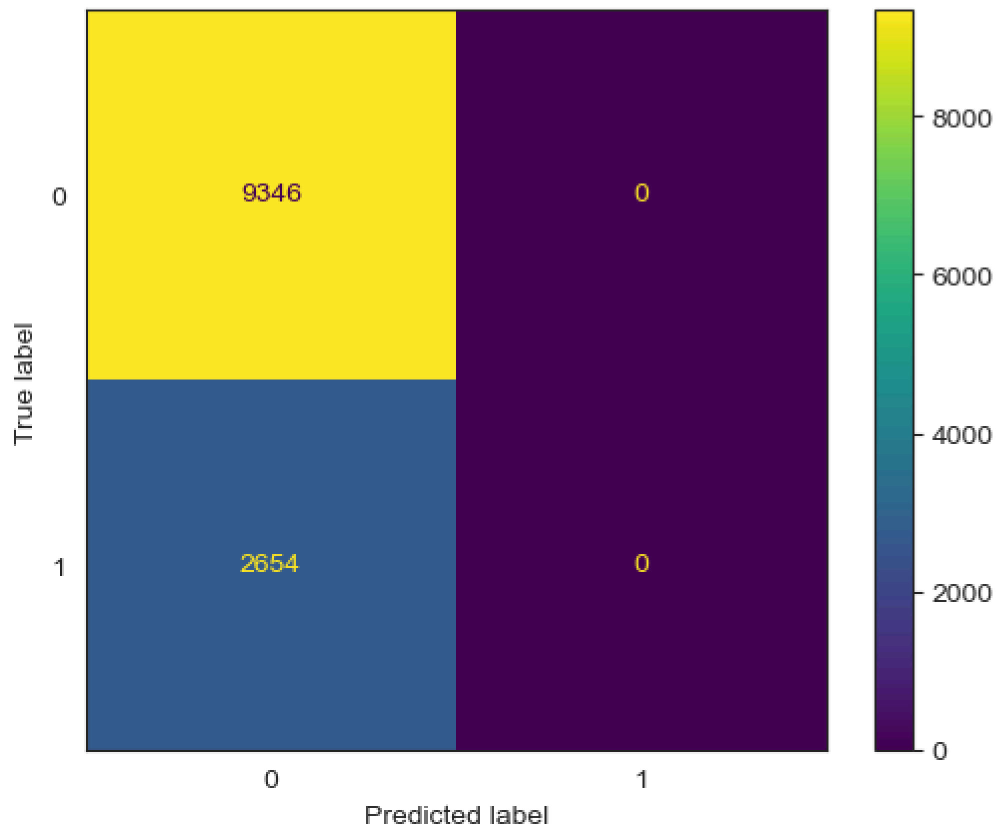
```
naivebayes = NaiveBayes_Scratch()
naivebayes.fit(X_train_scaled, y_train)
predictions = naivebayes.predict(X_test_scaled)
predictions_trained = naivebayes.predict(X_train_scaled)

# F1, Accuracy and ROC scores for the custom Naive Bayes on the training dataset
custom_naivebayes_f1_train = f1_score(y_train, predictions_trained, average='weighted', labels=np.unique(prediction
custom_naivebayes_accuracy_train = accuracy_score(y_train, predictions_trained)
custom_naivebayes_auc_train = roc_auc_score(y_train, predictions_trained)

# F1 score for the custom Naive Bayes
custom_naivebayes_f1 = f1_score(y_test, predictions, average='weighted', labels=np.unique(predictions))
```

```python
# Accuracy score for the custom Naive Bayes
custom_naivebayes_accuracy = accuracy_score(y_test, predictions)

# ROC AUC score for the custom Naive Bayes
custom_naivebayes_auc = roc_auc_score(y_test, predictions)

# Print statements for training data
print(f"F1 score for Naive Bayes from scratch (Training): {custom_naivebayes_f1_train:.5f}")
print(f"Accuracy score for Naive Bayes from scratch (Training): {custom_naivebayes_accuracy_train:.5f}")
print(f"ROC AUC score for Naive Bayes from scratch (Training): {custom_naivebayes_auc_train:.5f}\n")

# Print statements for testing data
print(f"F1 score for Naive Bayes from scratch: {custom_naivebayes_f1:.5f}")
print(f"Accuracy score for Naive Bayes from scratch: {custom_naivebayes_accuracy:.5f}")
print(f"ROC AUC score for Naive Bayes from scratch: {custom_naivebayes_auc:.5f}")
```

```
F1 score for Naive Bayes from scratch (Training): 0.87563
Accuracy score for Naive Bayes from scratch (Training): 0.77878
ROC AUC score for Naive Bayes from scratch (Training): 0.50000

F1 score for Naive Bayes from scratch: 0.87567
Accuracy score for Naive Bayes from scratch: 0.77883
ROC AUC score for Naive Bayes from scratch: 0.50000
```

```python
In [24]: matrix = confusion_matrix(y_test, predictions, labels=naivebayes.classes_)
         cm_display = ConfusionMatrixDisplay(confusion_matrix=matrix, display_labels=naivebayes.classes_)
         cm_display.plot()
```

```
Out[24]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1757b1430>
```



Summary of Custom Naive Bayes Classifier

As seen above, the F1 score for Naive Bayes from scratch reached 0.87566, which is on par with our custom KNN Classifier from scratch.

Fine-tuning for k-neighbours and distance metric with Cross Validation

In the section below, we will use KFold and Cross_Validate from the Scikit-learn library to perform cross validation. We will be using f1 and accuracy as the metrics for the cross validation scores.

In [27]:
```
k_fold = KFold(n_splits = 5, shuffle = True, random_state = 42)
scoring = ['f1_weighted','f1_micro','f1_macro','accuracy']
custom_nb_kf_scores = cross_validate(NaiveBayes_Scratch(), X, y, scoring = scoring,error_score="raise")
```

In [28]:
```
for score in custom_nb_kf_scores:
    print(f'{score}: {np.mean(custom_nb_kf_scores[score]):.5f}')
```

```
fit_time: 0.05592
score_time: 6.68533
test_f1_weighted: 0.68195
test_f1_micro: 0.77880
test_f1_macro: 0.43782
test_accuracy: 0.77880
```

The mean cross validated scores are 0.68193. This is a poorer performance than the KNN Classifier. This may be because Naive Bayes relies on Bayes' theorem and assumes independence among the features. In addition, KNN can achieve more complex decision boundaries.

4.3 Logistic Regression (From Scratch)

In the code bellow, I will implement logistic regression from scratch. Logistic regression predicts the class of a data point using a sigmoid function. It assumes a linear relationship between the class and the features.

In [29]:
```
import numpy as np
from scipy.special import expit # sigmoid
from sklearn.base import BaseEstimator, ClassifierMixin

class LogRegressor_Scratch(BaseEstimator, ClassifierMixin):
    """
    This is the Logistic Regression classifier implemented from scratch. Inheriting from BaseEstimator and Classifi
    we define the methods init, fit, pairwise and predict in the format that allows us to use
    Scikit-learn's libraries and helper functions.
    """
    def __init__(self, learning_rate = 0.01, max_iterations = 10000, tolerance = 1e-6):
        """
        Initialises the model
        """
        self.learning_rate = learning_rate
        self.max_iterations = max_iterations
        self.tolerance = tolerance
        self.weights = None
        self.classes_ = None

    def fit(self, X, y):
        """
        Fit the model with the training data. Get the weights using gradient descent.
        """
        # To find the unique classes - in this dataset, we expect two classes - 0 and 1
        self.classes_ = unique_labels(y)

        # add bias term
        self.classes_ = np.unique(y)
        X = np.hstack([np.ones((X.shape[0],1)),X])

        # initialise weights
        self.weights = np.zeros((X.shape[1],1))

        # reshape y
        y = y.values.reshape(-1,1)

        # perform gradient descent
        self.weights = self.gradient_descent(
            X, y, self.weights, self.learning_rate, self.max_iterations, self.tolerance
        )
        return self

    def predict_proba(self, X):
        """
        Return an array of the probabilities
        """
        # add bias term
        X = np.hstack([np.ones((X.shape[0],1)),X])
        # compute probabilities
        probabilities = expit(X @ self.weights)
        return np.hstack([1 - probabilities, probabilities])

    def predict(self,X):
        """
        Predict the class
        """
        # class labels
        probabilities = self.predict_proba(X)[:,1]
        return (probabilities >= 0.5).astype(int) # True or False, computed to int for Class 0 or 1

    def compute_gradient(self, X, y, weights):
        """
        Calculate the gradient with the logistic sigmoid function
        """
        # expit uses the sigmoid function
        predictions = expit(X @ weights) # perform matrix multiplication with X and the weights
        gradients = X.T @ (predictions - y) / X.shape[0] # matrix multiplication with transpose of X

        return gradients

    def gradient_descent(self,X, y, weights, learning_rate, max_iterations, tolerance):
        """
        Perform gradient descent and return the weights after either converging or reaching the max iterations
        """
        for iteration in range(max_iterations):
            gradients = self.compute_gradient(X, y, weights)
            weights = weights - learning_rate * gradients

            # checking for convergence
            if np.linalg.norm(gradients) < tolerance:
                print(f"converged after {iteration} iterations")
                break
        return weights
```

In [121…

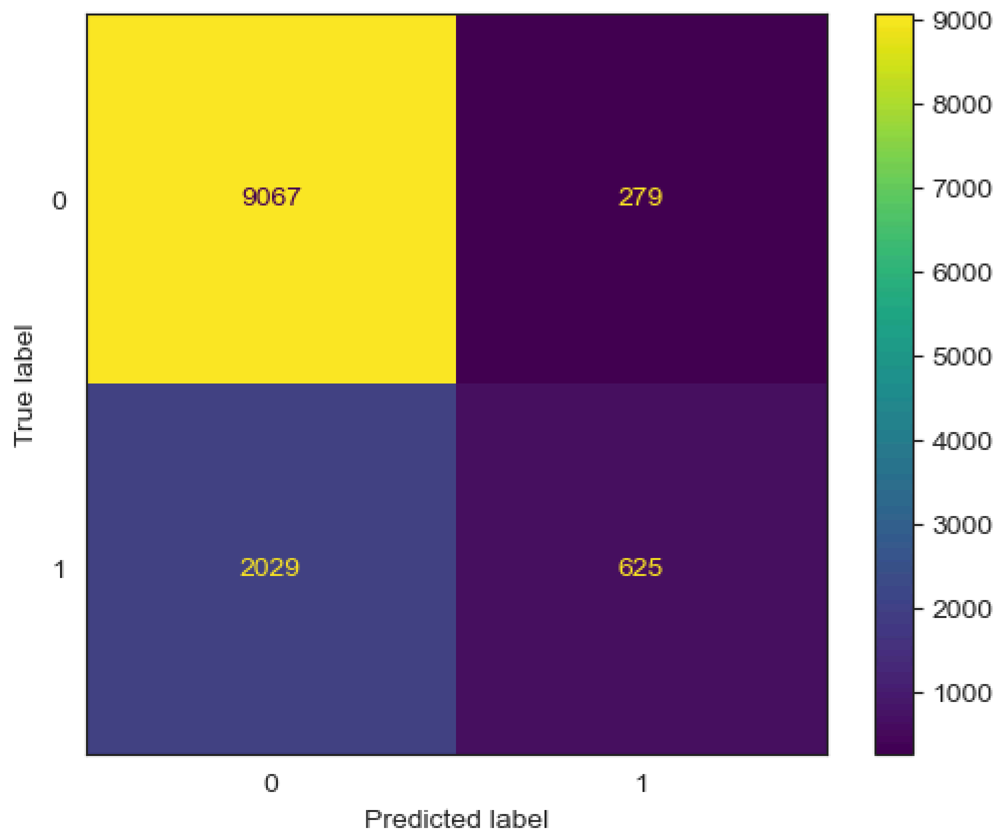
```
logres = LogRegressor_Scratch()
logres.fit(X_train_scaled, y_train)
predictions = logres.predict(X_test_scaled)
predictions_trained = logres.predict(X_train_scaled)

# F1 score for the custom Logistic Regressor
custom_logres_f1 = f1_score(y_test, predictions, average='weighted', labels=np.unique(predictions))

# Accuracy score for the custom Logistic Regressor
```

```
custom_logres_accuracy = accuracy_score(y_test, predictions)

# ROC AUC score for the custom Logistic Regressor
custom_logres_auc = roc_auc_score(y_test, predictions)

# F1, Accuracy and AUC scores for the training data
custom_logres_f1_train = f1_score(y_train, predictions_trained, average='weighted', labels=np.unique(predictions_tr
custom_logres_accuracy_train = accuracy_score(y_train, predictions_trained)
custom_logres_auc_train = roc_auc_score(y_train, predictions_trained)

# print statement for the training data
print(f"F1 score for Logistic Regression from scratch (Training): {custom_logres_f1_train:.5f}")
print(f"Accuracy score for Logistic Regression from scratch (Training): {custom_logres_accuracy_train:.5f}")
print(f"ROC AUC score for Logistic Regression from scratch (Training): {custom_logres_auc_train:.5f}\n")

# print statement for the scores
print(f"F1 score for Logistic Regression from scratch: {custom_logres_f1:.5f}")
print(f"Accuracy score for Logistic Regression from scratch: {custom_logres_accuracy:.5f}")
print(f"ROC AUC score for Logistic Regression from scratch: {custom_naivebayes_auc:.5f}")
```

```
F1 score for Logistic Regression from scratch (Training): 0.77759
Accuracy score for Logistic Regression from scratch (Training): 0.81406
ROC AUC score for Logistic Regression from scratch (Training): 0.61470

F1 score for Logistic Regression from scratch: 0.76860
Accuracy score for Logistic Regression from scratch: 0.80767
ROC AUC score for Logistic Regression from scratch: 0.50000
```

In [31]:
```
matrix = confusion_matrix(y_test, predictions, labels=logres.classes_)
cm_display = ConfusionMatrixDisplay(confusion_matrix=matrix, display_labels=logres.classes_)
cm_display.plot()
```

Out[31]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x3210b84a0>
```



Summary of Custom Logistic Regression Classifier

It is evident that the Logistic Regression performed poorer than both KNN and Naive Bayes. This may indicate that our data has no linear relationship.

5-fold cross validation with Cross_Val_Sore

In order to test the model's ability in predicting new data, we perform cross-validation. The Cross_Val_Score helper function splits the dataset into 5 folds by default and build a model for each fold. By checking the performance of the model on each fold, we can evaluate how well the model works on unseen data.

```
In [32]: logres_cross_val = cross_val_score(LogRegressor_Scratch(),X, y, scoring = 'f1_weighted', error_score = 'raise')
         logres_cross_val
```

```
Out[32]: array([0.68177111, 0.68199909, 0.68199909, 0.68199909, 0.68199909])
```

4.4 Decision Tree (Scikit-Learn)

In this section, I will implement the DecisionTreeClassifier from Scikit-learn's Tree library. It uses a tree structure to classify the data points. By setting the class weight to balanced, the Decision Tree Classifier sets the weights to be inversely proportionate to the frequencies of each class.

```
In [116…
clf = tree.DecisionTreeClassifier(class_weight = "balanced")
clf = clf.fit(X_train_scaled,y_train)
predictions = clf.predict(X_test_scaled)
predictions_train = clf.predict(X_train_scaled)

# F1 score for the Decision Tree
decision_tree_f1 = f1_score(y_test, predictions)

# Accuracy score for the Decision Tree
decision_tree_accuracy = accuracy_score(y_test, predictions)

# ROC AUC score for the Decision Tree
decision_tree_auc = roc_auc_score(y_test, predictions)

# F1 score for the Decision Tree
decision_tree_f1_train = f1_score(y_train, predictions_train)

# Accuracy score for the Decision Tree
decision_tree_accuracy_train = accuracy_score(y_train, predictions_train)

# ROC AUC score for the Decision Tree
decision_tree_auc_train = roc_auc_score(y_train, predictions_train)

print(f"F1 score for Decision Tree (Train): {decision_tree_f1_train:.5f}")
print(f"Accuracy score for Decision Tree (Train): {decision_tree_accuracy_train:.5f}")
print(f"ROC AUC score for Decision Tree (Train): {decision_tree_auc_train:.5f}")
print(f"F1 score for Decision Tree: {decision_tree_f1:.5f}")
print(f"Accuracy score for Decision Tree: {decision_tree_accuracy:.5f}")
print(f"ROC AUC score for Decision Tree: {decision_tree_auc:.5f}")
```

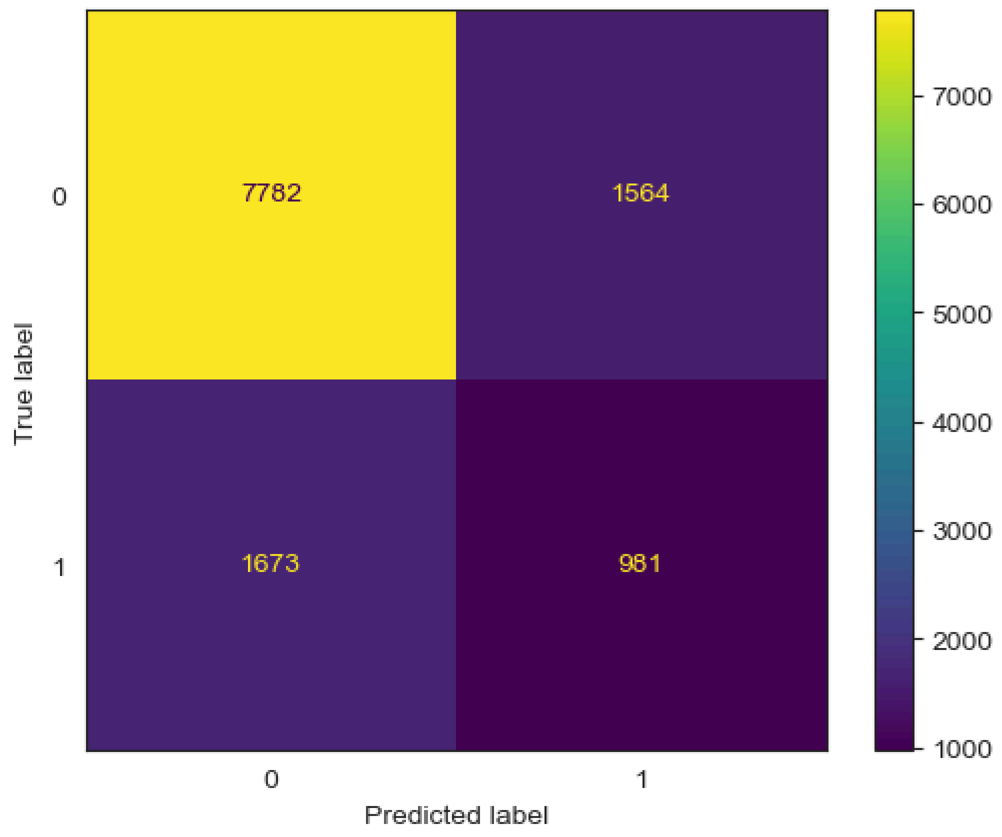
```
F1 score for Decision Tree (Train): 0.99862
Accuracy score for Decision Tree (Train): 0.99939
ROC AUC score for Decision Tree (Train): 0.99961
F1 score for Decision Tree: 0.38086
Accuracy score for Decision Tree: 0.73042
ROC AUC score for Decision Tree: 0.60314
```

In [34]:
```
X_train.shape, X_test.shape
```

Out[34]:
```
((18000, 23), (12000, 23))
```

In [35]:
```
matrix = confusion_matrix(y_test, predictions, labels=clf.classes_)
cm_display = ConfusionMatrixDisplay(confusion_matrix=matrix, display_labels=clf.classes_)
cm_display.plot()
```

Out[35]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x3210a0170>
```



5. Feature Engineering

5.1 Principal Component Analysis (PCA)

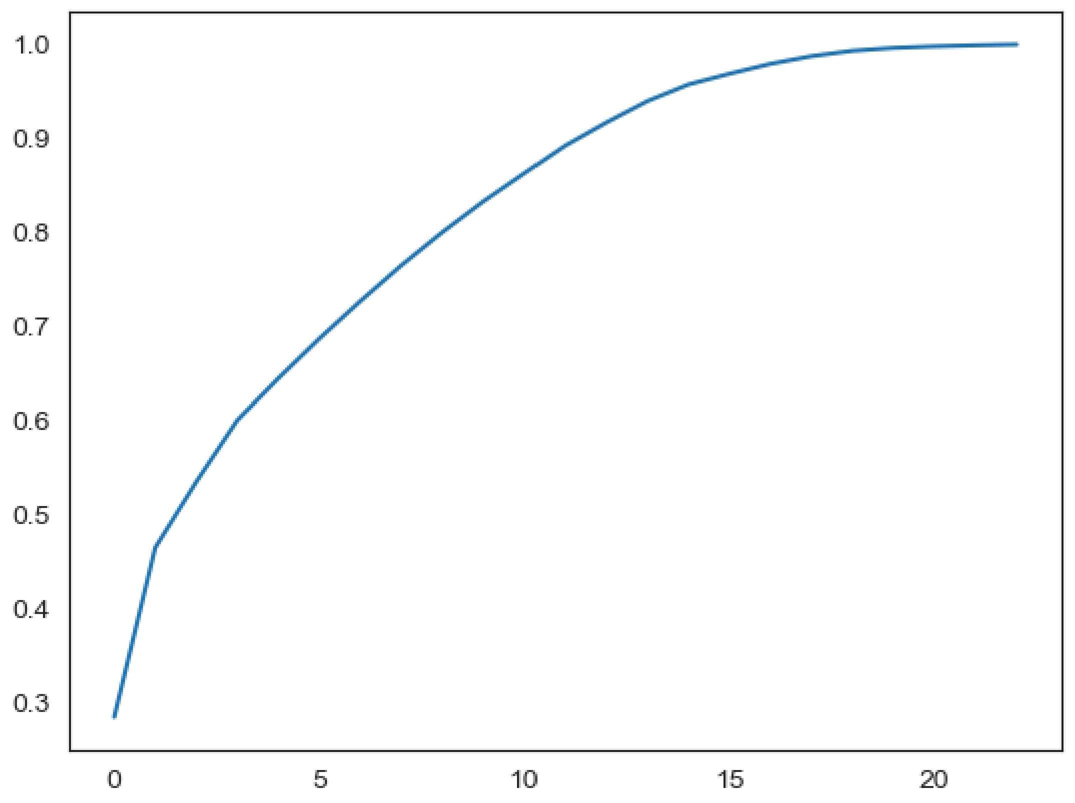
PCA is used to reduce the number of dimensions to principal components that retain most of the original information, by transforming correlated variables into a smaller set of features (principal components)

```
In [36]: pca = PCA() # initialise PCA
         pca.fit(X_train_scaled) # Fit with the scaled training data
         Xp_train = pca.transform(X_train_scaled) # Transform the scaled training data
```

```
Xp_test = pca.transform(X_test_scaled) # Transform the scaled testing data

nb_model = GaussianNB()
nb_model.fit(Xp_train,y_train)
predictions = nb_model.predict(Xp_test)

# F1 score for the Decision Tree
nb_pca_f1 = f1_score(y_test, predictions)

# Accuracy score for the Decision Tree
nb_pca_accuracy = accuracy_score(y_test, predictions)

# ROC AUC score for the Decision Tree
nb_pca_auc = roc_auc_score(y_test, predictions)
```

In [37]:
```
plt.plot(np.cumsum(pca.explained_variance_ratio_))
```

Out[37]:
```
[<matplotlib.lines.Line2D at 0x321318260>]
```



```
In [38]: pca.explained_variance_
```

```
Out[38]: array([6.55752052, 4.14198358, 1.60869028, 1.50411086, 1.02106891,
                0.96944467, 0.91509601, 0.87498664, 0.81188373, 0.75416253,
                0.69016914, 0.68047889, 0.56507991, 0.52289563, 0.40562259,
                0.26186693, 0.2419758 , 0.18731919, 0.13205484, 0.07028363,
                0.03853399, 0.02595118, 0.02009839])
```

According to Kaiser rule when selecting the number of principal components to keep, we have to keep components with eigenvalues greater than 1. Hence, we will keep the number of components as 5.

```
In [39]: pca = PCA(n_components = 5)
         pca.fit(X_train_scaled)
```

```
Xp_train = pca.transform(X_train_scaled)
Xp_test = pca.transform(X_test_scaled)
```

```
In [40]: Xp_train.shape, Xp_test.shape
```

```
Out[40]: ((18000, 5), (12000, 5))
```

```
In [41]: Xp_train.shape, y_train.shape
```

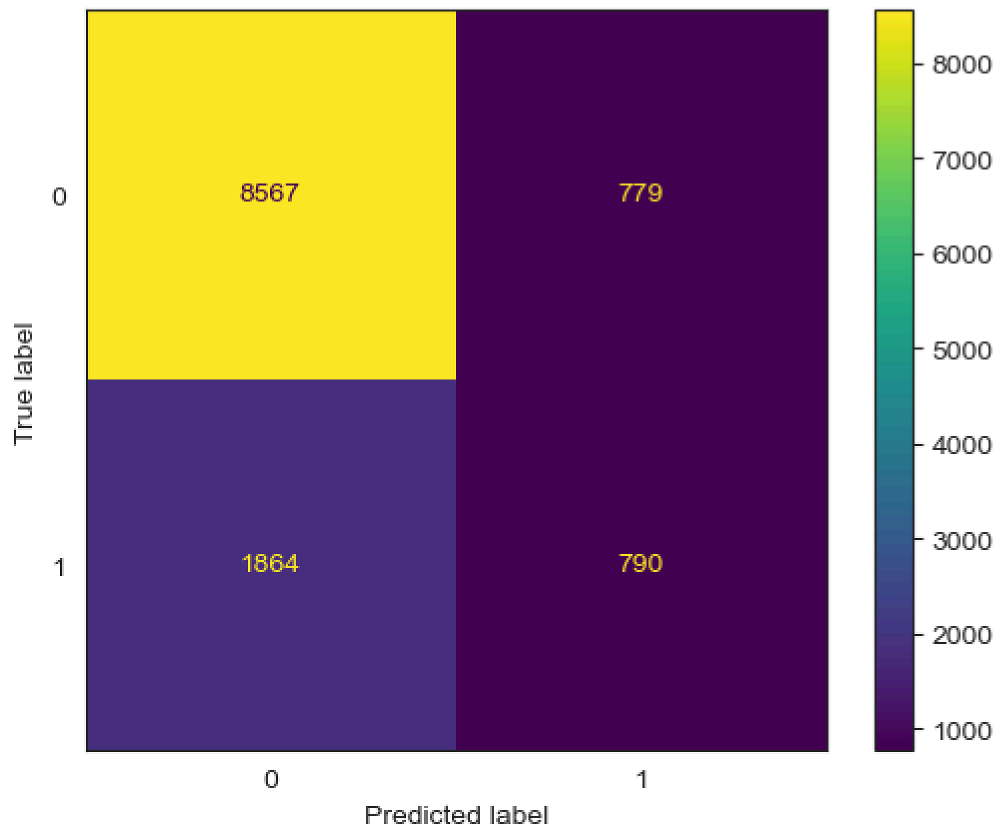
```
Out[41]: ((18000, 5), (18000, 1))
```

```
In [42]: k_fold = KFold(n_splits = 5, shuffle = True, random_state = 42)
         knc = KNeighborsClassifier(n_neighbors = 5, metric = 'euclidean') # using the best parameters found earlier
         knc.fit(Xp_train, y_train)
         predictions = knc.predict(Xp_test)

         # F1 score for the KNN classifier with 5 PCA components
         knc_pca_f1 = f1_score(y_test, predictions)

         # Accuracy score for the KNN classifier with 5 PCA components
         knc_pca_accuracy = accuracy_score(y_test, predictions)

         # ROC AUC score for the KNN classifier with 5 PCA components
         knc_pca_auc = roc_auc_score(y_test, predictions)

         print(f"F1 score for 5 PCA components: {knc_pca_f1:.5f}")
         print(f"Accuracy score for 5 PCA components: {knc_pca_accuracy:.5f}")
         print(f"ROC AUC score for 5 PCA components: {knc_pca_auc:.5f}")
```

```
F1 score for 5 PCA components: 0.37414
Accuracy score for 5 PCA components: 0.77975
ROC AUC score for 5 PCA components: 0.60716
```

```
In [43]: matrix = confusion_matrix(y_test, predictions, labels=knc.classes_)
         cm_display = ConfusionMatrixDisplay(confusion_matrix=matrix, display_labels=knc.classes_)
         cm_display.plot()
```

```
Out[43]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x3142d3620>
```



5.2 TSNE

In [44]:
```
tsne = TSNE(n_components=2, random_state=42)
X_train_tsne = tsne.fit_transform(X_train_scaled)
X_test_tsne = tsne.fit_transform(X_test_scaled)
tsne.kl_divergence_
```

Out[44]:
```
1.343996286392212
```

In [94]:
```
fig = px.scatter(x=X_train_tsne[:, 0], y=X_train_tsne[:, 1], color = y_train.values.ravel(),width=600, height=500)
fig.show()
```

In [46]:
```
k_fold = KFold(n_splits = 5, shuffle = True, random_state = 42)
knc = KNeighborsClassifier(n_neighbors = 5, metric = 'euclidean')
knc.fit(X_train_tsne, y_train)
predictions = knc.predict(X_test_tsne)

# F1 score for the KNN classifier with TSNE
knc_tsne_f1 = f1_score(y_test, predictions)

# Accuracy score for the KNN classifier with TSNE
knc_tsne_accuracy = accuracy_score(y_test, predictions)
```

```
# ROC AUC score for the KNN classifier with TSNE
knc_tsne_auc = roc_auc_score(y_test, predictions)

print(f"F1 score for 2 t-SNE components: {knc_tsne_f1:.5f}")
print(f"Accuracy score for 2 t-SNE components: {knc_tsne_accuracy:.5f}")
print(f"ROC AUC score for2 t-SNE components: {knc_tsne_auc:.5f}")
```

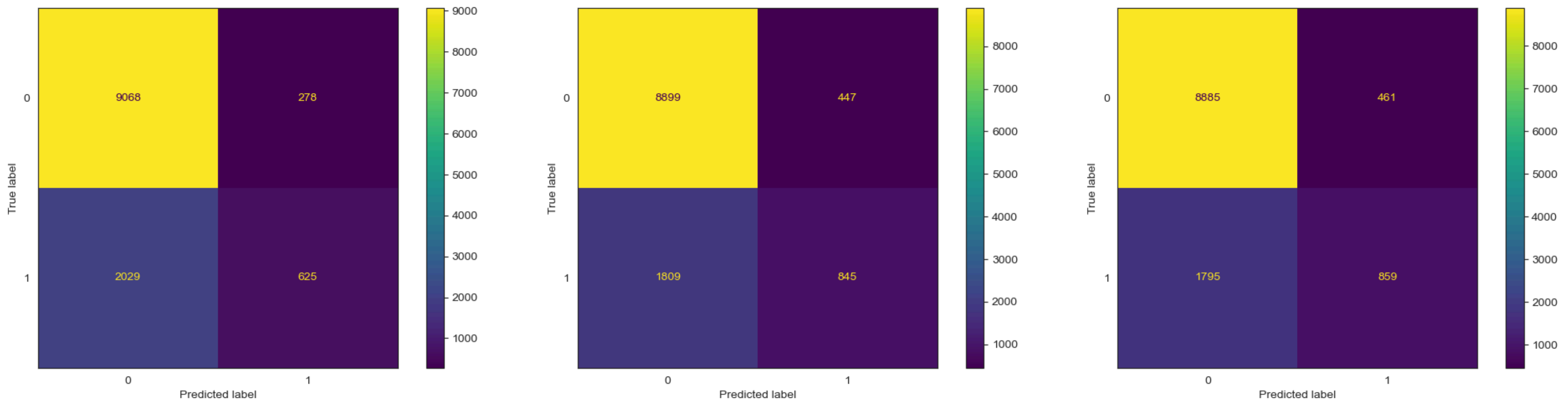
```
F1 score for 2 TSNE components: 0.34290
Accuracy score for 2 TSNE components: 0.75375
ROC AUC score for2 TSNE components: 0.58790
```

5.3 Polynomial Features

In [98]:
```
scores = {}

fig, axs = plt.subplots(1,3, layout = "tight", figsize = (20,5))

for degree in range(1,4):
    poly = PolynomialFeatures(degree = degree)
    Xp_train = poly.fit_transform(X_train_scaled)
    Xp_test = poly.fit_transform(X_test_scaled)
    num_features = Xp_train.shape[1]
    # knc = KNeighborsClassifier(n_neighbors = 5, metric = 'euclidean')
    model = LogisticRegression()
    # knc.fit(Xp_train, y_train)
    model.fit(Xp_train, y_train)
    predictions = model.predict(Xp_test)

    features = Xp_train.shape[1]

    # F1 score for the KNN classifier with polynomial features
    pf_f1 = f1_score(y_test, predictions)

    # Accuracy score for the KNN classifier with polynomial features
    pf_accuracy = accuracy_score(y_test, predictions)

    # ROC AUC score for the KNN classifier with polynomial features
    pf_auc = roc_auc_score(y_test, predictions)
    pf_matrix = confusion_matrix(y_test, predictions, labels=model.classes_)
    cm_display = ConfusionMatrixDisplay(confusion_matrix=pf_matrix, display_labels=model.classes_)
    cm_display.plot(ax = axs[degree - 1])

    scores[degree] = {'f1': pf_f1,
                 'accuracy':pf_accuracy,
                 'roc_auc':pf_auc,
                 'matrix': pf_matrix,
                 'features_number': features}

pd.options.display.float_format = '{:,.3f}'.format
polynomial_scores = pd.DataFrame.from_dict(scores)
polynomial_scores
```

Out[98]:

	1	2	3
f1	0.351	0.428	0.432
accuracy	0.808	0.812	0.812
roc_auc	0.603	0.635	0.637
matrix	[[9068, 278], [2029, 625]]	[[8899, 447], [1809, 845]]	[[8885, 461], [1795, 859]]
features_number	24	300	2600



5.4 RFE

In [108…

```python
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression

rfe_scores = {} # initialise empty dict

# new
logres = LogisticRegression()

for n in range(1,24):
    # rfe = RFE(estimator=LogisticRegression(), n_features_to_select=n)
    rfe = RFE(estimator = logres, n_features_to_select = n)
    rfe.fit(X_train_scaled, y_train)

    # Get the selected features
    selected_features = X_train_scaled.columns[rfe.support_]

    # create new training set
    # rfe.predict(X_test_scaled)
    # rfe.score(X_test_scaled, y_test
    X_train_rfe = X_train_scaled[selected_features]
    X_test_rfe = X_test_scaled[selected_features]

    logres.fit(X_train_rfe, y_train)
    predictions = logres.predict(X_test_rfe)

    # F1 score for the KNN classifier with polynomial features
    rfe_f1 = f1_score(y_test, predictions)

    # Accuracy score for the KNN classifier with polynomial features
    rfe_accuracy = accuracy_score(y_test, predictions)

    # ROC AUC score for the KNN classifier with polynomial features
    rfe_auc = roc_auc_score(y_test, predictions)

    rfe_scores[n] = {'f1': rfe_f1,
                     'accuracy':rfe_accuracy,
                      'roc_auc':rfe_auc,
                      'features_number': 24-n}


print(f"F1 score for rfe components: {rfe_f1:.5f}")
print(f"Accuracy score for rfe components: {rfe_accuracy:.5f}")
print(f"ROC AUC score for rfe components: {rfe_auc:.5f}")
```

```
F1 score for rfe components: 0.35188
Accuracy score for rfe components: 0.80783
ROC AUC score for rfe components: 0.60306
```

In [109…

```
pd.options.display.float_format = '{:,.3f}'.format
rfe_scores_df = pd.DataFrame.from_dict(rfe_scores)
rfe_scores_df.T
```

Out[109…

	f1	accuracy	roc_auc	features_number
1	0.435	0.817	0.639	23.000
2	0.414	0.814	0.629	22.000
3	0.343	0.808	0.600	21.000
4	0.336	0.806	0.597	20.000
5	0.336	0.806	0.597	19.000
6	0.339	0.807	0.598	18.000
7	0.348	0.808	0.602	17.000
8	0.360	0.810	0.606	16.000
9	0.354	0.809	0.604	15.000
10	0.355	0.808	0.605	14.000
11	0.353	0.807	0.603	13.000
12	0.357	0.808	0.605	12.000
13	0.350	0.807	0.602	11.000
14	0.353	0.808	0.603	10.000
15	0.353	0.808	0.603	9.000
16	0.353	0.808	0.603	8.000
17	0.351	0.808	0.603	7.000
18	0.352	0.808	0.603	6.000
19	0.353	0.808	0.603	5.000
20	0.352	0.808	0.603	4.000
21	0.352	0.808	0.603	3.000
22	0.352	0.808	0.603	2.000
23	0.352	0.808	0.603	1.000

6. Results

This section analyses the performance of the algorithms and hyper-parameters tuning.

6.1 Results for Algorithms

In [99]:
```
results_algos = {
    "Custom baseline" :{
        "Accuracy": baseline_accuracy.Y,
        "F1": 0.5,
        "AUC": 0.5,
    },
    "Custom K-Nearest Neighbour": {
        "Accuracy": custom_knn_accuracy,
        "F1": custom_knn_f1,
        "AUC": custom_knn_auc
    },
    "Custom Naive Bayes": {
        "Accuracy": custom_naivebayes_accuracy,
        "F1": custom_naivebayes_f1,
        "AUC": custom_naivebayes_auc
    },
    "Custom Logistic Regression": {
        "Accuracy":custom_logres_accuracy,
        "F1": custom_logres_f1,
        "AUC": custom_logres_auc
    },
    "Decision Tree (Scikit-learn)": {
        "Accuracy": decision_tree_accuracy,
        "F1": decision_tree_f1,
        "AUC": decision_tree_auc
    }
}
```

In [102…
```
pd.DataFrame.from_dict(results_algos).T
```

Out[102…

	Accuracy	F1	AUC
Custom baseline	0.779	0.500	0.500
Custom K-Nearest Neighbour	0.779	0.876	0.500
Custom Naive Bayes	0.779	0.876	0.500
Custom Logistic Regression	0.808	0.769	0.603
Decision Tree (Scikit-learn)	0.730	0.377	0.601

Based on the table above, we can see that the Custom Logistic Regression model performs the best overall, with the highest AUC and accuracy. While Custom KNN and Custom Naive Bayes have similar performance across the metrics, the AUC scores do not improve over baseline. The decision tree performs worst overall, in particular with its F1 score. This implies that the decision tree algorithm struggles with precision and recall.

6.2 Results for Cross-validation and Hyper-parameter tuning

In [105…
```
results_crossval = { "Grid Search with KNN": {
        "F1": grid_knn_f1
    },
    "Kfold with Naive Bayes": {
        "F1": custom_nb_kf_scores["test_f1_weighted"].mean()
    },
    "Cross Val with Log Res":logres_cross_val.mean()}
```

In [106…
```
results_crossval
pd.DataFrame.from_dict(results_crossval).T
```

Out[106…

	F1
Grid Search with KNN	0.721
Kfold with Naive Bayes	0.682
Cross Val with Log Res	0.682

Based onn the table above, we see that Grid Search with KNN performed the best, based on the F1 score. As Grid Search exhaustively tries every combination of a provided hyper-parameter to find the best model, it is not surprising that it produces the optimal model.

In [ ]:
```
"Grid Search with KNN": {
        "F1": grid_knn_f1
    },
```

6.3 Results for Feature Engineering

This section covers the results of feature engineering.

image.png

As seen above, feature engineering improved both AOC and Accuracy Score. Using 3 polynomial features, especially, yields an accuracy score of 0.812 and an AOC score of 0.637. However, the F1 score has decreased with feature engineering to worse than baseline, with RFE achieving the highest score of 0.435 from the batch. The reduced F1 score suggests that the models may be overfitting. As the number of features increase, the data also becomes sparse in the feature space, making it difficult for models to learn meaningful patterns between features.

7. Evaluation

7.1 Achieving the aim of the project

The primary objective of this project was two-fold – 1) to identify the most effective machine learning model for classifying whether a customer is likely to default on their credit card payment and 2) to develop a deeper understanding of machine learning algorithms by implementing them from scratch. Throughout the project, this objective was systematically addressed. By breaking down and manually implementing three classification algorithms — l=k-Nearest Neighbours (kNN), Naïve Bayes, and Logistic Regression — the underlying principles and mechanics of these models were thoroughly explored. This hands-on approach not only enhanced conceptual understanding but also facilitated a comprehensive analysis of their strengths and limitations in solving the given classification task.

Among the implemented models, the Custom Logistic Regression emerged as the best-performing model. It achieved an accuracy of 0.808, an F1 score of 0.769, and an AUC (Area Under the Curve) score of 0.603. This demonstrates its ability to strike a balance between precision, recall, and overall classification effectiveness. To further enhance model performance, advanced techniques such as cross-validation, hyperparameter tuning, and feature engineering were employed in Section 3 of this project. These steps helped refine the model's predictions and ensured that the results were robust and generalisable to unseen data. Overall, the project offered a thorough understanding of the machine learning pipeline — encompassing preprocessing, algorithm implementation, model evaluation, and optimisation techniques. It not only identified the most effective model for the problem at hand but also laid the groundwork for further exploration into real-world classification tasks.

7.2 Strengths

This project demonstrates three key strengths:

- Implementing three algorithms from scratch significantly deepened understanding of their inner workings. By researching on the algorithms and coding each step, I was able to grasp the nuances of each algorithm, such as how Naïve Bayes calculates probabilities using features likelihood and how Logistic Regression calculates for the max Log likelihood. This hands-on implementation allowed for a translation of theoretical concepts to practical implementation. Tackling three algorithms from scratch was a challenging undertaking, but it proved highly rewarding as it added depth and robustness to the project.
- Preprocessing the data was another crucial aspect of the project and was thoroughly explored in this project. Techniques such as feature scaling ensured that all features contributed equally to the model's performance. Removing outliers using the interquartile range (IQR) helped reduce noise and improve data quality, which in turn enhanced the stability and reliability of the models.
- Employing feature engineering also improved model performance. Creating new features and selecting the best ones reduced computational complexity and improved the accuracy of the models.

7.3 Weaknesses

Upon reflection, it is evident that the project exhibits three critical weaknesses. These shortcomings became apparent in the evaluation of the models in Section 5.

- Firstly, the lack of a singular performance metric has weakened the analysis of the models. The project uses multiple metrics, including accuracy, F1 score and AUC score, to assess model performance. However, not settling on one definitive metric makes it difficult to draw clear insights from the results. In the future, a streamlined approach with a single, well-chosen metric might provide a clearer and more concise understanding of model effectiveness.

- Secondly, the interpretability of the models is poor. It remains unclear why the models produce the established results – especially with the trade-off between the different metrics. One plausible explanation may be multi-collinearity of the features, where several independent variables in the model are correlated. In the future, the degree of multicollinearity can be explored through looking at the Variance Inflation Factor (VIF) using the VIF algorithm.

- Lastly, a more thorough examination of issues such as overfitting and underfitting should be examined. These aspects are fundamental to understanding if the models are generalising well to unseen data, or if they are over-tailored to the training data. This project only provided a shallow analysis of these factors.

In the future, these weaknesses should be addressed to produce more reliable and interpretable results.

Conclusion

In conclusion, this project achieved its primary objective of identifying the most effective machine learning algorithm for predicting whether a customer is likely to default on a credit card payment.

The development process began with thorough pre-processing of the dataset to ensure data quality and consistency. Outliers were identified and removed, and scaling was applied to normalize the data. Following this, three classification algorithms—k-Nearest Neighbours, Naïve Bayes, and Logistic Regression—were implemented from scratch. Extensive research was conducted to understand the underlying mechanics of each algorithm, forming the theoretical foundation. These algorithms were then coded in Python using the

NumPy library. Additionally, feature engineering techniques were applied to enhance the models' predictive performance, demonstrating a commitment to iterative refinement and optimization.

Among the four classifier algorithms, the custom logistic regression algorithm performs the best overall, delivering the highest ROC and accuracy score. For future expansion of this project, further analysis should focus on exploring multicollinearity among features and assessing the extent of overfitting and underfitting to ensure generalisable results.

References

[1] I. Yeh. "Default of Credit Card Clients," UCI Machine Learning Repository, 2009. [Online]. Available: https://doi.org/10.24432/C55S3H. [Accessed 10 December 2024]

[2] C. Boutin, “There’s More to AI Bias Than Biased Data, NIST Report Highlights,” 16 March 2022. [Online]. Available: https://www.nist.gov/news-events/news/2022/03/theres-more-ai-bias-biased-data-nist-report-highlights. [Accessed 10 December 2024]

[3] A. Suresh, “Outliers (Machine Learning Algorithms),” 2020. [Online]. Available: https://www.kaggle.com/discussions/general/241610. [Accessed 11 December 2024]

[4] The Pennsylvania State Univeristy, “3.2 - Identifying Outliers: IQR Method,” 2024. [Online]. Available: https://online.stat.psu.edu/stat200/lesson/3/3.2. [Accessed 11 December 2024]

[5] R. Sharma, “Detect and Remove the Outliers using Python,” 30 August 2024. [Online]. Available: https://www.geeksforgeeks.org/detect-and-remove-the-outliers-using-python/. [Accessed 11 December 2024]