```
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_style('whitegrid')
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report
from sklearn.tree import DecisionTreeClassifier
from catboost import CatBoostClassifier, Pool, metrics, cv
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
import warnings
warnings.filterwarnings('ignore')
data = pd.read_csv("/content/LoanStats3a.csv")
data.shape
pd.set_option('display.max_columns', None)
data.head(5)
data.shape
data.info()
data.isnull().sum()
pct = (data.isnull().sum().sum())/(data.shape[0]*data.shape[1])
print("Overall missing values in the data ≈ {:.2f} %".format(pct*100))
plt.figure(figsize=(16,14))
sns.heatmap(data.isnull())
plt.title('Null values heat plot', fontdict={'fontsize': 20})
plt.legend(data.isnull())
plt.show()
temp_df = pd.DataFrame()
temp_df['Percentage of null values'] = ['10% or less', '10% to 20%', '20% to 30%', '30% to
40%', '40% to 50%',
'50% to 60%', '60% to 70%', '70% to 80%', '80% to 90%', 'More than 90%']
# Calculate the percentage of null values for each column
null_percentages = (data.isnull().sum() / len(data))
# Store the columns count separately for each range
ten_percent = len(null_percentages[null_percentages <= 0.1])
ten_to_twenty_percent = len(null_percentages[(null_percentages <= 0.2) &
(null_percentages > 0.1)]) # Corrected logic
```

```
twenty_to_thirty_percent = len(null_percentages[(null_percentages <= 0.3) & (null_percentages > 0.2)]) # Corrected logic
thirty_to_forty_percent = len(null_percentages[(null_percentages <= 0.4) & (null_percentages > 0.3)]) # Corrected logic
forty_to_fifty_percent = len(null_percentages[(null_percentages <= 0.5) & (null_percentages > 0.4)]) # Corrected logic
fifty_to_sixty_percent = len(null_percentages[(null_percentages <= 0.6) & (null_percentages > 0.5)]) # Corrected logic
sixty_to_seventy_percent = len(null_percentages[(null_percentages <= 0.7) & (null_percentages > 0.6)]) # Corrected logic
seventy_to_eighty_percent = len(null_percentages[(null_percentages <= 0.8) & (null_percentages > 0.7)]) # Corrected logic
eighty_to_ninety_percent = len(null_percentages[(null_percentages <= 0.9) & (null_percentages > 0.8)]) # Corrected logic
hundred_percent = len(null_percentages[null_percentages > 0.9])
f1 = data[data.columns[((data.isnull().sum())/len(data)) < 0.4]]
df1.shape
# Checking columns that have only single values in them i.e, constant columns
const_cols = []
for i in df1.columns:
if df1[i].nunique() == 1:
const_cols.append(i)
print(const_cols)
# After observing the above output, we are dropping columns which have single values in them
print("Shape before:", df1.shape)
df1.drop(const_cols, axis=1, inplace = True)
print("Shape after:", df1.shape)
# Columns other than numerical value
colms = df1.columns[df1.dtypes == 'object']
colms
df1[colms].head(2)
dt_cols = ['issue_d', 'earliest_cr_line', 'last_pymnt_d', 'last_credit_pull_d']
for i in dt_cols:
df1[i] = pd.to_datetime(df1[i].astype('str'), format='%b-%y', yearfirst=False)
df1[['issue_d','earliest_cr_line','last_pymnt_d','last_credit_pull_d']].head()
# Considering only year of joining for 'earliest_cr_line' column
df1['earliest_cr_line'] = pd.DatetimeIndex(df1['earliest_cr_line']).year
```

```
# Adding new features by getting month and year from [issue_d, last_pymnt_d, and last_credit_pull_d] columns
df1['issue_d_year'] = pd.DatetimeIndex(df1['issue_d']).year
df1['issue_d_month'] = pd.DatetimeIndex(df1['issue_d']).month
df1['last_pymnt_d_year'] = pd.DatetimeIndex(df1['last_pymnt_d']).year
df1['last_pymnt_d_month'] = pd.DatetimeIndex(df1['last_pymnt_d']).month
df1['last_credit_pull_d_year'] = pd.DatetimeIndex(df1['last_credit_pull_d']).year
df1['last_credit_pull_d_month'] = pd.DatetimeIndex(df1['last_credit_pull_d']).month
# Feature extraction
df1.earliest_cr_line = 2019 - (df1.earliest_cr_line)
df1.issue_d_year = 2019 - (df1.issue_d_year)
df1.last_pymnt_d_year = 2019 - (df1.last_pymnt_d_year)
df1.last_credit_pull_d_year = 2019 - (df1.last_credit_pull_d_year)
# Dropping the original features to avoid data redundancy
df1.drop(['issue_d','last_pymnt_d','last_credit_pull_d'], axis=1, inplace=True)
df1.shape
# Checking for null values in the updated dataframe
plt.figure(figsize=(16,10))
sns.heatmap(df1.isnull())
plt.show()
# Checking for Percentage of null values
a = (df1.isnull().sum() / df1.shape[0]) * 100
b = a[a > 0.00]
b = pd.DataFrame(b, columns = ['Percentage of null values'])
b.sort_values(by= ['Percentage of null values'], ascending=False)
# Dropping the 29 rows which have null values in few columns
df1 = df1[df1['delinq_2yrs'].notnull()]
df1.shape
# Checking again for Percentage of null values
a = (df1.isnull().sum() / df1.shape[0]) * 100
b = a[a > 0.00]
b = pd.DataFrame(b, columns = ['Percentage of null values'])
b.sort_values(by= ['Percentage of null values'], ascending=False
# Imputing the null values with the median value
df1['last_pymnt_d_year'].fillna(df1['last_pymnt_d_year'].median(), inplace=True)
df1['last_pymnt_d_month'].fillna(df1['last_pymnt_d_month'].median(), inplace=True)
df1['last_credit_pull_d_year'].fillna(df1['last_credit_pull_d_year'].median(),
inplace=True)
```

```
df1['last_credit_pull_d_month'].fillna(df1['last_credit_pull_d_month'].median(),
inplace=True)
df1['tax_liens'].fillna(df1['tax_liens'].median(), inplace=True)
# For 'revol_util' column, fill null values with 50%
df1.revol_util.fillna('50%', inplace=True)
# Extracting numerical value from string
df1.revol_util = df1.revol_util.apply(lambda x: x[:-1])
# Converting string to float
df1.revol_util = df1.revol_util.astype('float')
df1.pub_rec_bankruptcies.value_counts()
df1['pub_rec_bankruptcies'].fillna(df1['pub_rec_bankruptcies'].median(), inplace=True)
df1['emp_length'].value_counts()
# Seperating null values by assigning a random string
df1['emp_length'].fillna('5000',inplace=True)
# Filling '< 1 year' as '0 years' of experience and '10+ years' as '10 years'
df1.emp_length.replace({'10+ years':'10 years', '< 1 year':'0 years'}, inplace=True)
# Then extract numerical value from the string
df1.emp_length = df1.emp_length.apply(lambda x: x[:2])
# Converting it's dattype to float
df1.emp_length = df1.emp_length.astype('float')
a = (df1.isnull().sum() / df1.shape[0]) * 100
b = a[a > 0.00]
b = pd.DataFrame(b, columns = ['Percentage of null values'])
b.sort_values(by= ['Percentage of null values'], ascending=False)
df1.drop(['desc', 'emp_title', 'title'], axis = 1, inplace = True)
df1.isnull().sum()
df1.head(2)
df1['term'].unique()
df1['int_rate'].unique()[:5]
df1.term = df1.term.apply(lambda x: x[1:3])
df1.term = df1.term.astype('float')
df1.int_rate = df1.int_rate.apply(lambda x: x[:2])
df1.int_rate = df1.int_rate.astype('float')
df1.head(2)
df2 = df1.drop('zip_code', axis = 1)
df2 = pd.get_dummies(df2, columns = ['home_ownership', 'verification_status',
'purpose', 'addr_state', 'debt_settlement_flag'], drop_first
df2.head(2)
```

```
le = LabelEncoder()
le.fit(df2.grade)
print(le.classes_)
# Update 'grade' column
df2.grade = le.transform(df2.grade)
# Label encoding on 'sub_grade' column
le2 = LabelEncoder()
le2.fit(df2.sub_grade)
le2.classes_
df2.sub_grade = le2.transform(df2.sub_grade)
df2['loan_status'].unique()
# Prediction features
X = df2.drop("loan_status", axis = 1)
# Target variable
y = df2['loan_status']
y.value_counts()
le3 = LabelEncoder()
le3.fit(y)
y_transformed = le3.transform(y)
y_transformed
x_train, x_test, y_train, y_test = train_test_split(X, y_transformed, test_size = 0.20,
stratify = y_transformed, random_state = 2)
x_train.shape, y_train.shape, x_test.shape, y_test.shape
giniDecisionTree = DecisionTreeClassifier(criterion='gini', random_state = 100,
max_depth=3, class_weight = 'balanced', min_samples_leaf = 5)
giniDecisionTree.fit(x_train, y_train)
giniPred = giniDecisionTree.predict(x_test)
print('Accuracy Score: ', accuracy_score(y_test, giniPred))
# Create CatBoostClassifier object
CatBoost_clf = CatBoostClassifier(iterations=5,
learning_rate=0.1,
#loss_function='CrossEntropy'
)
#cat_features = list(range(0, X.shape[1]))
CatBoost_clf.fit(x_train, y_train,
#cat_features=cat_features,
eval_set = (x_test, y_test),
verbose = False)
```

```
# Prediction using CatBoost
cbr_prediction = CatBoost_clf.predict(x_test)
print('Accuracy Score: ', accuracy_score(y_test, cbr_prediction))
print('Classification Report for CatBoost:')
print(classification_report(y_test, cbr_prediction))
# Create XGBClassifier object
XGB_clf = XGBClassifier(learning_rate = 0.1)
# Fit on training set
XGB_clf.fit(x_train, y_train,
eval_set = [(x_train, y_train), (x_test, y_test)],
verbose = False)
# Prediction using XGBClassifier
XGB_prediction = XGB_clf.predict(x_test)
print('Accuracy Score: ', accuracy_score(y_test, XGB_prediction))
print('Classification Report for XGBoost:')
print(classification_report(y_test, XGB_prediction))
# Create LGBMClassifier object
LGBM_clf = LGBMClassifier(learning_rate = 0.1)
from lightgbm import LGBMClassifier, log_evaluation # Import log_evaluation instead of callback
LGBM_clf = LGBMClassifier(learning_rate = 0.1)
# Fit on training set
# Instead of verbose, use callbacks=[lightgbm.log_evaluation(period=1)] to control verbosity
LGBM_clf.fit(x_train, y_train,
eval_set = [(x_train, y_train), (x_test, y_test)],
callbacks=[log_evaluation(period=1)]) # Use log_evaluation instead of callback.print_evaluation
```