CHAPTER 5
IMPLEMENTATION AND RESULTS

5.1. Implementation

Here we will import the numpy and pandas libraries using the following command, lines 1 and 2 are used to import libraries used in the classification process, NumPy to transform arrays, and pandas to handle data structures.

```
1. import numpy as np
2. import pandas as pd
```

Then the we will read the dataset file in the form of .csv, here using the Pandas function, namely read_csv which can be seen in the third line and the 4th line is used to display data.

```
3. df = pd.read_csv('train.csv')
4. df
```

The 5th row is used to find out the data structure of the dataset used, such as the data type, the number of columns that are not empty, the total number of rows, the number of columns, and so on.

```
5. df.info()
```

In line 6, this section is used to drop the unnecessary column like LOAN_ID. LOAN_ID dropped because it is not related to the classification process. And the 7th row is used to display the data. Axis is set as 1 to indicate that the data to be dropped is in the form of columns and inplace parameter is set as true so that the drop process is carried out on the original data.

```
6. df.drop('Loan_ID', axis=1, inplace=True)
7. df
```

In line 8 it is used to view statistics from data in the form of numeric data in the dataset, and in line 9 to view statistics from data in the form of objects in the dataset.

```
8. df.describe()
9. df.describe(include=['O'])
```

The 10th row contains "value_counts()" which functions to see the value in the Gender column, and in rows 13 to 16 there is a "loc" which is used to retrieve data according to the desired conditions and "shape[0]" to get the number of rows, like the number of Gender columns that are "Male” and Loan_Status is “Y”, the number of Gender columns that are “Male” and Loan_Status “N”, the number of Gender columns that are “Female” and

Loan_Status “Y”, the number of Gender columns that are “Female” and Loan_Status “N". We will do this counting function for every variable to see their effect on loan yield.

```
print(df['Gender'].value_counts())
print('\n')
print('Male and Loan Status accepted:', df.loc[(df['Gender'] == 'Male') & (df['Loan_Status'] == 'Y')].shape[0])
print('Male and Loan Status not accepted:', df.loc[(df['Gender'] == 'Male') & (df['Loan_Status'] == 'N')].shape[0])
print('Female and Loan Status accepted:', df.loc[(df['Gender'] == 'Female') & (df['Loan_Status'] == 'Y')].shape[0])
print('Female and Loan Status not accepted:', df.loc[(df['Gender'] == 'Female') & (df['Loan_Status'] == 'N')].shape[0])
```

Here the function is to take null data using "isna" and add it up with "sum". To deal with data that contains null rows where this data is categorical it will fill in null data with the mode of the column, and where the data is numerical will fill in null data with the mean value of a column.

```
df.isna().sum()
df['Gender'].fillna(df['Gender'].mode()[0], inplace=True)
df['Married'].fillna(df['Married'].mode()[0], inplace=True)
df['Dependents'].fillna(df['Dependents'].mode()[0], inplace=True)
df['Loan_Amount_Term'].fillna(df['Loan_Amount_Term'].mode()[0], inplace=True)
df['Credit_History'].fillna(df['Credit_History'].mode()[0], inplace=True)
df['Self_Employed'].fillna(df['Self_Employed'].mode()[0], inplace=True)
df['LoanAmount'].fillna(df['LoanAmount'].mean(), inplace=True)
```

Here we will encode the Loan_Status, dependents, education, property area columns with the replace function and the dictionary.

```
dic = {"N": 0, "Y": 1}
df = df.replace({"Loan_Status": dic})
dic = {"0": 0, "1": 1, "2": 2, "3+": 3}
df = df.replace({"Dependents": dic})
dic = {"Not Graduate": 0, "Graduate": 1}
df = df.replace({"Education": dic})
dic = {"Rural": 0, "Semiurban": 1, "Urban": 2}
df = df.replace({"Property_Area": dic})
```

Here we will looking the feature that influence loan status result.

```
corr_mat = df.corr()
corr_mat['Loan_Status'] = abs(corr_mat['Loan_Status'])
```

```
sorted_corr_mat = corr_mat.sort_values(by=['Loan_Status'], ascending=False)
sorted_corr_mat['Loan_Status']
```

Here we will use one hot label process on categorical columns that are not ordinal, with the get_dummies function. One hot label is the process of changing a column into several columns according to the value of that column.

```
catg = ["Gender", "Married", "Self_Employed"]
df = pd.get_dummies(df, columns=catg)
df['Loan_Status'].value_counts()
```

After that we will separate the data for training and testing with a total of 70% for training and 30% for testing where the testing data will be divided by 3 to see the result consistency. And separate data for training and testing with a total of 60% for training and 40% for testing where the testing data will be divided by 3 to see the result consistency, after that data will be shuffled so that the data we use is not sequential.

```
df_yes_train1 = df_yes.loc[df_yes.index[range(0, 253)]]
df_yes_test1 = df_yes.loc[df_yes.index[range(253, 422)]]
df_yes_test1_1 = df_yes.loc[df_yes.index[range(253, 309)]]
df_yes_test1_2 = df_yes.loc[df_yes.index[range(309, 365)]]
df_yes_test1_3 = df_yes.loc[df_yes.index[range(365, 422)]]
df_no_train1 = df_no.loc[df_no.index[range(0, 115)]]
df_no_test1 = df_no.loc[df_no.index[range(115, 192)]]
df_no_test1_1 = df_no.loc[df_no.index[range(115, 141)]]
df_no_test1_2 = df_no.loc[df_no.index[range(141, 167)]]
df_no_test1_3 = df_no.loc[df_no.index[range(167, 192)]]
df_yes_train2 = df_yes.loc[df_yes.index[range(0, 295)]]
df_yes_test2 = df_yes.loc[df_yes.index[range(295, 422)]]
df_yes_test2_1 = df_yes.loc[df_yes.index[range(295, 337)]]
df_yes_test2_2 = df_yes.loc[df_yes.index[range(337, 379)]]
df_yes_test2_3 = df_yes.loc[df_yes.index[range(379, 422)]]
df_no_train2 = df_no.loc[df_no.index[range(0, 134)]]
df_no_test2 = df_no.loc[df_no.index[range(134, 192)]]
df_no_test2_1 = df_no.loc[df_no.index[range(134, 153)]]
df_no_test2_2 = df_no.loc[df_no.index[range(153, 172)]]
df_no_test2_3 = df_no.loc[df_no.index[range(172, 192)]]
df_train1 = pd.concat([df_yes_train1, df_no_train1], ignore_index=True)
df_train2 = pd.concat([df_yes_train2, df_no_train2], ignore_index=True)
df_test1 = pd.concat([df_yes_test1, df_no_test1], ignore_index=True)
df_test1_1=pd.concat([df_yes_test1_1,df_no_test1_1], ignore_index=True)
```

```
df_test1_2=pd.concat([df_yes_test1_2,df_no_test1_2],
ignore_index=True)
df_test1_3=pd.concat([df_yes_test1_3,df_no_test1_3],
ignore_index=True)
df_test2 = pd.concat([df_yes_test2, df_no_test2], ignore_index=True)
df_test2_1=pd.concat([df_yes_test2_1,df_no_test2_1],
ignore_index=True)
df_test2_2=pd.concat([df_yes_test2_2,df_no_test2_2],
ignore_index=True)
df_test2_3=pd.concat([df_yes_test2_3,df_no_test2_3],
ignore_index=True)
df_train1 = df_train1.sample(frac=1).reset_index(drop=True)
df_train2 = df_train2.sample(frac=1).reset_index(drop=True)
df_test1 = df_test1.sample(frac=1).reset_index(drop=True)
df_test1_1 = df_test1_1.sample(frac=1).reset_index(drop=True)
df_test1_2 = df_test1_2.sample(frac=1).reset_index(drop=True)
df_test1_3 = df_test1_3.sample(frac=1).reset_index(drop=True)
df_test2 = df_test2.sample(frac=1).reset_index(drop=True)
df_test2_1 = df_test2_1.sample(frac=1).reset_index(drop=True)
df_test2_2 = df_test2_2.sample(frac=1).reset_index(drop=True)
df_test2_3 = df_test2_3.sample(frac=1).reset_index(drop=True)
var_input=['Dependents',                                      'Education',
'CoapplicantIncome','LoanAmount',
'Loan_Amount_Term','Credit_History','Property_Area','Gender_Female',
'Gender_Male','Married_No','Married_Yes','Self_Employed_No','Self_Emp
loyed_Yes']
X_train1 = df_train1[var_input]
y_train1 = df_train1['Loan_Status']
X_train2 = df_train2[var_input]
y_train2 = df_train2['Loan_Status']
X_test1 = df_test1[var_input]
y_test1 = df_test1['Loan_Status']
X_test1_1 = df_test1_1[var_input]
y_test1_1 = df_test1_1['Loan_Status']
X_test1_2 = df_test1_2[var_input]
y_test1_2 = df_test1_2['Loan_Status']
X_test1_3 = df_test1_3[var_input]
y_test1_3 = df_test1_3['Loan_Status']
X_test2 = df_test2[var_input]
y_test2 = df_test2['Loan_Status']
X_test2_1 = df_test2_1[var_input]
y_test2_1 = df_test2_1['Loan_Status']
X_test2_2 = df_test2_2[var_input]
y_test2_2 = df_test2_2['Loan_Status']
X_test2_3 = df_test2_3[var_input]
```

```
99. y_test2_3 = df_test2_3['Loan_Status']
```

Here we will calculate the accuracy, precision, recall, and f1-score. To calculate the precision. For the first time true positive and false positive is set to 0, when the original label and prediction are the same where the original label is 1, then the true positive variable is added 1 and if the original label and prediction are not the same where the original label is 0, then the false positive variable is added by 1. Then the precision is calculated with the true positive formula divided by the sum of the true positives and false positives. To calculate recall. For the first time true positive and false negative are set to 0, then if the original label and prediction are the same where the original label is 1, then the true positive variable is added 1 and if the original label and prediction are not the same where the original label is 1, then the false-negative variable is added by 1. Then the recall is calculated using the true positive formula divided by the sum of the true positives and false negatives. To calculate recall. For the first time true positive, false positive, and false negative is first set to 0, if true positive is 1 then the true positive variable is added 1, if the false positive is 1 then the false positive variable is added 1, and if the false negative is 1 then the variable false-negative plus 1. Then the recall is calculated using the formula 2 multiplied by the product of recall and precision which is then divided by the sum of recall and precision.

```
100.def accuracy(true, pred):
101.  accuracy = np.sum(true == pred) / len(true)
102.  return accuracy
```

Lines 144 to 146 serve to calculate accuracy.

```
103.def precision(true, pred):
104.  tp = 0
105.  fp = 0
106.  for i in range(len(true)):
107.    if (true[i] == pred[i] and true[i] == 1):
108.      tp = tp + 1
109.    if (true[i] != pred[i] and true[i] == 0):
110.      fp = fp + 1
111.  precision = tp / (tp + fp)
112.  return precision
113.def recall(true, pred):
114.  tp = 0
115.  fn = 0
116.  for i in range(len(true)):
117.    if (true[i] == pred[i] and true[i] == 1):
118.      tp = tp + 1
119.    if (true[i] != pred[i] and true[i] == 1):
      fn = fn + 1
  recall = tp / (tp + fn)
  return recall
def f1(true, pred):
  tp = 0
  fp = 0
  fn = 0
  for i in range(len(true)):
    if (true[i] == pred[i] and true[i] == 1):
      tp = tp + 1
    if (true[i] != pred[i] and true[i] == 0):
      fp = fp + 1
    if (true[i] != pred[i] and true[i] == 1):
      fn = fn + 1
  precision = tp / (tp + fp)
  recall = tp / (tp + fn)
  f1 = 2 * (recall * precision) / (recall + precision)
  return f1
```

And here we will create a class for logistic regression model to initiate a logistic regression model. There is a learning rate, namely the training parameter to calculate the weight correction value during the training process, the greater the learning rate value, the faster the training process will run. Then there is num iteration, which is the parameter for the number of iterations, the fit intercept which functions to divide the data, and verbose to debug by looking at the number of losses. And intercept the value which if the fit intercept is false, then the line will be forced to pass (0,0) and if true then the line will be fit to that data. initialized to the value 1, and concatenates the array to the final X value in 1 separate column. Then use the sigmoid functions to perform sigmoid calculations, if the value for the calculation is greater than 0.5 it will be 1 and if it is smaller it will be 0. "Z" is the result of the calculation of the linear regression formula, and "exp" is exponential. For performing loss calculations, to check and minimize errors from the model we will use the loss function in the model. "yp" is the label of the prediction result, and "y" is the label on the original data.

```
class LogisticRegression:
    def __init__(self, learning_rate=0.01, num_iterations=50000, fit_intercept=True, verbose=False):
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
        self.fit_intercept = fit_intercept
        self.verbose = verbose
    def b_intercept(self, X):
        intercept = np.ones((X.shape[0], 1))
        return np.concatenate((intercept, X), axis=1)
    def sigmoid_function(self, z):
        return 1 / (1 + np.exp(-z))
    def loss(self, yp, y):
        return (-y * np.log(yp) - (1 - y) * np.log(1 - yp)).mean()
```

After that we will use the fit function for training the logistic regression model. if the fit intercept parameter is true and the self fit intercept is obtained from the current model initialization, then the b intercept function will be called. and initialize weight and bias to 0 which will be updated in each iteration. The training processes with the number of iterations that are given in the parameters will be done, where the calculations will be carried out to find linear regression and calculations will be carried out to find sigmoid. Then the calculations are carried out to find the gradient of the error created by the model. Then there is the process of changing the weight value which will be used in the next iteration. Then that calculations will be carried out to find the new linear regression value which is used to find the new sigmoid value. After that loss functions will be done to calculate loss, and then we will print the loss value if the verbose is true or 1 and each 10,000 iterations.

```
    def fit(self, X, y):
        if self.fit_intercept:
            X = self.b_intercept(X)
        self.W = np.zeros(X.shape[1])
        self.b = 0
        print("----- Proses training -----\n")
        for i in range(self.num_iterations):
            z = np.dot(X, self.W) + self.b
            yp = self.sigmoid_function(z)
            gradient_w = np.dot(X.T, (yp - y)) / y.size
            gradient_b = np.sum((yp - y)) / y.size
            self.W -= self.learning_rate * gradient_w
            self.b -= self.learning_rate * gradient_b
            z = np.dot(X, self.W) + self.b
            yp = self.sigmoid_function(z)
            loss = self.loss(yp, y)
            if(self.verbose == True and i % 10000 == 0):
                print("--- loss: {:.6f} ---".format(loss))
    def predict_prob(self, X):
        if self.fit_intercept:
            X = self.b_intercept(X)
        return self.sigmoid_function(np.dot(X, self.W) + self.b)
```

Here we will calculate the prediction probability based on the weight value obtained from all iterations, by first checking whether the fit intercept that has been initialized is true or not and if true then the b intercept function will be executed with the input value X, after that the value X which has gone through the b intercept process will be entered into the sigmoid formula to start the calculation. The prediction function is to predict the result between 0 or 1, if it is less than 0.5 it will be 0 and if it is more than 0.5 it will be 1. Here we will set the parameters for the model training, and the model will be tested on a predefined dataset.

```
    def predict(self, X):
        return self.predict_prob(X).round()
model = LogisticRegression(learning_rate=0.0000001, num_iterations=500000, verbose=True)
model.fit(X_train, y_train)
pred = model.predict(X_test1)
print('accuracy :', accuracy(y_test1, pred))
print('precision :', precision(y_test1, pred))
print('recall :', recall(y_test1, pred))
print('f1-score :', f1(y_test1, pred))
    pred = model.predict(X_test2)
    print('accuracy :', accuracy(y_test2, pred))
    print('precision :', precision(y_test2, pred))
    print('recall :', recall(y_test2, pred))
    print('f1-score :', f1(y_test2, pred))
    pred = model.predict(X_test3)
    print('accuracy :', accuracy(y_test3, pred))
    print('precision :', precision(y_test3, pred))
    print('recall :', recall(y_test3, pred))
    print('f1-score :', f1(y_test3, pred))
```

Here we will start the extreme gradient boosting algorithm using the sigmoid function where exp is used to calculate the exponent for each x value in the input array, gradient function by entering the predicted sigmoid calculation into the preds variable, then the results are reduced by labels (according to the formula), hessian function by entering the predictive sigmoid calculation into the preds variable, then the result is multiplied by the result of 1 minus the preds variable (according to the formula).

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def grad(preds, labels):
    preds = sigmoid(preds)
    return (preds - labels)
def hess(preds, labels):
    preds = sigmoid(preds)
    return (preds * (1 - preds))
```

And we will do an initialization function which used to initialize a node, here it will initialize several attributes, namely is_leaf which is used to check whether the node is a leaf or not, leaf_score to store the score if the node is a leaf or not if true then it will not be split, whether the node is a left child or right child, and places NA or missing value on the left or right.

```
class TreeNode(object):
 def __init__(self, is_leaf=False, leaf_score=None, split_feature=None, split_threshold=None, left_child=None, right_child=None, NA_direction='left'):
         self.is_leaf = is_leaf
         self.leaf_score = leaf_score
     self.split_feature = split_feature
     self.split_threshold = split_threshold
         self.left_child = left_child
     self.right_child = right_child
     self.NA_direction = NA_direction
```

After that we will create tree classes to initiate a tree where the root denotes a node that is the main branch, here we define the minimum sample required for split, subsampling column fraction, lambda, gamma, and minimum weight for split.

```
class Tree(object):
     def __init__(self, root=None, min_sample_split=None, col_sub_frac=None, lamda=None, gamma=None, num_thread=None, min_child_weight=None):
         self.root = root
         self.min_sample_split = min_sample_split
         self.col_sub_frac = col_sub_frac
         self.lamda = lamda
         self.gamma = gamma
         self.min_child_weight = min_child_weight
```

And we will calculate the predicted score for each leaf node that is formed using leaf score formula.

```
     def cal_leaf_score(self, Y):
         return - (Y['grad'].sum() / (Y['hess'].sum() + self.lamda))
```

Here the functions that used to perform split gain calculations. If the missing value is left it will create a “GL” or Gradient left variable which contains the number of gradients left plus the gradient value, “HL” or Hessian left which contains the number of Hessian left plus the hessian value, “GR” or Gradient right which contains the number of gradient right, “HR” or Hessian right which contains the number of Hessian rights. Then if other, then create a “GL” or Gradient left variable which contains the number of gradients left, “HL” or Hessian left which contains the number of Hessian left, “GR” or Gradient right which contains the number of gradient right plus the value of gradient, “HR” or Hessian right which contains the sum of the hessian rights plus the hessian value. Then in line 230, the gain is calculated using the split gain formula.

```
219.     def cal_split_gain(self, left_Y, right_Y, NA_grad, NA_hess,
   NA_direction='left'):
220.         if (NA_direction == 'left'):
221.            GL = left_Y['grad'].sum() + NA_grad
222.            HL = left_Y['hess'].sum() + NA_hess
223.            GR = right_Y['grad'].sum()
224.            HR = right_Y['hess'].sum()
225.         else:
226.            GL = left_Y['grad'].sum()
227.            HL = left_Y['hess'].sum()
228.            GR = right_Y['grad'].sum() + NA_grad
229.            HR = right_Y['hess'].sum() + NA_hess
230.         gain    =    0.5    *    (    (GL**2/(HL+self.lamda))    +
   (GR**2/(HR+self.lamda))  -  ((GL+GR)**2/(HL+HR+self.lamda))  )  -
   self.gamma
```

Here we will do functions to select the best feature to split, the best split value, and the best gain value, set “best_split_value” to none, “best_gain” to infinity, “best_NA_direction” to left, “selected_dt” the feature column along with its label, gradient , and hessian, then “mask” to find the missing value, “ NA_dt” contains the mask variable, “Non_NA_dt” contains data that is not in the “mask” (looking for data that is not a missing value), “NA_grad” contains the number of gradients from the missing value, “NA_hess” contains the hessian number of missing values, Then resets the index "Non_NA_dt", adds 1 column named "feature _index" whose contents are filled in according to the order of the value of a feature, "Non_NA_dt" contains the hessian number of data that is not a missing value. After that, the rank calculation is carried out.

Line 211 is looped for the amount of data -1, then the current rank is searched and the next rank is used using the rank formula. If the result of the current rank minus the next rank is greater than the epsilon value, then the looping will continue if it is smaller, then the data will be separated using the split gain formula, filling the variable "left_Y" with the initial value up to "j" and the variable "right_Y" with value "j" to the last. From these two variables, a split gain calculation will be carried out on each data entered in the "go_left" and "go_right" variables, if the "go_left" data is greater then it will become the left branch and set "this_gain" to " go_left" and if other then it will be the right branch and set “this_gain” to “go_right”. If “this_gain” is greater than “best_gain”, it will set “best_split_value” to split_value, “best_gain” to “this_gain”, “best_NA_direction” to “this_direction”.

```
    def weighted_quantile_sketch(self, dt, feature):
        best_split_value = None
        best_gain = -np.inf
        best_NA_direction = 'left'
        selected_dt = dt[[feature, 'label', 'grad', 'hess']]
        mask = selected_dt[feature].isnull()
        NA_dt = selected_dt[mask]
        Non_NA_dt = selected_dt[~mask]
        NA_grad = NA_dt['grad'].sum()
        NA_hess = NA_dt['hess'].sum(
        Non_NA_dt.reset_index(inplace = True)
        Non_NA_dt['feature_index'] = Non_NA_dt[feature].argsort()
        Non_NA_dt = Non_NA_dt.iloc[Non_NA_dt['feature_index']]
        hess_sum = Non_NA_dt['hess'].sum()
 Non_NA_dt['rank'] = Non_NA_dt.apply(lambda x : (1/hess_sum)*sum(Non_NA_dt[Non_NA_dt[feature] < x[feature]]['hess']), axis=1)
        for j in range(Non_NA_dt.shape[0]-1):
            rk_sk_j, rk_sk_j_1 = Non_NA_dt['rank'].iloc[j:j+2]
            if (abs(rk_sk_j-rk_sk_j_1) >= self.eps):
                continue
        split_value = (Non_NA_dt[feature].iloc[j+1] + Non_NA_dt[feature].iloc[j])/2
            left_Y = Non_NA_dt.iloc[:(j+1)]
            right_Y = Non_NA_dt.iloc[(j+1):]
        go_left = self.cal_split_gain(left_Y, right_Y, NA_grad, NA_hess, NA_direction = 'left')
            go_right = self.cal_split_gain(left_Y, right_Y, NA_grad, NA_hess, NA_direction = 'right')
            if (go_left > go_right):
                this_gain = go_left
                this_direction = 'left'
            else:
                this_gain = go_right
                this_direction = 'right'
            if (this_gain > best_gain):
                best_split_value = split_value
                best_gain = this_gain
                best_NA_direction = this_direction
    return feature, best_split_value, best_gain, best_NA_direction
```

Here we will select the best feature for split, best split value, best gain value. Here "best_gain" will be set to infinity, then "best_feature", "best_split_value", "result" is set to none, "features" is filled with a list of features or columns, "data" is filled with a combination of inputs and labels. Then the number of features is looped, the contents of which are printing the features, adding the WQS. Then it will look for the best split by sorting the results of the WQS calculation. Then it will print the calculation of the best split, and fill in "best_feature", "best_split_value", "best_gain", "best_NA_direction".

```
    def find_best_split_value_and_feature(self, X, Y):
        best_gain = -np.inf
        best_feature, best_split_value, results = None, None, None
        features = list(X.columns) # get a list of all features
        data = pd.concat([X, Y], axis = 1)
        results = []
        for j in range(len(features)):
            print("----- Fitur dan WQS -----\n", features[j])
            results.append(self.weighted_quantile_sketch(data, features[j]))
        best = sorted(results, key = lambda x: float(x[2]), reverse = True)[0]
        print("----- Best split calc -----\n", best)
        best_feature = best[0]
        best_split_value = best[1]
        best_gain = best[2]
        best_NA_direction = best[3]
        return best_feature, best_split_value, best_gain, best_NA_direction
```

And we will be done in a split direction. Here it will combine inputs and labels, define a list of x and y columns, and if the NA_Direction is left then data with a value more than the same as the split value will be placed in the right variable, then for data less than the split value, it will be placed in the left variable along with the value NA (if any). Otherwise, data with a value less than the split value will be placed in the left variable, while for data more than equal to the split value, it will be placed in the right variable along with the NA value (if any). This split will be returned in the form of data that has been separated.

```
    def split(self, X, Y, feature, split_value, NA_direction):
        data = pd.concat([X, Y], axis = 1)
         X_cols, Y_cols = list(X.columns), list(Y.columns)
        print("----- Splitting -----\n", feature, split_value)
        if(NA_direction == 'left'):
           mask = (data[feature] >= split_value)
           left = data[~mask] # left take all NA
           right = data[mask]
        else:
           mask = (data[feature] < split_value)
           left = data[mask]
           right = data[~mask] # right take all NA
        return left[X_cols], left[Y_cols], right[X_cols], right[Y_cols]
```

Below is the function to create a tree, if the input is less than "min_sample_split" or the tree depth is 0 or the number of hessian labels is less than the "min_child_weight", it will calculate the leaf score.

```
def build_tree(self, X, Y, depth):
      print("----- Build a tree -----\n")
        if (X.shape[0] < self.min_sample_split) or (depth == 0) or (Y['hess'].sum() < self.min_child_weight):
            print("----- Buat leaf (X.shape[0] < min_sample_split / depth == 0 / Y['hess'].sum() < min_child_weight) -----\n")
            print(X.shape[0], self.min_sample_split, Y['hess'].sum(), self.min_child_weight)
           l_score = self.cal_leaf_score(Y)
           return TreeNode(is_leaf=True, leaf_score=l_score)
```

For checking whether there is overfitting or not we will use the function below. By searching the “best_feature”, “best_split_value”, “best_gain”, “best_NA_direction” using the “find_best_split_value_and_feature(X_sub, Y)” function. If the “best_gain” value is less than equal to 0, then a leaf score will be calculated where the return process will be returned in the form of a leaf node with the result of the leaf score. Then we will be splitting defined data into “left_X”, “left_Y”, “right_X”, “right_Y”. After that the left branch, right branch tree will be created, and merges the right and left branch trees will be created too.

```
        X_sub = X.sample(frac=self.col_sub_frac, axis=1)
        best_feature, best_split_value, best_gain, best_NA_direction = self.find_best_split_value_and_feature(X_sub, Y)
  print("----- Best split value and feature -----\n", best_feature, best_split_value, best_gain, best_NA_direction)
```

```
        if (best_gain <= 0):
            print("----- Buat leaf (best gain <= 0) -----\n")
            l_score = self.cal_leaf_score(Y)
            return TreeNode(is_leaf=True, leaf_score=l_score)
        left_X, left_Y, right_X, right_Y = self.split(X_sub, Y, best_feature, best_split_value, best_NA_direction)
        print("----- Buat cabang tree kiri -----\n")
        left_child = self.build_tree(left_X, left_Y, depth - 1)
        print("----- Buat cabang tree kanan -----\n")
        right_child = self.build_tree(right_X, right_Y, depth - 1)
        print("----- Gabung cabang tree kanan dan kiri -----\n")
        sub_tree = TreeNode(is_leaf=False, leaf_score=None, split_feature=best_feature, split_threshold=best_split_value, left_child=left_child, right_child=right_child, NA_direction=best_NA_direction)
        return sub_tree
```

Here is the function for the process of creating a tree. The function below is used to search for nodes that match the tree that has been defined, if the tree_node is a leaf, it will be returned as "tree_node.leaf_score". If the data is not a missing value and the direction is left, then the left branch will be used. If the split value of a feature is less than the split limit value, predictions will be made using the left branch tree. Otherwise it will be predicted using the right branch tree.

```
    def fit(self, X, Y, max_depth=3, min_child_weight=1, col_sub_frac=1, min_sample_split=10, lamda=1, gamma=0.05, eps=0.001):
        self.min_child_weight = min_child_weight
        self.col_sub_frac = col_sub_frac
        self.min_sample_split = min_sample_split
        self.lamda = lamda
        self.gamma = gamma
        self.eps = eps
        self.root = self.build_tree(X, Y, max_depth)
    def predict_one(self, tree_node, X):
        if tree_node.is_leaf == True:
            return tree_node.leaf_score
        elif (type(X[tree_node.split_feature].item()) != int) and (type(X[tree_node.split_feature].item()) != float) and (tree_node.NA_direction == 'left'):
            return self.predict_one(tree_node.left_child, X)
        elif ((X[tree_node.split_feature] < tree_node.split_threshold).item()):
            return self.predict_one(tree_node.left_child, X)
        else:
            return self.predict_one(tree_node.right_child, X)
```

Here are functions for making predictions. As long as n is the number of data inputs, predictions will be made using the tree that has been made.

```
    def predict(self, X):
        preds = []
        for n in range(X.shape[0]):
            preds.append(self.predict_one(self.root, X.iloc[[n]]))
        return np.array(preds)
```

The functions below are for training the XGBoost model. Here we will initialize variables and initialize prediction labels ("best metric value" is filled with infinity, "best round" is filled with none, creates a variable "metric_value_list", resets index of input and label), loop for training where in this process a tree will be created, as long as parameter “i” is in the parameter range "max_round". And predictions will be made using the existing tree to find out the accuracy value or f1-score in the current model, calculate the gradient and hessian used. to create a tree in the next iteration.

```
def xgboost_train(X, Y, eta, max_round, max_depth, row_sub_frac, col_sub_frac, min_child_weight, min_sample_split, lamda, gamma, eps, metric):
    trees = []
    initialize_pred = 1
    best_metric_value, best_round = -np.inf, None
    metric_value_list = []
    X.reset_index(drop=True, inplace=True)
    Y = Y.to_frame(name='label')
    Y.reset_index(drop=True, inplace=True)
    Y['y_pred'] = initialize_pred
    Y['grad'] = grad(Y['y_pred'], Y['label'])
    Y['hess'] = hess(Y['y_pred'], Y['label'])
    print("----- Initialization -----\n", Y['y_pred'], Y['grad'], Y['hess'])
    print("----- Training loop -----\n")
    for i in range(max_round):
        print("Training step: ", i, "--------------\n")
        data = pd.concat([X, Y], axis=1)
        print("----- Data -----\n", data)
        data = data.sample(frac=row_sub_frac, axis=0)
        print("----- Data sub-sampling -----\n", data)
        Y_Selected = data[['label', 'y_pred', 'grad', 'hess']]
        print("----- Y selected -----\n", Y_Selected)
        X_Selected = data[list(X.columns)]
        print("----- X selected -----\n", X_Selected)
        print("----- Tree inizialitation -----\n")
        tree = Tree()
        tree.fit(X_Selected, Y_Selected, max_depth=max_depth, min_child_weight=min_child_weight, col_sub_frac=col_sub_frac, min_sample_split=min_sample_split, lamda=lamda, gamma=gamma, eps=eps)
        preds = tree.predict(X)
        print("----- Prediction -----\n", preds)
        Y['y_pred'] = Y['y_pred'] + eta * preds
        Y['grad'] = grad(Y['y_pred'], Y['label'])
        Y['hess'] = hess(Y['y_pred'], Y['label'])
        print("----- In Iter -----\n", Y['y_pred'], Y['grad'], Y['hess'])
        trees.append(tree)
        print("----- Tress -----\n", trees)
        print("----- Test step: ", i, "-----\n")
        test_perf = []
        avg = Y['y_pred'].mean()
        for j in Y['y_pred']:
            if (j > avg):
                test_perf.append(1)
            else:
                test_perf.append(0)
        if (metric == 'f1'):
            m = f1(Y['label'], test_perf)
            print("F1-Score: ", m)
        if (metric == 'accuracy'):
            m = accuracy(Y['label'], test_perf)
            print("Accuracy: ", m, "\n")
        metric_value_list.append(m)
        if (m > best_metric_value):
            best_metric_value = m
            best_round = i
    best_trees = trees[:(i+1)]
    return best_trees, eta
```

Then we will make predictions using the previously trained XGBoost model. Here, predictions will be made using the existing tree in the XGBoost model, by multiplying the predictions from the test data with the learning rate and then adding up the old predictions. And by creating the variables for the final prediction and the average of the predictions. From all predictions that have been obtained if the value is above the average then it becomes 1, if below it becomes 0.

```
def xgboost_predict(trees, X_test, eta):
    preds = np.ones(X_test.shape[0])
    print("----- Initialization predict -----\n", preds)
    print("----- Tree loop -----\n")
    for tree in trees:
        print("----- Before predict -----\n", preds)
        preds = preds + tree.predict(X_test) * eta
        print("----- After predict -----\n", preds)
    adj_preds = [] # buat prediksi akhir
    avg = preds.mean()
    print("----- Predict score average -----\n", avg)
    for i in preds:
        if (i > avg):
            adj_preds.append(1)
        else:
            adj_preds.append(0)
    return adj_preds
```

And finally after we train the XGBoost with the dataset that we are using, then after being trained it will be tested with the test dataset that has been defined previously. Here the parameters will be set to find the best results. predictions were made using the XGBoost model that had been previously trained and using a previously defined dataset. Then the results of accuracy, precision, recall, and f1-score will be printed.

```
model, eta = xgboost_train(X_train, y_train,
          eta = 0.4,
          max_round = 45,
          max_depth = 3,
          row_sub_frac = 0.95,
          col_sub_frac = 1,
          min_child_weight = 1,
          min_sample_split = 10,
          lambda = 1,
          gamma = 0,
          eps = 0.003,
          metric = 'accuracy')
pred = xgboost_predict(model, X_test1, eta)
print('accuracy :', accuracy(y_test1, pred))
print('precision :', precision(y_test1, pred))
print('recall :', recall(y_test1, pred))
print('f1-score :', f1(y_test1, pred))
pred = xgboost_predict(model, X_test2, eta)
print('accuracy :', accuracy(y_test2, pred))
print('precision :', precision(y_test2, pred))
print('recall :', recall(y_test2, pred))
print('f1-score :', f1(y_test2, pred))
pred = xgboost_predict(model, X_test3, eta)
print('accuracy :', accuracy(y_test3, pred))
print('precision :', precision(y_test3, pred))
print('recall :', recall(y_test3, pred))
print('f1-score :', f1(y_test3, pred))
```

5.2. Result

In this project, the existing code produces the following:

- Logistic Regression

In the Logistic Regression model, 2 parameters are used, namely the learning rate and number iteration. Here is the Logistic Regression result of several testing from different parameter and dataset ratio.

Table 5.2.1 Logistic Regression Result 60:40

Pembagian data	**Iteration number**	**Learning rate**	**Testing**	**accuracy**	**precision**	**Recall**	**F1 score**
60:40	50.000	0.0000001	1	0.6829	0.6829	1.0	0.8115
			2	0.6829	0.6829	1.0	0.8115
			3	0.6951	0.6951	1.0	0.8201
		0.000001	1	0.6829	0.6829	1.0	0.8115
			2	0.6829	0.6829	1.0	0.8115
			3	0.6951	0.6951	1.0	0.8201
		0.0000025	1	0.6463	0.6956	0.8571	0.7679
			2	0.6097	0.6875	0.7857	0.7333
			3	0.6463	0.7058	0.8421	0.7679
	500.000	0.0000001	1	0.6829	0.6829	1.0	0.8115
			2	0.6829	0.6829	1.0	0.8115
			3	0.6951	0.6951	1.0	0.8201
		0.000001	1	0.6829	0.6829	1.0	0.8159
			2	0.6829	0.6829	1.0	0.8159
			3	0.6951	0.6951	1.0	0.8201
		0.0000025	1	0.6585	0.7	0.875	0.7777
			2	0.5853	0.6718	0.7678	0.7166
			3	0.6463	0.7058	0.8421	0.7679

Table 5.2.2 Logistic Regression Result 70:30

Pembagian data	Iteration number	Learning rate	Testing	accuracy	precision	Recall	F1 score
70:30	50.000	0.0000001	1	0.6885	0.6885	1.0	0.8155
			2	0.6721	0.6833	0.9762	0.8039
			3	0.6825	0.6825	1.0	0.1113
		0.000001	1	0.6885	0.6885	1.0	0.8155
			2	0.6721	0.6833	0.9762	0.8039
			3	0.6825	0.6825	1.0	0.8113
		0.0000025	1	0.6721	0.6833	0.9762	0.8039
			2	0.6393	0.6786	0.9048	0.7755
			3	0.7460	0.7368	0.9767	0.8399
	500.000	0.0000001	1	0.6885	0.6885	1.0	0.8155
			2	0.6721	0.6833	0.9762	0.8039
			3	0.6825	0.6825	1.0	0.8113
		0.000001	1	0.6885	0.6885	1.0	0.8155
			2	0.6885	0.6855	1.0	0.8155
			3	0.6825	0.6825	1.0	0.8113
		0.0000025	1	0.6721	0.6833	0.9762	0.8039
			2	0.6721	0.6964	0.9286	0.7959
			3	0.7460	0.7288	1.0	0.8431

- XGBoost

The XGBoost model uses 10 parameters, namely learning rate, max_round, max_depth, row_sub_frac, xol_sub_frac, min_child_weight, min_sample_split, lambda, gamma and epsilon. Here is the extreme gradient boosting result of several testing from different parameter and dataset ratio.

Table 5.2.3 Extreme Gradient Boosting Result 60:40

Pembagian data	**Iteration number**	**Learning rate**	**Testing**	**accuracy**	**precision**	**Recall**	**F1 score**
60:40	30	0.3	1	0.8170	0.8474	0.8928	0.8695
			2	0.7195	0.8	0.7857	0.7927
			3	0.7926	0.8225	0.8947	0.8571
		0.4	1	0.8658	0.8461	0.9821	0.9090
			2	0.7073	0.7758	0.8035	0.7894
			3	0.8170	0.85	0.8947	0.8717
		0.5	1	0.8170	0.8596	0.875	0.8672
			2	0.6829	0.7884	0.7321	0.7592
			3	0.7560	0.8245	0.8245	0.8245
	45	0.3	1	0.8292	0.8387	0.9285	0.8813
			2	0.7195	0.7619	0.8571	0.8067
			3	0.7926	0.8225	0.8947	0.8571
		0.4	1	0.7814	0.8412	0.9464	0.8907
			2	0.6585	0.7592	0.7321	0.7454
			3	0.7748	0.8474	0.8771	0.8620
		0.5	1	0.8658	0.8571	0.9642	0.9075
			2	0.7251	0.7719	0.7857	0.7787
			3	0.8536	0.8461	0.9649	0.9016

Table 5.2.4 Extreme Gradient Boosting Result 70:30

Pembagian data	**Iteration number**	**Learning rate**	**Testing**	**accuracy**	**precision**	**Recall**	**F1 score**
70:30	30	0.3	1	0.8196	0.8163	0.9523	0.8791
			2	0.8032	0.8	0.9523	0.8695
			3	0.8253	0.9767	0.9767	0.8842
		0.4	1	0.8032	0.8125	0.9285	0.8666
			2	0.7613	0.7906	0.8095	0.8
			3	0.7777	0.8085	0.8837	0.8444
		0.5	1	0.7704	0.8043	0.8809	0.8409
			2	0.7377	0.7826	0.8571	0.8181
			3	0.7936	0.8125	0.9069	0.8571
	45	0.3	1	0.8032	0.8125	0.9285	0.8666
			2	0.7840	0.8	0.8571	0.8275
			3	0.7936	0.8260	0.8837	0.8837
		0.4	1	0.7704	0.8043	0.8809	0.8409
			2	0.7377	0.7826	0.8571	0.8181
			3	0.7936	0.8125	0.9069	0.8571
		0.5	1	0.7540	0.8139	0.8333	0.8235
			2	0.7613	0.8048	0.7857	0.7951
			3	0.7619	0.8333	0.8139	0.8235

5.3. Analysis

In this project, for the first time, data collection was carried out by reading the dataset. Then EDA is carried out, here will be analyzed the variables in the dataset used, which ones affect and do not affect the loan status. After that, data cleaning is carried out by removing unused columns, filling in missing values with the mode or mean of the variable column. And the separation of the data in 2 ways, namely 70% training data, 30% test data and 60% training data, 40% test data, each of them will be divided into three tests to see the consistency of the model, then it will be used for testing the Extreme Gradient Boosting and Logistic Regression algorithm models.

The way Logistic Regression works is by calculating the linear regression function, then predictions are made using the sigmoid function which is then calculated from the gradient of the error created by this algorithm. From the error, the gradient is multiplied by the learning rate, where the results of the calculation used in the next iteration have drawbacks, by using this model and several parameters applied, namely the learning rate and number iteration.

The way extreme gradient boosting works is the first prediction is made, then gradient and hessian calculations will be carried out, then the results will be made a tree where the tree is made based on the best variables. Then the tree that has been made will be predicted, the process will be looped until the iteration process is complete, so that it will produce some of the best trees. With this model and several parameters applied. The accuracy result can change because this algorithm forms a tree with random data that is selected to be a sample of the entire dataset which aims to reduce overfit. The one of sample data that used in this experiment as shown below :

Table 5.3.1 Extreme Gradient Boosting Sample Data

	Dependents	Education	Applicant Income	….	y_pred	grad	hess
427	1	1	14692	….	0.914935	0.714009	0.2042
357	0	1	12500	….	0.914935	0.714009	0.2042
363	0	1	14166	….	0.914935	-0.285991	0.2042
110	0	1	12137	….	0.914935	-0.285991	0.2042
233	2	1	6250	….	0.914935	-0.285991	
….	….	….	….	….	…..	….	….
383	0	1	2132	….	0.914935	-0.285991	0.2042
158	3	1	5266	….	0.914935	-0.285991	0.2042
57	2	1	5417	….	0.914935	-0.285991	0.2042
26	0	1	4843	….	0.914935	-0.285991	0.2042
253	0	1	2980	….	0.914935	-0.285991	0.2042

If the two models are compared, the better one is extreme gradient boosting because it has gradient and hessian as the error calculation while logistic regression only uses gradient for error calculation. Then extreme gradient boosting can choose the best variable using the gain calculation function, while logistic regression cannot choose a variable because this model uses all existing variables. Extreme gradient boosting can have better accuracy because apart from being able to choose the best few variables, this model has its weighting for the variables used.

From the three trials, we can see that in this project the results between the two are consistent, which indicates that there is no overfitting. For comparison of dataset ratios between 70:30 and 60:40, logistic regression shows that the classification result is better at 70:30, although extreme gradient produces better accuracy than logistic regression, but extreme gradient boosting have the highest accuracy result at 60:40, although the highest accuracy results in this project were obtained with a data ratio of 60:40, overall the average accuracy result was better at 70:30 this is due to poor training data, which the dataset has many missing values, and the influence of the parameters used on the dataset. In addition, the Project also conducted several trials to get the best accuracy results by changing the parameters in the model, the parameters that were changed were the learning rate and the number of iterations. The best parameter from these experiments for extreme gradient boosting is a learning rate of

0.5 and the number of iterations is 45. While for Logistic Regression the best parameter is a learning rate of 0.0000025 and the number of iterations is 500000.