1. Overview

Based on the descriptive and exploratory analysis done in notebook 00_data_understanding, this Python Script will work on preprocessing the data, preparing it so that we can then work on the model training in the future.

2. Data Understanding

2.1 Data Description

This file will use the df_train_transform excel sheet created in the previous notebook: 00_data_understanding

2.2 Import Necessary Libraries

In [1]:
```
# pip install category_encoders
```

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from sklearn.preprocessing import OneHotEncoder
from category_encoders import TargetEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split


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

```

3. Code

3.1 Import the database

In [3]:

```
df = pd.read_excel('df_train_transform.xlsx')
df.head()
```

Out[3]:

	amount_tsh	gps_height	population	basin	region	public_meeting	permit	extraction_typ
0	6000.0	1390	109	lake nyasa	iringa	1.0	0.0	
1	0.0	1399	280	lake victoria	mara	NaN	1.0	
2	25.0	686	250	pangani	manyara	1.0	1.0	
3	0.0	263	58	ruvuma southern coast	mtwara	1.0	1.0	sub
4	0.0	0	0	lake victoria	kagera	1.0	1.0	

3.2 Class Imbalance checking

In [4]:

```
# Check class distribution in y_train
print("Class distribution of status_group:")
print(df['status_group'].value_counts(normalize=False))
```

```
Class distribution of status_group:
functional                 32259
non functional             22824
functional needs repair     4317
Name: status_group, dtype: int64
```

In [5]:

```
# Check class distribution in y_train
print("Class distribution of status_group:")
print(df['status_group'].value_counts(normalize=True))
```

```
Class distribution of status_group:
functional                 0.543081
non functional             0.384242
functional needs repair    0.072677
Name: status_group, dtype: float64
```

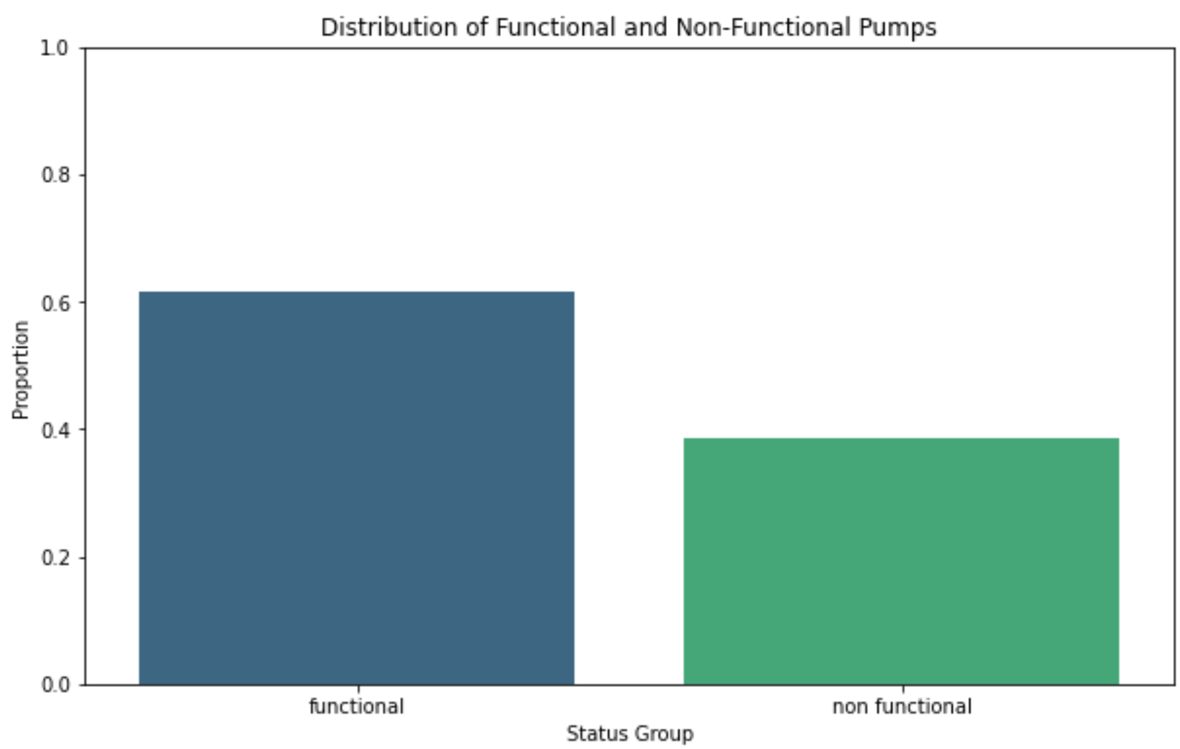
We decide to group together into a same class functional needs repair and functional. In this way, we have a binary classification problem

In [6]:
```
# Replace 'functional needs repair' with 'functional'
df['status_group'] = df['status_group'].replace('functional needs repair',

# Verify changes by checking the class distribution again in y_train and y
print("Class distribution in y_train after replacement:")
print(df['status_group'].value_counts(normalize=True))
```

```
Class distribution in y_train after replacement:
functional        0.615758
non functional    0.384242
Name: status_group, dtype: float64
```

To have further insight, let's do a bar graph representation of the distribution of the target variable

In [7]:
```
class_distribution = df['status_group'].value_counts(normalize=True)

# Plotting the bar plot
plt.figure(figsize=(10, 6))
sns.barplot(x=class_distribution.index, y=class_distribution.values, palet
plt.xlabel('Status Group')
plt.ylabel('Proportion')
plt.title('Distribution of Functional and Non-Functional Pumps')
plt.ylim(0, 1)
plt.show()
```



3.3 Define predictor and target variables

In [8]:
```
y = df['status_group']
X = df.drop('status_group', axis=1)
```

3.4 Do a train test split

In [9]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, r
```

3.5 Dealing with null values

In [10]:
```
# For train data
(X_train.isna().sum()/len(df))*100
```

Out[10]:
```
amount_tsh                 0.000000
gps_height                 0.000000
population                 0.000000
basin                      0.000000
region                     0.000000
public_meeting             4.526936
permit                     4.106061
extraction_type_class      0.000000
management_group           0.000000
payment_type               0.000000
quality_group              0.000000
quantity_group             0.000000
source_type                0.000000
waterpoint_type            0.000000
funder_type                0.000000
installer_type             0.000000
scheme_management_grouped  0.000000
dtype: float64
```

Column 'public_meeting'

In [11]:
```
X_train["public_meeting"].value_counts(normalize=True)
```

Out[11]:
```
1.0    0.908813
0.0    0.091187
Name: public_meeting, dtype: float64
```

In [12]:
```
# Given that the null values are only 6%, lets replace them with the mode

# Calculate the mode of the 'public_meeting' column
public_meeting_mode = X_train['public_meeting'].mode()[0]

# Fill missing values in 'public_meeting' of X_train with the mode from X_
X_train['public_meeting'].fillna(public_meeting_mode, inplace=True)

# Fill missing values in 'public_meeting' of X_test with the mode from X_t
X_test['public_meeting'].fillna(public_meeting_mode, inplace=True)

# Convert the 'public_meeting' column to type object in both X_train and X
X_train['public_meeting'] = X_train['public_meeting'].astype(object)
X_test['public_meeting'] = X_test['public_meeting'].astype(object)

# Verify if all NA values are filled
print(df['public_meeting'].value_counts(normalize=True))
```

```
1.0    0.909838
0.0    0.090162
Name: public_meeting, dtype: float64
```

In [13]:
```
public_meeting_mode
```

Out[13]:
```
1.0
```

Column 'permit'

In [14]:
```
df["permit"].value_counts(normalize=True)
```

Out[14]:
```
1.0    0.68955
0.0    0.31045
Name: permit, dtype: float64
```

In [15]:

```
# Given that the null values are only 5%, lets replace them with the mode

# Calculate the mode of the 'permit' column
permit_mode = X_train['permit'].mode()[0]

# Fill missing values in 'permit' of X_train with the mode of X_train
X_train['permit'].fillna(permit_mode, inplace=True)

# Fill missing values in 'permit' of X_test with the mode of X_train
X_test['permit'].fillna(permit_mode, inplace=True)

# Convert the 'permit' column to type object in both X_train and X_test
X_train['permit'] = X_train['permit'].astype(object)
X_test['permit'] = X_test['permit'].astype(object)

# Verify if all NA values are filled
print(X_train['permit'].value_counts(normalize=True))
```

```
1.0    0.704272
0.0    0.295728
Name: permit, dtype: float64
```

In [16]:

```
permit_mode
```

Out[16]:

```
1.0
```

3.6 Doing target enconder on the categorical columns

Let's perform a one hot enconder on the categorical columns that have less than 6 categories

In [17]:

```
# Identifying categorical columns
categorical_columns = X_train.select_dtypes(include=['object', 'category']

# Printing the list of categorical columns
print("Categorical columns in X_train:")
print(categorical_columns)
```

```
Categorical columns in X_train:
Index(['basin', 'region', 'public_meeting', 'permit', 'extraction_type_clas
s',
       'management_group', 'payment_type', 'quality_group', 'quantity_group',
       'source_type', 'waterpoint_type', 'funder_type', 'installer_type',
       'scheme_management_grouped'],
      dtype='object')
```

X_train

Let's do a code to apply one hot encnder on the columns that have less than 6 variables and a target encnder on the columns that have more than 6 variables. The reason why we decide to not apply target encoding to all the columns directly is to avoid overfitting

In [18]:

```
# Check if 'y_train' and 'y_test' need to be converted to a numeric type
if y_train.dtype == 'object':
    y_train = y_train.astype('category').cat.codes
if y_test.dtype == 'object':
    y_test = y_test.astype('category').cat.codes

# Capture categorical columns from X_train for encoding
categorical_columns = X_train.select_dtypes(include=['object', 'category']

# Initialize encoders
target_encoder = TargetEncoder()

# Encoding the categorical columns in X_train and X_test
for col in categorical_columns:
    if X_train[col].nunique() <= 6:
        # Apply OneHotEncoder for columns with 6 or fewer unique values
        X_train = pd.get_dummies(X_train, columns=[col], drop_first=True)
        X_test = pd.get_dummies(X_test, columns=[col], drop_first=True)
    else:
        # Apply TargetEncoder for columns with more than 6 unique values
        X_train[col] = target_encoder.fit_transform(X_train[col], y_train)
        X_test[col] = target_encoder.transform(X_test[col])
        pickle.dump(target_encoder, open(f"model_objects/{col}_target_enco

# Display the DataFrame to check the results
X_train.head()

```

Out[18]:

	amount_tsh	gps_height	population	basin	region	extraction_type_class	payment_t
3607	50.0	2092	160	0.346722	0.315956	0.300187	0.277
50870	0.0	0	0	0.346722	0.443875	0.309484	0.475
20413	0.0	0	0	0.485901	0.398196	0.805243	0.475
52806	0.0	0	0	0.311216	0.398196	0.300187	0.226
50091	300.0	1023	120	0.432348	0.398697	0.805243	0.308

5 rows × 34 columns

3.7 Dealing with numerical columns

X_train

In [19]:
```
# Capture numerical columns
numerical_columns = X_train.select_dtypes(include=['int64', 'float64']).cc
 
# Initialize the StandardScaler
scaler = StandardScaler()

# Fit and transform the numerical columns
scaler.fit(X_train[numerical_columns])

X_train[numerical_columns] = scaler.transform(X_train[numerical_columns])

# Save the fitted variables
pickle.dump(scaler, open(f"model_objects/numerical_columns_scaler.pkl", 'w

# Display the DataFrame to check the results
X_train.head()
```

Out[19]:

	amount_tsh	gps_height	population	basin	region	extraction_type_class	payment_
3607	-0.084999	2.053863	-0.041306	-0.540016	-0.633090	-0.521411	-0.89
50870	-0.100621	-0.965049	-0.379739	-0.540016	0.555492	-0.463637	0.77
20413	-0.100621	-0.965049	-0.379739	1.471270	0.131062	2.617222	0.77
52806	-0.100621	-0.965049	-0.379739	-1.053126	0.131062	-0.521411	-1.33
50091	-0.006889	0.511216	-0.125914	0.697368	0.135714	2.617222	-0.64

5 rows × 34 columns

In [20]:
```
X_train[numerical_columns].columns
```

Out[20]:
```
Index(['amount_tsh', 'gps_height', 'population', 'basin', 'region',
       'extraction_type_class', 'payment_type', 'source_type',
       'waterpoint_type', 'installer_type'],
      dtype='object')
```

In [21]:
```
numerical_columns

print(len(numerical_columns))
```

```
10
```

X_test

In [22]:
```
X_test[numerical_columns] = scaler.transform(X_test[numerical_columns])

# Display the DataFrame to check the results
X_test.head()
```

Out[22]:

	amount_tsh	gps_height	population	basin	region	extraction_type_class	payment_
2980	-0.100621	-0.965049	-0.379739	0.205860	-0.699807	2.617222	1.0
5246	-0.100621	-0.965049	-0.379739	0.205860	1.453840	-0.463637	0.7
22659	-0.097497	1.452101	-0.066689	-0.540016	-0.633090	-0.521411	-0.8
39888	-0.100621	-0.965049	-0.379739	1.471270	0.131062	-0.463637	0.7
13361	-0.084999	0.635320	0.117334	-0.540016	0.663779	1.165688	-0.8

5 rows × 34 columns

3.8 Concatenate train on one side and test on the other

In [23]:
```
# Concatenate all train
df_train = pd.concat([X_train, y_train], axis=1)

# Concatenate all test
df_test = pd.concat([X_test, y_test], axis=1)

# Create a label column
df_train['is_test'] = 0
df_test['is_test'] = 1
```

3.9 Concatenate everything in one dataframe

In [24]:
```
data_processed = pd.concat([df_train,df_test], axis=0)

# Reset index
data_processed = data_processed.reset_index(drop=True)

# Rename column 0 to status_group
data_processed = data_processed.rename(columns={0: 'status_group'})

data_processed
```

Out[24]:

	amount_tsh	gps_height	population	basin	region	extraction_type_class	payment
0	-0.084999	2.053863	-0.041306	-0.540016	-0.633090	-0.521411	-0.89
1	-0.100621	-0.965049	-0.379739	-0.540016	0.555492	-0.463637	0.77
2	-0.100621	-0.965049	-0.379739	1.471270	0.131062	2.617222	0.77
3	-0.100621	-0.965049	-0.379739	-1.053126	0.131062	-0.521411	-1.33
4	-0.006889	0.511216	-0.125914	0.697368	0.135714	2.617222	-0.64
...	...	...	...	...	...	...	
59395	-0.038133	1.596408	0.741319	-1.230325	-1.769052	-0.521411	-1.33
59396	0.055600	1.704639	-0.062458	-0.569630	-1.180350	-0.521411	-0.64
59397	-0.100621	-0.965049	-0.379739	0.335579	0.103144	-0.521411	0.77
59398	-0.100621	-0.038596	-0.377623	0.697368	0.135714	-0.521411	0.77
59399	-0.100621	1.098547	-0.377623	-0.569630	0.234762	-0.521411	0.77

59400 rows × 36 columns

4. Export the data

In [25]:
```
data_processed.to_excel('df_data_processed.xlsx', index=False)
```