



DATA MINING

Higher Diploma in Science in Data Analytics

Name: Eduardo Campos

Contents

Introduction.....	1
Pump it Up: Data Mining the Water Table.....	2
1) Data Loading	2
2) Analyze by Describing Data.....	3
2.2) Describing Numerical Values	4
2.3.) Describing Categorical Values	4
3) Data Cleaning	6
4) Analyze by Pivoting Features	9
5) Analyze by Pivoting Features	14
6) Converting Categorical Features to Numeric.....	20
7) Split the Dataset into Training and Testing Sets	23
8) Balance the Dataset	24
9) Model, Predict and Solve.....	26
9.1. Multinomial Logistic Regression	26
9.2. Random Forest	27
9.3. K-Nearest Neighbors (KNN).....	28
9.4. Neural Networks (MLP Classifier)	29
Conclusion.....	32

Index of Figures

Figure 1. Load the data using Pandas.....	2
Figure 2. Union of the train values and labels datasets.....	3
Figure 3. Analyzing the Values in the Dataset	4
Figure 4. Describing Numerical Values.....	4
Figure 5. Describing Categorical Values.....	5
Figure 6. Identifying missing values.	5
Figure 7. Number of Variables with Missing Values.....	6
Figure 8. Percentage of Missing Values per Variable	6
Figure 9. Assignment of the Mean per Region for Missing Values in the column "Longitude"	7
Figure 10. Establish a Range of Population	7
Figure 11. Filling Missing Values in the column "public_meeting".....	8
Figure 12. Establish Range per Decade for the column "construction_year"	8
Figure 13. Definition of the column "Days_Since_Recorded".....	9
Figure 14. Analysis of the "basin" column using a Pivot Table.....	10
Figure 15. Analysis of the "region" column using a Pivot Table.	11
Figure 16. Analysis of "permit" column using a Pivot Table	11
Figure 17. Analysis of "extraction_type" column using a Pivot Table	12
Figure 18. Analysis of "water_quality" column using a Pivot Table	12
Figure 19. Analysis of "quantity" column using a Pivot Table.....	13
Figure 20. Analysis of "source" column using a Pivot Table	13
Figure 21. Analysis of "waterpoint_type" column using a Pivot Table	14
Figure 22. Analysis of "Year_Range" column using a Pivot Table	14
Figure 23. Visualization through Boxplot for the variable "amount_tsh".	15
Figure 24. Visualization through Boxplot for the variable "gps_height".....	16
Figure 25. Visualization through Boxplot for the variable "population".....	17
Figure 26. Visualization through Histogram for the variable "days_since_recorded".	17
Figure 27. Removal of Irrelevant Variables.....	18
Figure 28. Correlation Heatmap	19
Figure 29. GeoMap for the "Latitude" and "Longitude" of each region.	20
Figure 30. Transforming the Dependent Variable into Numeric Values	21
Figure 31. Application of One-Hot Encoding for Transforming Independent Variables into Binary Format.	22
Figure 32. Dataset with Numeric Values.....	22
Figure 33. Variables defined as Integers and Floats	23
Figure 34. Splitting the Dataset into Training and Testing Sets.	24
Figure 35. Disbalance of Dependent Variables	24
Figure 36. Balancing Dependent Variables Using the SMOTE Function	25
Figure 37. Scaling of Independents Variables	25

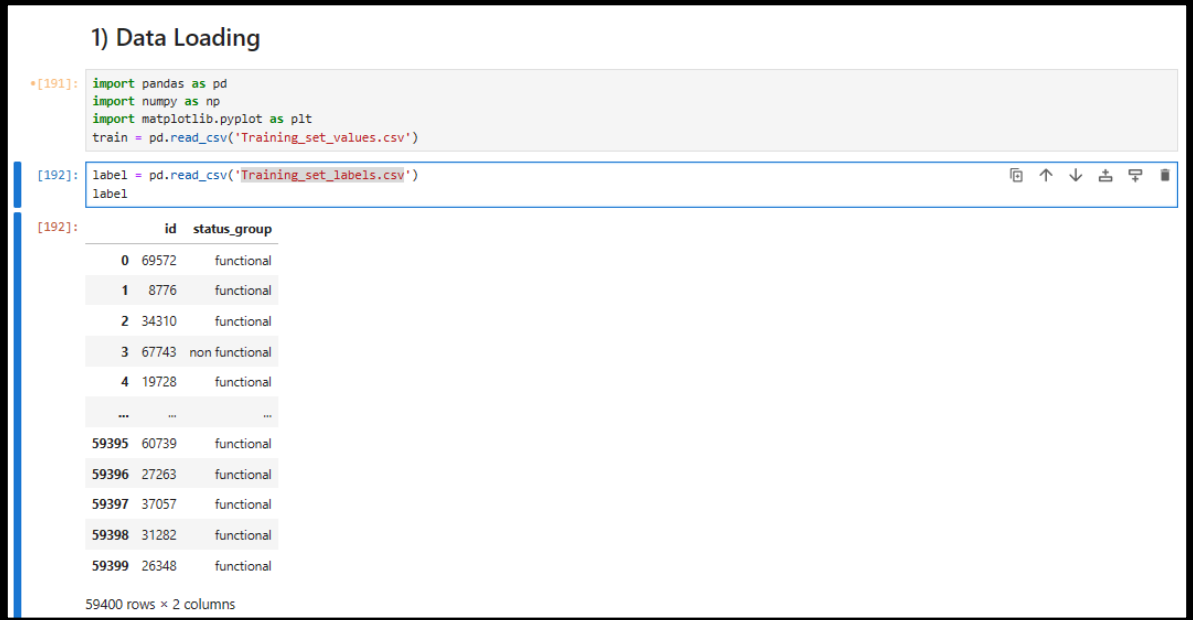
Figure 38. Accuracy of Multinomial Logistic Regression	27
Figure 39. Accuracy of Random Forest	28
Figure 40. Accuracy of K-Nearest Neighbors (KNN)	29
Figure 41. Accuracy of Neural Networks (MLP Classifier)	29
Figure 42. Model Evaluation Table	30
Figure 43. Correlation Matrix	31

Introduction

This project delves into a systematic exploration of the operational status of individual Waterpoint records within the dataset. The undertaking comprises meticulous tasks, including the importation of libraries, loading of data, generation of scatterplots, and comprehensive data cleaning. Employing advanced techniques, graphical representations, and sophisticated models, my objective is to craft a precise predictive model. Following this, a rigorous assessment will be conducted to evaluate the model's accuracy against predefined standards, ensuring its utmost reliability.

Pump it Up: Data Mining the Water Table

1) Data Loading



The screenshot shows a Jupyter Notebook interface with the title "1) Data Loading". It contains two code cells. The first cell, labeled "[191]:", imports the necessary libraries: `import pandas as pd`, `import numpy as np`, `import matplotlib.pyplot as plt`, and loads the training data with `train = pd.read_csv('Training_set_values.csv')`. The second cell, labeled "[192]:", loads the labels with `label = pd.read_csv('Training_set_labels.csv')` and displays the variable `label`. Below the code, a preview of the `label` DataFrame is shown, displaying columns `id` and `status_group`. The preview includes rows 0 through 4, an ellipsis indicating more rows, and rows 59395 through 59399. At the bottom, it states "59400 rows x 2 columns".

```
[191]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
train = pd.read_csv('Training_set_values.csv')

[192]: label = pd.read_csv('Training_set_labels.csv')
label
```

	id	status_group
0	69572	functional
1	8776	functional
2	34310	functional
3	67743	non functional
4	19728	functional
...
59395	60739	functional
59396	27263	functional
59397	37057	functional
59398	31282	functional
59399	26348	functional

59400 rows x 2 columns

Figure 1. Load the data using Pandas

In this phase, I use essential libraries for analysis:

1. **pandas (pd)**: Facilitates data manipulation and analysis with DataFrame structures.
2. **matplotlib.pyplot (plt)**: Creates various plots and charts for visual data representation.
3. **numpy (np)**: Supports numerical operations, handling large arrays and matrices.

I employed the 'pandas' library to import the necessary data, which, in this case, is divided into two CSV files. The first file, 'training_set_values.csv,' contains all the independent variables crucial for predicting the operational status of waterpoints—whether they are functional, non-functional, or in need of repair. The corresponding

To synchronize the labels (dependent values) with the training data values (independent values), I utilized the 'merge' function from pandas. This association was established through the common column "id," facilitating a seamless connection between the two datasets.

```

•[193]: merge = pd.merge(train,label, on="id", how="outer")

water_table = merge

water_table

```

0	69572	6000.0	2011-03-14	Roman	1390	Roman	34.938093	-9.856322	none	0	Lake Nyasa	Mnyusi B	Iringa
1	8776	0.0	2013-03-06	Grumeti	1399	GRUMETI	34.698766	-2.147466	Zahanati	0	Lake Victoria	Nyamara	Mara
2	34310	25.0	2013-02-25	Lottery Club	686	World vision	37.460664	-3.821329	Kwa Mahundi	0	Pangani	Majengo	Manyara
3	67743	0.0	2013-01-28	Unicef	263	UNICEF	38.486161	-11.155298	Zahanati Ya Nanyumbu	0	Ruvuma / Southern Coast	Mahakamani	Mtwara
4	19728	0.0	2011-07-13	Action In A	0	Artisan	31.130847	-1.825359	Shuleni	0	Lake Victoria	Kyanyamisa	Kagera

2) Analyze by Describing Data

By employing the 'head' function, one can conveniently examine the initial values within the dataset. The majority of these values predominantly consist of categorical variables.

2) Analyze by Describing Data

[195]: water_table.head()

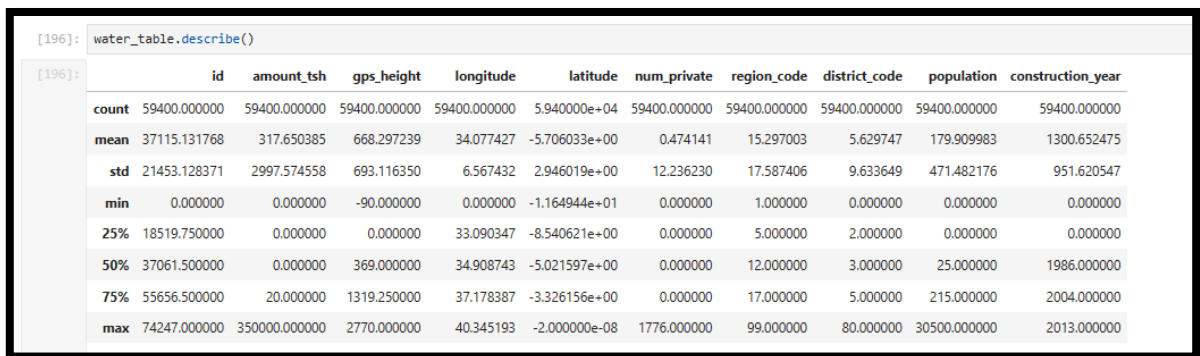
	ID	Value	Date	Name	Age	Gender	Income	Height	Weight	Education	Region	District	Ward	Count
1	8776	0.0	2013-03-06	Grumeti	1399	GRUMETI	34.698766	-2.147466	Zahanati	0	Lake Victoria	Nyamara	Mara	20
2	34310	25.0	2013-02-25	Lottery Club	686	World vision	37.460664	-3.821329	Kwa Mahundi	0	Pangani	Majengo	Manyara	21
3	67743	0.0	2013-01-28	Unicef	263	UNICEF	38.486161	-11.155298	Zahanati Ya Nanyumbu	0	Ruvuma / Southern Coast	Mahakamani	Mtwara	90
4	19728	0.0	2011-07-13	Action In A	0	Artisan	31.130847	-1.825359	Shuleni	0	Lake Victoria	Kyanyamisa	Kagera	18

Figure 3. Analyzing the Values in the Dataset

2.2) Describing Numerical Values

To enhance the characterization of my numerical variables, I employed the "describe" function. Key insights derived from this analysis include:

- The dataset encompasses 59,400 waterpoint records.
- A significant 75% of these records originate from the region identified by the region code 17 and district 5.
- The highest recorded population for a waterpoint is 30,500 people.
- The construction years of the waterpoints range from 1986 to 2013, with the majority being constructed in 1968.



	id	amount_tsh	gps_height	longitude	latitude	num_private	region_code	district_code	population	construction_year
count	59400.000000	59400.000000	59400.000000	59400.000000	5.940000e+04	59400.000000	59400.000000	59400.000000	59400.000000	59400.000000
mean	37115.131768	317.650385	668.297239	34.077427	-5.706033e+00	0.474141	15.297003	5.629747	179.909983	1300.652475
std	21453.128371	2997.574558	693.116350	6.567432	2.946019e+00	12.236230	17.587406	9.633649	471.482176	951.620547
min	0.000000	0.000000	-90.000000	0.000000	-1.164944e+01	0.000000	1.000000	0.000000	0.000000	0.000000
25%	18519.750000	0.000000	0.000000	33.090347	-8.540621e+00	0.000000	5.000000	2.000000	0.000000	0.000000
50%	37061.500000	0.000000	369.000000	34.908743	-5.021597e+00	0.000000	12.000000	3.000000	25.000000	1986.000000
75%	55656.500000	20.000000	1319.250000	37.178387	-3.326156e+00	0.000000	17.000000	5.000000	215.000000	2004.000000
max	74247.000000	350000.000000	2770.000000	40.345193	-2.000000e-08	1776.000000	99.000000	80.000000	30500.000000	2013.000000

Figure 4. Describing Numerical Values

2.3.) Describing Categorical Values

After gaining insights into numerical values, it was time to examine categorical values. Using the same function but including objects, the following outputs were obtained:

- There are nine distinct types of basins, with Lake Victoria being the most frequent.
- Most of the pumps are classified as public meetings.
- Gravity extraction is the predominant method for most water points.
- A significant portion of them has never been paid for.
- The majority are functional, providing a sufficient and reliable water supply to people.


```
[197]: water_table.describe(include=['O'])
```

payment_type	water_quality	quality_group	quantity	quantity_group	source	source_type	source_class	waterpoint_type	waterpoint_type_group	status_group
59400	59400	59400	59400	59400	59400	59400	59400	59400	59400	59400
7	7	8	6	5	5	10	7	3	7	6
never pay	never pay	soft	good	enough	enough	spring	spring	groundwater	communal standpipe	communal standpipe
25348	25348	50818	50818	33186	33186	17021	17021	45794	28522	34625
										32259

Figure 5. Describing Categorical Values

After gaining an understanding of the dataset, I proceeded to assess information about missing values. Employing the "info()" function, I identified certain variables with null values.

```
[198]: water_table.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 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         55763 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       59398 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
```

Figure 6. Identifying missing values.

For a more detailed account of the specific count of missing values, I employed the "isnull()" function to identify them, followed by the "sum()" function to calculate the total. Variables such as funder, installer, subvillage, "wpt_name," "public_meetings," "schema_management," "schema_name," and "permit" exhibited missing values. Notably, "schema_name" stood out with the highest count of missing values.

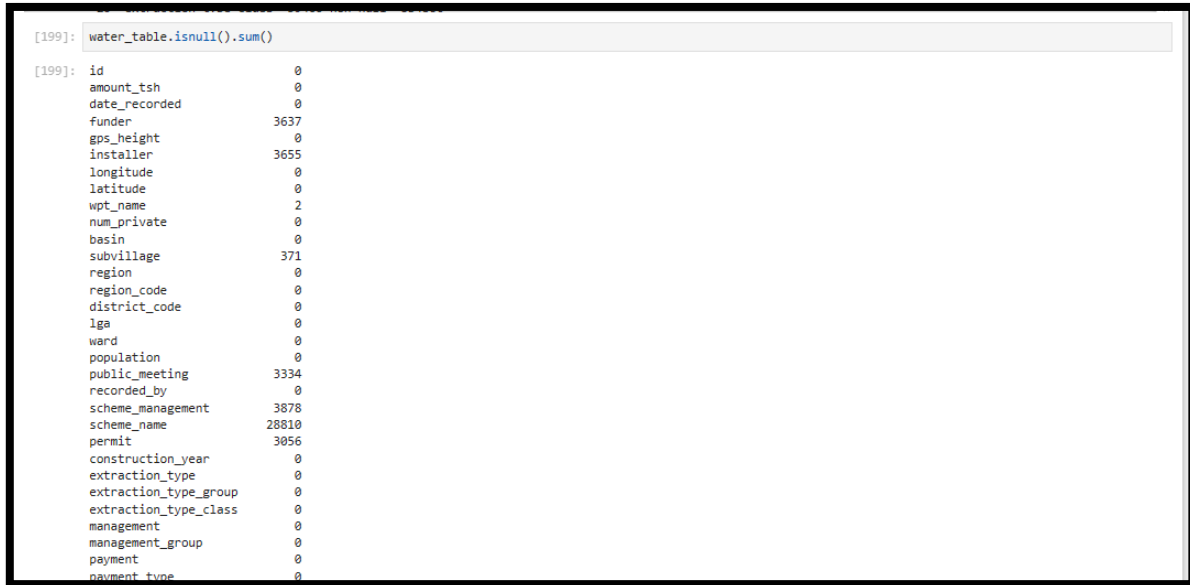


Figure 7. Number of Variables with Missing Values

Next, I identified variables with the highest number of zero values in the numerical dataset, aiming to evaluate if this could potentially impact the accuracy of those values. Upon consideration, the variable "amount_tsh" denotes the quantity of water available to the waterpoint, allowing for instances where a waterpoint may have no water due to factors like dry seasons or maintenance. Additionally, "gps_height" signifies the altitude of the well and might be zero, serving as a reference point for the height of planetary features. However, other zero values needed addressing as leaving them unattended could potentially compromise the accuracy of the model results.

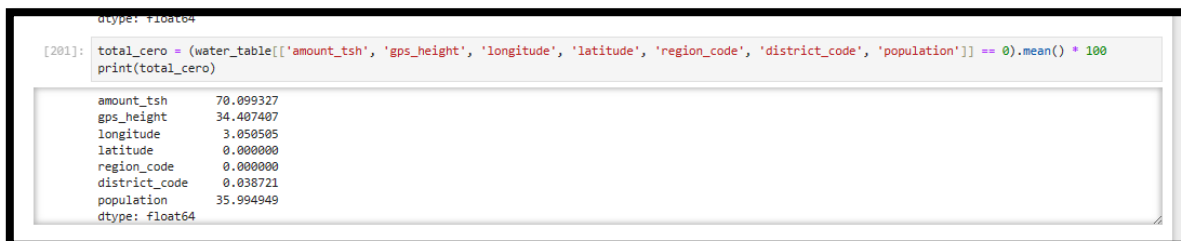


Figure 8. Percentage of Missing Values per Variable

3) Data Cleaning

In the case of the variable "Longitude," the optimal solution for me was to approximate the values with zero by calculating the mean based on the "region code" variable. I implemented a loop to assign the mean longitude corresponding to each

"region code." Subsequently, I executed a query to verify if there were any zero values, and the output confirmed the absence of values meeting that condition.

```
[204]: zero_sum = (water_table['longitude'] == 0).sum()
total_rows = len(water_table)

percentage_zero = (zero_sum / total_rows) * 100

print(f"Sum of zero values in longitude is: {zero_sum}")
print(f"Percentage of zero values is: {percentage_zero:.2f}%")

Sum of zero values in longitude is: 1812
Percentage of zero values is: 3.05%

[205]: # "I have implemented a loop that filters data based on the 'region_code' (which is defined in the dataset from 1 to 99) and assigns the mean of each reg
pd.set_option('display.max_columns', None)

for i in range(1, 100):
    region_index = water_table.query(f'region_code == {i}').copy()
    x = region_index.loc[region_index['longitude'] != 0, 'longitude'].mean()
    region_index['longitude'].replace(0, x, inplace=True) # Use inplace=True to modify the original DataFrame
    water_table.loc[water_table['region_code'] == i, 'longitude'] = region_index['longitude']

water_table_longitude_cero2 = water_table.query('longitude == 0')
water_table_longitude_cero2

[205]: id amount_tsh date_recorded funder gps_height installer longitude latitude wpt_name num_private basin subvillage region region_code district_code lga
```

Figure 9. Assignment of the Mean per Region for Missing Values in the column "Longitude"

I additionally introduced a column named "Population_Range" as an alternative approach to handle zero values in the "population" column. This column was devised to categorize populations into specific ranges based on predefined thresholds. Any population value falling outside the defined ranges would be assigned the label "No Identified."

```
[207]: water_table["Population_Range"] = water_table["population"]

water_table["Population_Range"] = pd.to_numeric(water_table["Population_Range"], errors='coerce')

conditions = [
    (water_table["Population_Range"] < 100),
    ((water_table["Population_Range"] >= 100) & (water_table["Population_Range"] <= 500)),
    ((water_table["Population_Range"] > 500) & (water_table["Population_Range"] <= 1000)),
    ((water_table["Population_Range"] > 1000) & (water_table["Population_Range"] <= 1500)),
    ((water_table["Population_Range"] > 1500) & (water_table["Population_Range"] <= 10000)),
    ((water_table["Population_Range"] > 10000) & (water_table["Population_Range"] <= 15000)),
    (water_table["Population_Range"] > 15000)
]

choices = [
    "Population less than 100",
    "Population between 100 to 500",
    "Population between 501 to 1000",
    "Population between 1001 to 1500",
    "Population between 1501 to 10000",
    "Population between 10001 to 15000",
    "Population more than 15000",
]

## To handle easier the conditional assignment I use the function np.select(), it is going to assign a value according the condition.
water_table["Population_Range"] = np.select(conditions, choices, default="No Identified")

water_table
```

Figure 10. Establish a Range of Population

The missing values in the "public_meeting" and "permit" columns were addressed by substituting them with the string "Not Specified."

```
[208]: water_table["public_meeting"].replace(pd.NA,"Not Specified", inplace=True)
       water_table["public_meeting"]

[208]: 0      True
       1  Not Specified
       2      True
       3      True
       4      True
       ...
       59395      True
       59396      True
       59397      True
       59398      True
       59399      True
       Name: public_meeting, Length: 59400, dtype: object

[209]: ##pd.reset_option('display.max_columns',)
       pd.set_option('display.max_columns', None)
       water_table["permit"].replace(pd.NA,"Not Specified", inplace=True)
       query = water_table.query('permit == "Not Specified"')
       query
```

Figure 11. Filling Missing Values in the column "public_meeting".

A similar approach was employed for the "construction_year" column, where I categorized the years into decades. A new column was introduced to specify the decade corresponding to each year. Even instances without a defined year were labeled as "unknown." This straightforward method simplifies the handling of inputs during model execution.

```
[210]: ##pd.reset_option('display.max_columns',)
       pd.set_option('display.max_columns', None)
       water_table["Year_Range"] = water_table["construction_year"]

       water_table["Year_Range"] = pd.to_numeric(water_table["Year_Range"], errors='coerce')

       conditions1 = [
           ((water_table["Year_Range"] >= 1960) & (water_table["Year_Range"] < 1970)),
           ((water_table["Year_Range"] >= 1970) & (water_table["Year_Range"] < 1980)),
           ((water_table["Year_Range"] >= 1980) & (water_table["Year_Range"] < 1990)),
           ((water_table["Year_Range"] >= 1990) & (water_table["Year_Range"] < 2000)),
           ((water_table["Year_Range"] >= 2000) & (water_table["Year_Range"] < 2010)),
           (water_table["Year_Range"] >= 2010)
       ]

       choices2 = [
           '60s',
           '70s',
           '80s',
           '90s',
           '00s',
           '10s',
       ]

       ## To handle easier the conditional assignment I use the function np.select(), it is going to assign a value according the condition.
       water_table["Year_Range"] = np.select(conditions1, choices2, default='unknown')

       water_table
```

Figure 12. Establish Range per Decade for the column "construction_year"

I have computed the duration, in terms of days, between the data acquisition date for a particular pump, as denoted by the 'date_recorded' variable, and the most recent date in the dataset. This approach is grounded on the assumption that data recorded more recently is more likely to accurately represent the pump's functionality. The initial step involved converting the column to a datetime type for further analysis.

```
[213]: water_table['date_recorded'] = pd.to_datetime(water_table['date_recorded'])
reference_date = pd.to_datetime('2013-12-3')

water_table['days_since_recorded'] = (reference_date - water_table['date_recorded']).dt.days

water_table
```

	id	amount_tsh	date_recorded	funder	gps_height	installer	longitude	latitude	wpt_name	num_private	basin	subvillage	region	reg
0	69572	6000.0	2011-03-14	Roman	1390	Roman	34.938093	-9.856322	none	0	Lake Nyasa	Mnyusi B	Iringa	
1	8776	0.0	2013-03-06	Grumeti	1399	GRUMETI	34.698766	-2.147466	Zahanati	0	Lake Victoria	Nyamara	Mara	
2	34310	25.0	2013-02-25	Lottery Club	686	World vision	37.460664	-3.821329	Kwa Mahundi	0	Pangani	Majengo	Manyara	
3	67743	0.0	2013-01-28	Unicef	263	UNICEF	38.486161	-11.155298	Zahanati Ya Nanyumbu	0	Ruvuma / Southern Coast	Mahakamani	Mtwara	
4	19728	0.0	2011-07-13	Action In A	0	Artisan	31.130847	-1.825359	Shuleni	0	Lake Victoria	Kyanyamisa	Kagera	
...
59395	60739	10.0	2013-05-03	Germany Republi	1210	CES	37.169807	-3.253847	Area Three Namba 27	0	Pangani	Kiduruni	Kilimanjaro	

Figure 13. Definition of the column "Days_Since_Recorded".

4) Analyze by Pivoting Features

At this stage, I aggregated certain categorical variables to identify patterns among functional, functional-but-need-repair, and non-functional pumps.

For the initial grouping, I focused on the "basin" column. As illustrated in the image below, it is noteworthy that over 50% of the functional pumps are distributed across the Internal, Lake Nyasa, Pangani, Rufiji, and Wami/Ruvu basins. In contrast, 50% of the non-functional pumps are primarily found in the Ruvuma/Southern Coast basin. Additionally, some basins, such as Lake Rukwa and Lake Tanganyika, exhibit a distinct pattern, with approximately 10% of the pumps being functional but in need of repair.

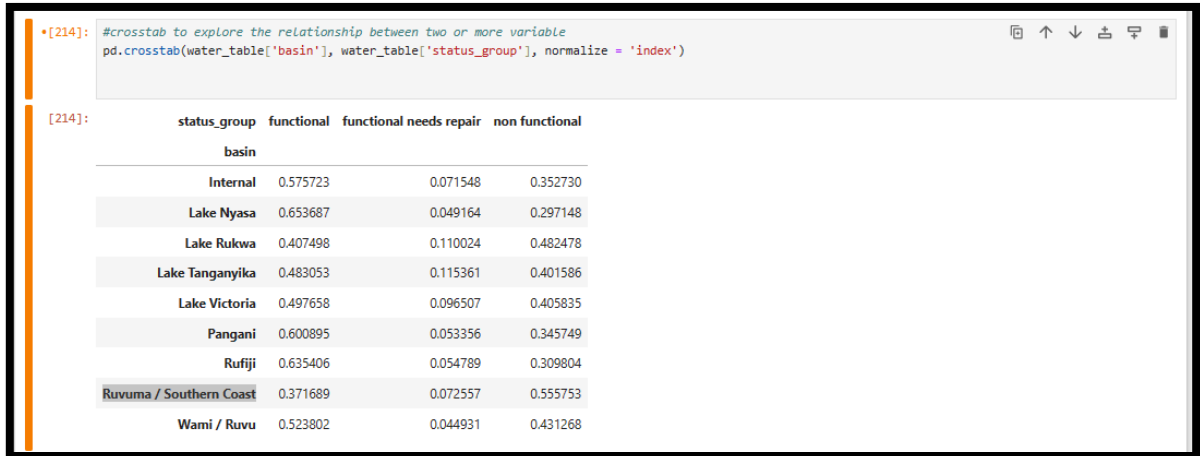


Figure 14. Analysis of the "basin" column using a Pivot Table.

Within the "regions" column, the dataset encompasses 21 defined regions. Notably, over 50% of functional pumps are concentrated in Arusha, Dar es Salaam, Iringa, Kagera, Kilimanjaro, Morogoro, Pwani, Ruvuma, and Tanga. Conversely, more than half of the non-functional pumps originate from Lindi, Mara, Mtwara, Rukwa, and Tabora. Additionally, regions like Kigoma and Shinyanga exhibit a distinct pattern, with around 10% of pumps being functional but in need of repair.

```
[215]: pd.crosstab(water_table['region'], water_table['status_group'], normalize = 'index')
```

```
[215]:
```

status_group	functional	functional needs repair	non functional
region			
Arusha	0.684776	0.052239	0.262985
Dar es Salaam	0.572671	0.003727	0.423602
Dodoma	0.458428	0.094957	0.446615
Iringa	0.782206	0.023234	0.194560
Kagera	0.520808	0.091677	0.387515
Kigoma	0.484020	0.214134	0.301847
Kilimanjaro	0.602877	0.073533	0.323590
Lindi	0.297542	0.060155	0.642303
Manyara	0.623500	0.060644	0.315856
Mara	0.449975	0.030472	0.519553
Mbeya	0.499892	0.108644	0.391464
Morogoro	0.528957	0.074888	0.396156
Mtwara	0.302890	0.072832	0.624277
Mwanza	0.484204	0.058994	0.456802
Pwani	0.590512	0.013662	0.395825
Rukwa	0.391040	0.074668	0.534292
Ruvuma	0.560606	0.062121	0.377273
Shinyanga	0.559815	0.127459	0.312726
Singida	0.483039	0.061156	0.455805

Figure 15. Analysis of the "region" column using a Pivot Table.

Columns such as "public_meeting," "permit," "payment," "source_class," and the newly created "Population_Range" column were deemed less informative for the model. This decision was based on the observation that the percentages of functional, functional-needs-repair, and non-functional pumps were similar for each value within these columns. Consequently, in the "Extraction of Similar Variables" step, these columns were considered for removal to streamline the model's input features.

```
[218]: pd.crosstab(water_table['permit'], water_table['status_group'], normalize = 'index')
```

```
[218]:
```

status_group	functional	functional needs repair	non functional
permit			
False	0.517094	0.075463	0.407443
True	0.554437	0.069417	0.376145
Not Specified	0.547448	0.098168	0.354385

Figure 16. Analysis of "permit" column using a Pivot Table

In the "extraction_type" column, a notable pattern emerges over 50% of functional pumps are associated with extraction types such as "afridev," "gravity," "india mark ii," "nira/tanira," "other – rope pump," "other - swn 81," "submersible," and "swn 80." Conversely, 50% of non-functional pumps tend to be linked to extraction types like "climax," "india mark iii," "mono," "windmill," and others, excluding "rope pump" and

"swn 81." Additionally, pumps originating from "walimi" exhibit a unique characteristic, with over 10% being functional but in need of repair.

```
[219]: pd.crosstab(water_table['extraction_type'], water_table['status_group'], normalize = 'index')
```

```
[219]:
```

	status_group	functional	functional needs repair	non functional
extraction_type				
	afridev	0.677966	0.023729	0.298305
	cemo	0.500000	0.100000	0.400000
	climax	0.250000	0.000000	0.750000
	gravity	0.599253	0.100859	0.299888
	india mark ii	0.603333	0.032917	0.363750
	india mark iii	0.448980	0.010204	0.540816
	ksb	0.496820	0.018375	0.484806
	mono	0.377661	0.045026	0.577312
	nira/tanira	0.664827	0.078612	0.256561
	other	0.160031	0.032037	0.807932
	other - mkulima/shinyanga	0.000000	0.000000	1.000000
	other - play pump	0.341176	0.011765	0.647059
	other - rope pump	0.649667	0.037694	0.312639
	other - swn 81	0.524017	0.030568	0.445415
	submersible	0.551217	0.047649	0.401134
	swn 80	0.569482	0.057766	0.372752
	walimi	0.479167	0.250000	0.270833
	windmill	0.427350	0.059829	0.512821

Figure 17. Analysis of "extraction_type" column using a Pivot Table

Concerning the "water_quality" column, the majority—more than 50%—of pumps exhibit the qualities of "fluoride," "milky," "salty abandoned," and "soft." However, a notable distinction is observed with "fluoride abandoned," where over 50% of pumps are non-functional. Additionally, the "salty abandoned" quality stands out, with more than 10% of pumps being functional but in need of repairs.

```
[222]: pd.crosstab(water_table['water_quality'], water_table['status_group'], normalize = 'index')
```

```
[222]:
```

	status_group	functional	functional needs repair	non functional
water_quality				
	coloured	0.502041	0.110204	0.387755
	fluoride	0.755000	0.065000	0.180000
	fluoride abandoned	0.352941	0.000000	0.647059
	milky	0.544776	0.017413	0.437811
	salty	0.457166	0.046334	0.496499
	salty abandoned	0.513274	0.212389	0.274336
	soft	0.565941	0.076823	0.357236
	unknown	0.140725	0.018657	0.840618

Figure 18. Analysis of "water_quality" column using a Pivot Table

The "quantity" column provides insights into water availability, with categories such as "enough," "insufficient," and "seasonal" demonstrating functional water pumps exceeding a 50% rate. Conversely, when water pumps indicate a dry condition, they exhibit a non-functional rate of over 90%. Interestingly, only 10% of water pumps with a seasonal quantity are identified as needing repair.

```
[224]: pd.crosstab(water_table['quantity'], water_table['status_group'], normalize = 'index')
```

```
[224]: status_group  functional  functional needs repair  non functional
quantity
dry      0.025136      0.005924      0.968940
enough   0.652323      0.072320      0.275357
insufficient  0.523234      0.095842      0.380924
seasonal  0.574074      0.102716      0.323210
unknown  0.269962      0.017744      0.712294
```

Figure 19. Analysis of "quantity" column using a Pivot Table

In the "source" column, the analysis reveals that over 50% of functional water pumps originate from sources such as "hand dtw," "machine dbh," "rainwater harvesting," "river," and "spring." Conversely, more than 50% of non-functional pumps are associated with sources like "dam" and "lake." Additionally, a noteworthy observation is that more than 10% of water pumps, deemed functional but in need of repair, are sourced from "rainwater harvesting" and "river".

```
[225]: pd.crosstab(water_table['source'], water_table['status_group'], normalize = 'index')
```

```
[225]: status_group  functional  functional needs repair  non functional
source
dam      0.385671      0.036585      0.577744
hand dtw  0.568650      0.019451      0.411899
lake     0.211765      0.015686      0.772549
machine dbh  0.489571      0.044334      0.466095
other     0.594340      0.004717      0.400943
rainwater harvesting  0.603922      0.136819      0.259259
river     0.568560      0.127029      0.304411
shallow well  0.494769      0.056883      0.448348
spring    0.622290      0.074966      0.302744
unknown   0.484848      0.060606      0.454545
```

Figure 20. Analysis of "source" column using a Pivot Table

In the "waterpoint_type" column, the data indicates that over 50% of functional water pumps fall into categories such as "cattle trough," "communal standpipe," "dam," "hand pump," and "improved spring." Conversely, water points categorized as "communal standpipe multiple" show a non-functional rate exceeding 50%. Interestingly, both "communal standpipe multiple" and "improved spring" types exhibit a unique characteristic, with 10% of them being functional but in need of repairs.

```
[227]: pd.crosstab(water_table['waterpoint_type'], water_table['status_group'], normalize = 'index')
```

```
[227]:
```

	status_group	functional	functional needs repair	non functional
waterpoint_type				
cattle trough		0.724138	0.017241	0.258621
communal standpipe		0.621485	0.079237	0.299278
communal standpipe multiple		0.366213	0.106177	0.527609
dam		0.857143	0.000000	0.142857
hand pump		0.617852	0.058840	0.323307
improved spring		0.718112	0.108418	0.173469
other		0.131661	0.045925	0.822414

Figure 21. Analysis of "waterpoint_type" column using a Pivot Table

In the 'Year_Range' category, the analysis exposes that a substantial portion—over 50%—of non-functional water pumps originates from the decades spanning the 60s, 70s, and 80s. Furthermore, a significant proportion, also exceeding 50%, is associated with the subsequent decades of the 90s, 00s, and 10s. This unveils a notable correlation between the age of water pumps and the likelihood of non-functionality, with older pumps exhibiting a higher probability of dysfunction.

```
[228]: pd.crosstab(water_table['Year_Range'], water_table['status_group'], normalize = 'index')
```

```
[228]:
```

	status_group	functional	functional needs repair	non functional
Year_Range				
00s		0.651598	0.063731	0.284671
10s		0.735129	0.042627	0.222244
60s		0.289963	0.078067	0.631970
70s		0.319110	0.078983	0.601906
80s		0.397992	0.075834	0.526174
90s		0.539073	0.067465	0.393462
unknown		0.509682	0.086388	0.403931

Figure 22. Analysis of "Year_Range" column using a Pivot Table

5) Analyze by Pivoting Features

After examining the categorical variables, it was time to shift the focus to the numerical ones. The graphic for "amount_tsh" assesses the quantity of water available to the waterpoint and categorizes whether a water pump is functional, non-functional, or in need of repair. Notably, if the total static head of water exceeds 150,000, it is more likely that the pump is functional. Conversely, if the value is less than that threshold, it suggests the possibility of water pumps needing repair or being non-functional.

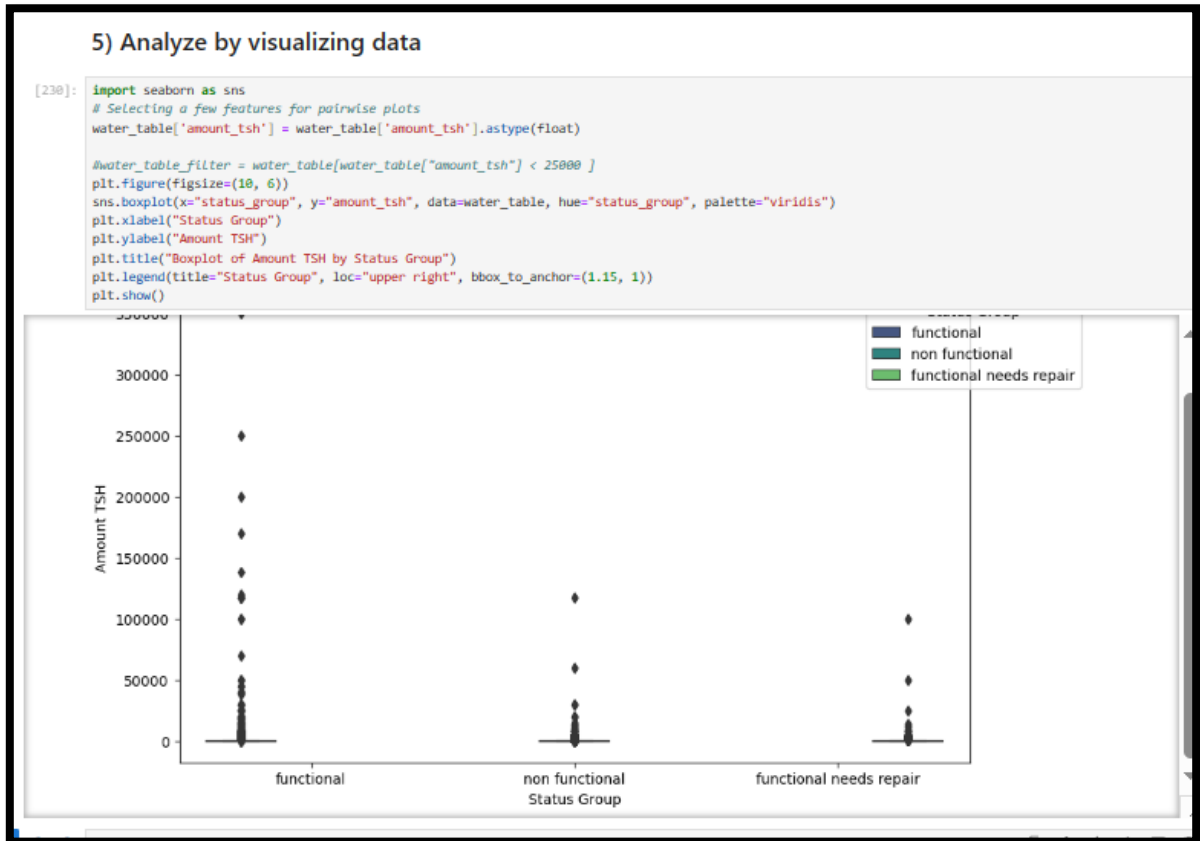


Figure 23. Visualization through Boxplot for the variable "amount_tsh".

Altitude could potentially serve as a determinant for the functionality status of a water pump in the column "gps_height". Notably, functional water pumps exhibit an average altitude close to 500. This observation suggests a heightened likelihood of functionality for wells situated at similar altitudes.

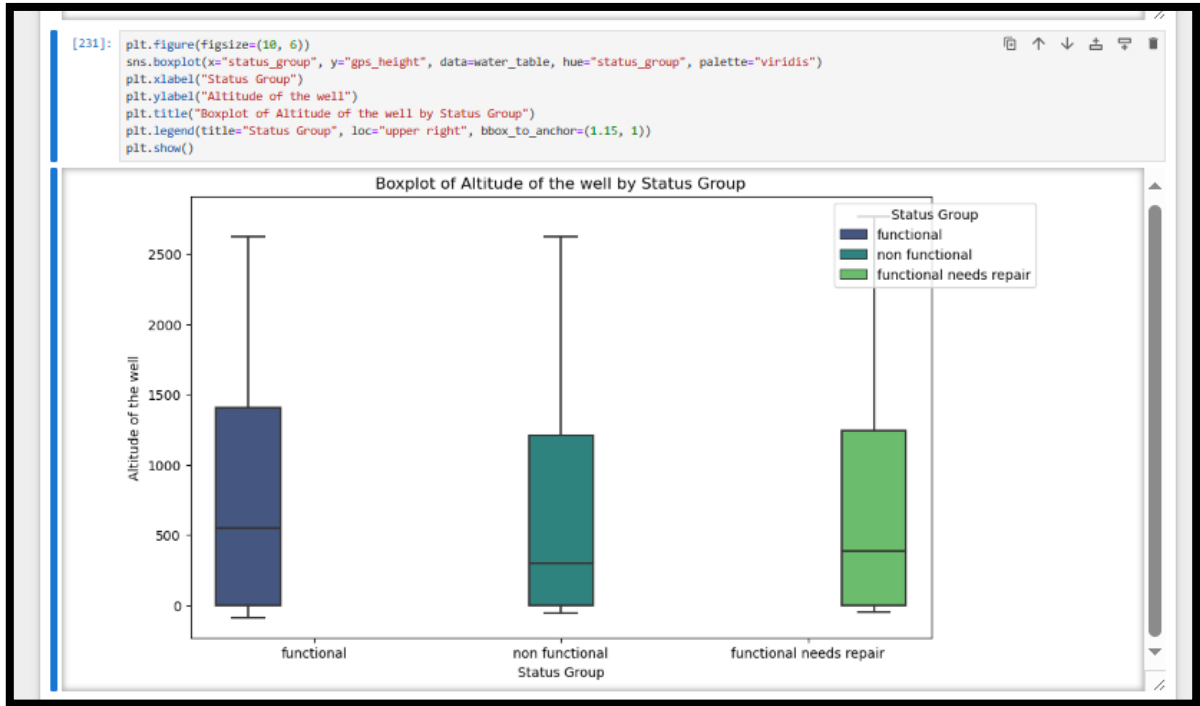


Figure 24. Visualization through Boxplot for the variable "gps_height".

Upon analyzing the population graphic, it becomes evident that this variable might be dispensable. The graphic indicates that the population's quantity does not conclusively determine the functionality status of water pumps—whether they are functional, non-functional, or in need of repair. The values and the mean are distributed evenly, suggesting limited discriminatory power in predicting pump status based on population alone.

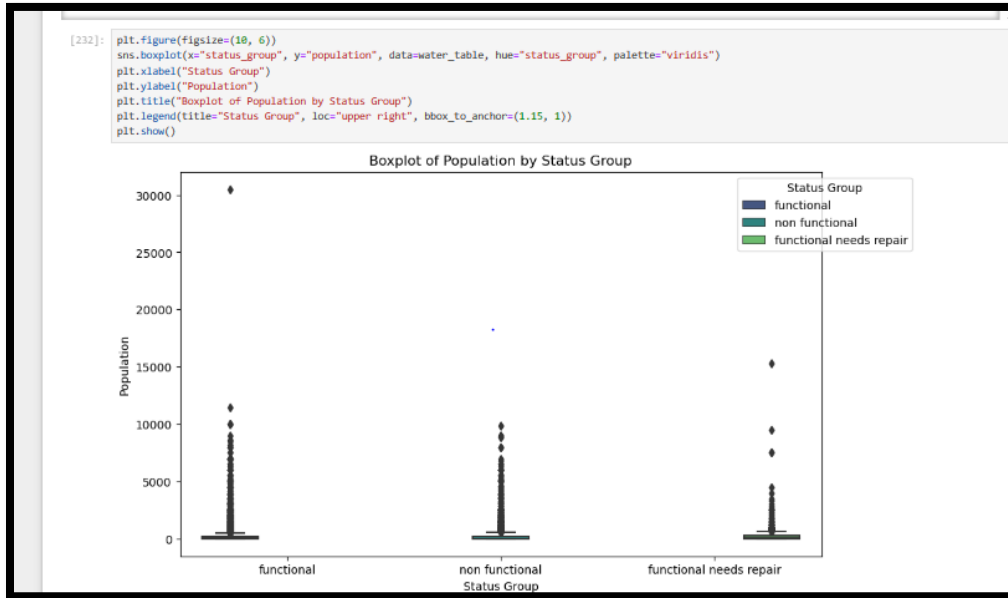


Figure 25. Visualization through Boxplot for the variable "population".

Based on the graphic, it appears that the "days_since_recorded" variable does not contribute significantly to distinguishing the functionality status of water pumps—whether they are functional, non-functional, or in need of repair—based on the recorded days in the report. Consequently, this column emerges as another potential candidate for removal.

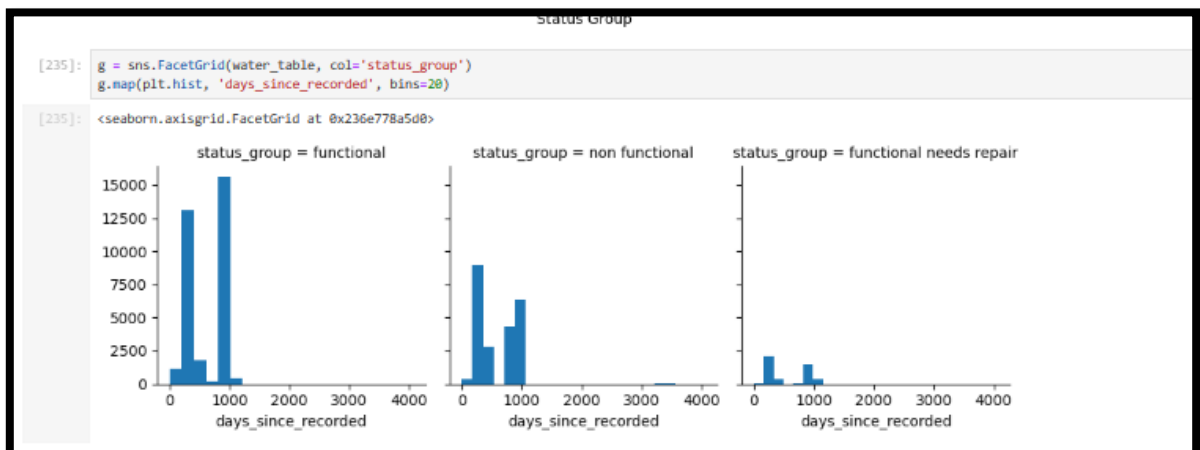


Figure 26. Visualization through Histogram for the variable "days_since_recorded".

Several variables were excluded from the analysis due to various reasons. The criteria for removal included considerations of irrelevance, redundancy, or the presence of an excessive number of blank spaces, which could potentially impact the model's results. The decision to eliminate these variables was driven by the goal of optimizing the model's performance and focusing on the most informative and meaningful features for the analysis. This refinement process aims to enhance the model's efficiency and interpretability while reducing unnecessary complexity and noise in the dataset.



```
[239]: water_table = water_table.drop(['date_recorded', 'funder', 'installer', 'wpt_name', 'num_private', 'subvillage', 'region_code', 'district_code',
                                     'lga', 'ward', 'public_meeting', 'recorded_by', 'scheme_name', 'permit', 'construction_year', 'extraction_type_group',
                                     'management', 'management_group', 'payment', 'payment_type', 'quality_group', 'quantity_group', 'source_type', 'source_class',
                                     'waterpoint_type_group', 'Population_Range', 'normalized_population', 'population'], axis=1)
```

Figure 27. Removal of Irrelevant Variables.

The Correlation Heatmap revealed that there are no strong negative or positive correlations among the variables: "amount_tsh," 'gps_height,' 'population,' and 'days_since_recorded.' These variables appear to be independent of each other.



Figure 28. Correlation Heatmap

Specific regions within the "status_group" exhibit varying counts of functional, non-functional, or pumps in need of repair. To visualize this, a graphic was generated using latitude (Y values) and longitude (X values) to highlight these geographical zones. For clarity, the data was filtered to include only rows where latitude is less than 0 (presumably in the Southern Hemisphere) and longitude is greater than 0 (presumably in the Eastern Hemisphere).

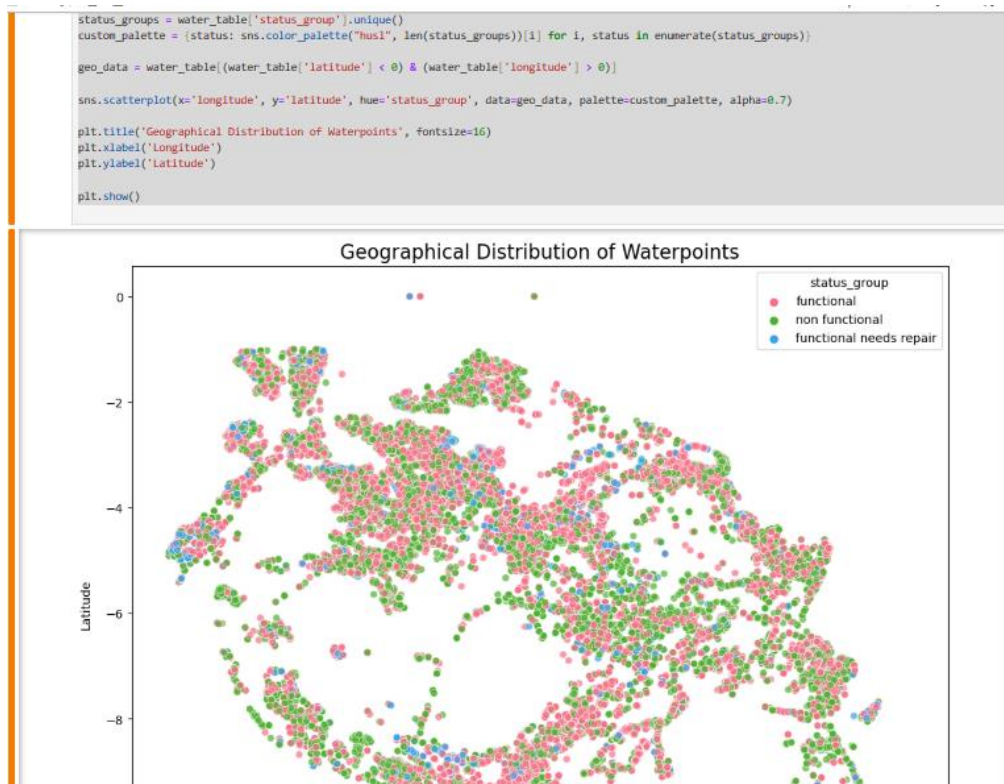


Figure 29. GeoMap for the "Latitude" and "Longitude" of each region.

6) Converting Categorical Features to Numeric

To enhance the model's performance, I transformed categorical features into numeric representations. The primary motivation behind this process is that numeric representations facilitate distance calculations between instances, making it more seamless for certain algorithms to handle categorical features. Consequently, the conversion of categorical features into numeric forms expands the spectrum of algorithms that can be applied.

The initial focus was on the dependent variable "num_status," which categorizes water pumps as functional, non-functional, or in need of repair. Employing label encoding, I assigned the label "2" for functional, "1" for functional but in need of repair, and "0" for non-functional. Utilizing the "map" function, I systematically associated each categorical value with its corresponding integer label.

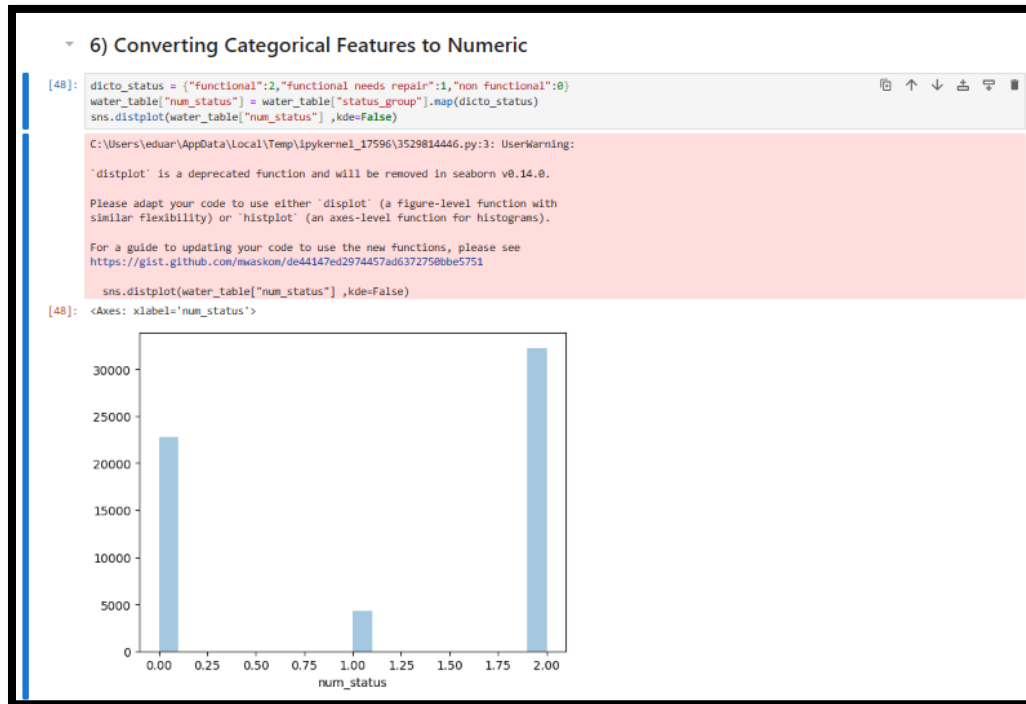


Figure 30. Transforming the Dependent Variable into Numeric Values

For the remaining independent variables, I applied One-Hot Encoding using the Sklearn library. This process generated binary columns for each category, signifying the presence or absence of each value. These binary columns play a crucial role in determining the status of the water pumps.

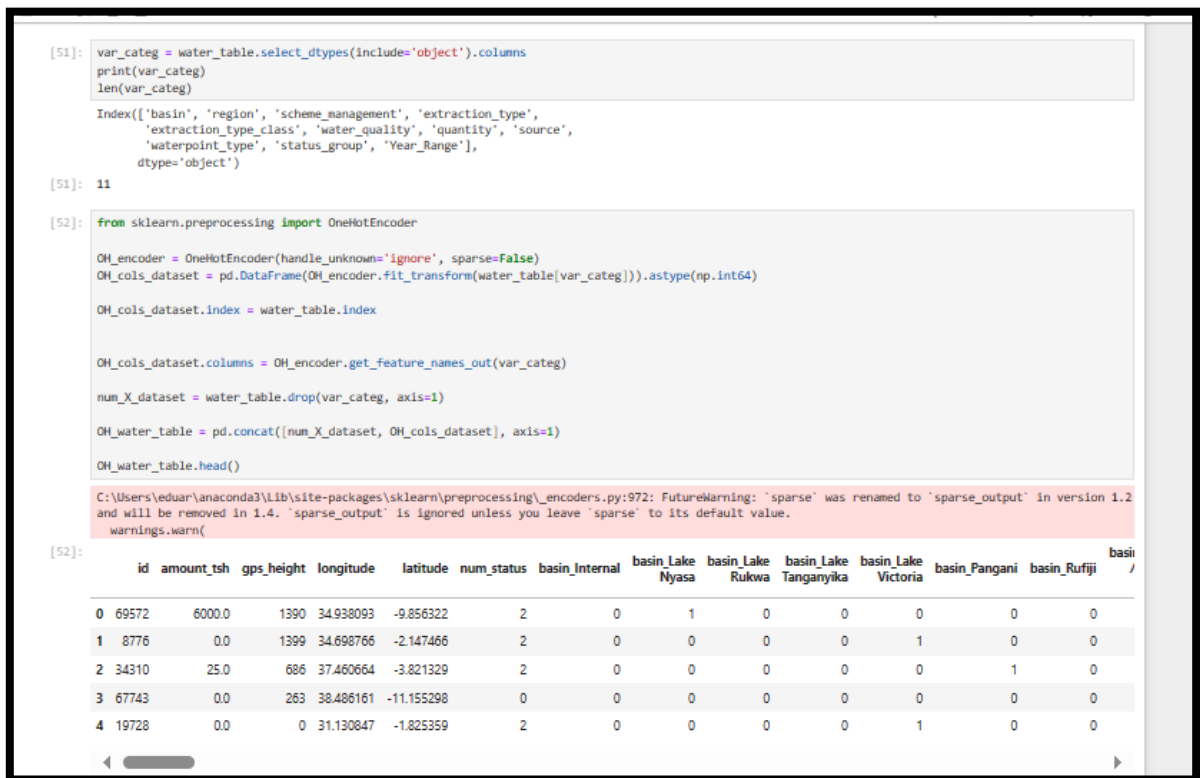


Figure 31. Application of One-Hot Encoding for Transforming Independent Variables into Binary Format.

The dependent variables were influenced by this operation; hence, I opted to exclude them from the new table designated as "OH_water_table," which will be utilized in my modeling process.

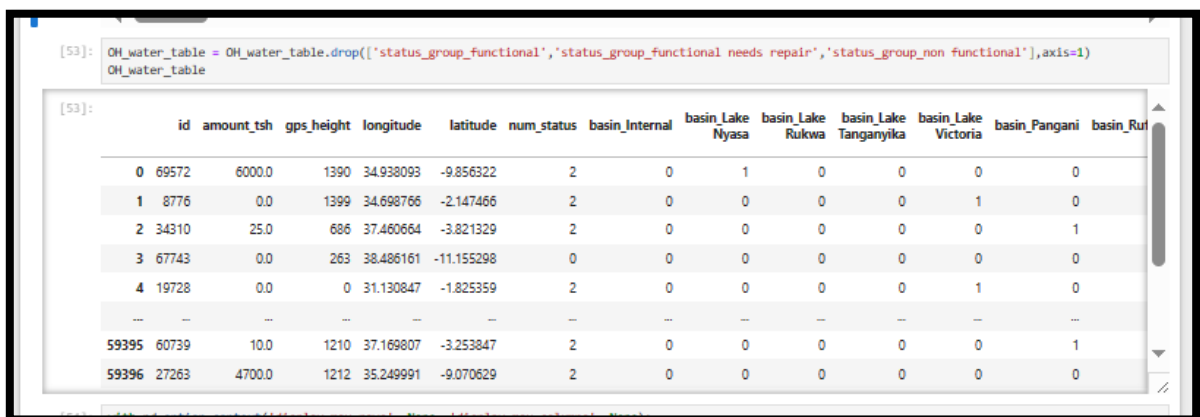


Figure 32. Dataset with Numeric Values

All variables are now defined as integers or floats, indicating that the dataset is nearly prepared for modeling.

```
[54]: with pd.option_context('display.max_rows', None, 'display.max_columns', None):
      print(OH_water_table.dtypes)
```

id	int64
amount_tsh	float64
gps_height	int64
longitude	float64
latitude	float64
num_status	int64
basin_Internal	int64
basin_Lake Nyasa	int64
basin_Lake Rukwa	int64
basin_Lake Tanganyika	int64
basin_Lake Victoria	int64
basin_Pangani	int64
basin_Rufiji	int64
basin_Ruvuma / Southern Coast	int64
basin_Wani / Ruvu	int64
region_Arusha	int64
region_Dar es Salaam	int64
region_Dodoma	int64
region_Iringa	int64
region_Kagera	int64
region_Kigoma	int64
region_Kilimanjaro	int64
region_Lindi	int64
region_Manyara	int64
region_Mara	int64
region_Mbeya	int64
region_Morogoro	int64
region_Mtwara	int64
region_Mwanza	int64
region_Pwani	int64

Figure 33. Variables defined as Integers and Floats

7) Split the Dataset into Training and Testing Sets

The dataset was prepared for division into training and testing sets. The variable X was defined for independent variables, excluding the columns "id" and "num_status." The "status" column, representing the target variable, was assigned to the variable Y. Utilizing the train_test_split function, the features (X) and the target variable (Y) were split into training and testing sets. The test_size parameter, set to 0.2, specifies that 20% of the data will be allocated for testing. The random_state parameter is set to 42 for reproducibility, ensuring consistent splits if the code is run multiple times.

```
7) Split the Dataset into Training and Testing Sets

[55]: from sklearn.model_selection import train_test_split
      X = OH_water_table.drop(['id', 'num_status'], axis=1)
      Y = OH_water_table['num_status']
      X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)
```

Figure 34. Splitting the Dataset into Training and Testing Sets.

8) Balance the Dataset

This marks the last step before modeling, wherein two crucial functions, SMOTE and MinMax Scaler, are employed. Let's delve into the role of each and the rationale behind their use.

Upon inspecting the percentage distribution for each status of the water pumps, a noteworthy observation is the relatively low percentage associated with the "functional need repair" category compared to the other two. This presents a potential challenge for the model, as an imbalanced dataset may lead it to overlook this minority class. To address this, the SMOTE function comes into play, generating synthetic samples for the minority class and thereby mitigating the issue of class imbalance.

```
[56]: count_by_status = water_table.groupby('status_group')['num_status'].count()
      percentage_by_status = (count_by_status / count_by_status.sum()) * 100
      print(percentage_by_status)

status_group
functional      54.388881
functional needs repair  7.267677
non functional  38.424242
Name: num_status, dtype: float64
```

Figure 35. Disbalance of Dependent Variables

SMOTE operates by generating synthetic instances of the minority class through interpolation between existing minority class instances. This process effectively balances the class distribution, enhancing the model's capacity to learn from the minority class. As depicted in the figure below, there is a noticeable difference between the pre- and post-application of the function. The three variables are now balanced and primed for utilization.

8) Balance the Dataset

```
[56]: from imblearn.over_sampling import SMOTE

print('Class Distribution Before:')
print('Train Set')
print(Y_train.value_counts())

smote = SMOTE()
X_train_resampled, Y_train_resampled = smote.fit_resample(X_train, Y_train)

print('\nClass Distribution After:')
print('Train Set')
print(pd.Series(Y_train_resampled).value_counts())
```

Class Distribution Before:
Train Set
num_status
2 25802
0 18252
1 3466
Name: count, dtype: int64

Class Distribution After:
Train Set
num_status
2 25802
0 25802
1 25802
Name: count, dtype: int64

Figure 36. Balancing Dependent Variables Using the SMOTE Function

I employ the Min-Max Scaler function primarily due to the substantial magnitude of float variables. This function involves subtracting the minimum value of each feature and dividing it by the range (the difference between the maximum and minimum values). The objective is to bring all features to a comparable scale, preventing any feature from dominating the learning process solely based on its larger magnitude.

```
[57]: from sklearn.preprocessing import MinMaxScaler

X_train_resampled, X_test, Y_train_resampled, Y_test = train_test_split(X, Y, test_size = 0.2, random_state = 21)

scale = MinMaxScaler()

X_train_resampled = scale.fit_transform(X_train_resampled)
X_test = scale.transform(X_test)
```

Figure 37. Scaling of Independent Variables

9) Model, Predict and Solve

At this point, the critical task was to choose a suitable model for this multinomial distribution. The ideal model needed to effectively capture the membership of water pumps in one of three statuses based on known variables. For this classification scenario, several popular models commonly used for multinomial classification tasks were considered:

- **Multinomial Logistic Regression:** A natural choice for modeling multinomial distributions, extending binary logistic regression to handle multiple classes. Implementation can be done using libraries like Scikit-Learn's LogisticRegression with the 'multinomial' option for the multi_class parameter.
- **Random Forest:** Known for its versatility in handling classification tasks. Random forests can effectively manage multinomial distributions, excelling at capturing complex relationships within the data.
- **K-Nearest Neighbors (KNN):** This algorithm classifies data points based on the majority class among their k-nearest neighbors. Trained on a labeled dataset, it uses input features to learn the mapping between inputs and corresponding class labels.
- **Neural Networks (MLP Classifier):** Capable of learning intricate patterns and non-linear relationships in the data, especially suitable for large and high-dimensional datasets. While neural networks, including MLP, excel in handling large datasets, they may require more data to prevent overfitting.

Upon selecting these models, I proceeded to evaluate each one, obtaining crucial results that will inform the subsequent steps.

9.1. Multinomial Logistic Regression

The Multinomial Logistic Regression demonstrates an accuracy of 71.98%. A comprehensive classification report provides insights into precision, recall, F1-score, and support. Cross-validation results indicate a consistent accuracy range between

71% and 72%, with an average accuracy of 72%. While the model's performance is favorable, it doesn't stand out as one of the top-performing results.

```
[58]: from sklearn.linear_model import LogisticRegression
      from sklearn.metrics import classification_report
      from sklearn.model_selection import cross_val_score

      logreg_multinomial = LogisticRegression(solver="lbfgs", multi_class="multinomial", max_iter=9500)
      logreg_multinomial.fit(X_train_resampled, Y_train_resampled)
      Y_pred1 = logreg_multinomial.predict(X_test)

      acc_logreg_multinomial = round(logreg_multinomial.score(X_train_resampled, Y_train_resampled) * 100, 2)

      print("Accuracy:", acc_logreg_multinomial)
      print("\nClassification Report:\n", classification_report(Y_test, Y_pred1))

Accuracy: 71.98

Classification Report:
              precision    recall  f1-score   support

     0       0.78       0.59       0.67       4614
     1       0.30       0.02       0.04        832
     2       0.70       0.90       0.78       6434

   accuracy                   0.72       11800
  macro avg       0.59       0.50       0.50       11800
 weighted avg       0.70       0.72       0.69       11800

[59]: scores_logreg_multinomial = cross_val_score(logreg_multinomial, X_train_resampled, Y_train_resampled, cv=5)
      scores_logreg_multinomial

[59]: array([0.72106481, 0.7165404 , 0.7170665 , 0.71475168, 0.72159091])

[60]: print("%0.2f accuracy with a standard deviation of %0.2f" % (scores_logreg_multinomial.mean(), scores_logreg_multinomial.std()))

0.72 accuracy with a standard deviation of 0.00
```

Figure 38. Accuracy of Multinomial Logistic Regression

9.2. Random Forest

Following a similar procedure, the accuracy for this model reached 79.44%. The detailed classification report highlights elevated regular precision, recall, F1-score, and support metrics. Cross-validation results consistently show accuracy per split in the range of 77% to 78%, with an average accuracy of 78%. Notably, this model achieved the highest performance compared to the other models.



Figure 39. Accuracy of Random Forest

9.3. K-Nearest Neighbors (KNN)

In the initial iteration, KNN achieved the highest accuracy at 86.56%. However, a closer examination through the classification report revealed that it had the lowest regular precision, recall, F1-score, and support compared to Random Forest. Cross-validation results consistently indicated accuracy per split in the range of 76% to 77%, averaging at 77%. While the model's performance was commendable, it did not surpass the performance of the best model.

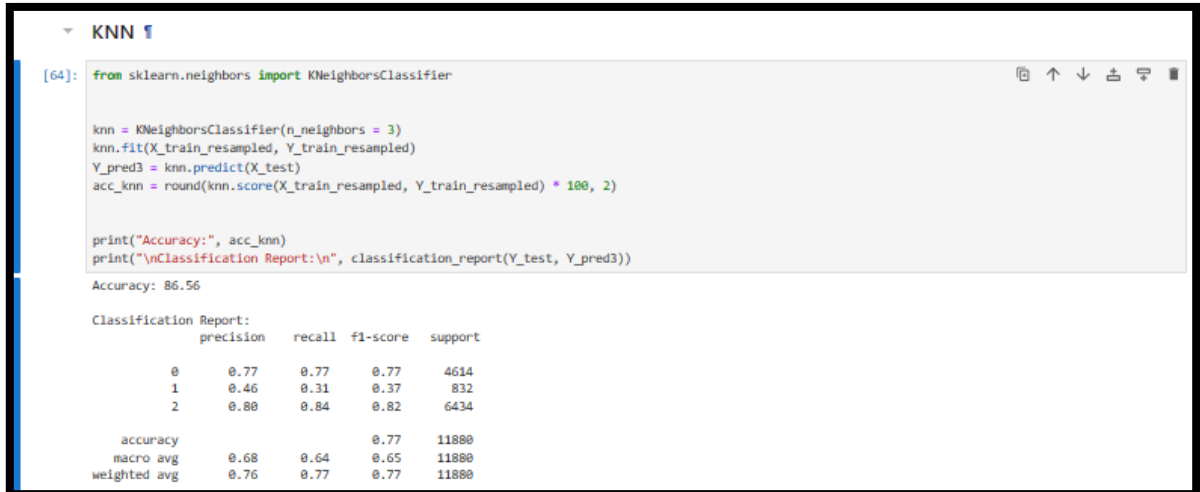


Figure 40. Accuracy of K-Nearest Neighbors (KNN)

9.4. Neural Networks (MLP Classifier)

Much like KNN, Neural Networks (MLP Classifier) displayed a commendable level of accuracy. However, a closer inspection of the classification report revealed that it exhibited the lowest regular precision, recall, F1-score, and support in comparison to the other models. The cross-validation results consistently showed accuracy per split within the range of 75% to 76%, with an average of 77%. While the model's performance was noteworthy, it did not surpass the results achieved by the best-performing model.

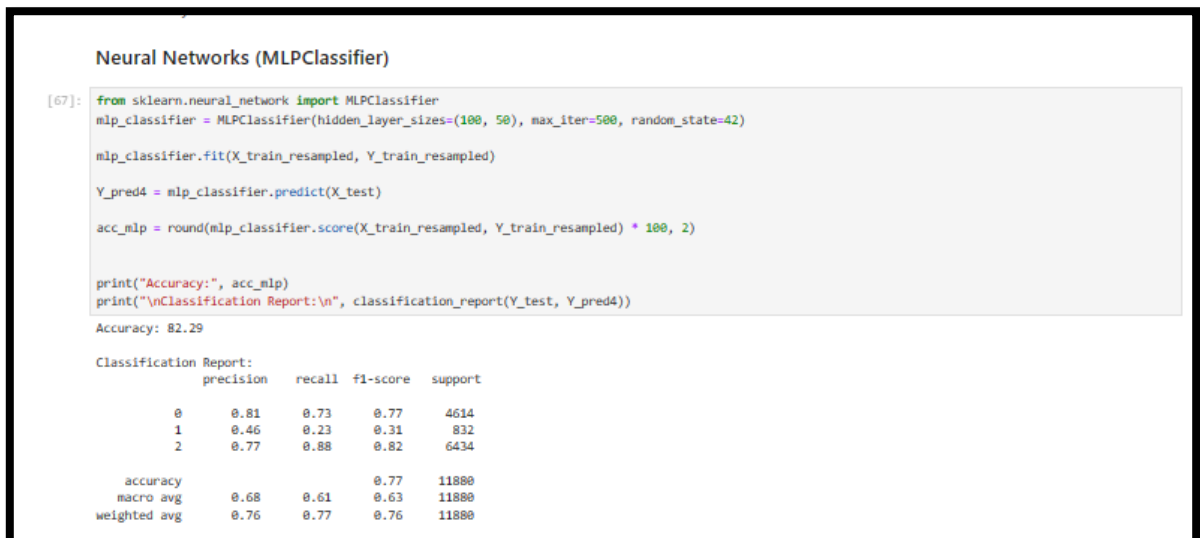


Figure 41. Accuracy of Neural Networks (MLP Classifier)

The average accuracy results for each model have been gathered and summarized in the model evaluation. A comprehensive overview of each result is provided, with the Random Forest model achieving the highest score at 78%.

▼ **Model Evaluation**

```

71]: models = pd.DataFrame({
      'Model': ['Multinomial Logistic Regression', 'Random Forest', 'KNN', 'Neural Networks'],
      'Score': [scores_logreg_multinomial.mean(), scores_random_forest.mean(), scores_knn.mean(),
               scores_mlp_classifier.mean()]})
      models.sort_values(by='Score', ascending=False)
71]:

```

	Model	Score
1	Random Forest	0.782618
2	KNN	0.767235
3	Neural Networks	0.762858
0	Multinomial Logistic Regression	0.718203

Figure 42. Model Evaluation Table

To validate the results, a Correlation Matrix was generated using one of the functions from Sklearn. The predictions outputted by the Random Forest model were examined, yielding the following outcomes:

- 3,594 water pumps were accurately predicted as 'Non-Functional.'
- 312 water pumps were correctly predicted as 'Functional Needs Repair.'
- 5,532 water pumps were accurately predicted as 'Functional,' aligning with the labels in the "Y_test" file derived from the cleaned data.

Out of a total of 59,400 values, less than 950 were inaccurately classified. This performance indicates a robust result for the Random Forest model.

Correlation Matrix

```
[72]: from sklearn.metrics import confusion_matrix

class_labels = ["No-Functional", "Needs Repair", "Functional"]

# Confusion Matrix
confu_matrix = confusion_matrix(Y_test, Y_pred2)

plt.figure(figsize=(6, 4))
sns.heatmap(confu_matrix, annot=True, fmt="d", cmap="Blues", cbar=True, xticklabels=class_labels, yticklabels=class_labels)

plt.title('Confusion Matrix')
plt.xlabel('Actual')
plt.ylabel('Model Prediction')

plt.show()
```

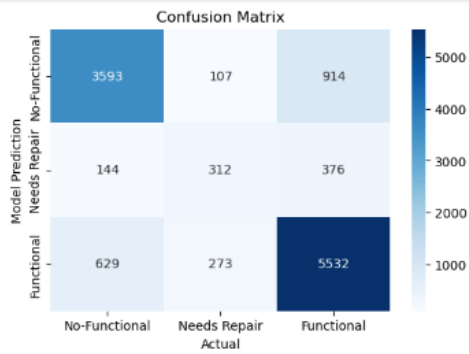


Figure 43. Correlation Matrix

Conclusion

In conclusion, this project successfully transforms raw data into valuable insights, enabling the prediction of the operational condition of water points in Tanzania. Leveraging essential tools such as pandas, matplotlib, numpy, and sklearn, the project identifies and addresses redundant or anomalous data, which is pivotal for accurate predictions in this context. The strategic application of these tools streamlines the analysis, ultimately enhancing precision and reliability in predictive models.