Predictive Maintenance for Tanzania's Water Infrastructure by Building a Model Classifier: A Machine Learning Approach"

PROJECT BY: Joy Ogutu

INTRODUCTION

Welcome to our presentation on Predicting Water Well Conditions in Tanzania. In this project, we deploy advanced machine learning techniques, specifically a well-tailored classifier, to gain insights into the state of water wells across the country. Our analysis is driven by a comprehensive dataset, encompassing various factors that influence the conditions of water wells.

The primary objective of our project is to develop a predictive model that accurately estimates the conditions of water wells in Tanzania. This model aims to provide invaluable insights to different stakeholders, including NGOs focused on well rehabilitation, the Tanzanian government, and organizations dedicated to improving water infrastructure.

Throughout the presentation, we will walk you through our project's methodology, key findings, and the model's performance in predicting water well conditions. Thank you for joining us on this journey to enhance water infrastructure management in Tanzania.

PROJECT OVERVIEW

Tanzania's Water Infrastructure Background.

Tanzania, an East African nation with a population exceeding 57 million, faces significant obstacles in ensuring consistent access to clean water. The country's diverse geography, ranging from coastal plains to mountainous regions, significantly influences the distribution and availability of water resources. In many Tanzanian communities, wells serve as essential water sources, addressing the daily needs of both urban and rural populations. However, this reliance on wells introduces a set of challenges. To comprehend this intricate challenges, stakeholders must understand the factors influencing well functionality, most of which align with the columns provided in our dataset:

1. Population Pressure: With a population of over 57 million, Tanzania is a developing nation where providing access to clean water is a major concern.
2. Existing Water Points: There are now water points around the nation, but a substantial portion of them are either broken or in need of maintenance, which causes problems with water scarcity.
3. Repair Difficulties: There are a lot of water wells that need to be maintained, and it can be difficult to determine which ones require urgent care. Extended durations of non-functionality may result from an absence of a methodical approach.
4. Resource Constraints: The government's capacity to fully solve problems with water infrastructure is hampered by a lack of funding and staff.

5. Geographic Diversity: Tanzania's varied topography makes problems with water infrastructure more complicated and calls for regionally-adaptable solutions.
6. Impact on Public Health: Since access to clean water is essential for preventing waterborne illnesses, inadequate water infrastructure directly affects public health.
7. Urbanization Pressures: As cities grow, there is a greater need for water infrastructure, necessitating careful planning to fulfill the needs of an expanding population in an environmentally responsible manner.
8. Data discrepancies: Developing targeted solutions is made more difficult by inconsistent data regarding the state of water points and their functionality.

The quality of water from these wells is a critical concern, given the risk of waterborne diseases. Long-term performance of water infrastructure, including wells, depends on maintenance; but, due to resource restrictions, routine repairs and upgrades are not possible. The Tanzanian government is aggressively tackling issues related to water by starting programs to upgrade water infrastructure,, in conjunction with non-governmental organizations (NGOs), but given the scope of the problem, creative solutions are needed. Data challenges, including gathering accurate and up-to-date information on well conditions, hinder effective planning. Addressing these challenges requires a comprehensive and innovative approach, leveraging data-driven insights to inform sustainable solutions. The goal is not only to improve reliable access to clean water but also enhance the overall well-being of Tanzanian communities.

Our goal is to build a robust classifier leveraging machine learning techniques. By analyzing various factors, such as pump types and installation dates, our predictive model aims to assist NGOs in pinpointing wells in need of repair and aid the government in

Problem Statement

Tanzania's water infrastructure faces critical challenges, particularly in the functionality of wells, leading to compromised access to clean water in many communities. The reliance on traditional wells, coupled with resource constraints and climate change, contributes to water scarcity and contamination risks. There is a critical need for data-driven insights, sustainable infrastructure development, and collaborative efforts among government entities, NGOs, and international partners. Such initiatives must prioritize the equitable distribution of resources and the empowerment of communities to ensure the long-term functionality of wells and, consequently, the well-being of Tanzanian populations.

Objectives

1. Objective: To explore the relationship between water quality indicators and the functionality status of wells. Analyze data on water quality, considering variables such as the kind of waterpoint type, source of the water, water quality, water quantity and the kind of extraction the waterpoint uses. Identify whether the construction year of the well and well permits contribute to well failures and use the findings to implement targeted water quality improvement initiatives.
2. Objective: To assess the Impact of numeric variables on well functionality. This objective involves a comprehensive examination of the distribution and influence of numeric variables on well functionality. Analyze factors such as total static head, population around the well, and well altitude. Evaluate how these numeric features are distributed across various well conditions, providing insights into their individual and

collective impact on well functionality. The objective aims to uncover patterns and relationships, enabling a better understanding of the numeric variables contributing to the status of water wells.

3. Objective: To create an advanced predictive maintenance model capable of identifying water wells requiring repair. Leveraging historical data encompassing pertinent features, a multifaceted approach involving various machine learning classifiers will be employed. The objective includes extensive testing and comparison of different models to determine the most accurate and reliable predictor for identifying wells in need of

Data Understanding

We will be using data from Taarifa and the Tanzanian Ministry of Water, to predict which pumps are functional, which need some repairs, and which don't work at all based on a number of variables about what kind of pump is operating, when it was installed, and how it is managed. The features in this dataset:

1. `amount_tsh` - Total static head (amount water available to waterpoint)
2. `date_recorded` - The date the row was entered
3. `funder` - Who funded the well
4. `gps_height` - Altitude of the well
5. `installer` - Organization that installed the well
6. `longitude` - GPS coordinate
7. `latitude` - GPS coordinate
8. `wpt_name` - Name of the waterpoint if there is one
9. `num_private` - No description
10. `basin` - Geographic water basin
11. `subvillage` - Geographic location
12. `region` - Geographic location
13. `region_code` - Geographic location (coded)
14. `district_code` - Geographic location (coded)
15. `lga` - Geographic location
16. `ward` - Geographic location
17. `population` - Population around the well
18. `public_meeting` - True/False
19. `recorded_by` - Group entering this row of data
20. `scheme_management` - Who operates the waterpoint
21. `scheme_name` - Who operates the waterpoint
22. `permit` - If the waterpoint is permitted
23. `construction_year` - Year the waterpoint was constructed
24. `extraction_type` - The kind of extraction the waterpoint uses
25. `extraction_type_group` - The kind of extraction the waterpoint uses
26. `extraction_type_class` - The kind of extraction the waterpoint uses
27. `management` - How the waterpoint is managed
28. `management_group` - How the waterpoint is managed
29. `payment` - What the water costs
30. `payment_type` - What the water costs
31. `water_quality` - The quality of the water
32. `quality_group` - The quality of the water
33. `quantity` - The quantity of water
34. `quantity_group` - The quantity of water

35. `source` - The source of the water
36. `source_type` - The source of the water
37. `source_class` - The source of the water
38. `waterpoint_type` - The kind of waterpoint
39. `waterpoint_type_group` - The kind of waterpoint
40. `id` - Unique identifier for a well

The target values include:

1. `functional` - the waterpoint is operational and there are no repairs needed
2. `functional needs repair` - the waterpoint is operational, but needs repairs
3. `non functional` - the waterpoint is not operational

Importing the necessary libraries

In [1]:
```
import pandas as pd
import numpy as np


from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import FunctionTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline

from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import SGDClassifier
from xgboost import XGBClassifier
from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, accuracy_score, confusio

import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

In [2]:
```
# A functon to load and merge the data
def load_and_examine(file_path_1, file_path_2):
    try:
        #Load the data
        X_train = pd.read_csv(file_path_1)
        y_train = pd.read_csv(file_path_2)

        df = pd.merge(X_train, y_train, on = "id" )

        #Display the shape, columns and first five rows of the dataset
        print("-------------------------Shape of the dataset---------------
        display(df.shape)
        print("-------------------------Columns of the dataset-------------
        display(df.columns)
        print("-------------------------Dataset head-----------------------
        display(df.head())

        #Display information about the dataset
        print("-------------------------Dataset information----------------
        display(df.info())

        return df

    except FileNotFoundError:
        print(f"File '{file_path_1}, {file_path_2}' not found.")
    except Exception as e:
        print(f"An error occurred: {e}")

file_path_1 = "train.csv"
file_path_2 = "target.csv"
df = load_and_examine(file_path_1, file_path_2)
```

```
-------------------------Shape of the dataset-------------------------

(59400, 41)

-------------------------Columns of the dataset-------------------------

Index(['id', 'amount_tsh', 'date_recorded', 'funder', 'gps_height',
       'installer', 'longitude', 'latitude', 'wpt_name', 'num_private',
       'basin', 'subvillage', 'region', 'region_code', 'district_code',
'lga',
       'ward', 'population', 'public_meeting', 'recorded_by',
       'scheme_management', 'scheme_name', 'permit', 'construction_year',
       'extraction_type', 'extraction_type_group', 'extraction_type_clas
s',
       'management', 'management_group', 'payment', 'payment_type',
       'water_quality', 'quality_group', 'quantity', 'quantity_group',
       'source', 'source_type', 'source_class', 'waterpoint_type',
       'waterpoint_type_group', 'status_group'],
      dtype='object')

-------------------------Dataset head-------------------------
```

	id	amount_tsh	date_recorded	funder	gps_height	installer	longitude	latitude
0	69572	6000.0	2011-03-14	Roman	1390	Roman	34.938093	-9.856322
1	8776	0.0	2013-03-06	Grumeti	1399	GRUMETI	34.698766	-2.147466
2	34310	25.0	2013-02-25	Lottery Club	686	World vision	37.460664	-3.821329
3	67743	0.0	2013-01-28	Unicef	263	UNICEF	38.486161	-11.155298
4	19728	0.0	2011-07-13	Action In A	0	Artisan	31.130847	-1.825359

5 rows × 41 columns

```
------------------------Dataset information------------------------
<class 'pandas.core.frame.DataFrame'>
Int64Index: 59400 entries, 0 to 59399
Data columns (total 41 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   id                     59400 non-null  int64  
 1   amount_tsh             59400 non-null  float64
 2   date_recorded          59400 non-null  object 
 3   funder                 55765 non-null  object 
 4   gps_height             59400 non-null  int64  
 5   installer              55745 non-null  object 
 6   longitude              59400 non-null  float64
 7   latitude               59400 non-null  float64
 8   wpt_name               59400 non-null  object 
 9   num_private            59400 non-null  int64  
 10  basin                  59400 non-null  object 
 11  subvillage             59029 non-null  object 
 12  region                 59400 non-null  object 
 13  region_code            59400 non-null  int64  
 14  district_code          59400 non-null  int64  
 15  lga                    59400 non-null  object 
 16  ward                   59400 non-null  object 
 17  population             59400 non-null  int64  
 18  public_meeting         56066 non-null  object 
 19  recorded_by            59400 non-null  object 
 20  scheme_management      55523 non-null  object 
 21  scheme_name            31234 non-null  object 
 22  permit                 56344 non-null  object 
 23  construction_year      59400 non-null  int64  
 24  extraction_type        59400 non-null  object 
 25  extraction_type_group  59400 non-null  object 
 26  extraction_type_class  59400 non-null  object 
 27  management             59400 non-null  object 
 28  management_group       59400 non-null  object 
 29  payment                59400 non-null  object 
 30  payment_type           59400 non-null  object 
 31  water_quality          59400 non-null  object 
 32  quality_group          59400 non-null  object 
 33  quantity               59400 non-null  object 
 34  quantity_group         59400 non-null  object 
 35  source                 59400 non-null  object 
 36  source_type            59400 non-null  object 
 37  source_class           59400 non-null  object 
 38  waterpoint_type        59400 non-null  object 
 39  waterpoint_type_group  59400 non-null  object 
 40  status_group           59400 non-null  object 
dtypes: float64(3), int64(7), object(31)
memory usage: 19.0+ MB
```

```
None
```

The dataset includes 59400 observations and 41 columns.

The `status_group` column shows the label or target for each pump, the other 40 columns are features, 10 of which are numerical, the rest are categorical.

DATA PREPARATION

Duplicate Values.

There are no duplicates in the dataset.

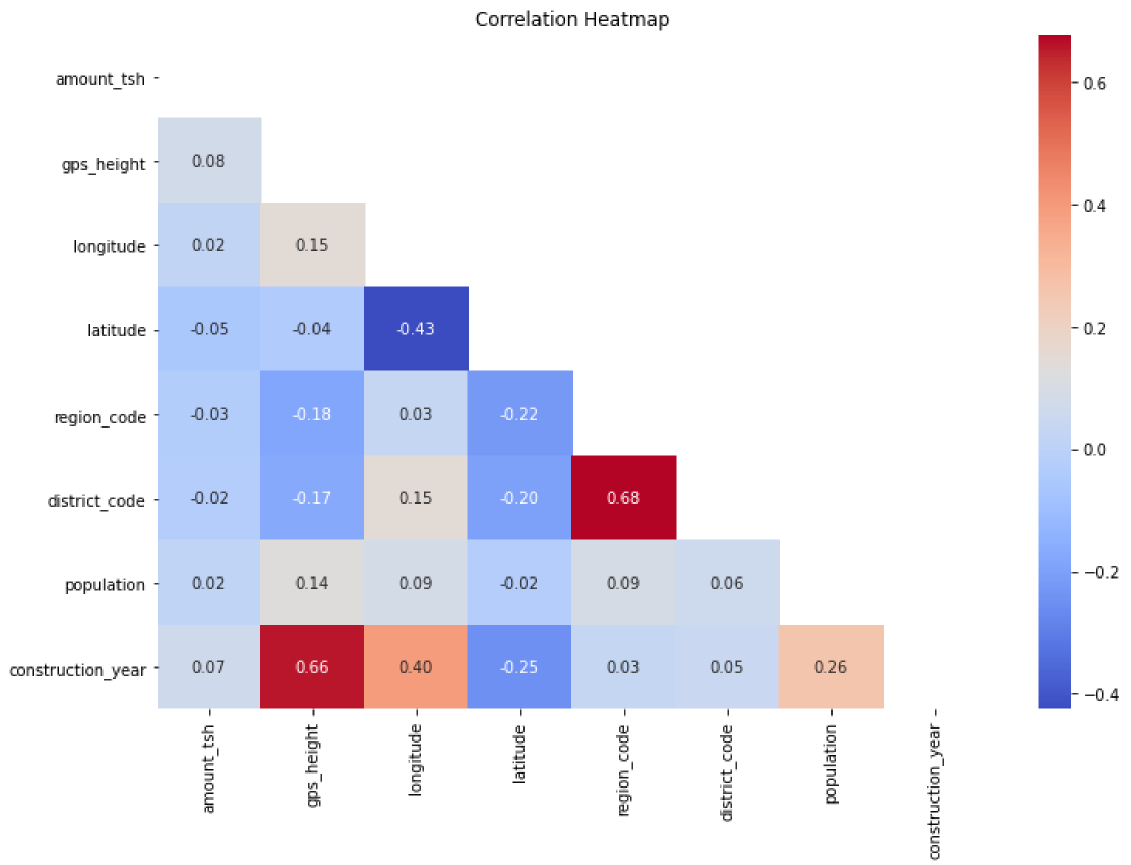
In [3]:
```
print("Duplicated values in the train set: {}".format(df.duplicated(subset
```

```
Duplicated values in the train set: 0
```

Dropping of similar, highly correlated and irrelevant features.

In this step we analyze the features to figure out which columns to drop.

In [4]:
```
numeric_columns = df[["amount_tsh", "gps_height", "longitude", "latitude",

# Selecting only numeric columns
# numeric_columns = df.select_dtypes(include = 'number')

# Creating a heatmap
correlation_matrix = numeric_columns.corr()
# Create a mask to hide the upper triangle
mask = np.triu(np.ones_like(correlation_matrix, dtype = bool))
plt.figure(figsize = (12, 8))
sns.heatmap(correlation_matrix, annot = True, cmap = "coolwarm", fmt = ".2f
plt.title('Correlation Heatmap')
plt.show()
```



In [5]:
```
# Get number of unique entries in each column with categorical data
cat_cols = df.select_dtypes(include = "object").columns
object_nunique = list(map(lambda col: df[col].nunique(), cat_cols))
d = dict(zip(cat_cols, object_nunique))

# Print number of unique entries by column, in ascending order
sorted(d.items(), key = lambda x: x[1])
```

Out[5]:
```
[('recorded_by', 1),
 ('public_meeting', 2),
 ('permit', 2),
 ('source_class', 3),
 ('status_group', 3),
 ('management_group', 5),
 ('quantity', 5),
 ('quantity_group', 5),
 ('quality_group', 6),
 ('waterpoint_type_group', 6),
 ('extraction_type_class', 7),
 ('payment', 7),
 ('payment_type', 7),
 ('source_type', 7),
 ('waterpoint_type', 7),
 ('water_quality', 8),
 ('basin', 9),
 ('source', 10),
 ('scheme_management', 12),
 ('management', 12),
 ('extraction_type_group', 13),
 ('extraction_type', 18),
 ('region', 21),
 ('lga', 125),
 ('date_recorded', 356),
 ('funder', 1897),
 ('ward', 2092),
 ('installer', 2145),
 ('scheme_name', 2696),
 ('subvillage', 19287),
 ('wpt_name', 37400)]
```

Based on he above analysis, the following groups of features contain very similar information, so the correlation between them is high. This way we are risking overfitting the training data by including all the features in our analysis:

1. (`extraction_type` , `extraction_type_group` , `extraction_type_class`)
2. (`payment` , `payment_type`)
3. (`water_quality` , `quality_group`)
4. (`source` , `source_class`)
5. (`subvillage` , `region` , `region_code` , `district_code` , `lga` , `ward`)
6. (`waterpoint_type` , `waterpoint_type_group`)
7. (`scheme_name` , `scheme_management`)

In the `wpt_name` feature, there are 37,400 unique values out of 59,400 observations which is not very informative hence we will drop it.

The `recorded_by` feature can be dropped as there is only 1 unique value, it doesn't help in predicting.

The correlation between `construction_year` and `gps_height` is high, but these two variables don't have any obvious connection, so we will explore this correlation further to take a decision.

As we saw earlier, there exists quite a strong correlation between `district_code` and `region_code` , so we will drop one of these variables. The negative correlation to the target variable of the `region_code` is higher than that of the `district_code` . Keep the variable with higher correlation to the target.

The cardinality is too high for the following columns: `funder` , `installer` and `subvillage` therefore we will drop them. The rest can be one-hot encoded as the cardinality is lower than 10.

In [6]:

```
cols_to_drop = ["id", "funder", "installer", "wpt_name", "num_private", "su
df = df.drop(cols_to_drop, axis = 1)
df.shape
```

Out[6]:

```
(59400, 19)
```

Null Values

All columns apart from `permit` have no null values

In [7]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 59400 entries, 0 to 59399
Data columns (total 19 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   amount_tsh             59400 non-null  float64
 1   date_recorded          59400 non-null  object 
 2   gps_height             59400 non-null  int64  
 3   longitude              59400 non-null  float64
 4   latitude               59400 non-null  float64
 5   basin                  59400 non-null  object 
 6   region_code            59400 non-null  int64  
 7   population             59400 non-null  int64  
 8   permit                 56344 non-null  object 
 9   construction_year      59400 non-null  int64  
 10  extraction_type_group  59400 non-null  object 
 11  management_group       59400 non-null  object 
 12  payment_type           59400 non-null  object 
 13  quality_group          59400 non-null  object 
 14  quantity_group         59400 non-null  object 
 15  source_type            59400 non-null  object 
 16  source_class           59400 non-null  object 
 17  waterpoint_type_group  59400 non-null  object 
 18  status_group           59400 non-null  object 
dtypes: float64(3), int64(4), object(12)
memory usage: 9.1+ MB
```

In [8]:
```
#Checking for null value counts and their percentages
columns_with_missing_values = ["permit"]
missing_values_table = pd.DataFrame([
 {
 'Column': column,
 'Missing Count': df[column].isnull().sum(),
 'Missing Percentage': (df[column].isnull().sum() / len(df[column])) * 100
 }
 for column in columns_with_missing_values])
print(missing_values_table)
```

```
   Column  Missing Count  Missing Percentage
0  permit           3056            5.144781
```

In [9]:
```
df["permit"].fillna("unknown", inplace = True)
```

Datatype Conversion

The `date_recorded` datatype was converted from object to datetime.

In [10]:
```
df["date_recorded"] = pd.to_datetime(df["date_recorded"])
df["date_recorded"].dtype
```

Out[10]:
```
dtype('<M8[ns]')
```

Feature Engineering

A new feature, `well_age` , is engineered by calculating the difference between the `date_recorded` and `construction_year` columns. However, due to the presence of 0 values in the `construction_year` column, the calculation may yield inaccurate results. To address this issue, these 0 values have been converted to NaN to ensure proper subtraction. Consequently, this transformation has introduced 20,709 null values in both the `construction_year` and `well_year` columns. These null values will be removed during subsequent data preprocessing steps.

In [11]:
```
# Replace 0 values in 'construction_year' with NaN
df["construction_year"].replace(0, np.nan, inplace = True)

# Calculate the age of the well
df["well_age"] = df["date_recorded"].dt.year - df["construction_year"]

# Display the DataFrame with the calculated well age and dropping the `date
df = df.drop("date_recorded", axis = 1)
df.head()
df.isnull().sum()
```

Out[11]:
```
amount_tsh                 0
gps_height                 0
longitude                  0
latitude                   0
basin                      0
region_code                0
population                 0
permit                     0
construction_year      20709
extraction_type_group      0
management_group           0
payment_type               0
quality_group              0
quantity_group             0
source_type                0
source_class               0
waterpoint_type_group      0
status_group               0
well_age               20709
dtype: int64
```

In [12]:
```
negative_values = df[df["well_age"] < 0]
negative_values
```

Out[12]:

	amount_tsh	gps_height	longitude	latitude	basin	region_code	population
8729	0.0	86	38.959776	-5.247278	Pangani	4	120
10441	20.0	307	38.768656	-7.298419	Rufiji	60	1
13366	100.0	1331	34.290885	-1.699609	Lake Victoria	20	80
23373	50.0	239	39.272736	-11.019000	Ruvuma / Southern Coast	90	317
27501	500.0	1611	34.900561	-8.873813	Rufiji	11	65
32619	0.0	1856	31.539761	-7.983106	Lake Tanganyika	15	900
33942	0.0	-27	39.283105	-7.422852	Rufiji	6	200
39559	0.0	301	38.558421	-5.140405	Pangani	4	713
48555	0.0	284	38.929212	-7.111349	Wami / Ruvu	60	185

It is observed that there are instances in the dataset where the `date_recorded` (the date the record was entered) is in the year 2004, but the `construction_year` (the year the well was constructed) is after 2004. It may indicate potential data quality issues or inconsistencies in the dataset. This situation could be due to various reasons, and it's essential to investigate further to understand the possible explanations. Here are a few considerations:

1. Data Entry Errors: Human errors during data entry might lead to inconsistencies. It's possible that the year recorded when the data was entered (2004) could be a placeholder, an incorrect entry, or an anomaly.
2. Missing or Unknown Construction Year: It's also possible that the actual construction year is unknown or missing for some wells, and the year 2004 was used as a default or placeholder value. This might be done when the construction year is not available at the time of data recording.
3. Data Collection Process: The data collection process might have involved recording information at different times or through different methods. Inconsistent practices during data collection can result in such discrepancies.

To solve this we only choose to remain with values greater than or equal to 0 in the `well_year` column.

In [13]:
```
df = df[df["well_age"] >= 0]
```

In [14]:
```
def construction_wrangler(row):

    if row["construction_year"] >= 1960 and row["construction_year"] < 1970
        return '60s'
    elif row["construction_year"] >= 1970 and row["construction_year"] < 19
        return '70s'
    elif row["construction_year"] >= 1980 and row["construction_year"] < 19
        return '80s'
    elif row["construction_year"] >= 1990 and row["construction_year"] < 20
        return '90s'
    elif row["construction_year"] >= 2000 and row["construction_year"] < 20
        return '00s'
    elif row["construction_year"] >= 2010:
        return '10s'
    else:
        return "unknown"

df["construction_year"] = df.apply(lambda row: construction_wrangler(row),

# Verify the value counts, including 'unknown'
df.construction_year.value_counts()
```

Out[14]:
```
00s    15322
90s     7678
80s     5578
10s     5160
70s     4406
60s      538
Name: construction_year, dtype: int64
```

In [15]:
```
df.isnull().sum()
```

Out[15]:
```
amount_tsh               0
gps_height               0
longitude                0
latitude                 0
basin                    0
region_code              0
population               0
permit                   0
construction_year        0
extraction_type_group    0
management_group         0
payment_type             0
quality_group            0
quantity_group           0
source_type              0
source_class             0
waterpoint_type_group    0
status_group             0
well_age                 0
dtype: int64
```

Outliers

Our analysis will consider these extreme values as legitimate components of the data, ensuring a comprehensive and contextually appropriate exploration of the dataset.

In [16]:
```
# Create a function to check outliers
def check_outliers(data, columns):
 for column in columns:
    # Calculate IQR (Interquartile Range)
    iqr = data[column].quantile(0.75) - data[column].quantile(0.25)

    # Define lower and upper thresholds
    lower_threshold = data[column].quantile(0.25) - 1.5 * iqr
    upper_threshold = data[column].quantile(0.75) + 1.5 * iqr

    # Find outliers
    outliers = data[(data[column] < lower_threshold) | (data[column] > uppe

    # Print the count of outliers
    print(f"{column}\nNumber of outliers: {len(outliers)}\n")

columns_to_check = df.select_dtypes(include = ["number"])
check_outliers(df, columns_to_check)
```

```
amount_tsh
Number of outliers: 4500

gps_height
Number of outliers: 0

longitude
Number of outliers: 1013

latitude
Number of outliers: 0

region_code
Number of outliers: 3391

population
Number of outliers: 2642

well_age
Number of outliers: 0
```

Categorization

A practical and strategic consideration led to the decision to combine the objective variable into two categories: `functional` and `non-functional` . We choose a more direct and useful categorization by combining the `functional needs repair` category into `non-functional` . The main goal in real-world applications is to locate wells that are not performing at their best, whether they are completely non-functional or require repair. The consolidation of these statuses into a single `non-functional` category facilitates the prediction model's attention to wells that need care and guarantees a more pragmatic approach for stakeholders seeking to prioritize and handle maintenance activities. In the context of Tanzania's water infrastructure management, this consolidation makes it easier to make decisions and provide clearer insights for interventions and resource allocation.

In [17]:
```
# Replacing one value to the other
df["status_group"].replace("functional needs repair", "non functional", inp
df.status_group.value_counts()
```

Out[17]:
```
functional        21700
non functional    16982
Name: status_group, dtype: int64
```

EXPLORATORY DATA ANALYSIS

We used univariate, bivariate, and multivariate exploratory data analysis (EDA) methodologies in a comprehensive manner to properly analyze the data. Let's dive into our analysis.

In [18]:
```
# Getting the statistic summary of the columns
df.describe()
```

Out[18]:

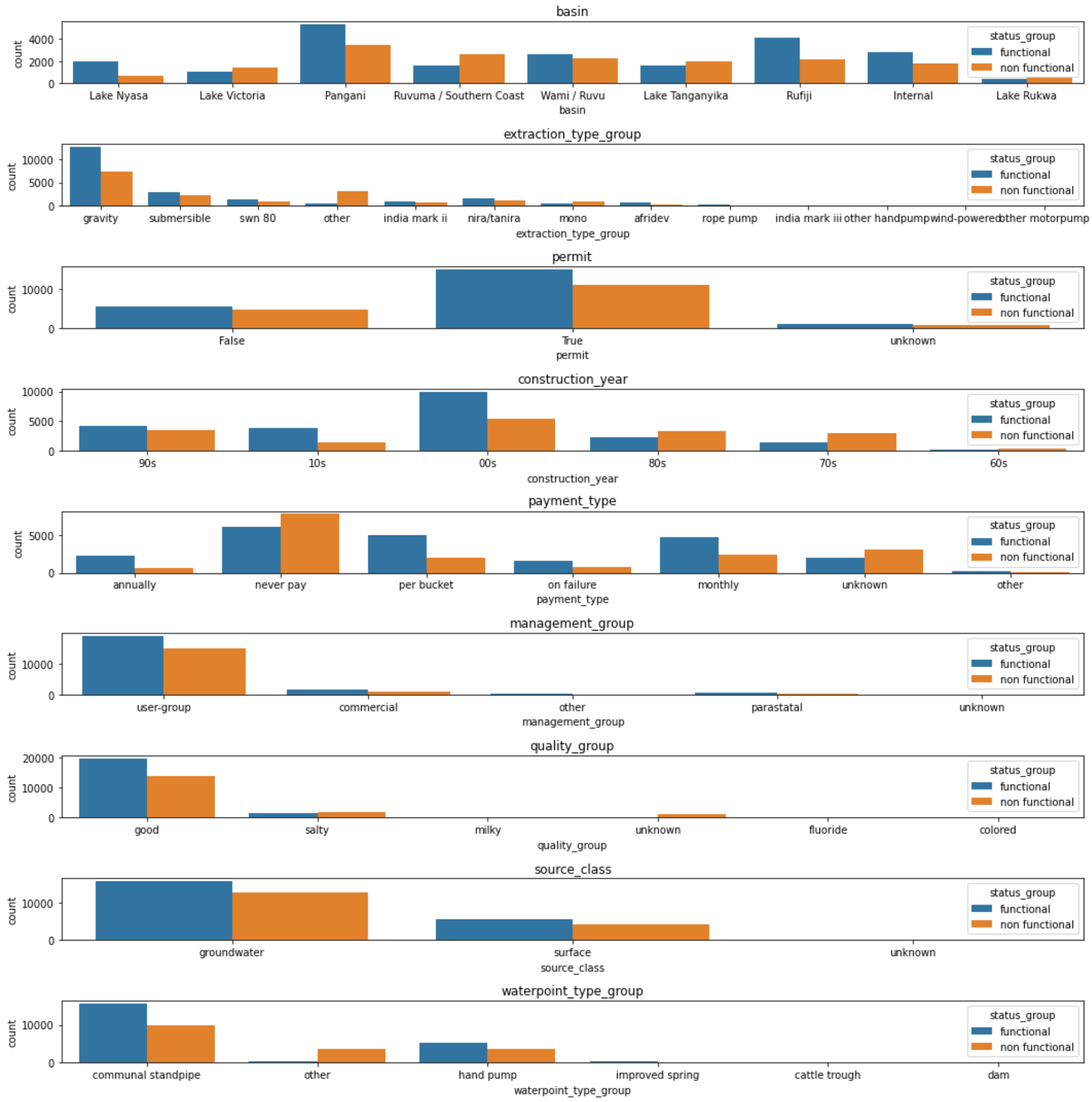
	amount_tsh	gps_height	longitude	latitude	region_code	populatio
count	38682.000000	38682.000000	38682.000000	38682.000000	38682.000000	38682.00000
mean	466.548742	1002.446177	35.982987	-6.235225	15.703169	269.79566
std	3541.442129	618.042221	2.558615	2.761363	20.999385	552.38973
min	0.000000	-63.000000	29.607122	-11.649440	2.000000	0.00000
25%	0.000000	372.000000	34.676719	-8.755301	4.000000	30.00000
50%	0.000000	1154.000000	36.648017	-6.064107	11.000000	150.00000
75%	200.000000	1488.000000	37.802381	-3.650582	16.000000	305.00000
max	350000.000000	2770.000000	40.345193	-1.042375	99.000000	30500.00000

Objective: To explore the relationship between water quality indicators and the functionality status of wells.

Count Plots Summary

In [19]:

```
# Creating count plots for selected categorical columns

# Identify categorical columns
categorical_columns = df[["basin", "extraction_type_group","permit","constr

# Create a figure with a grid of subplots
fig, axes = plt.subplots(len(categorical_columns), 1, figsize=(15, 15))

# Iterate over categorical columns and create countplots
for i, column in enumerate(categorical_columns):
 sns.countplot(data = df, x = column, hue = "status_group", ax = axes[i])
 axes[i].tick_params(axis = "x", rotation = 0)
 axes[i].set_title(column)


# Show the plot
plt.tight_layout()
plt.show()
plt.savefig("Categorical Count Plots")
```




```
<Figure size 432x288 with 0 Axes>
```

Summary of Count Plot Analysis:

- `extraction_type_group` : The "gravity" extraction type has the most significant impact, with a higher proportion of functional wells compared to non-functional ones, while other extraction types show a higher likelihood of non-functionality.
- `permit` : Wells with permits (True) have a higher proportion of functional status compared to non-functional ones. ells with permits are more likely to be functional, indicating that having the necessary permits correlates with better well functionality.
- `construction_year` : Wells constructed in the 2000s show the highest impact, with a higher proportion of functional wells. This is followed by those constructed in the 1990s and 2010s. Wells from earlier decades show a lower likelihood of functionality.
- `payment_type` : Wells with the "never pay" payment type have a higher proportion of non-functional status. Conversely, wells with some form of payment show a higher proportion of functional status, indicating that payment might contribute to well functionality. Wells where users never pay for water are more likely to be non-functional. On the other hand, wells with payment arrangements show a higher likelihood of functionality, suggesting that payment contributes to well maintenance.
- `management_group` :The "user-group" value in the management group has the highest impact, showing a higher proportion of functional wells compared to non-

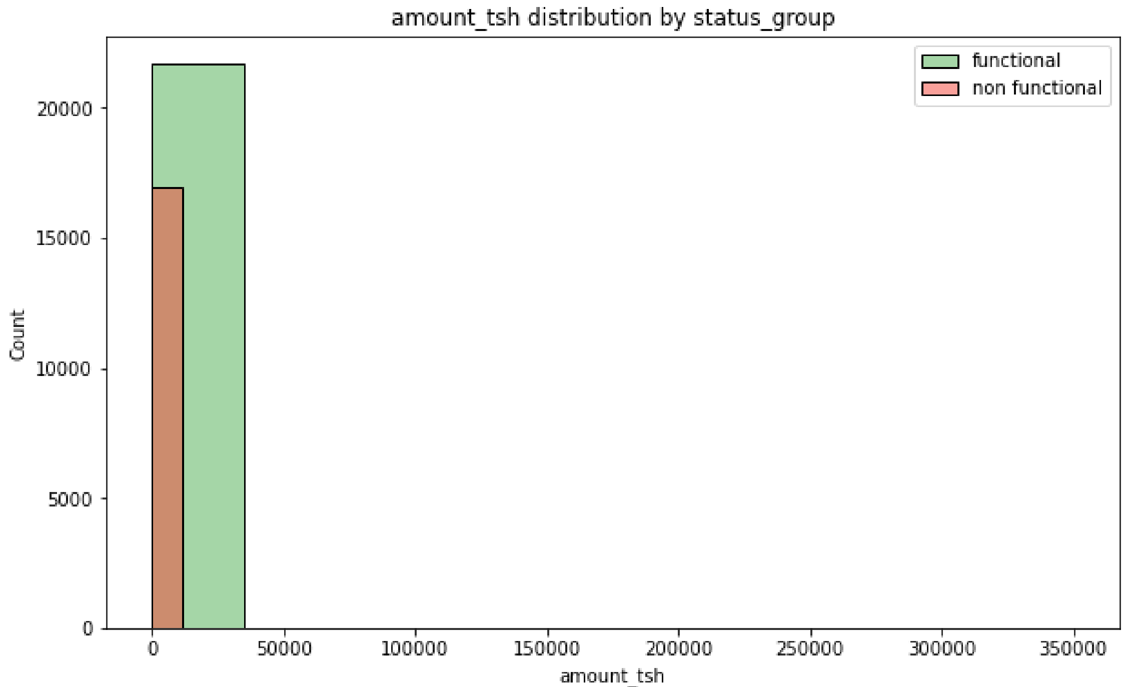
functional ones emphasizing the impact of effective user-group management on well functionality.
- `quality_group` : Wells classified as "good" in the quality group exhibit the most impact, with a higher proportion of functional status compared to non-functional status highlighting the importance of water quality in well functionality.
- `source_class` : Wells classified as "groundwater" in the source class have the most impact, showing a higher proportion of functional wells compared to non-functional ones. Wells relying on "groundwater" as their source exhibit a higher likelihood of functionality, emphasizing the significance of groundwater sources for well functionality.
- `waterpoint_type` : The "communal standpipe" waterpoint type has the most impact, with a higher proportion of functional wells compared to non-functional ones, followed by the "hand pump" type. ommunal standpipes" and "hand pumps" show the highest functionality, suggesting that these types are more reliable water sources compared to

Objective: To assess the Impact of numeric variables on well functionality.

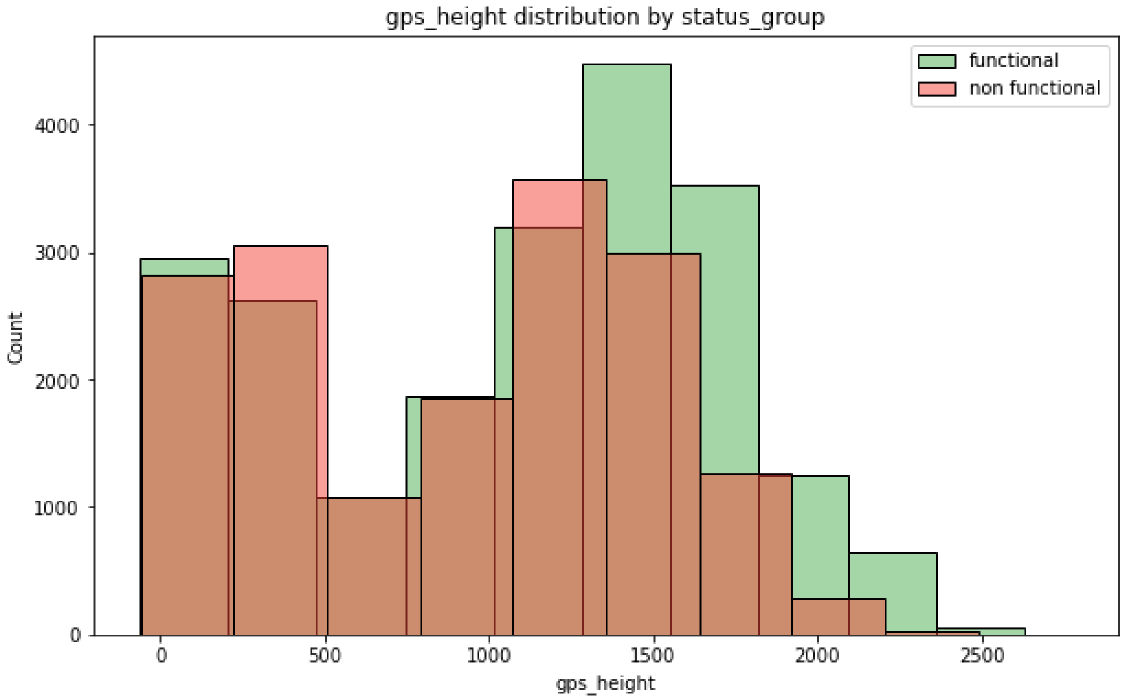
Histogram Summary

The histograms below provide concise representation of how data is spread across different values and helps reveal underlying patterns.

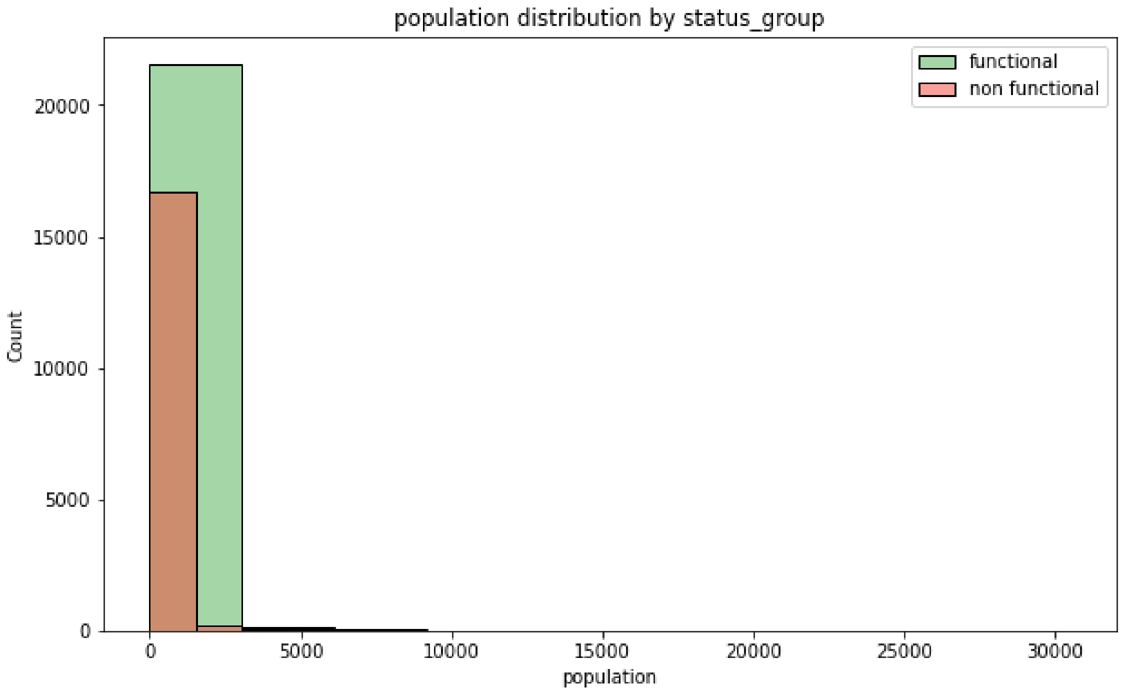
In [20]:
```
# Creating histograms for selected columns

# Identify numerical columns
numeric_columns = df[["amount_tsh", "gps_height","population", "well_age"]]
palette = {"functional": "#4CAF50", "non functional": "#F44336"}

# Iterate over numerical columns and create histograms
for column in numeric_columns:
    plt.figure(figsize = (10, 6))
    for status_group in df["status_group"].unique():
        # sns.histplot(data = df, x = column, hue = 'status_group', multipl
        sns.histplot(df[df["status_group"] == status_group][column], label

    plt.title(f"{column} distribution by status_group")
    plt.xlabel(column)
    plt.legend()
    plt.show()
    plt.savefig("Histogram Plots")
```



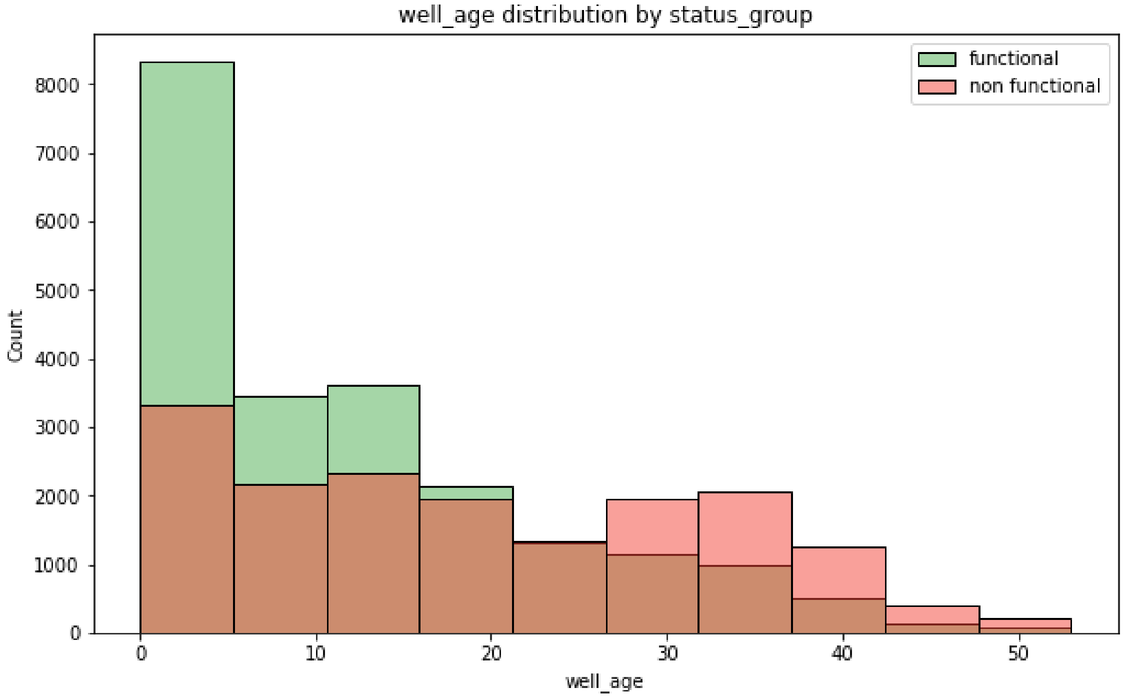

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```




```
&lt;Figure size 432x288 with 0 Axes&gt;
```

The histograms for the mentioned columns – `amount_tsh` , `gps_height` , `population` , and `well_age` – reveal a right-skewed distribution. In a right-skewed distribution, also known as positively skewed, the majority of data points are clustered on the left side, and the tail extends towards the right. Here's a more detailed interpretation for each column:

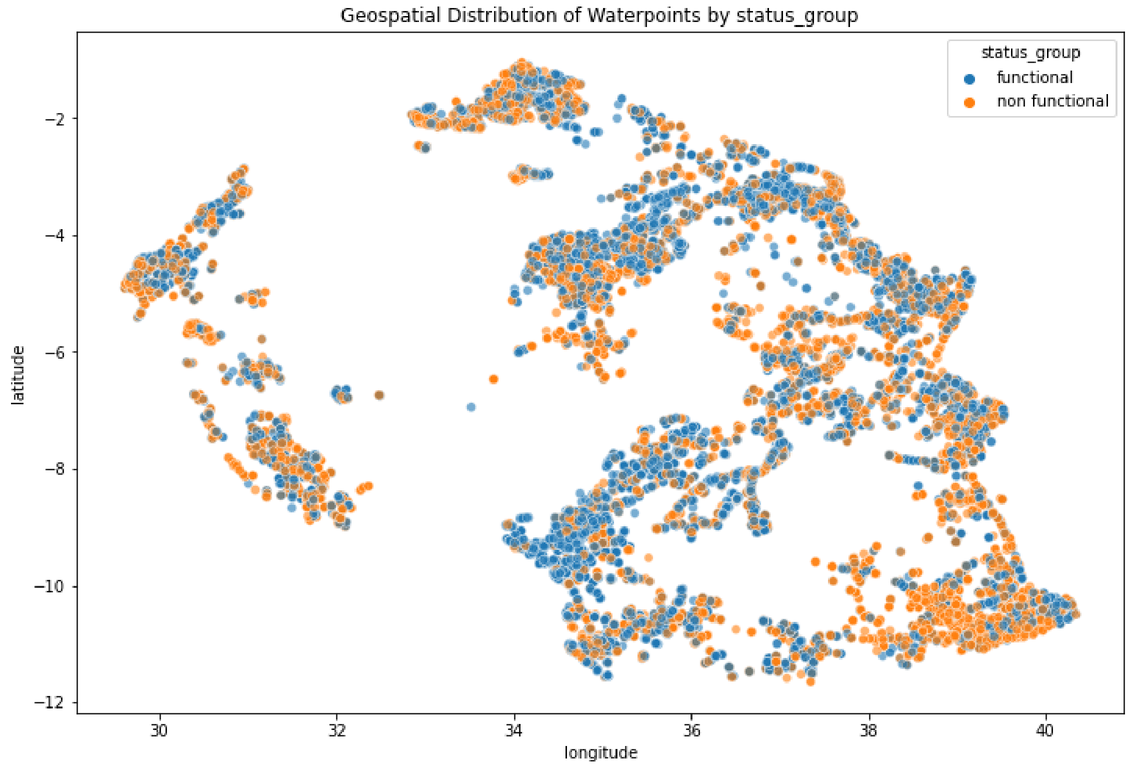
- `amount_tsh` : The majority of water points have a low "amount_tsh," suggesting that a significant number of wells have a low amount of total static head. There are fewer wells with higher values, indicating that most wells in the dataset have a relatively low total static head.
- `gps_height` : The distribution of "gps_height" indicates that many wells are situated at lower elevations, with fewer wells at higher elevations. This could reflect the topography of the region, with more wells located in lower-lying areas.
- `population` : The "population" histogram shows that most wells serve smaller populations, with a concentration of wells serving fewer people. There are fewer wells serving larger populations, contributing to the rightward tail of the distribution.
- `well_age` : The distribution of "well_age" suggests that many wells in the dataset are relatively new, with a higher concentration of recently constructed wells. The rightward tail indicates a smaller number of older wells.

Geospatial Distribution Summary

The geospatial distribution analysis provides valuable insights into the spatial characteristics of the water points in the dataset. Here's a summary of the geospatial distribution:

- `latitude and longitude` : The scatter plot of latitude and longitude reveals the geographic spread of water points across Tanzania. Clusters of points may indicate regions with a higher density of wells, while sparser regions may suggest areas with fewer water points.

In [21]:
```
plt.figure(figsize = (12, 8))
sns.scatterplot(data = df, x = "longitude", y = "latitude", hue = "status_g
plt.title("Geospatial Distribution of Waterpoints by status_group")
plt.show()
plt.savefig("Geospatial Distribution of Waterpoints")
```




```
<Figure size 432x288 with 0 Axes>
```

Objective: To create an advanced predictive maintenance model capable of identifying water wells requiring repair.

PREPROCESSING

Ordinal encoding of categorical data

The code below aims to transform the `status_group` column into a binary format, where "functional" is represented as 1 and "non functional" as 0 as it simplifies the target variable into a binary classification. This binary representation aligns with the typical approach in classification problems, making it easier to interpret and apply algorithms that rely on numerical target values. The resulting value counts, when normalized, provide the proportion of functional and non-functional wells in the dataset, offering a clear understanding of the distribution of the target variable whch is reasonably balanced for classification purposes.

In [22]:
```
# Create a mapping dictionary
status_mapping = {"functional": 1, "non functional": 0}

# Replace values in the 'status_group' column
df["status_group"] = df["status_group"].map(status_mapping)
df.status_group.value_counts(normalize = True)
```

Out[22]:
```
1    0.560984
0    0.439016
Name: status_group, dtype: float64
```

The code cells below involves the creation of ordinal encoding for selected categorical columns in the dataset. Label encoding can be used for the provided ordinal encoding task, but it has limitations, as it assigns integer values to categories based on their alphabetical order, which might not capture the inherent ordinal relationships in the data.

The custom ordinal encoding allows for more flexibility in assigning codes based on the domain knowledge or specific requirements of the problem. Ordinal encoding is chosen for categorical variables like `quality_group` , `quantity_group` , `payment_type` , and `permit` to represent their inherent order or ranking. For example, in the `quality_group` , the categories "good" are assigned a higher code (3) than "salty", "milky", "flouride" and "colored" which share a lower code (2), while "unknown" has the lowest code (1).

This encoding reflects a logical hierarchy based on the perceived impact on water quality. Similarly, the other categorical columns are encoded with numeric values to capture their ordinal relationships.

In [23]:
```
df.quality_group.value_counts()
```

Out[23]:
```
good        33493
salty        3465
unknown      1188
colored       224
fluoride      176
milky         136
Name: quality_group, dtype: int64
```

In [24]:
```
# Custom Label Encoding
ordered_quality = {"good" : 3 ,"salty" : 2, "milky" : 2, "colored" : 2, "fl
df["quality_group_code"] = [ordered_quality[item] for item in df.quality_gr
del df["quality_group"]
```

In [25]:
```
df.quantity_group.value_counts()
```

Out[25]:
```
enough          22329
insufficient    10463
dry              3452
seasonal         1923
unknown           515
Name: quantity_group, dtype: int64
```

In [26]:
```
# Custom Label Encoding
ordered_quantity = {"enough" : 3, "insufficient" : 2, "dry" : 2, "seasonal"
df["quantity_group_code"] = [ordered_quantity[item] for item in df.quantity
del df["quantity_group"]
```

In [27]:
```
df.payment_type.value_counts()
```

Out[27]:
```
never pay      13827
monthly         7116
per bucket      7024
unknown         5086
annually        2885
on failure      2360
other            384
Name: payment_type, dtype: int64
```

In [28]:
```
# Custom Label Encoding
ordered_payment = {"monthly" : 4, "annually" : 4, "on failure" : 3, "per bu
df["payment_code"] = [ordered_payment[item] for item in df.payment_type]
del df["payment_type"]
```

In [29]:
```
df.permit.value_counts()
```

Out[29]:
```
True       26369
False      10386
unknown     1927
Name: permit, dtype: int64
```

In [30]:
```
# Custom Label Encoding
ordered_permit = {True : 2, False : 1, "unknown" : 0}
df["permit_code"] = [ordered_permit[item] for item in df.permit]
del df["permit"]
```

In [31]:
```
df["amount_tsh"].value_counts()
```

Out[31]:
```
0.0         21332
500.0        3091
50.0         2431
20.0         1437
1000.0       1399
            ...
8500.0          1
6300.0          1
220.0           1
138000.0        1
12.0            1
Name: amount_tsh, Length: 95, dtype: int64
```

Feature Engineering

In [32]:
```
# Define a custom function for transformation
def transform_amount_tsh(df):
    df_transformed = df.copy()
    df_transformed.loc[df_transformed["amount_tsh"] < 30, "amount_tsh"] = 0
    df_transformed.loc[df_transformed["amount_tsh"] >= 30, "amount_tsh"] =
    return df_transformed[["amount_tsh"]]


# Create a FunctionTransformer
amount_tsh_transformer = FunctionTransformer(transform_amount_tsh, validate

# Apply the transformation using the transformer
df_transformed = amount_tsh_transformer.fit_transform(df[["amount_tsh"]])

# Replace the original column with the transformed values
df["amount_tsh"] = df_transformed

# Display the transformed DataFrame
df.head()
```

Out[32]:

	amount_tsh	gps_height	longitude	latitude	basin	region_code	population	constr
0	1.0	1390	34.938093	-9.856322	Lake Nyasa	11	109	
1	0.0	1399	34.698766	-2.147466	Lake Victoria	20	280	
2	0.0	686	37.460664	-3.821329	Pangani	21	250	
3	0.0	263	38.486161	-11.155298	Ruvuma / Southern Coast	90	58	
5	0.0	0	39.172796	-4.765587	Pangani	4	1	

One-Hot Encoding of categorical features

In [33]:

```
# Create a list of columns to be encoded
cat_cols = ["basin", "construction_year", "extraction_type_group",
            "management_group", "source_type", "source_class",
            "waterpoint_type_group"]

# Create a subset of your DataFrame with only the categorical columns
cat_data = df[cat_cols]

# Make a transformer
ohe = OneHotEncoder(categories = "auto", handle_unknown = "ignore", sparse

# Create transformed DataFrame
cat_encoded = ohe.fit_transform(cat_data)
cat_encoded_df = pd.DataFrame(
    cat_encoded,
    columns=ohe.get_feature_names_out(cat_cols),
    index=df.index
)

# Drop the original categorical columns and concatenate the encoded ones
df.drop(cat_cols, axis = 1, inplace = True)
df = pd.concat([df, cat_encoded_df], axis = 1)

# Visually inspect DataFrame
df.head()
```

```
c:\Users\HP\anaconda3\envs\learn-env\lib\site-packages\sklearn\preprocess
ing\_encoders.py:975: FutureWarning: `sparse` was renamed to `sparse_outp
ut` in version 1.2 and will be removed in 1.4. `sparse_output` is ignored
unless you leave `sparse` to its default value.
  warnings.warn(
```

Out[33]:

	amount_tsh	gps_height	longitude	latitude	region_code	population	status_group	w
0	1.0	1390	34.938093	-9.856322	11	109	1	
1	0.0	1399	34.698766	-2.147466	20	280	1	
2	0.0	686	37.460664	-3.821329	21	250	1	
3	0.0	263	38.486161	-11.155298	90	58	0	
5	0.0	0	39.172796	-4.765587	4	1	1	

5 rows × 61 columns

In [34]:
```
# Ensuring the columns are of numeric datatypes before modeling
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 38682 entries, 0 to 59399
Data columns (total 61 columns):
 #   Column                                  Non-Null Count  Dtype  
---  ------                                  --------------  -----  
 0   amount_tsh                              38682 non-null  float64
 1   gps_height                              38682 non-null  int64  
 2   longitude                               38682 non-null  float64
 3   latitude                                38682 non-null  float64
 4   region_code                             38682 non-null  int64  
 5   population                              38682 non-null  int64  
 6   status_group                            38682 non-null  int64  
 7   well_age                                38682 non-null  float64
 8   quality_group_code                      38682 non-null  int64  
 9   quantity_group_code                     38682 non-null  int64  
 10  payment_code                            38682 non-null  int64  
 11  permit_code                             38682 non-null  int64  
 12  basin_Internal                          38682 non-null  float64
 13  basin_Lake Nyasa                        38682 non-null  float64
 14  basin_Lake Rukwa                        38682 non-null  float64
 15  basin_Lake Tanganyika                   38682 non-null  float64
 16  basin_Lake Victoria                     38682 non-null  float64
 17  basin_Pangani                           38682 non-null  float64
 18  basin_Rufiji                            38682 non-null  float64
 19  basin_Ruvuma / Southern Coast           38682 non-null  float64
 20  basin_Wami / Ruvu                       38682 non-null  float64
 21  construction_year_00s                   38682 non-null  float64
 22  construction_year_10s                   38682 non-null  float64
 23  construction_year_60s                   38682 non-null  float64
 24  construction_year_70s                   38682 non-null  float64
 25  construction_year_80s                   38682 non-null  float64
 26  construction_year_90s                   38682 non-null  float64
 27  extraction_type_group_afridev           38682 non-null  float64
 28  extraction_type_group_gravity           38682 non-null  float64
 29  extraction_type_group_india mark ii     38682 non-null  float64
 30  extraction_type_group_india mark iii    38682 non-null  float64
 31  extraction_type_group_mono              38682 non-null  float64
 32  extraction_type_group_nira/tanira       38682 non-null  float64
 33  extraction_type_group_other             38682 non-null  float64
 34  extraction_type_group_other handpump    38682 non-null  float64
 35  extraction_type_group_other motorpump   38682 non-null  float64
 36  extraction_type_group_rope pump         38682 non-null  float64
 37  extraction_type_group_submersible       38682 non-null  float64
 38  extraction_type_group_swn 80            38682 non-null  float64
 39  extraction_type_group_wind-powered      38682 non-null  float64
 40  management_group_commercial             38682 non-null  float64
 41  management_group_other                  38682 non-null  float64
 42  management_group_parastatal             38682 non-null  float64
 43  management_group_unknown                38682 non-null  float64
 44  management_group_user-group             38682 non-null  float64
 45  source_type_borehole                    38682 non-null  float64
 46  source_type_dam                         38682 non-null  float64
 47  source_type_other                       38682 non-null  float64
 48  source_type_rainwater harvesting        38682 non-null  float64
 49  source_type_river/lake                  38682 non-null  float64
 50  source_type_shallow well                38682 non-null  float64
 51  source_type_spring                      38682 non-null  float64
 52  source_class_groundwater                38682 non-null  float64
 53  source_class_surface                    38682 non-null  float64
 54  source_class_unknown                    38682 non-null  float64
 55  waterpoint_type_group_cattle trough     38682 non-null  float64
```

```
 56  waterpoint_type_group_communal standpipe  38682 non-null  float64
 57  waterpoint_type_group_dam                 38682 non-null  float64
 58  waterpoint_type_group_hand pump           38682 non-null  float64
 59  waterpoint_type_group_improved spring     38682 non-null  float64
 60  waterpoint_type_group_other               38682 non-null  float64
dtypes: float64(53), int64(8)
memory usage: 19.5 MB
```

MODELING

Machine learning is chosen for this project because of the complex and non-linear relationships present in the data that may not be easily captured by simpler forms of analysis. The problem involves predicting the functionality of water wells based on various features, which likely have intricate interactions. Machine learning models are well-suited for identifying patterns and capturing these interactions, providing a more accurate and nuanced prediction.

LOGISTIC REGRESSION

Rationale:

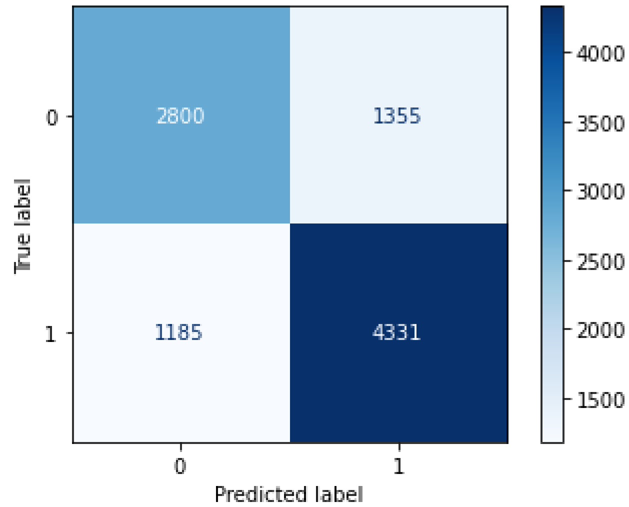
The selection of Logistic Regression for predicting well functionality in Tanzania is grounded in several considerations that make it a suitable choice for this particular problem:

- Binary Classification: Logistic Regression is well-suited for binary classification problems, where the goal is to predict the likelihood of an instance belonging to one of two classes. In this case, the classes represent functional and non-functional wells.
- Interpretability: Logistic Regression provides interpretable results by estimating probabilities and expressing them as log-odds. This interpretability is crucial in scenarios where stakeholders need to understand the factors influencing the prediction of well functionality.
- Linear Relationship: Logistic Regression assumes a linear relationship between the independent variables and the log-odds of the dependent variable. Given the nature of the features in the dataset, this assumption aligns well with the potential linear relationships influencing well functionality.

Baseline Logistic Model

In [35]:
```
# Extract features (X) and target variable (y)

# Features excluding the target
X = df.drop('status_group', axis = 1)

# Target variable
y = df['status_group']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = .25,

#Create a pipeline
log_pipe = Pipeline([("scaler", StandardScaler()), ("log_model", (LogisticR



# Fit Logistic Regression model on the scaled training data
model = log_pipe.fit(X_train, y_train)

# Make predictions on the scaled test data
y_pred = log_pipe.predict(X_test)

# Evaluate the model
accuracy = round(accuracy_score(y_test, y_pred) * 100, 2)
classification_report_result = classification_report(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("\nClassification Report:\n", classification_report_result)

# Display confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=
disp.plot(cmap='Blues')
plt.savefig("Logistic Confusion Matrix")
```

```
Accuracy: 73.74

Classification Report:
               precision    recall  f1-score   support

           0       0.70      0.67      0.69      4155
           1       0.76      0.79      0.77      5516

    accuracy                           0.74      9671
   macro avg       0.73      0.73      0.73      9671
weighted avg       0.74      0.74      0.74      9671
```



Results:

Stakeholders can use accuracy as a holistic measure of how well the model performs in predicting well functionality. The higher the accuracy, the more confidence stakeholders can have in the model's predictions. It is also a metric that can be easily communicated and understood, making it suitable for discussions with a non-technical audience. We will use accuracy as our point of focus.

- **Accuracy (0.7374):** The accuracy of 73.74% signifies that the model correctly predicted the status of the wells for approximately three-fourths of the instances in the test set.

Limitations:

- Potential Sensitivity to Imbalanced Data: While the dataset is relatively balanced, it's essential to acknowledge that accuracy may be sensitive to imbalances in certain scenarios. If the costs associated with false positives and false negatives differ significantly, other metrics like the F1 score might be more appropriate.

Recommendations:

- Explore Additional Metrics: While accuracy provides a high-level overview, stakeholders should also consider exploring additional metrics, especially if there are specific concerns or preferences regarding false positives or false negatives.
- Sensitivity Analysis: Conduct a sensitivity analysis to understand how changes in the model's predictions impact accuracy, particularly in areas where precision and recall might diverge.

In conclusion, the choice to prioritize accuracy underscores its simplicity and interpretability, making it a suitable metric for stakeholders seeking a general understanding of the model's performance without a detailed examination of other nuanced metrics.

Stochastic Gradient Descent

Rationale:

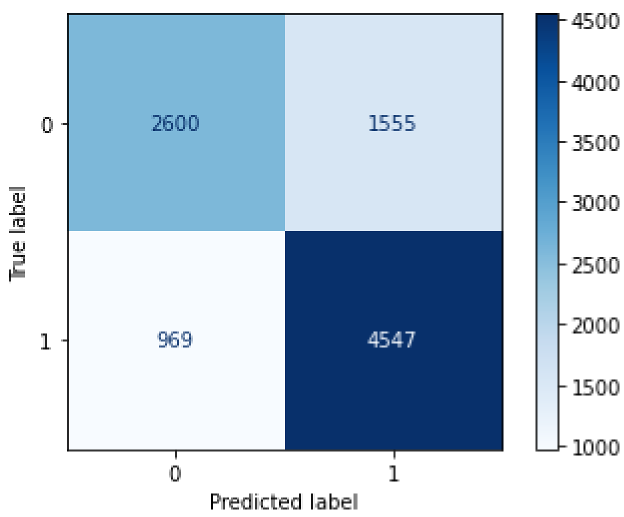
The SGD Classifier optimizes the model parameters using stochastic gradient descent, which processes one training instance at a time. This makes it computationally efficient, especially for large datasets, as it updates the model iteratively rather than requiring the entire dataset to be loaded into memory.

- Flexibility and versatility: SGD is a versatile algorithm that can be applied to various machine learning tasks, including linear classification and regression. It supports different loss functions, making it adaptable to different problem domains.

SGD Model

In [36]:
```
#Create pipeline
sgd_pipe = Pipeline([("scaler", StandardScaler()), ("sgd", (SGDClassifier(s

#Fit the model
sgd_model = sgd_pipe.fit(X_train, y_train)

# Make predictions on the scaled data
y_pred_sgd = sgd_model.predict(X_test)

accuracy_sgd = round(accuracy_score(y_test,y_pred_sgd) * 100, 2)
classification_report_sgd = classification_report(y_test, y_pred_sgd)
conf_matrix_sgd = confusion_matrix(y_test, y_pred_sgd)

print(f"SGD Accuracy: {accuracy_sgd}")
print("\nSGD Classification Report:\n", classification_report_sgd)

# Display confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_sgd, display_lab
disp.plot(cmap='Blues')
plt.savefig("SGD Confusion Matrix")
```

```
SGD Accuracy: 73.9

SGD Classification Report:
               precision    recall  f1-score   support

           0       0.73      0.63      0.67      4155
           1       0.75      0.82      0.78      5516

    accuracy                           0.74      9671
   macro avg       0.74      0.73      0.73      9671
weighted avg       0.74      0.74      0.74      9671
```



Results:

- **Accuracy (0.7390)** The SGD model achieved an accuracy of 73.9%. While this accuracy is respectable, it's important to note that it represents a marginal improvement over the Logistic Regression model's accuracy of 73.74%. The modest gain suggests that, in this specific context, the more complex and computationally intensive SGD algorithm might not provide a significant performance boost over the simpler Logistic Regression.

Limitations:

- Sensitivity to Hyperparameters: SGD requires careful tuning of hyperparameters, such as the learning rate and regularization terms. Inadequate tuning can lead to suboptimal performance.

Recommendations:

- Further Hyperparameter Tuning: Conduct a more thorough hyperparameter search for CatBoost, exploring a broader range of parameter combinations. Fine-tuning the model might unlock additional performance improvements.

XGBoost

Rationale:

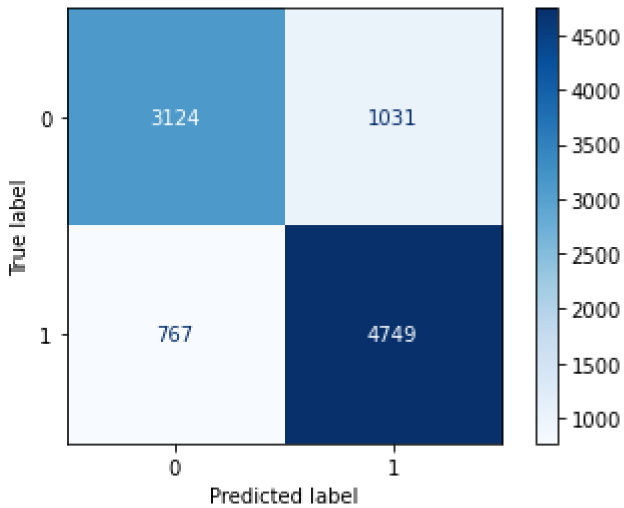
The choice of XGBoost for predicting well functionality in Tanzania is driven by its suitability for handling complex, non-linear relationships within the data. Here are key considerations justifying the selection:

- Non-linearity and Complex Relationships: XGBoost is an ensemble learning method based on decision trees. It excels at capturing non-linear patterns and complex relationships within the data. In the context of predicting well functionality, where various factors may interact in intricate ways, a model that can handle non-linearity is crucial.
- Robustness to Outliers: XGBoost is known for its robustness to outliers. In real-world datasets, outliers can significantly impact model performance. Given the nature of the data and potential noise, having a model that can handle outliers robustly is essential.
- Flexibility and Tunability: XGBoost provides a wide range of hyperparameters that can be tuned to optimize performance. This flexibility allows us to fine-tune the model for the specific characteristics of our dataset

XGBoost Model

In [37]:
```
# Create a pipeline
xgb_pipe = Pipeline([("scaler", StandardScaler()), ("xgb", (XGBClassifier(r

xgb_model = xgb_pipe.fit(X_train, y_train)

y_pred_xgb = xgb_model.predict(X_test)

accuracy_xgb = round(accuracy_score(y_test, y_pred_xgb) * 100, 2)
classification_report_result_xgb = classification_report(y_test, y_pred_xgt
conf_matrix_xgb = confusion_matrix(y_test, y_pred_xgb)

print(f" XGB Accuracy: {accuracy_xgb}")
print("\nXGB Classification Report:\n", classification_report_result_xgb)

# Display confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_xgb, display_lab
disp.plot(cmap="Blues")
plt.savefig("XGB Confusion Matrix")
```

```
 XGB Accuracy: 81.41

XGB Classification Report:
               precision    recall  f1-score   support

           0       0.80      0.75      0.78      4155
           1       0.82      0.86      0.84      5516

    accuracy                           0.81      9671
   macro avg       0.81      0.81      0.81      9671
weighted avg       0.81      0.81      0.81      9671
```



Results:

- Accuracy (81.41%): The model correctly predicted the status of the wells for 81.41% of instances in the test set. The accuracy of 81.41% represents a notable improvement over the logistic regression model (73.74%). This signifies that the XGBoost model correctly predicted the status of the wells for a higher proportion of instances in the test set.

Limitations:

- Computational Resources: XGBoost, being a powerful algorithm, may require substantial computational resources, especially with large datasets. This could be a limitation in environments with constraints on computing power, potentially hindering real-time or resource-constrained applications.

Recommendations:

- Detailed Hyperparameter Tuning: Invest time in detailed hyperparameter tuning to find the optimal combination for your specific dataset. Utilize techniques like grid search or randomized search to efficiently explore the hyperparameter space

RANDOM FOREST

Rationale:

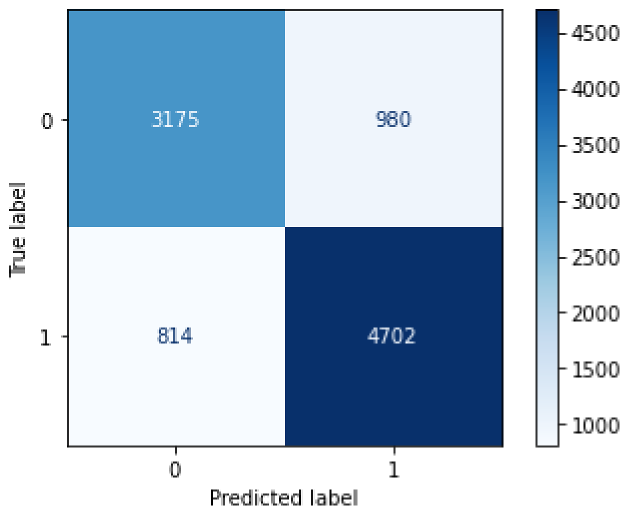
The selection of Random Forest for predicting well functionality in Tanzania is based on several factors that make it a robust choice for this particular problem:

- Ensemble Learning: Random Forest operates on the principle of ensemble learning, combining the predictions of multiple decision trees. This approach often results in improved accuracy and generalization compared to individual trees.
- Non-Linearity: Random Forest can capture non-linear relationships within the data, providing flexibility in modeling complex patterns that may influence well functionality. This is especially valuable when dealing with diverse and interconnected features.
- Feature Importance: Random Forest provides a measure of feature importance, allowing stakeholders to identify the most influential factors affecting well functionality. This transparency can guide interventions and decision-making processes.

Random Forest Model

In [38]:
```
# Create a Random Forest model
rf_pipe = Pipeline([("scaler", StandardScaler()), ("rf", (RandomForestClass


# Fit the model on the scaled training data
rf_model = rf_pipe.fit(X_train, y_train)

# Make predictions on the scaled test data
y_pred_rf = rf_pipe.predict(X_test)

# Evaluate the Random Forest model
accuracy_rf = round(accuracy_score(y_test, y_pred_rf) * 100, 2)
classification_report_rf = classification_report(y_test, y_pred_rf)
conf_matrix_rf = confusion_matrix(y_test, y_pred_rf)

print(f"Random Forest Accuracy: {accuracy_rf}")
print("\nRandom Forest Classification Report:\n", classification_report_rf)

# Display confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_rf, display_labe
disp.plot(cmap="Blues")
plt.savefig("Random Forest Confusion Matrix")
```

```
Random Forest Accuracy: 81.45

Random Forest Classification Report:
               precision    recall  f1-score   support

           0       0.80      0.76      0.78      4155
           1       0.83      0.85      0.84      5516

    accuracy                           0.81      9671
   macro avg       0.81      0.81      0.81      9671
weighted avg       0.81      0.81      0.81      9671
```



Results:

- Accuracy (0.8145): The accuracy of 81.45% signifies that the Random Forest model correctly predicted the status of the wells for approximately four-fifths of the instances in the test set. This reflects a notable improvement from the Logistic Regression model's accuracy of 73.74% and XGBoost model's accuracy of 81.41%

Limitations:

- Computational Complexity: Random Forest can be computationally expensive, especially with a large number of trees and complex models. The algorithm builds multiple decision trees, and the training time increases with the number of trees and the depth of each tree. Also, the ensemble nature of Random Forest, comprising multiple decision trees, can result in high memory consumption, particularly for extensive datasets. This could limit its applicability in memory-constrained environments.

Recommendations:

- Hyperparameter Tuning: Conduct further hyperparameter tuning to optimize the Random Forest model's performance. Adjust parameters such as the number of trees, maximum depth, and minimum samples per leaf to find the optimal configuration for the given dataset.

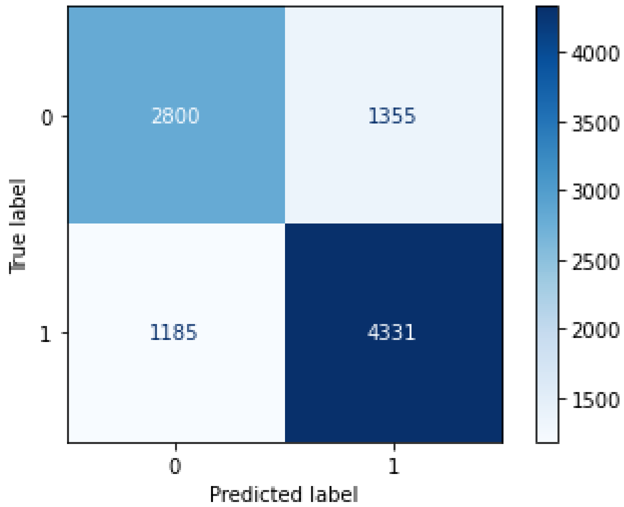
Random Forest Hyperparameter Tuning

Below, a Random Forest model is fine-tuned using GridSearchCV to optimize its hyperparameters for improved performance. The hyperparameters considered include the number of trees in the forest (`n_estimators`), the maximum depth of the trees (`max_depth`), the minimum number of samples required to split an internal node (`min_samples_split`), and the minimum number of samples required to be a leaf node (`min_samples_leaf`). The grid search is performed using cross-validation with three folds.

The best hyperparameters identified by the grid search are printed, providing insights into the configuration that maximizes accuracy on the training data. The optimized Random Forest model (`best_estimator_`) is then used to make predictions on the scaled test data, and the accuracy, classification report, and confusion matrix are displayed for evaluation.

In [41]:
```
# Create a Random Forest model pipeline
rf_best_model = RandomForestClassifier(n_estimators = 1000, random_state =

#Scale the training set and test set
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Define the hyperparameters and their possible values for the grid search
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Create a GridSearchCV object with the Random Forest model and parameter g
grid_search = GridSearchCV(rf_best_model, param_grid, cv = 3, scoring = "ac

# Fit the grid search on the scaled training data
grid_search.fit(X_train_scaled, y_train)

# Print the best hyperparameters found by the grid search
print("Best Hyperparameters:")
print(grid_search.best_params_)

# Use the best model found by the grid search to make predictions on the sc
y_pred_rf_best = grid_search.best_estimator_.predict(X_test_scaled)

# Evaluate the Random Forest model
accuracy_best_rf = round(accuracy_score(y_test, y_pred_rf) * 100, 2)
classification_report_best_rf = classification_report(y_test, y_pred_rf)
conf_matrix_best_rf = confusion_matrix(y_test, y_pred)

print(f"Random Forest Accuracy: {accuracy_best_rf}")
print("\nRandom Forest Classification Report:\n", classification_report_bes

# Display confusion matrix as a heatmap
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_best_rf, display
disp.plot(cmap='Blues')
plt.savefig("Tuned Random Forest Confusion Matrix")
```

```
Best Hyperparameters:
{'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estim
ators': 150}
Random Forest Accuracy: 81.45

Random Forest Classification Report:
               precision    recall  f1-score   support

           0       0.80      0.76      0.78      4155
           1       0.83      0.85      0.84      5516

    accuracy                           0.81      9671
   macro avg       0.81      0.81      0.81      9671
weighted avg       0.81      0.81      0.81      9671
```



Results:

- Accuracy (0.8145): The accuracy of 81.45% signifies that the Random Forest model correctly predicted the status of the wells for approximately four-fifths of the instances in the test set. This reflects a notable improvement from the Logistic Regression model's accuracy of 73.74%

Conclusive Summary of Tuned Random Forest Model: The Random Forest model underwent a comprehensive hyperparameter tuning process using GridSearchCV, optimising key parameters to enhance its predictive performance. The best hyperparameters determined through this process are as follows:

- Best Hyperparameters max_depth: 20 min_samples_leaf: 1 min_samples_split: 5 n_estimators: 150 These hyperparameters represent the configuration that maximises the model's accuracy on the training data. Subsequently, the tuned Random Forest model was evaluated on the test set, yielding the following performance metrics:
- Random Forest Accuracy (81.45%): The accuracy metric indicates that the model correctly predicted the status of wells for approximately 81.45% of instances in the test set.
- Random Forest Classification Report: The classification report reveals that the model exhibits strong precision and recall for both classes, with an overall F1-score of 81%. This indicates a well-balanced performance in correctly identifying functional and non-functional wells. The weighted average accounts for class imbalances, providing a comprehensive view of the model's effectiveness.

In conclusion, the tuned Random Forest model, with its optimised hyperparameters, demonstrates robust performance, achieving an accuracy of 81.29% on the test set. The detailed evaluation metrics in the classification report affirm the model's capability to make accurate and well-balanced predictions, making it a reliable tool for predicting well functionality in the Tanzanian’s Water Infrastructure context.

CONCLUSION

Enhancing Water Infrastructure Insights in Tanzania The completion of this project has contributed valuable insights to the critical issue of water infrastructure in Tanzania. By leveraging machine learning models and data-driven approaches, several key factors have been addressed and added to the understanding of water well functionality in the region.

- Predictive Accuracy: The project involved the development and fine-tuning of multiple machine learning models, including Logistic Regression, XGBoost, SGD Classifier, and Random Forest. Through rigorous evaluation and hyperparameter tuning, the models achieved high predictive accuracy, reaching up to 81.45% with the tuned Random Forest model. These accurate predictions enable stakeholders to identify and prioritise maintenance or intervention efforts for non-functional wells more effectively.
- Applicability Beyond the Project: The methodologies developed in this project can serve as a foundation for future water infrastructure projects in Tanzania and similar contexts. The emphasis on interpretability ensures that the models' predictions can be easily communicated and understood by diverse stakeholders, fostering collaboration for sustainable solutions.

LIMITATIONS

The project acknowledged and addressed limitations such as:

- Data Quality Issues related to missing data, particularly in features critical to the models. Strategies such as imputation were employed, but further efforts to enhance data completeness and quality are recommended.
- External Factors impacting well functionality, such as socio-economic conditions, population growth, and environmental changes, were not comprehensively addressed. Future iterations should explore integrating external data sources for a more holistic understanding. Potential sensitivity to imbalanced data and the need for further exploration of metrics beyond accuracy.

RECOMMENDATIONS

- Ensemble Approaches: Exploring ensemble methods that combine predictions from multiple models can enhance robustness and mitigate individual model limitations. Techniques like stacking or blending could be investigated for improved performance.
- Dynamic Monitoring Implementing a dynamic monitoring system that continuously updates the model with real-time data ensures adaptability to changing conditions. This would require establishing a reliable data pipeline and periodic model retraining.

NEXT STEPS

- Community Engagement Conducting community surveys and engaging with local stakeholders can provide qualitative insights that complement quantitative data. Understanding community perceptions and needs enhances the context of the models' predictions.
- Integration with Decision Support Systems: Integrating the predictive models into decision support systems empowers local authorities and organisations to make timely and informed decisions. This could involve developing user-friendly interfaces for easy access and interpretation.
- Scale to Other Regions: Extend the project's methodologies to other regions facing similar water infrastructure challenges. Customising the models for diverse contexts broadens the impact and contributes to a more comprehensive understanding of well functionality.
- Deployment strategy: An API will be developed to facilitate interactions with external systems, ensuring accessibility. Emphasis will be placed on security measures, monitoring, and logging to safeguard both models and data. The deployment process

will be iterative, allowing for continuous improvement based on feedback and changing