Learning Objectives

At the end of the experiment, you will be able to :

- perform data preprocessing
- perform feature transformation
- implement CatBoost, XGBoost and LightGBM model to perform classification using Lending Club dataset

Introduction

XGBoost was originally produced by University of Washington researchers and is maintained by open-source contributors. XGBoost is available in Python, R, Java, Ruby, Swift, Julia, C, and C++. Similar to LightGBM, XGBoost uses the gradients of different cuts to select the next cut, but XGBoost also uses the hessian, or second derivative, in its ranking of cuts. Computing this next derivative comes at a slight cost, but it also allows a greater estimation of the cut to use.

CatBoost is developed and maintained by the Russian search engine Yandex and is available in Python, R, C++, Java, and also Rust. CatBoost distinguishes itself from LightGBM and XGBoost by focusing on optimizing decision trees for categorical variables, or variables whose different values may have no relation with each other (eg. apples and oranges).

LightGBM is a boosting technique and framework developed by Microsoft. The framework implements the LightGBM algorithm and is available in Python, R, and C. LightGBM is unique in that it can construct trees using Gradient-Based One-Sided Sampling, or GOSS for short.

To know more on comparisons between CatBoost, XgBoost and LightGBM, refer below

- Article 1
- Article 2

Dataset Description

Lending Club is a lending platform that lends money to people in need at an interest rate based on their credit history and other factors. We will analyze this data and pre-process it based on our need and build a machine learning model that can identify a potential defaulter based on his/her history of transactions with Lending Club.

This dataset contains 42538 rows and 144 columns. **Out of these 144 columns, many columns have majorly null values.**

To know more about the Lending Club dataset features, refer here.

Import required packages

```
!pip -qq install catboost
```

```
# 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')
!pip -qq install catboost  # Install catboost

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  # Import catboost modules
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
import warnings
warnings.filterwarnings('ignore')
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 98.7/98.7 MB 8.1 MB/s eta 0:00:00
/usr/local/lib/python3.10/dist-packages/dask/dataframe/__init__.py:42: FutureWarning:
Dask dataframe query planning is disabled because dask-expr is not installed.

You can install it with `pip install dask[dataframe]` or `conda install dask`.
This will raise in a future version.

  warnings.warn(msg, FutureWarning)
```

Load Dataset

```
# Load the raw loan stats dataset
# data = pd.read_csv("LoanStats3a.csv")
# data.shape
# Load the raw loan stats dataset
# Load the raw loan stats dataset
# data = pd.read_csv("LoanStats3a.csv")
# data.shape
# Load the raw loan stats dataset
try:
    # Use on_bad_lines='skip' instead of error_bad_lines=False for newer pandas versions
    data = pd.read_csv("LoanStats3a.csv", on_bad_lines='skip')
    print("Data loaded successfully, but some rows might have been skipped due to errors.")
except pd.errors.ParserError as e:
    print(f"ParserError: {e}")
    print("Trying to load data with 'quotechar=\"'\"' to handle potential quote issues...")
    try:
        # Use on_bad_lines='skip' instead of error_bad_lines=False for newer pandas versions
        data = pd.read_csv("LoanStats3a.csv", on_bad_lines='skip', quotechar="\"'")
        print("Data loaded successfully with quotechar=\"'\"'.")
    except pd.errors.ParserError as e:
        print(f"ParserError: {e}")
        print("Please check the file for unclosed quotes or other data inconsistencies.")

data.shape
```

```
Data loaded successfully, but some rows might have been skipped due to errors.
(42538, 144)
```

Data Preprocessing

```
# View the top 5 rows of data
pd.set_option('display.max_columns', None)

data.head(5)
```

	member_id	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	installment	grade	sub_grade	emp_title	emp_length	home_ow
0	NaN	5000.0	5000.0	4975.0	36 months	10.65%	162.87	B	B2	NaN	10+ years	
1	NaN	2500.0	2500.0	2500.0	60 months	15.27%	59.83	C	C4	Ryder	< 1 year	
2	NaN	2400.0	2400.0	2400.0	36 months	15.96%	84.33	C	C5	NaN	10+ years	
3	NaN	10000.0	10000.0	10000.0	36 months	13.49%	339.31	C	C1	AIR RESOURCES BOARD	10+ years	
4	NaN	3000.0	3000.0	3000.0	60 months	12.69%	67.79	B	B5	University Medical Group	1 year	

```
# Size of the dataset
data.shape
```

```
(42538, 144)
```

```
# Checking info of the raw dataframe
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 42538 entries, 0 to 42537
Columns: 144 entries, member_id to settlement_term
dtypes: float64(115), object(29)
memory usage: 46.7+ MB
```

Check for missing values in the dataset

```
# Check missing values
data.isnull().sum()
```

	0
member_id	42538
loan_amnt	3
funded_amnt	3
funded_amnt_inv	3
term	3
...	...
settlement_status	42378
settlement_date	42378
settlement_amount	42378
settlement_percentage	42378
settlement_term	42378

144 rows × 1 columns

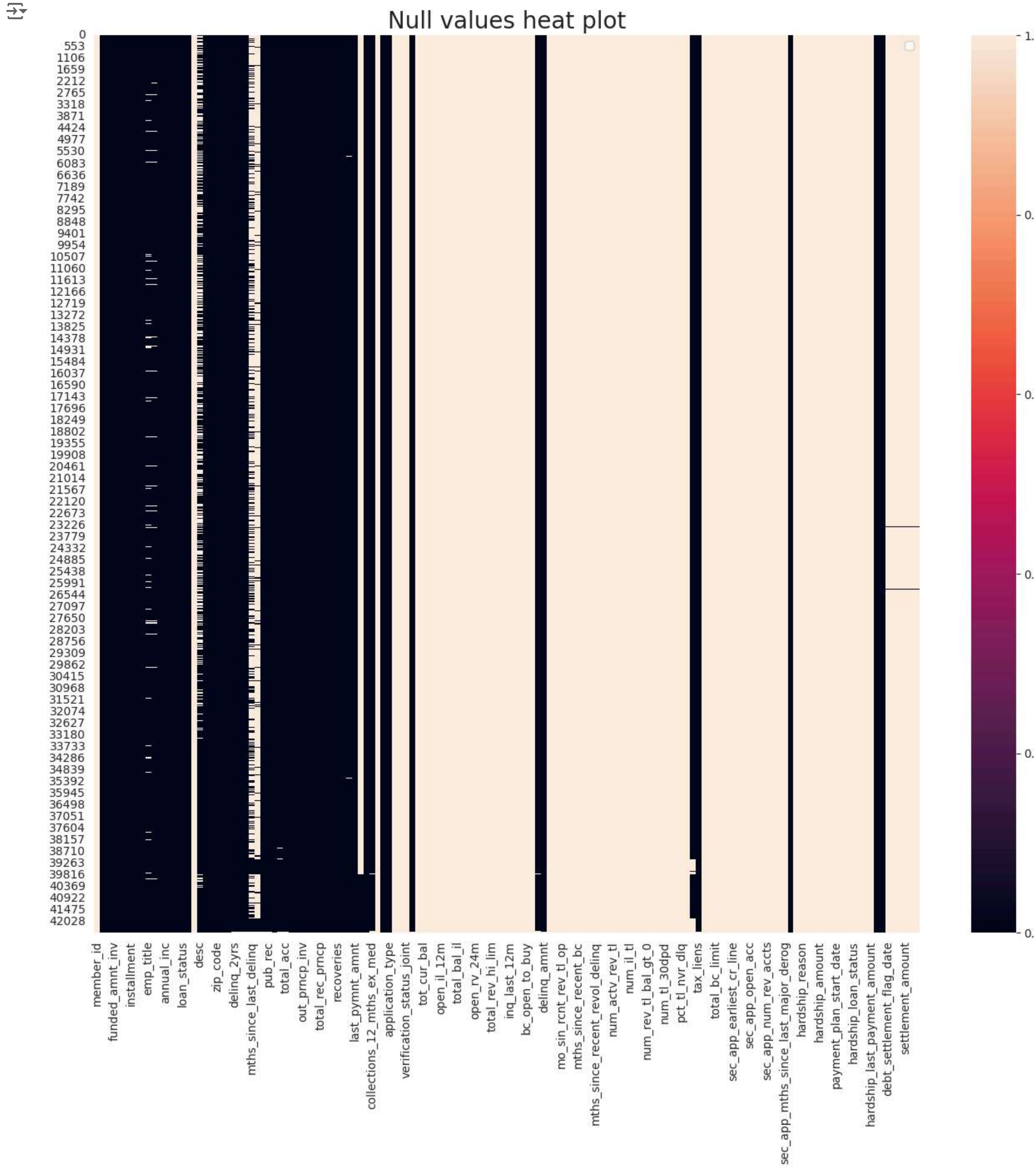
```
# Total percentage of null values in the data
pct = (data.isnull().sum().sum())/(data.shape[0]*data.shape[1])
print("Overall missing values in the data ≈ {:.2f} %".format(pct*100))
```

```
Overall missing values in the data ≈ 62.44 %
```

From above we can see that, about 63% of the values in the overall data are null values.

Let's visualize the null values using the heatmap.

```
# Checking for null values using a heatmap as a visualizing tool
plt.figure(figsize=(16,14))
sns.heatmap(data.isnull())
plt.title('Null values heat plot', fontdict={'fontsize': 20})
plt.legend(data.isnull())
plt.show()
```



As we can see from the above heatmap, there are lot of null values in the dataset. We have to carefully deal with these null values.

Handling missing values in the data

- Select columns having null values less than 40%

```
# # Creating a dataframe to display percentage of null values in different number of columns
# 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%']

# # Store the columns count separately for each range
# ten_percent = len(data.columns[((data.isnull().sum())/len(data)) <= 0.1])
# ten_to_twenty_percent = len(data.columns[((data.isnull().sum())/len(data)) <= 0.2] & data.columns[((data.isnull().sum())/len(data)) > 0.1]
# twenty_to_thirty_percent = len(data.columns[((data.isnull().sum())/len(data)) <= 0.3] & data.columns[((data.isnull().sum())/len(data)) > 0
# thirty_to_forty_percent = len(data.columns[((data.isnull().sum())/len(data)) <= 0.4] & data.columns[((data.isnull().sum())/len(data)) > 0.
# forty_to_fifty_percent = len(data.columns[((data.isnull().sum())/len(data)) <= 0.5] & data.columns[((data.isnull().sum())/len(data)) > 0.4
# fifty_to_sixty_percent = len(data.columns[((data.isnull().sum())/len(data)) <= 0.6] & data.columns[((data.isnull().sum())/len(data)) > 0.5
# sixty_to_seventy_percent = len(data.columns[((data.isnull().sum())/len(data)) <= 0.7] & data.columns[((data.isnull().sum())/len(data)) > 0
# seventy_to_eighty_percent = len(data.columns[((data.isnull().sum())/len(data)) <= 0.8] & data.columns[((data.isnull().sum())/len(data)) >
# eighty_to_ninety_percent = len(data.columns[((data.isnull().sum())/len(data)) <= 0.9] & data.columns[((data.isnull().sum())/len(data)) > 0
# hundred_percent = len(data.columns[((data.isnull().sum())/len(data)) > 0.9])

# temp_df['No. of columns'] = [ten_percent, ten_to_twenty_percent, twenty_to_thirty_percent, thirty_to_forty_percent, forty_to_fifty_percent
#                              fifty_to_sixty_percent, sixty_to_seventy_percent, seventy_to_eighty_percent, eighty_to_ninety_percent, hundre
# temp_df


# Creating a dataframe to display percentage of null values in different number of columns
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])  # Corrected logic


temp_df['No. of columns'] = [ten_percent, ten_to_twenty_percent, twenty_to_thirty_percent, thirty_to_forty_percent, forty_to_fifty_percent,
                             fifty_to_sixty_percent, sixty_to_seventy_percent, seventy_to_eighty_percent, eighty_to_ninety_percent, hundred_
temp_df
```

	Percentage of null values	No. of columns
0	10% or less	53
1	10% to 20%	0
2	20% to 30%	0
3	30% to 40%	1
4	40% to 50%	0
5	50% to 60%	0
6	60% to 70%	1
7	70% to 80%	0
8	80% to 90%	0
9	More than 90%	89

From the above results, we can see that there are only 53 columns out of 144 columns that have null values less than 40%.

```
# Considering only those columns which have null values less than 40% in that particular column
df1 = data[data.columns[((data.isnull().sum())/len(data)) < 0.4]]
df1.shape
```

```
(42538, 54)
```

By considering columns with less number of null values, we were able to decrease total number of columns from 144 to 53.

Note that we will deal with null values present in these selected 53 columns later below.

Removing columns having single distinct value

```
# 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)
```

```
['pymnt_plan', 'initial_list_status', 'out_prncp', 'out_prncp_inv', 'collections_12_mths_ex_med', 'policy_code', 'application_type', 'ch
```

```
# 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)
```

```
Shape before: (42538, 54)
Shape after: (42538, 44)
```

Extract features from datetime columns

```
# Columns other than numerical value
colms = df1.columns[df1.dtypes == 'object']
colms
```

```
Index(['term', 'int_rate', 'grade', 'sub_grade', 'emp_title', 'emp_length',
       'home_ownership', 'verification_status', 'issue_d', 'loan_status',
       'desc', 'purpose', 'title', 'zip_code', 'addr_state',
       'earliest_cr_line', 'revol_util', 'last_pymnt_d', 'last_credit_pull_d',
       'debt_settlement_flag'],
      dtype='object')
```

```
# Check which columns needs to be converted to datetime
df1[colms].head(2)
```

	term	int_rate	grade	sub_grade	emp_title	emp_length	home_ownership	verification_status	issue_d	loan_status	desc	purpo
0	36 months	10.65%	B	B2	NaN	10+ years	RENT	Verified	Dec-11	Fully Paid	Borrower added on 12/22/11 > I need to upgra...	credit_c
1	60 months	15.27%	C	C4	Ryder	< 1 year	RENT	Source Verified	Dec-11	Charged Off	Borrower added on 12/22/11 > I plan to use t...	

```
# Converting objects to datetime columns
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)
```

```
# Checking the new datetime columns
df1[['issue_d','earliest_cr_line','last_pymnt_d','last_credit_pull_d']].head()
```

	issue_d	earliest_cr_line	last_pymnt_d	last_credit_pull_d
0	2011-12-01	1985-01-01	2015-01-01	2018-07-01
1	2011-12-01	1999-04-01	2013-04-01	2016-10-01
2	2011-12-01	2001-11-01	2014-06-01	2017-06-01
3	2011-12-01	1996-02-01	2015-01-01	2016-04-01
4	2011-12-01	1996-01-01	2017-01-01	2018-04-01

```
# 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
```

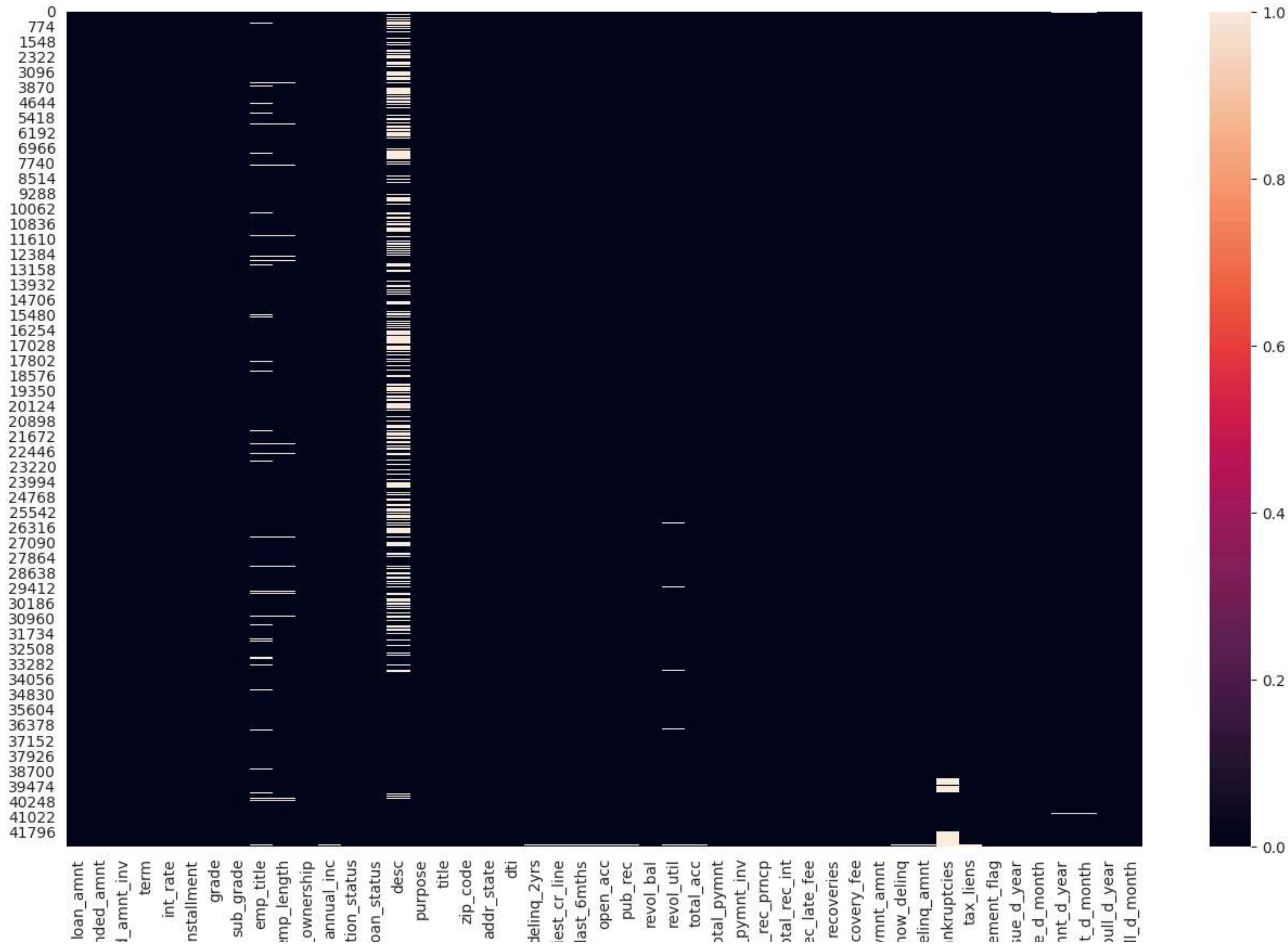
```
# Feature extraction
df1.earliest_cr_line = 2019 - (df1.earliest_cr_line)
df1.issue_d_year = 2019 - (df1.issue_d_year)
df1.last_pymnt_d_year = 2019 - (df1.last_pymnt_d_year)
df1.last_credit_pull_d_year = 2019 - (df1.last_credit_pull_d_year)
```

```
# 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
```

```
(42538, 47)
```

Check for missing values in reduced dataset

```
# Checking for null values in the updated dataframe
plt.figure(figsize=(16,10))
sns.heatmap(df1.isnull())
plt.show()
```



Handling Null values in reduced dataset

```
# 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)
```

	Percentage of null values
desc	31.261460
emp_title	6.180356
pub_rec_bankruptcies	3.215948
emp_length	2.621186
tax_liens	0.253891
revol_util	0.218628
last_pymnt_d_month	0.202172
last_pymnt_d_year	0.202172
total_acc	0.075227
inq_last_6mths	0.075227
pub_rec	0.075227
open_acc	0.075227
acc_now_delinq	0.075227
delinq_2yrs	0.075227
delinq_amnt	0.075227
earliest_cr_line	0.075227
title	0.037613
last_credit_pull_d_year	0.016456
last_credit_pull_d_month	0.016456
annual_inc	0.016456
dti	0.007053
collection_recovery_fee	0.007053
funded_amnt_inv	0.007053
term	0.007053
int_rate	0.007053
issue_d_month	0.007053
issue_d_year	0.007053
debt_settlement_flag	0.007053
installment	0.007053
grade	0.007053
sub_grade	0.007053

```
# Dropping the 29 rows which have null values in few columns
df1 = df1[df1['delinq_2yrs'].notnull()]
df1.shape
```

(42506, 47)

addr_state	0.007053

```
# 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)
```

	Percentage of null values
desc	31.275585
emp_title	6.149720
pub_rec_bankruptcies	3.143086
emp_length	2.616101
last_pymnt_d_year	0.195267
last_pymnt_d_month	0.195267
tax_liens	0.178798
revol_util	0.143509
title	0.030584
last_credit_pull_d_year	0.007058
last_credit_pull_d_month	0.007058

Now, imputing the missing values with the median value for columns **'last_pymnt_d_year', 'last_pymnt_d_month', 'last_credit_pull_d_year', 'last_credit_pull_d_month', 'tax_liens'** as null values in these columns are less than 0.5% of the size.

```
# 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, filling null values with median(string) which is close to 50:

```
# 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')
```

```
# Unique values in 'pub_rec_bankruptcies' column
df1.pub_rec_bankruptcies.value_counts()
```

pub_rec_bankruptcies	count
0.0	39316
1.0	1846
2.0	8

From the above we can see that the **'pub_rec_bankruptcies'** column is highly imbalanced. So, it is better to fill it with median(0) value as even after building model the model will be skewed very much towards 0.

```
# Fill 'pub_rec_bankruptcies' column
df1['pub_rec_bankruptcies'].fillna(df1['pub_rec_bankruptcies'].median(), inplace=True)
```

```
# Unique values in 'emp_length' column
df1['emp_length'].value_counts()
```

emp_length	count
10+ years	9366
< 1 year	5044
2 years	4742
3 years	4362
4 years	3649
1 year	3592
5 years	3458
6 years	2374
7 years	1875
8 years	1592
9 years	1340

dtype: int64

```
# 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')
```

```
# 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)
```

	Percentage of null values
desc	31.275585
emp_title	6.149720
title	0.030584

```
# Removing redundant features and features which have percentage null values > 5%
df1.drop(['desc', 'emp_title', 'title'], axis = 1, inplace = True)
df1.isnull().sum()
```

	0
loan_amnt	0
funded_amnt	0
funded_amnt_inv	0
term	0
int_rate	0
installment	0
grade	0
sub_grade	0
emp_length	0
home_ownership	0
annual_inc	0
verification_status	0
loan_status	0
purpose	0
zip_code	0
addr_state	0
dti	0
delinq_2yrs	0
earliest_cr_line	0
inq_last_6mths	0
open_acc	0
pub_rec	0
revol_bal	0
revol_util	0
total_acc	0
total_pymnt	0
total_pymnt_inv	0
total_rec_prncp	0
total_rec_int	0
total_rec_late_fee	0
recoveries	0

Converting categorical columns to numerical columns

```
df1.head(2)
```

	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	installment	grade	sub_grade	emp_length	home_ownership	annual_inc	veri
0	5000.0	5000.0	4975.0	36 months	10.65%	162.87	B	B2	10.0	RENT	24000.0	
1	2500.0	2500.0	2500.0	60 months	15.27%	59.83	C	C4	0.0	RENT	30000.0	

```
# Unique values in 'term' column
df1['term'].unique()
```

```
array([' 36 months', ' 60 months'], dtype=object)
```

last_credit_pull_d_month 0

```
# Unique values in 'int_rate' column
df1['int_rate'].unique()[:5]
```

```
array(['10.65%', '15.27%', '15.96%', '13.49%', '12.69%'], dtype=object)
```

```
# Converting 'term' and 'int_rate' to numerical columns
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)
```

	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	installment	grade	sub_grade	emp_length	home_ownership	annual_inc	verifi
0	5000.0	5000.0	4975.0	36.0	10.0	162.87	B	B2	10.0	RENT	24000.0	
1	2500.0	2500.0	2500.0	60.0	15.0	59.83	C	C4	0.0	RENT	30000.0	

Among the address related features, considering **'addr_state'** column and excluding **'zip_code'** column.

```
df2 = df1.drop('zip_code', axis = 1)
```

```
# One hot encoding on categorical columns
df2 = pd.get_dummies(df2, columns = ['home_ownership', 'verification_status', 'purpose', 'addr_state', 'debt_settlement_flag'], drop_first =
df2.head(2)
```

	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	installment	grade	sub_grade	emp_length	annual_inc	loan_status	dti	de
0	5000.0	5000.0	4975.0	36.0	10.0	162.87	B	B2	10.0	24000.0	Fully Paid	27.65	
1	2500.0	2500.0	2500.0	60.0	15.0	59.83	C	C4	0.0	30000.0	Charged Off	1.00	

```
# Label encoding on 'grade' column
le = LabelEncoder()
le.fit(df2.grade)
print(le.classes_)
```

```
['A' 'B' 'C' 'D' 'E' 'F' 'G']
```

```
# Update 'grade' column
df2.grade = le.transform(df2.grade)
```

```
# Label encoding on 'sub_grade' column
le2 = LabelEncoder()
le2.fit(df2.sub_grade)
le2.classes_
```

```
array(['A1', 'A2', 'A3', 'A4', 'A5', 'B1', 'B2', 'B3', 'B4', 'B5', 'C1',
       'C2', 'C3', 'C4', 'C5', 'D1', 'D2', 'D3', 'D4', 'D5', 'E1', 'E2',
       'E3', 'E4', 'E5', 'F1', 'F2', 'F3', 'F4', 'F5', 'G1', 'G2', 'G3',
       'G4', 'G5'], dtype=object)
```

```
# Update 'sub_grade' column
df2.sub_grade = le2.transform(df2.sub_grade)
```

```
df2.head(2)
```

	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	installment	grade	sub_grade	emp_length	annual_inc	loan_status	dti	de
0	5000.0	5000.0	4975.0	36.0	10.0	162.87	1	6	10.0	24000.0	Fully Paid	27.65	
1	2500.0	2500.0	2500.0	60.0	15.0	59.83	2	13	0.0	30000.0	Charged Off	1.00	

```
# Target feature
df2['loan_status'].unique()
```

```
array(['Fully Paid', 'Charged Off',
       'Does not meet the credit policy. Status:Fully Paid',
```

```
         'Does not meet the credit policy. Status:Charged Off'],
        dtype=object)
```

```
# Prediction features
X = df2.drop("loan_status", axis = 1)
# Target variable
y = df2['loan_status']
y.value_counts()
```

loan_status	count
Fully Paid	34116
Charged Off	5670
Does not meet the credit policy. Status:Fully Paid	1962
Does not meet the credit policy. Status:Charged Off	758

dtype: int64

```
# Label encoding the target variable
le3 = LabelEncoder()
le3.fit(y)
y_transformed = le3.transform(y)
y_transformed
```

```
array([3, 0, 3, ..., 2, 2, 2])
```

```
X.head(2)
```

	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	installment	grade	sub_grade	emp_length	annual_inc	dti	delinq_2yrs	ea
0	5000.0	5000.0	4975.0	36.0	10.0	162.87	1	6	10.0	24000.0	27.65	0.0	
1	2500.0	2500.0	2500.0	60.0	15.0	59.83	2	13	0.0	30000.0	1.00	0.0	

Split data into training and testing set

```
# Split the data into train and test
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
```

```
((34004, 106), (34004,), (8502, 106), (8502,))
```

Model Building

```
# Using DecisionTree as base model
giniDecisionTree = DecisionTreeClassifier(criterion='gini', random_state = 100,
                                          max_depth=3, class_weight = 'balanced', min_samples_leaf = 5)
giniDecisionTree.fit(x_train, y_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(class_weight='balanced', max_depth=3, min_samples_leaf=5,
                       random_state=100)
```

```
# Prediciton using DecisionTree
giniPred = giniDecisionTree.predict(x_test)
print('Accuracy Score: ', accuracy_score(y_test, giniPred))
```

```
Accuracy Score:  0.9426017407668784
```

CatBoost

```
# 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)
```

```
<catboost.core.CatBoostClassifier at 0x7b41a8bdabc0>
```

```
# Prediction using CatBoost
cbr_prediction = CatBoost_clf.predict(x_test)
print('Accuracy Score: ', accuracy_score(y_test, cbr_prediction))
```

```
Accuracy Score:  0.9704775346977181
```

```
#  Classification report for CatBoost model
print('Classification Report for CatBoost:')
print(classification_report(y_test, cbr_prediction))
```

```
Classification Report for CatBoost:
              precision    recall  f1-score   support

           0       0.97      0.97      0.97      1134
           1       0.83      0.72      0.77       152
           2       0.84      0.68      0.75       392
```