

# Table of Contents

- [1 Introduction](#)
- [2 Business Problem](#)
- [3 Data Understanding](#)
  - [3.1 Initial Observations](#)
    - [3.1.1 Missing Values](#)
    - [3.1.2 Data types](#)
    - [3.1.3 Duplicate and Similar Data](#)
- [4 Data Cleaning](#)
  - [4.1 Drop duplicate and columns with similar information](#)
  - [4.2 Dealing with null values](#)
  - [4.3 Investigate management and management\\_group](#)
  - [4.4 Construction year](#)
  - [4.5 Latitude/Longitude zeros](#)
  - [4.6 Installer - Several different spellings for same installer](#)
  - [4.7 Reduce dimensionality for installer](#)
- [5 Modified Features Exploration](#)
  - [5.1 Column EDA](#)
    - [5.1.1 Construction year](#)
    - [5.1.2 Population](#)
    - [5.1.3 Extraction type class](#)
    - [5.1.4 Management](#)
    - [5.1.5 Payment type](#)
    - [5.1.6 Water quality](#)
    - [5.1.7 Source type](#)
    - [5.1.8 Basin](#)
    - [5.1.9 Quantity](#)
    - [5.1.10 Region](#)
    - [5.1.11 Waterpoint type](#)
    - [5.1.12 Installer](#)
  - [5.2 Well Function map](#)
  - [5.3 Change status\\_group to integer format](#)
- [6 Modeling](#)
  - [6.1 Data Preprocessing](#)
    - [6.1.1 Create dummies](#)
    - [6.1.2 Separate target and perform train test split](#)
    - [6.1.3 Model Statistics Function](#)
  - [6.2 Dummy Classifier Model](#)
  - [6.3 Logistic Regression](#)
  - [6.4 K Nearest Neighbors](#)
  - [6.5 Decision Tree Model](#)
    - [6.5.1 Function to plot feature importances](#)
    - [6.5.2 Decision Tree Feature Importances](#)
  - [6.6 Random Forests](#)
    - [6.6.1 Random Forests Feature Importances](#)
  - [6.7 XG Boost](#)

- [6.7.1 XGB Feature Importances](#)
- [6.7.2 SMOTE and XGBoost](#)
- [6.8 ROC AUC Analysis](#)
- [7 Conclusions](#)

## ▼ Tanzanian Water Wells Status Prediction

By Melody Bass



## ▼ 1 Introduction

Tanzania is a developing country that struggles to get clean water to its population of 59 million people. The focus of this project is to build a classification model to predict the functionality of waterpoints in Tanzania given data provided by Taarifa and the Tanzanian Ministry of Water. The model was built from a dataset containing information about the source of water and status of the waterpoint (functional, functional but needs repairs, and non functional) using an iterative approach and can be found [here](#) ([./data/training\\_set\\_values.csv](#)). The dataset contains 60,000 waterpoints in Tanzania and the following features:

- `amount_tsh` - Total static head (amount water available to waterpoint)
- `date_recorded` - The date the row was entered
- `funder` - Who funded the well
- `gps_height` - Altitude of the well
- `installer` - Organization that installed the well
- `longitude` - GPS coordinate

- latitude - GPS coordinate
- wpt\_name - Name of the waterpoint if there is one
- num\_private -
- basin - Geographic water basin
- subvillage - Geographic location
- region - Geographic location
- region\_code - Geographic location (coded)
- district\_code - Geographic location (coded)
- lga - Geographic location
- ward - Geographic location
- population - Population around the well
- public\_meeting - True/False
- recorded\_by - Group entering this row of data
- scheme\_management - Who operates the waterpoint
- scheme\_name - Who operates the waterpoint
- permit - If the waterpoint is permitted
- construction\_year - Year the waterpoint was constructed
- extraction\_type - The kind of extraction the waterpoint uses
- extraction\_type\_group - The kind of extraction the waterpoint uses
- extraction\_type\_class - The kind of extraction the waterpoint uses
- management - How the waterpoint is managed
- management\_group - How the waterpoint is managed
- payment - What the water costs
- payment\_type - What the water costs
- water\_quality - The quality of the water
- quality\_group - The quality of the water
- quantity - The quantity of water
- quantity\_group - The quantity of water
- source - The source of the water
- source\_type - The source of the water
- source\_class - The source of the water
- waterpoint\_type - The kind of waterpoint
- waterpoint\_type\_group - The kind of waterpoint

The first sections focus on investigating, cleaning, wrangling, and reducing dimensionality for modeling. The next section contains 6 different classification models and evaluation of each, ultimately leading to us to select our best model for predicting waterpoint status based on the accuracy of the model. Finally, I will make recommendations to the Tanzanian Government and provide insight on predicting the status of waterpoints.

## ▼ 2 Business Problem

Tanzania is a developing country that struggles to provide its 59 million people with access to clean drinking water. DrivenData has started a competition to help officials predict statuses of waterpoints by building a model to predict which pumps are functional, functional but need repair, and non functional. The data is provided by Taarifa and the Tanzanian Ministry of Water with the

hope that the information provided by each waterpoint can aid understanding in which waterpoints will fail to improve the government's maintenance operations and ensure that it's residents have access to safe drinking water.

## ▼ 3 Data Understanding

The dataset used for this analysis can be found [here](#) (`./data/training_set_values.csv`). It contains a wealth of information about waterpoints in Tanzania and the status of their operation. The target variable has 3 different options for it's status:

- functional - the waterpoint is operational and there are no repairs needed
- functional needs repair - the waterpoint is operational, but needs repairs
- non functional - the waterpoint is not operational

Below I will import the dataset and start my investigation of relevant information it may contain. Let's get started!

```
In [1]: # Import standard packages
1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5 %matplotlib inline
6
7 import warnings
8 warnings.filterwarnings("ignore")
9
10 from sklearn.model_selection import train_test_split, GridSearchCV, c
11 from sklearn.pipeline import Pipeline
12 from imblearn.over_sampling import SMOTE
13
14 # Classification Models
15 from sklearn.linear_model import LogisticRegression
16 from sklearn.tree import DecisionTreeClassifier
17 from sklearn.ensemble import RandomForestClassifier
18 from sklearn.neighbors import KNeighborsClassifier
19 import xgboost as xgb
20 from sklearn.dummy import DummyClassifier
21 from xgboost.sklearn import XGBClassifier
22
23
24 from sklearn.metrics import plot_confusion_matrix, accuracy_score, f1_
25 from sklearn.metrics import roc_curve, auc, roc_auc_score
26
27 # Scalers
28 from sklearn.impute import SimpleImputer
29 from sklearn.preprocessing import StandardScaler, label_binarize
30
31 # Categorical Create Dummies
32 from sklearn.preprocessing import OneHotEncoder
```

```
In [2]: ▾ 1 # Data Import Train Set
2 df_train_set = pd.read_csv('data/training_set_values.csv', index_col=
3 df_train_set
```

Out[2]:

	amount_tsh	date_recorded	funder	gps_height	installer	longitude	latitude	wpt_name
	id							
69572	6000.0	2011-03-14	Roman	1390	Roman	34.938093	-9.856322	none
8776	0.0	2013-03-06	Grumeti	1399	GRUMETI	34.698766	-2.147466	Zahanat
34310	25.0	2013-02-25	Lottery Club	686	World vision	37.460664	-3.821329	Kwae Mahund
67743	0.0	2013-01-28	Unicef	263	UNICEF	38.486161	-11.155298	Zahanat Yé Nanyumbu
19728	0.0	2011-07-13	Action In A	0	Artisan	31.130847	-1.825359	Shulen
...	...	...	...	...	...	...	...	...

```
In [3]: ▾ 1 # Data import Training set labels
2 df_train_labels = pd.read_csv('data/training_set_labels.csv', index_col=
3 df_train_labels
```

Out[3]:

	status_group
	id
69572	functional
8776	functional
34310	functional
67743	non functional
19728	functional
...	...
60739	functional
27263	functional
37057	functional
31282	functional
26348	functional

59400 rows × 1 columns

```
In [4]: ▾ 1 #Merge datasets
2 df = pd.merge(df_train_labels, df_train_set, how = 'inner', on='id')
```

```
In [5]: ▾ 1 #Reset index
  2 df.reset_index(inplace=True)
  3 df.head()
```

Out[5]:

	id	status_group	amount_tsh	date_recorded	funder	gps_height	installer	longitude	latitude
0	69572	functional	6000.0	2011-03-14	Roman	1390	Roman	34.938093	-9.8
1	8776	functional	0.0	2013-03-06	Grumeti	1399	GRUMETI	34.698766	-2.1
2	34310	functional	25.0	2013-02-25	Lottery Club	686	World vision	37.460664	-3.8
3	67743	non functional	0.0	2013-01-28	Unicef	263	UNICEF	38.486161	-11.1
4	19728	functional	0.0	2011-07-13	Action In A	0	Artisan	31.130847	-1.8

5 rows × 41 columns

```
In [6]: ▾ 1 # Check datatypes
  2 df.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   status_group     59400 non-null   object  
 2   amount_tsh       59400 non-null   float64 
 3   date_recorded   59400 non-null   object  
 4   funder           55765 non-null   object  
 5   gps_height       59400 non-null   int64  
 6   installer        55745 non-null   object  
 7   longitude        59400 non-null   float64 
 8   latitude         59400 non-null   float64 
 9   wpt_name         59400 non-null   object  
 10  num_private     59400 non-null   int64  
 11  basin            59400 non-null   object  
 12  subvillage       59029 non-null   object  
 13  region           59400 non-null   object  
 ..  

```

```
In [7]: ▾ 1 #Get stats on numeric columns
  2 df.describe()
```

Out[7]:

	<b>id</b>	<b>amount_tsh</b>	<b>gps_height</b>	<b>longitude</b>	<b>latitude</b>	<b>num_private</b>	<b>reg</b>
<b>count</b>	59400.000000	59400.000000	59400.000000	59400.000000	5.940000e+04	59400.000000	5940
<b>mean</b>	37115.131768	317.650385	668.297239	34.077427	-5.706033e+00	0.474141	1
<b>std</b>	21453.128371	2997.574558	693.116350	6.567432	2.946019e+00	12.236230	1
<b>min</b>	0.000000	0.000000	-90.000000	0.000000	-1.164944e+01	0.000000	
<b>25%</b>	18519.750000	0.000000	0.000000	33.090347	-8.540621e+00	0.000000	
<b>50%</b>	37061.500000	0.000000	369.000000	34.908743	-5.021597e+00	0.000000	1
<b>75%</b>	55656.500000	20.000000	1319.250000	37.178387	-3.326156e+00	0.000000	1
<b>max</b>	74247.000000	350000.000000	2770.000000	40.345193	-2.000000e-08	1776.000000	9

```
In [8]: ▾ 1 #Check for duplicates
  2 sum(df.duplicated())
```

Out[8]: 0

```
In [9]:  
    1 # Print all value counts to make observations  
    2 for col in df.columns:  
        print(df[col].value_counts())  
  
2047      1  
72310     1  
49805     1  
51852     1  
62091     1  
...  
46396     1  
36155     1  
34106     1  
38200     1  
0         1  
Name: id, Length: 59400, dtype: int64  
functional          32259  
non functional      22824  
functional needs repair 4317  
Name: status_group, dtype: int64  
0.0            41639  
500.0          3102  
50.0           2472  
1000.0         1488  
20.0           1463  
...  
8500.0          1  
6300.0          1  
220.0           1  
138000.0        1  
12.0            1  
Name: amount_tsh, Length: 98, dtype: int64  
2011-03-15      572  
2011-03-17      558  
2013-02-03      546  
2011-03-14      520  
2011-03-16      513  
...  
2011-09-20      1  
2011-09-21      1  
2011-09-27      1  
2011-09-01      1  
2012-01-21      1  
Name: date_recorded, Length: 356, dtype: int64  
Government Of Tanzania 9084  
Danida          3114  
Hesawa          2202  
Rwssp           1374  
World Bank       1349  
...  
Yasi Naini      1  
Rodri           1  
Kassim          1  
Bhws            1  
Mtambo          1  
Name: funder, Length: 1897, dtype: int64  
0            20438
```

```
-15          60
-16          55
-13          55
-20          52
...
2285         1
2424         1
2552         1
2413         1
2385         1
Name: gps_height, Length: 2428, dtype: int64
DWE           17402
Government    1825
RWE           1206
Commu         1060
DANIDA        1050
...
JI            1
District Community j   1
SIMBA         1
Private company  1
Britain        1
Name: installer, Length: 2145, dtype: int64
0.000000     1812
37.540901     2
33.010510     2
39.093484     2
32.972719     2
...
37.579803     1
33.196490     1
34.017119     1
33.788326     1
30.163579     1
Name: longitude, Length: 57516, dtype: int64
-2.000000e-08  1812
-6.985842e+00  2
-3.797579e+00  2
-6.981884e+00  2
-7.104625e+00  2
...
-5.726001e+00  1
-9.646831e+00  1
-8.124530e+00  1
-2.535985e+00  1
-2.598965e+00  1
Name: latitude, Length: 57517, dtype: int64
none          3563
Shuleni        1748
Zahanati       830
Msikitini      535
Kanisani        323
...
Naftali Dembe   1
Matakani        1
Nyabikungu B    1
Kwa Benadeta Limo  1
```

```
Enannueli           1
Name: wpt_name, Length: 37400, dtype: int64
0      58643
6       81
1       73
5       46
8       46
...
180      1
213      1
23       1
55       1
94       1
Name: num_private, Length: 65, dtype: int64
Lake Victoria      10248
Pangani            8940
Rufiji             7976
Internal           7785
Lake Tanganyika    6432
Wami / Ruvu        5987
Lake Nyasa          5085
Ruvuma / Southern Coast  4493
Lake Rukwa          2454
Name: basin, dtype: int64
Madukani            508
Shulenzi           506
Majengo             502
Kati                373
Mtakuja            262
...
Majimaji Magengeni  1
Nyangoso            1
Kambi Ya Tanga Juu  1
Sasago               1
Mpalang'Ombe         1
Name: subvillage, Length: 19287, dtype: int64
Iringa              5294
Shinyanga           4982
Mbeya               4639
Kilimanjaro         4379
Morogoro            4006
Arusha              3350
Kagera              3316
Mwanza              3102
Kigoma              2816
Ruvuma              2640
Pwani               2635
Tanga               2547
Dodoma              2201
Singida              2093
Mara                1969
Tabora              1959
Rukwa               1808
Mtwara              1730
Manyara              1583
Lindi                1546
Dar es Salaam        805
```

```
Name: region, dtype: int64
11      5300
17      5011
12      4639
3       4379
5       4040
18      3324
19      3047
2       3024
16      2816
10      2640
4       2513
1       2201
13      2093
14      1979
20      1969
15      1808
6       1609
21      1583
80      1238
60      1025
90      917
7       805
99      423
9       390
24      326
8       300
40      1

Name: region_code, dtype: int64
1      12203
2      11173
3      9998
4      8999
5      4356
6      4074
7      3343
8      1043
30     995
33     874
53     745
43     505
13     391
23     293
63     195
62     109
60     63
0      23
80     12
67     6

Name: district_code, dtype: int64
Njombe        2503
Arusha Rural   1252
Moshi Rural    1251
Bariadi        1177
Rungwe         1106
...
Moshi Urban     79
```

```
Kigoma Urban      71
Arusha Urban      63
Lindi Urban       21
Nyamagana         1
Name: lga, Length: 125, dtype: int64
Igosi            307
Imalinyi          252
Siha Kati         232
Mdandu            231
Nduruma           217
...
Ikweha            1
Ifinga            1
Kinungu           1
Nyamtinga         1
Simbay            1
Name: ward, Length: 2092, dtype: int64
0                21381
1                7025
200              1940
150              1892
250              1681
...
3241              1
1960              1
1685              1
2248              1
1439              1
Name: population, Length: 1049, dtype: int64
True              51011
False             5055
Name: public_meeting, dtype: int64
GeoData Consultants Ltd    59400
Name: recorded_by, dtype: int64
VWC               36793
WUG               5206
Water authority   3153
WUA               2883
Water Board        2748
Parastatal         1680
Private operator   1063
Company            1061
Other              766
SWC                97
Trust              72
None               1
Name: scheme_management, dtype: int64
K                  682
None              644
Borehole           546
Chalinze wate     405
M                  400
...
Burieni water supply      1
Shimassawe branch line    1
Kitangwi Water Supply    1
Magoto piped water suplly 1
```

```
Namtibwili          1
Name: scheme_name, Length: 2696, dtype: int64
True      38852
False     17492
Name: permit, dtype: int64
0        20709
2010    2645
2008    2613
2009    2533
2000    2091
2007    1587
2006    1471
2003    1286
2011    1256
2004    1123
2012    1084
2002    1075
1978    1037
1995    1014
2005    1011
1999    979
1998    966
1990    954
1985    945
1980    811
1996    811
1984    779
1982    744
1994    738
1972    708
1974    676
1997    644
1992    640
1993    608
2001    540
1988    521
1983    488
1975    437
1986    434
1976    414
1970    411
1991    324
1989    316
1987    302
1981    238
1977    202
1979    192
1973    184
2013    176
1971    145
1960    102
1967     88
1963     85
1968     77
1969     59
1964     40
1962     30
```

```
1961      21
1965      19
1966      17
Name: construction_year, dtype: int64
gravity          26780
nira/tanira     8154
other            6430
submersible      4764
swn 80           3670
mono             2865
india mark ii    2400
afridev          1770
ksb              1415
other - rope pump 451
other - swn 81   229
windmill         117
india mark iii   98
cemo              90
other - play pump 85
walimi            48
climax            32
other - mkulima/shinyanga 2
Name: extraction_type, dtype: int64
gravity          26780
nira/tanira     8154
other            6430
submersible      6179
swn 80           3670
mono             2865
india mark ii    2400
afridev          1770
rope pump        451
other handpump   364
other motorpump  122
wind-powered     117
india mark iii   98
Name: extraction_type_group, dtype: int64
gravity          26780
handpump         16456
other            6430
submersible      6179
motorpump        2987
rope pump        451
wind-powered     117
Name: extraction_type_class, dtype: int64
vwc              40507
wug              6515
water board      2933
wua              2535
private operator  1971
parastatal       1768
water authority  904
other            844
company          685
unknown          561
other - school   99
trust             78
```

```
Name: management, dtype: int64
user-group      52490
commercial     3638
parastatal     1768
other          943
unknown         561
Name: management_group, dtype: int64
never pay        25348
pay per bucket   8985
pay monthly       8300
unknown          8157
pay when scheme fails 3914
pay annually      3642
other             1054
Name: payment, dtype: int64
never pay        25348
per bucket       8985
monthly          8300
unknown          8157
on failure        3914
annually          3642
other             1054
Name: payment_type, dtype: int64
soft              50818
salty             4856
unknown           1876
milky             804
coloured          490
salty abandoned    339
fluoride           200
fluoride abandoned 17
Name: water_quality, dtype: int64
good              50818
salty             5195
unknown           1876
milky             804
colored            490
fluoride           217
Name: quality_group, dtype: int64
enough            33186
insufficient      15129
dry                6246
seasonal           4050
unknown             789
Name: quantity, dtype: int64
enough            33186
insufficient      15129
dry                6246
seasonal           4050
unknown             789
Name: quantity_group, dtype: int64
spring             17021
shallow well       16824
machine dbh        11075
river               9612
rainwater harvesting 2295
hand dtw            874
```

```
lake          765
dam          656
other        212
unknown       66
Name: source, dtype: int64
spring        17021
shallow well  16824
borehole      11949
river/lake    10377
rainwater harvesting  2295
dam           656
other         278
Name: source_type, dtype: int64
groundwater   45794
surface       13328
unknown       278
Name: source_class, dtype: int64
communal standpipe  28522
hand pump     17488
other          6380
communal standpipe multiple  6103
improved spring  784
cattle trough   116
dam             7
Name: waterpoint_type, dtype: int64
communal standpipe  34625
hand pump     17488
other          6380
improved spring  784
cattle trough   116
dam             7
Name: waterpoint_type_group, dtype: int64
```

```
In [10]: ▾ 1 # Check null values
          2 df.isna().sum()
```

```
Out[10]: id                      0
status_group                  0
amount_tsh                     0
date_recorded                   0
funder                      3635
gps_height                     0
installer                      3655
longitude                      0
latitude                      0
wpt_name                      0
num_private                     0
basin                          0
subvillage                      371
region                          0
region_code                     0
district_code                   0
lga                            0
ward                           0
population                     0
public_meeting                 3334
recorded_by                     0
scheme_management                3877
scheme_name                     28166
permit                          3056
construction_year                0
extraction_type                  0
extraction_type_group            0
extraction_type_class             0
management                       0
management_group                  0
payment                          0
payment_type                     0
water_quality                    0
quality_group                     0
quantity                         0
quantity_group                     0
source                           0
source_type                      0
source_class                      0
waterpoint_type                  0
waterpoint_type_group              0
dtype: int64
```

```
In [11]: ▾ 1 # Check unique values for categorical data
  2 obj_df = df.select_dtypes(include=['object'])
  3 obj_df.unique()
```

```
Out[11]: status_group            3
date_recorded          356
funder                  1897
installer                2145
wpt_name                 37400
basin                      9
subvillage                19287
region                     21
lga                        125
ward                      2092
public_meeting              2
recorded_by                  1
scheme_management             12
scheme_name                 2696
permit                      2
extraction_type              18
extraction_type_group        13
extraction_type_class         7
management                  12
management_group               5
payment                      7
payment_type                  7
water_quality                  8
quality_group                  6
quantity                      5
quantity_group                  5
source                      10
source_type                  7
source_class                  3
waterpoint_type                  7
waterpoint_type_group           6
dtype: int64
```

## ▼ 3.1 Initial Observations

### ▼ 3.1.1 Missing Values

**scheme\_name** has the most missing values, followed by **funder**, **installer**, **public\_meeting**, **scheme\_management**, and **permit** with ~3,000 null values, and then **subvillage** with 371 null values. Several of these columns will be deleted as they appear to duplicate other columns, and I will investigate **installer**, **permit**, and **subvillage** further.

### 3.1.2 Data types

- **wpt\_name**, **subvillage**, **ward**, **scheme\_name**, **installer**, **funder**, and **date\_recorded** are categorical features that have unique values in the thousands. This will be a problem with dummy variables, will likely remove or feature engineer.
- I will drop **recorded\_by** as it has the same value for all rows.

- **num\_private** is not defined on the DrivenData site, and it is not obvious what the feature indicates.
- **id** column will be dropped.
- **public\_meeting** and **permit** are boolean.
- **construction\_year**, **latitude**, **longitude**, **gps\_height**, **\_amount\_tsh**, and **population** all have thousands of rows of 0 entered. I will drop rows for most of these variables that have 0 entered, and will have to investigate further for real data on some columns.

### 3.1.3 Duplicate and Similar Data

The following columns all contain duplicate or similar data, will remove features that will cause multicollinearity:

- **extraction\_type**, **extraction\_type\_group**, and **extraction\_type\_class**
- **payment** and **payment\_type**
- **water\_quality** and **quality\_group**
- **quanity** and **quantity\_group**
- **source** and **source\_type**
- **waterpoint\_type** and **waterpoint\_type\_group**
- **region** and **region\_code**

## 4 Data Cleaning

In this section, I will clean the dataset by removing similar and unnecessary columns and trim the dataset of remaining null values. I will also further investigate whether some columns contain the same information if it was not immediately obvious. There are several rows containing 0 enteries in some column information. I will investigate whether I believe the data to be real instead of a placeholder.

### 4.1 Drop duplicate and columns with similar information

I will keep **extraction\_type\_class** and remove **extraction\_type** and **extraction\_type\_group** as it's columns values appear to be the most relevant for the project. **scheme\_name** will be dropped for it's many null values. Other columns will be removed at this point due to irrelavancy, duplicates, null values, and some others will have to be investigated after the first drop.

```
In [12]: ▾ 1 # Columns to be dropped
      2 dropped_columns = ['extraction_type', 'extraction_type_group', 'paymen
      3           'quantity_group', 'source', 'waterpoint_type_group'
      4           'wpt_name', 'ward', 'funder', 'date_recorded', 'pub
      5           'scheme_management', 'source_class']
```

```
In [13]: 1 df = df.drop(dropped_columns, axis=1)
```

In [14]: 1 df.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 59400 entries, 0 to 59399
Data columns (total 19 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   status_group      59400 non-null   object  
 1   amount_tsh        59400 non-null   float64 
 2   gps_height        59400 non-null   int64  
 3   installer         55745 non-null   object  
 4   longitude         59400 non-null   float64 
 5   latitude          59400 non-null   float64 
 6   basin             59400 non-null   object  
 7   region            59400 non-null   object  
 8   population        59400 non-null   int64  
 9   permit             56344 non-null   object  
 10  construction_year 59400 non-null   int64  
 11  extraction_type_class 59400 non-null   object  
 12  management         59400 non-null   object  
 13  management_group   59400 non-null   object  
 14  payment_type       59400 non-null   object  
 15  water_quality      59400 non-null   object  
 16  quantity           59400 non-null   object  
 17  source_type        59400 non-null   object  
 18  waterpoint_type    59400 non-null   object  
dtypes: float64(3), int64(3), object(13)
memory usage: 8.6+ MB
```

## 4.2 Dealing with null values

In [15]: 1 #Check for nulls  
2 df.isna().sum()

```
Out[15]: status_group          0
amount_tsh              0
gps_height              0
installer               3655
longitude               0
latitude                0
basin                   0
region                  0
population              0
permit                  3056
construction_year        0
extraction_type_class   0
management              0
management_group         0
payment_type             0
water_quality            0
quantity                 0
source_type              0
waterpoint_type          0
dtype: int64
```

```
In [16]: 1 # Drop all remaining null values from our dataset
          2 df = df.dropna()
```

```
In [17]: 1 #Check to see that it worked
          2 df.isna().sum()
```

```
Out[17]: status_group  
amount_tsh  
gps_height  
installer  
longitude  
latitude  
basin  
region  
population  
permit  
construction_year  
extraction_type_class  
management  
management_group  
payment_type  
water_quality  
quantity  
source_type  
waterpoint_type  
dtype: int64
```

```
In [18]: 1 # Convert boolean permit to integers
2 df['permit'] = df['permit'].astype(int)
```

```
In [19]: 1 # Check to see that it worked
          2 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 55102 entries, 0 to 59399
Data columns (total 19 columns):
 #   Column           Non-Null Count   Dtype  
--- 
 0   status_group     55102 non-null    object  
 1   amount_tsh       55102 non-null    float64 
 2   gps_height      55102 non-null    int64  
 3   installer        55102 non-null    object  
 4   longitude        55102 non-null    float64 
 5   latitude         55102 non-null    float64 
 6   basin            55102 non-null    object  
 7   region           55102 non-null    object  
 8   population       55102 non-null    int64  
 9   permit            55102 non-null    int64  
 10  construction_year 55102 non-null    int64  
 11  extraction_type_class 55102 non-null    object  
 12  management        55102 non-null    object  
 13  management_group 55102 non-null    object  
 14  management_type  55102 non-null    object 
```

#### 4.3 Investigate management and management\_group

I need to investigate these 2 columns further to see if they contain similar information.

```
In [20]: 1 df['management'].value_counts()
```

```
Out[20]: vwc          37416
wug          6314
water board  2705
wua          2307
private operator 1891
parastatal    1588
water authority 825
other         733
company        656
unknown        491
other - school 99
trust          77
Name: management, dtype: int64
```

```
In [21]: 1 df['management_group'].value_counts()
```

```
Out[21]: user-group   48742
commercial     3449
parastatal     1588
other          832
unknown        491
Name: management_group, dtype: int64
```

The most data is contained in the user-group subcategory of **management\_group**. I will groupby to investigate if the information is similar.

```
In [22]: 1 df.loc[df['management_group']=='user-group']['management'].value_counts()
```

```
Out[22]: vwc          37416
wug          6314
water board  2705
wua          2307
Name: management, dtype: int64
```

The data is identical to the data contained in the management column in the subcategory of 'user-group'. I will drop **management\_group** from our features.

```
In [23]: ▾ 1 #Drop column
2 df = df.drop('management_group', axis=1)
```

In [24]:

```
1 #Check to see that it worked
2 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 55102 entries, 0 to 59399
Data columns (total 18 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   status_group      55102 non-null   object  
 1   amount_tsh        55102 non-null   float64 
 2   gps_height        55102 non-null   int64  
 3   installer         55102 non-null   object  
 4   longitude         55102 non-null   float64 
 5   latitude          55102 non-null   float64 
 6   basin             55102 non-null   object  
 7   region            55102 non-null   object  
 8   population        55102 non-null   int64  
 9   permit             55102 non-null   int64  
 10  construction_year 55102 non-null   int64  
 11  extraction_type_class 55102 non-null   object  
 12  management         55102 non-null   object  
 13  payment_type       55102 non-null   object  
 14  water_quality      55102 non-null   object  
 15  quantity           55102 non-null   object  
 16  source_type        55102 non-null   object  
 17  waterpoint_type    55102 non-null   object  
dtypes: float64(3), int64(4), object(11)
memory usage: 8.0+ MB
```

In [25]:

```
1 for col in df.columns:
2     print(df[col].value_counts())
```

```
functional                29885
non functional            21381
functional needs repair   3836
Name: status_group, dtype: int64
0.0                      37811
500.0                     3071
50.0                      2333
1000.0                    1442
20.0                      1427
...
38000.0                  1
1400.0                   1
8500.0                   1
6300.0                   1
26.0                      1
Name: amount_tsh, Length: 95, dtype: int64
0                      18310
-15                     54
303                     51
16                     51
```

After our first round of cleaning, there are several features we need to examine further:

- **status\_group** is an unbalanced target, may need to look into further during modeling and apply SMOTE.

- There are several columns with thousands of 0 entries - **amount\_tsh, gps\_height, longitude, latitude, population, construction\_year**.

## ▼ 4.4 Construction year

In [26]: 1 df['construction\_year'].value\_counts()

```
Out[26]: 0      18392
2008    2568
2009    2490
2010    2427
2000    1565
2007    1557
2006    1447
2003    1276
2011    1211
2004    1107
2002    1064
1978    1027
2012    1025
2005     983
1995     978
1999     950
1985     941
1998     921
1984     777
...     ...
```

In [27]: 1 # Finding mean and median without zero values
2 df.loc[df['construction\_year']!=0].describe()

Out[27]:

	amount_tsh	gps_height	longitude	latitude	population	permit	const
count	36710.000000	36710.000000	36710.000000	36710.000000	36710.000000	36710.000000	36710.000000
mean	471.881843	982.395015	36.015003	-6.358975	268.881694	0.717379	36710.000000
std	3074.841656	623.784917	2.609370	2.762486	542.812926	0.450280	36710.000000
min	0.000000	-63.000000	29.607122	-11.649440	0.000000	0.000000	36710.000000
25%	0.000000	351.000000	34.671850	-8.855908	30.000000	0.000000	36710.000000
50%	0.000000	1116.500000	36.691907	-6.351197	150.000000	1.000000	36710.000000
75%	200.000000	1471.000000	37.896261	-3.731978	304.000000	1.000000	36710.000000
max	250000.000000	2770.000000	40.345193	-1.042375	30500.000000	1.000000	36710.000000

In [28]: 1 #Replace 0 values in construction\_year with mean of data without 0s.
2 df['construction\_year'].replace(to\_replace = 0, value = 1996, inplace=

```
In [29]: ▾ 1 #Check to see if it worked
  2 df[ 'construction_year' ].value_counts()
```

```
Out[29]: 1996    19158
2008     2568
2009     2490
2010     2427
2000     1565
2007     1557
2006     1447
2003     1276
2011     1211
2004     1107
2002     1064
1978     1027
2012     1025
2005      983
1995      978
1999      950
1985      941
1998      921
1984      777
1982      741
1972      705
1994      703
1974      675
1990      666
1980      647
1992      632
1997      612
1993      595
2001      530
1988      520
1983      487
1975      437
1986      431
1976      411
1991      322
1989      316
1970      310
1987      297
1981      237
1977      199
1979      192
1973      183
2013      173
1971      145
1963       84
1967       83
1968       68
1969       59
1960       45
1964       40
1962       29
1961       20
1965       19
```

```
1966      17
Name: construction_year, dtype: int64
```

It is unfortunate that there are 19,000 entries with 0 for the `construction_year`. I chose to replace the 0 values with 1996, the mean of the dataset, since it was such a large representation of the dataset.

## ▼ 4.5 Latitude/Longitude zeros

```
In [30]: 1 df.longitude.value_counts()
```

```
Out[30]: 0.000000    1793
32.984790      2
37.540901      2
37.328905      2
37.252194      2
...
39.002868      1
37.095964      1
36.658462      1
33.116994      1
38.592731      1
Name: longitude, Length: 53261, dtype: int64
```

```
In [31]: ▼ 1 # Investigate longitude entries that are 0
2 df.loc[df['longitude'] == 0]
```

Out[31]:

	status_group	amount_tsh	gps_height	installer	longitude	latitude	basin	region
21	functional	0.0	0	DWE	0.0	-2.000000e-08	Lake Victoria	Shinyanga
53	non functional	0.0	0	Government	0.0	-2.000000e-08	Lake Victoria	Mwanza
168	functional	0.0	0	WVT	0.0	-2.000000e-08	Lake Victoria	Shinyanga
177	non functional	0.0	0	DWE	0.0	-2.000000e-08	Lake Victoria	Shinyanga
253	functional needs repair	0.0	0	DWE	0.0	-2.000000e-08	Lake Victoria	Mwanza
...	...	...	...	...	...	...	...	...
59189	functional needs repair	0.0	0	DWE	0.0	-2.000000e-08	Lake Victoria	Shinyanga

The 0s that are entered into the longitude column are also 0s in gps\_height and -2e8 for latitude columns. I will drop these values from the dataset.

```
In [32]: ▼ 1 # Drop rows with 0 entered in longitude column
2 df = df.loc[df['longitude'] != 0]
```

```
In [33]: 1 # Check to see if it worked
          2 df.describe()
```

Out[33]:

	amount_tsh	gps_height	longitude	latitude	population	permit	cons
count	53309.000000	53309.000000	53309.000000	53309.000000	53309.000000	53309.000000	53309.000000
mean	337.580181	692.509670	35.186804	-5.849440	188.814515	0.702508	
std	2714.547122	691.264883	2.670974	2.806529	474.147131	0.457159	
min	0.000000	-90.000000	29.607122	-11.649440	0.000000	0.000000	
25%	0.000000	0.000000	33.167340	-8.441371	0.000000	0.000000	
50%	0.000000	438.000000	35.295878	-5.144420	45.000000	1.000000	
75%	40.000000	1322.000000	37.353028	-3.359390	240.000000	1.000000	
max	250000.000000	2770.000000	40.345193	-0.998464	30500.000000	1.000000	

In [34]:

```
1 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 53309 entries, 0 to 59399
Data columns (total 18 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   status_group     53309 non-null   object 
 1   amount_tsh       53309 non-null   float64
 2   gps_height       53309 non-null   int64  
 3   installer        53309 non-null   object 
 4   longitude        53309 non-null   float64
 5   latitude         53309 non-null   float64
 6   basin            53309 non-null   object 
 7   region            53309 non-null   object 
 8   population        53309 non-null   int64  
 9   permit            53309 non-null   int64  
 10  construction_year 53309 non-null   int64  
 11  extraction_type_class 53309 non-null   object 
 12  management        53309 non-null   object 
 13  payment_type      53309 non-null   object 
 14  water_quality     53309 non-null   object 
 15  quantity           53309 non-null   object 
 16  source_type        53309 non-null   object 
 17  waterpoint_type    53309 non-null   object 
dtypes: float64(3), int64(4), object(11)
memory usage: 7.7+ MB
```

Looks like it all worked! I believe the **amount\_tsh** and **population** 0 values are real so I will leave all data as is for vanilla models.

## ▼ 4.6 Installer - Several different spellings for same installer

```
In [35]: ▾ 1 #Check unique values after initial cleaning
      2 df.nunique()
```

```
Out[35]: status_group            3
amount_tsh                  95
gps_height                 2426
installer                   2024
longitude                  53260
latitude                   53262
basin                      9
region                      21
population                 1026
permit                      2
construction_year           54
extraction_type_class      7
management                  12
payment_type                7
water_quality               8
quantity                     5
source_type                 7
waterpoint_type             7
dtype: int64
```

Upon checking the unique values for our categorical variables after trimming the dataset, installer still has 2024 unique entries, which will be a problem when we create dummies. We will need to cut down the amount of unique entries to not overload our model.

```
In [36]: ▾ 1 #Investigate 2024 unique values for installer
      2 # pd.set_option("display.max_rows", None)
      3 df['installer'].value_counts()
```

```
Out[36]: DWE                    16214
Government                1633
RWE                      1178
Commu                     1060
DANIDA                    1049
...
ABDUL                      1
JI                         1
KAEM                      1
District Community j       1
Morrovian                  1
Name: installer, Length: 2024, dtype: int64
```

There are several entries with typos and different variations of the same installer. I will attempt to fix some of the clerical errors and narrow down the amount of unique identifiers we will use for our model.

```
In [37]:  
    1 # Correct variations and misspellings in the installer column  
    2 df['installer'] = df['installer'].replace(to_replace = ('Central gove  
    3                                         'Central Government', 'Tanzan  
    4                                         'Centra Goverment', 'centra  
    5                                         'TANZANIA GOVERNMENT', 'Centr  
    6                                         'Tanzanian Government', 'Tan  
    7  
    8 df['installer'] = df['installer'].replace(to_replace = ('District COU  
    9                                         'Counc', 'District council',  
   10                                         'Council', 'COUN', 'Distri  
   11                                         value = 'District Council')  
   12  
   13 df['installer'] = df['installer'].replace(to_replace = ('villigers',  
   14                                         'Villi', 'Village Council',  
   15                                         'Village community', 'Villag  
   16                                         'Villege Council', 'Village  
   17                                         'Villager', 'Village Technici  
   18                                         'Village community members'  
   19                                         'Village govt', 'VILLAGERS'  
   20  
   21 df['installer'] = df['installer'].replace(to_replace = ('District Water  
   22                                         'Distric Water Department')  
   23  
   24 df['installer'] = df['installer'].replace(to_replace = ('FinW', 'Fini  
   25                                         'Finwater', 'FINN WATER', 'I  
   26                                         value ='Fini Water')  
   27  
   28 df['installer'] = df['installer'].replace(to_replace = ('RC CHURCH',  
   29                                         'RC church', 'RC CATHORIC',  
   30  
   31 df['installer'] = df['installer'].replace(to_replace = ('world vision',  
   32                                         'WORLD VISION', 'World Vissi  
   33  
   34 df['installer'] = df['installer'].replace(to_replace = ('Unisef', 'Unic  
   35  
   36 df['installer'] = df['installer'].replace(to_replace = 'DANID', value  
   37  
   38 df['installer'] = df['installer'].replace(to_replace = ('Commu', 'Commu  
   39                                         'Adra /Community', 'Community  
   40                                         value ='Community')  
   41  
   42 df['installer'] = df['installer'].replace(to_replace = ('GOVERNMENT',  
   43                                         'Gover', 'Gove', 'Governme  
   44  
   45 df['installer'] = df['installer'].replace(to_replace = ('Hesawa', 'hes  
   46  
   47 df['installer'] = df['installer'].replace(to_replace = ('JAICA', 'JICA  
   48                                         value ='Jaica')  
   49  
   50 df['installer'] = df['installer'].replace(to_replace = ('KKKT _ Konde  
   51                                         value ='KKKT')  
   52  
   53 df['installer'] = df['installer'].replace(to_replace = '0', value ='Un
```

```
In [38]: 1 df['installer'].value_counts().head(20)
```

```
Out[38]: DWE                16214
Government          2468
Community            1791
DANIDA              1601
HESAWA               1180
RWE                 1178
District Council    1173
Central Government   1115
KKKT                1102
Fini Water           952
Unknown              780
TCRS                702
World Vision         660
CES                  610
RC Church             484
Villagers            482
LGA                  408
WEDECO               397
TASAF                371
- .
```

## ▼ 4.7 Reduce dimensionality for installer

```
In [39]: ▼ 1 # Keep only top 20 installers as unique values
  2 installer_20 = df.installer.value_counts(normalize=True).head(20).index
  3
  4
▼ 5 df['installer'] = [type_ if type_ in installer_20
  6                   else "OTHER" for type_ in df['installer']]
```

```
In [40]: 1 df.installer.value_counts()
```

```
Out[40]: OTHER          19283
DWE              16214
Government      2468
Community       1791
DANIDA          1601
HESAWA          1180
RWE              1178
District Council 1173
Central Government 1115
KKKT              1102
Fini Water       952
Unknown           780
TCRS              702
World Vision     660
CES                610
RC Church         484
Villagers         482
LGA                408
WEDECO            397
TASAF              371
Jaica              358
Name: installer, dtype: int64
```

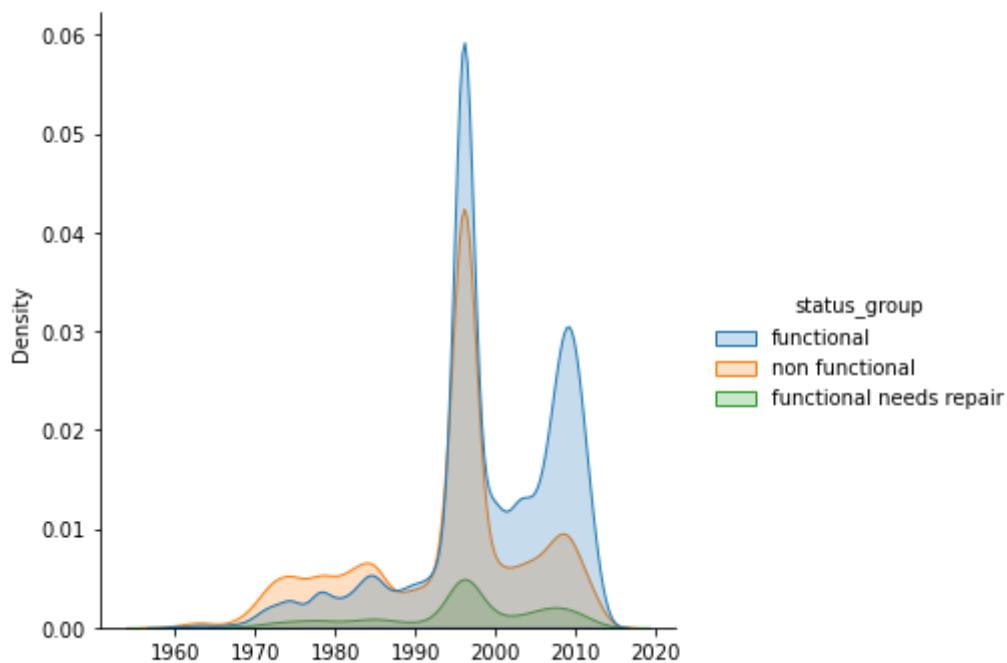
To reduce the dimensionality of the dataset, I made an "Other" category for installer if they were not in the top 20 installers of the dataset.

## ▼ 5 Modified Features Exploration

### ▼ 5.1 Column EDA

#### ▼ 5.1.1 Construction year

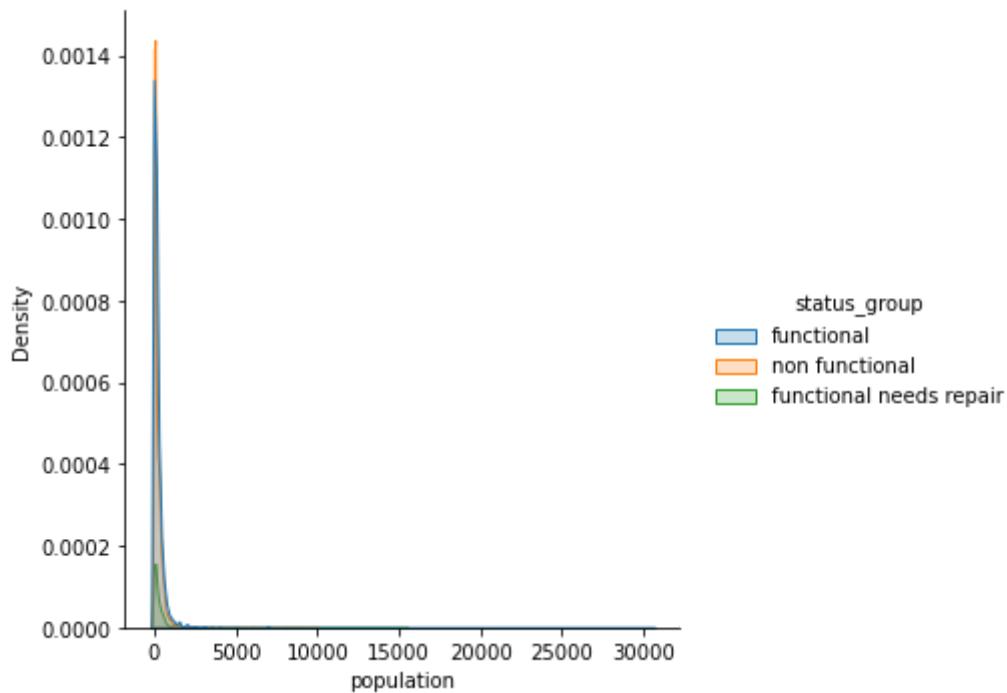
```
In [41]: 1 sns.displot(data=df, x='construction_year', hue='status_group', kind=
```



There is not a lot of separation between status of pump and the year it was constructed. We can also see the large amount of data in the year 1996.

### 5.1.2 Population

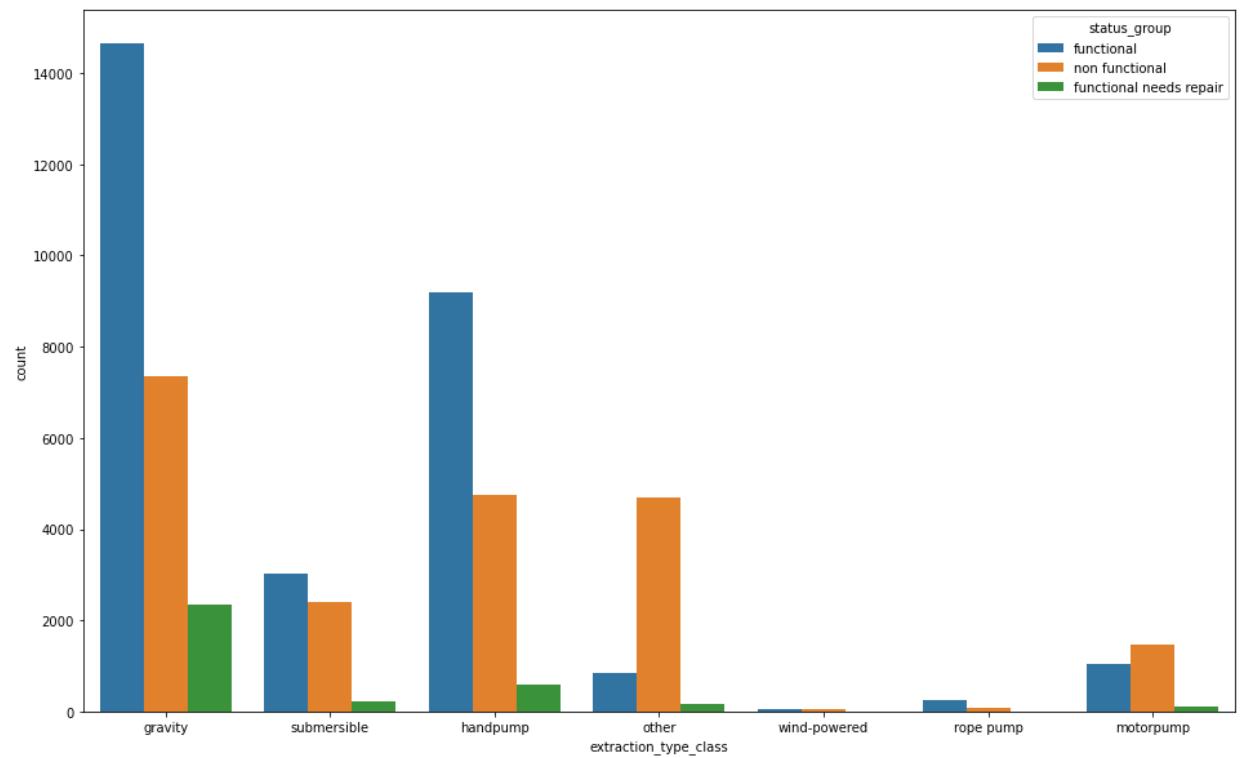
```
In [42]: 1 sns.displot(data=df, x='population', hue='status_group', kind='kde', :
```



### 5.1.3 Extraction\_type\_class

In [43]:

```
1 plt.figure(figsize=(16,10))
2 ax = sns.countplot(x='extraction_type_class', hue='status_group', data=
```

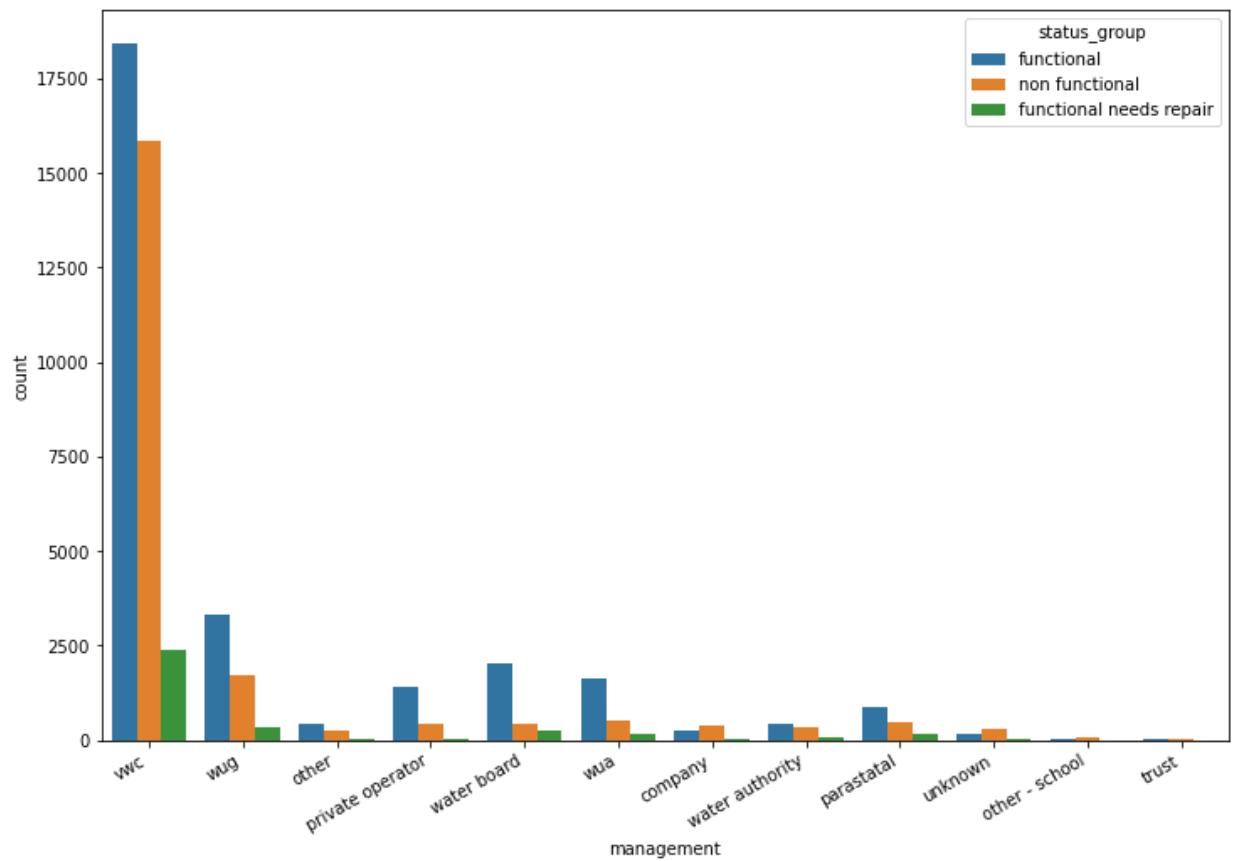


Other type and motorpump are especially non functioning. Gravity and handpump are the 2 largest types, and both have more functioning, but half non functioning.

#### ▼ 5.1.4 Management

In [44]:

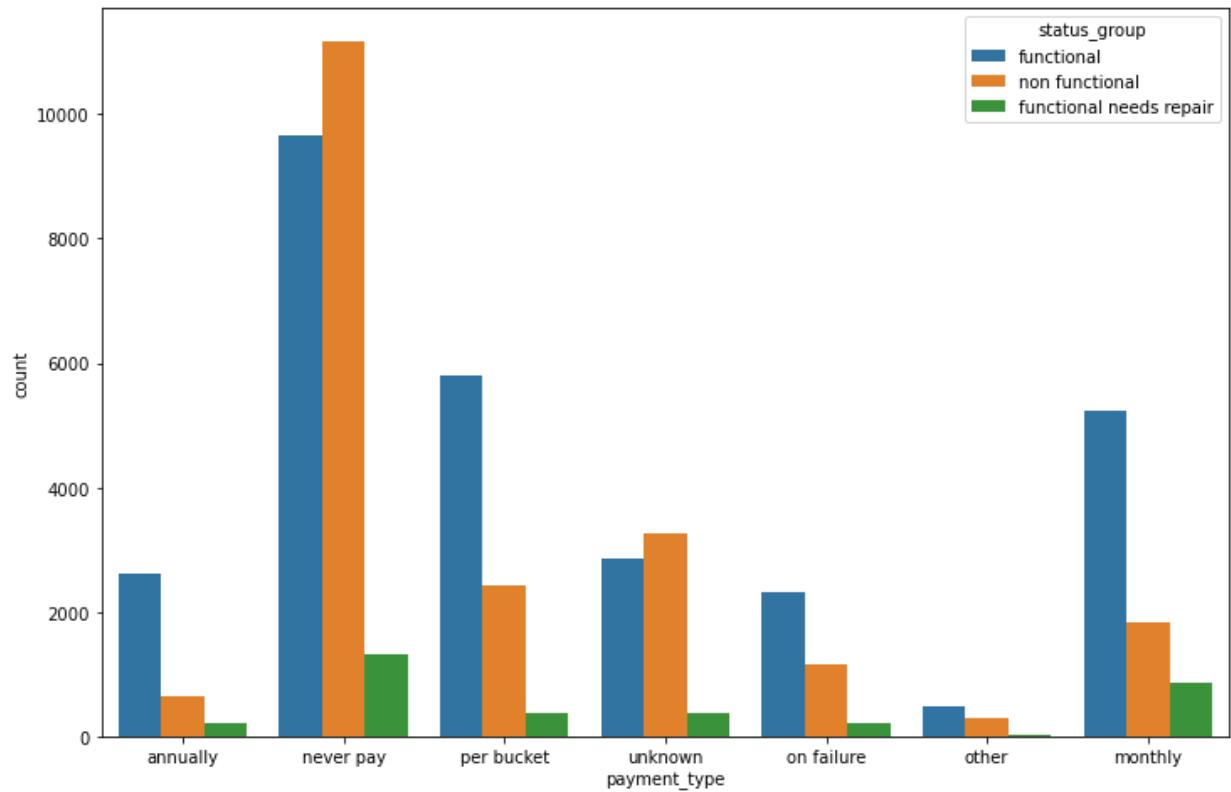
```
1 plt.figure(figsize=(12,8))
2 ax = sns.countplot(x='management', hue='status_group', data=df)
3 plt.xticks(rotation=30, ha='right');
```



## ▼ 5.1.5 Payment\_type

In [45]:

```
1 plt.figure(figsize=(12,8))
2 ax = sns.countplot(x='payment_type', hue='status_group', data=df);
```

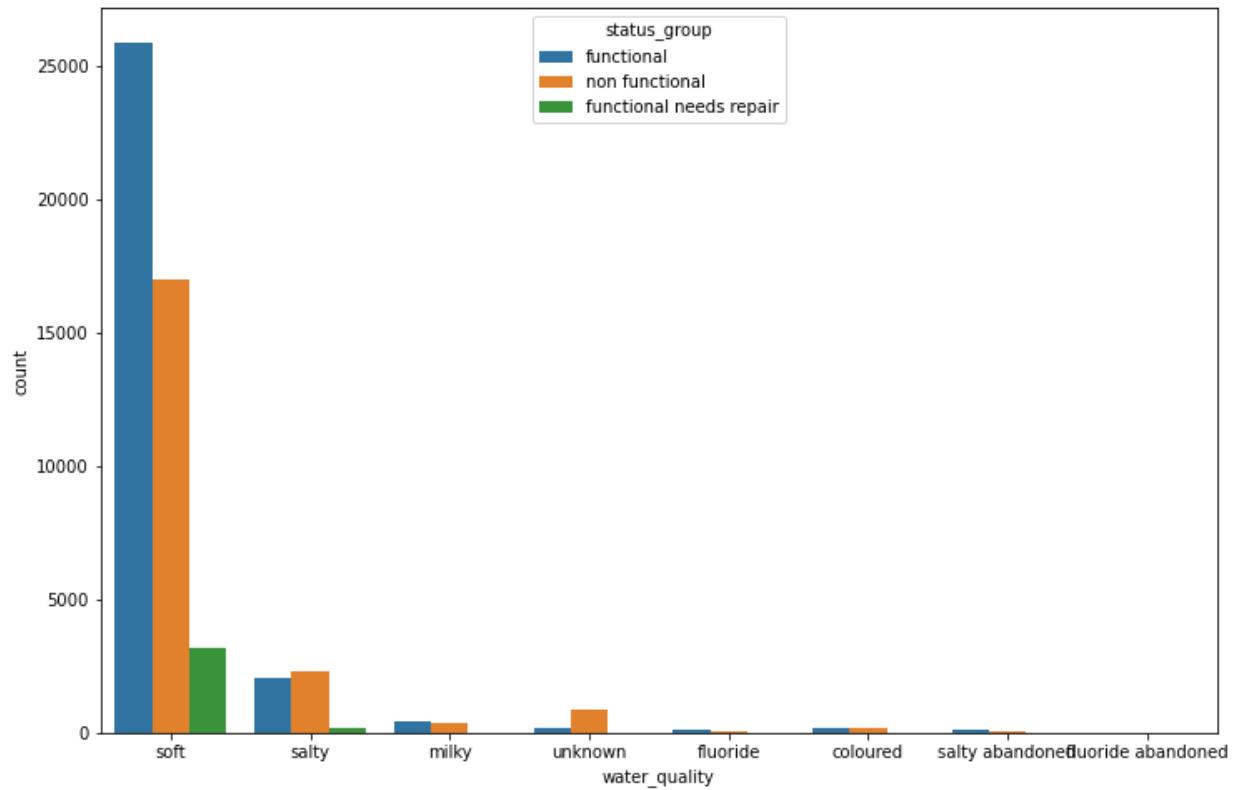


Never pay pumps have more non functioning waterpoints than functioning waterpoints.

▼ **5.1.6 Water quality**

In [46]:

```
1 plt.figure(figsize=(12,8))
2 ax = sns.countplot(x='water_quality', hue='status_group', data=df);
```

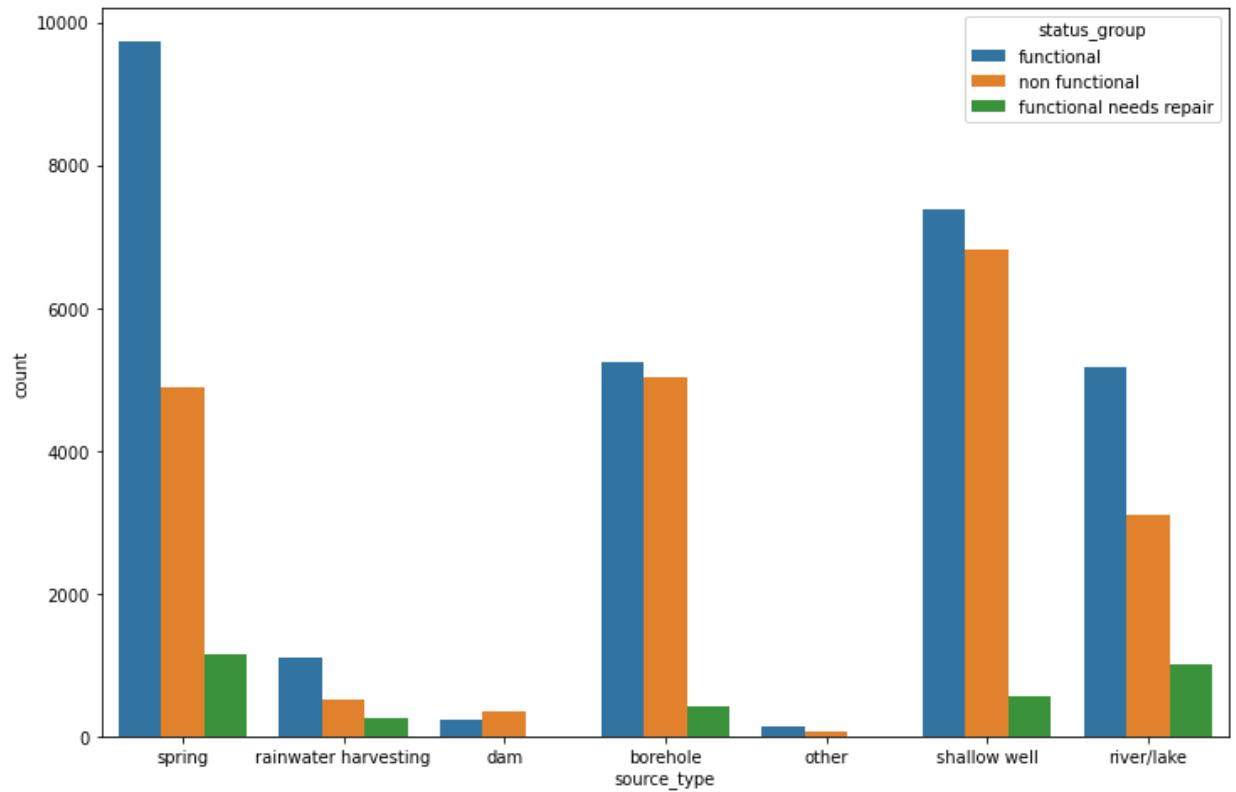


Soft water quality has a high rate of functional waterpoints, salty has a high rate of non functional waterpoints.

#### ▼ 5.1.7 Source type

In [47]:

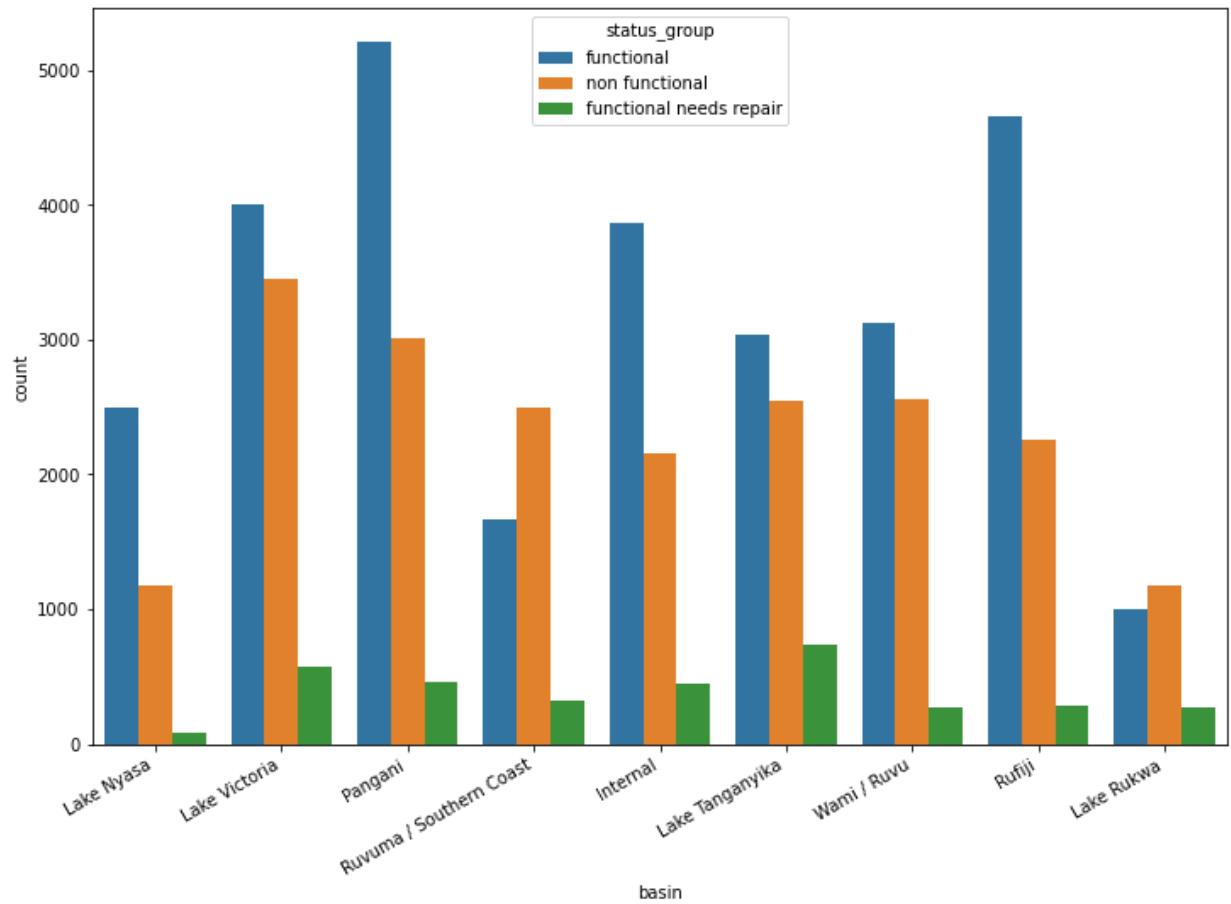
```
1 plt.figure(figsize=(12,8))
2 ax = sns.countplot(x='source_type', hue='status_group', data=df);
```



▼ 5.1.8 Basin

In [48]:

```
1 plt.figure(figsize=(12,8))
2 ax = sns.countplot(x='basin', hue='status_group', data=df)
3 plt.xticks(rotation=30, ha='right');
```



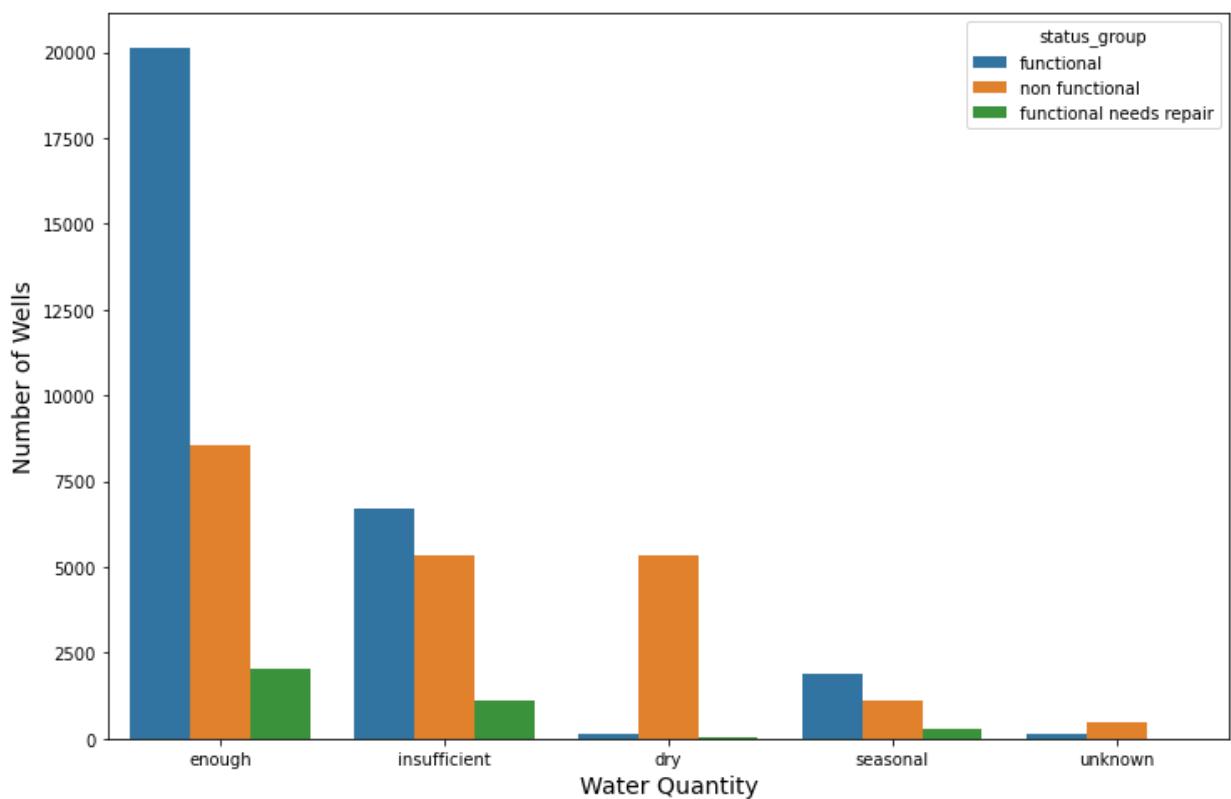
The Ruvuma/Southern Coast and Lake Rukwa basins have more non functioning wells than functional.

#### ▼ 5.1.9 Quantity

In [50]:

```
1 fig, ax = plt.subplots(figsize=(12,8))
2 ax = sns.countplot(x='quantity', hue="status_group", data=df)
3
4 fig.suptitle('Quantity of Water in Wells', fontsize=18)
5 plt.xlabel("Water Quantity", fontsize=14)
6 plt.ylabel("Number of Wells", fontsize=14)
7 plt.show()
8
9 fig.savefig('./images/quantity_function.jpeg');
```

Quantity of Water in Wells



Dry waterpoints have a high chance of being non functional, as expected. If the waterpoint has enough water, there is a high chance of functionality.

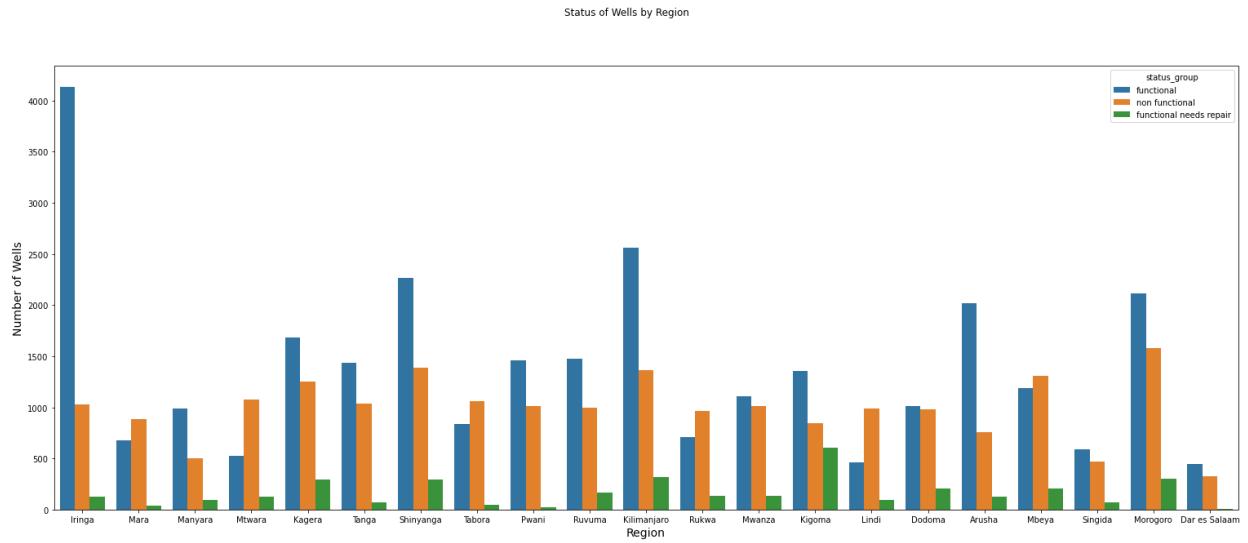
#### ▼ 5.1.10 Region

In [51]:

```

1 fig, ax = plt.subplots(figsize=(26,10))
2 ax = sns.countplot(x='region', hue="status_group", data=df)
3
4 fig.suptitle('Status of Wells by Region')
5 plt.xlabel("Region", fontsize=14)
6 plt.ylabel("Number of Wells", fontsize=14)
7 plt.show()
8
9 fig.savefig('./images/region_function.jpeg');

```

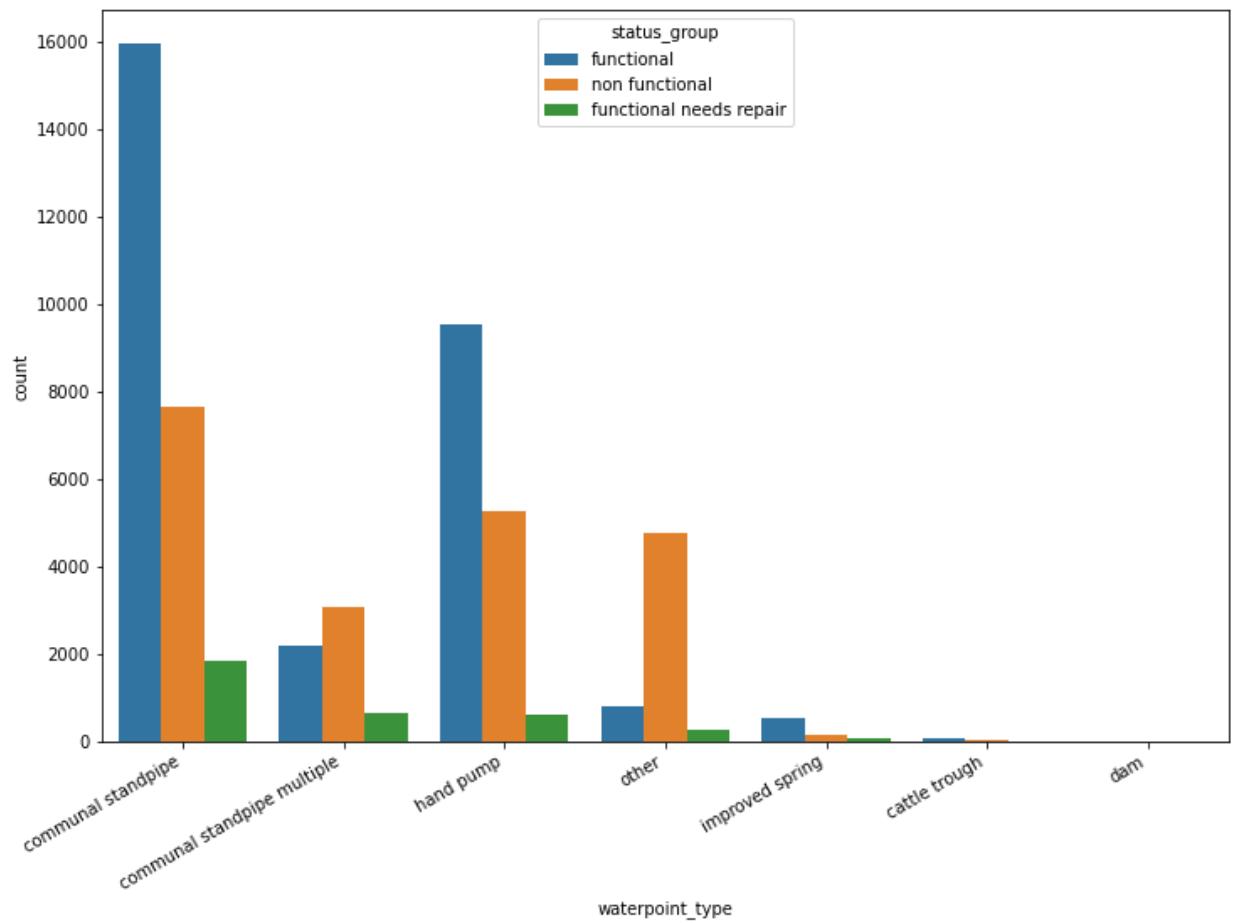


The Iringa region has a very high rate of functioning wells, followed by Kilimanjaro, Arusha, and Shinyanga. The worst regions for well performance are Mtwara, Mara, Rukwa, and Lindi.

#### ▼ 5.1.11 Waterpoint type

In [51]:

```
1 plt.figure(figsize=(12,8))
2 ax = sns.countplot(x='waterpoint_type', hue='status_group', data=df)
3 plt.xticks(rotation=30, ha='right');
```



Other and communal standpipe multiple have the highest rate of being non functioning.



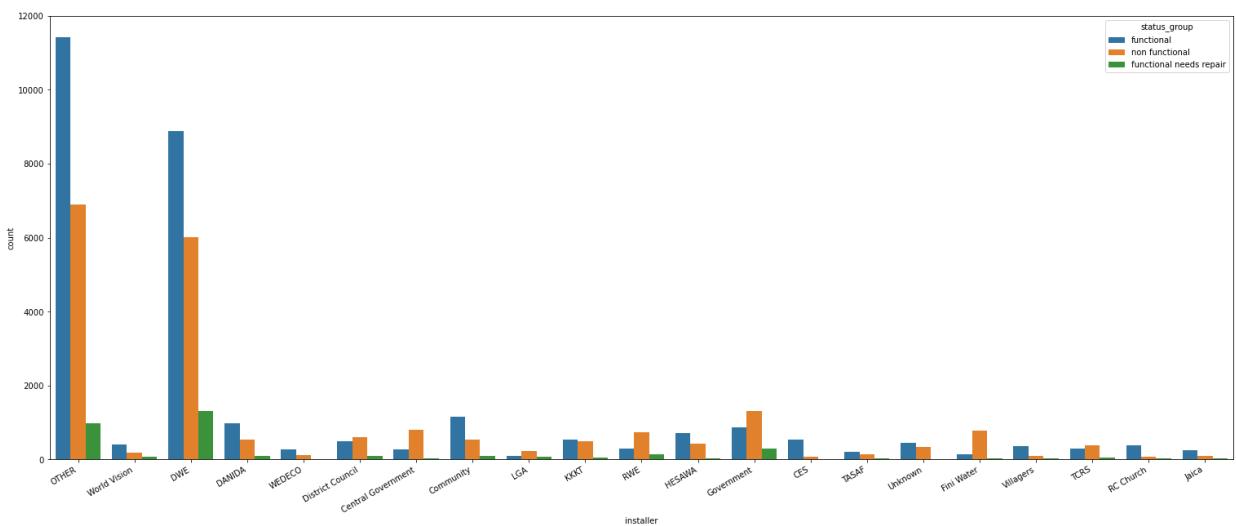
### 5.1.12 Installer

In [52]:

```

1 plt.figure(figsize=(26,10))
2 ax = sns.countplot(x='installer', hue='status_group', data=df)
3 plt.xticks(rotation=30, ha='right')
4 plt.savefig('./images/installer_function.png');

```



The government, Fini Water, RWE, and District Council have a high rate of non functioning wells. Other is our largest category.

## ▼ 5.2 Well Function map

In [53]:

```
1 import folium
```

In [54]:

```

1 # Create 3 dataframes for each status_group
2 df_f = df[df['status_group'] == 'functional']
3 df_nf = df[df['status_group'] == 'non functional']
4 df_fnr = df[df['status_group'] == 'functional needs repair']

```

In [55]:

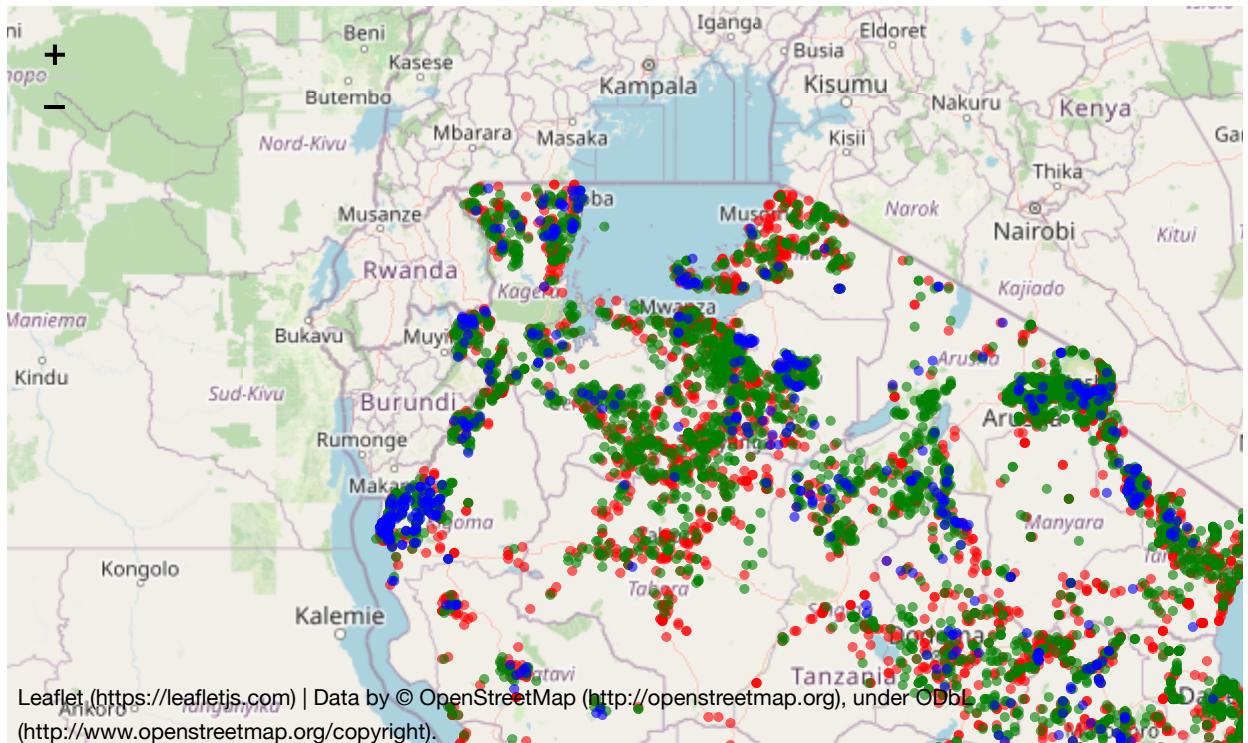
```

1 # Create lists of latitude and longitude values
2 lat_f = [x for x in df_f['latitude']]
3 long_f = [x for x in df_f['longitude']]
4
5 lat_nf = [x for x in df_nf['latitude']]
6 long_nf = [x for x in df_nf['longitude']]
7
8 lat_fnr = [x for x in df_fnr['latitude']]
9 long_fnr = [x for x in df_fnr['longitude']]
10
11 lat_long_f = [(lat_f[i], long_f[i]) for i in range(len(lat_f))]
12 lat_long_nf = [(lat_nf[i], long_nf[i]) for i in range(len(lat_nf))]
13 lat_long_fnr = [(lat_fnr[i], long_fnr[i]) for i in range(len(lat_fnr))]

```

```
In [56]: 1 #Create map
2 this_map = folium.Map()
3
4 # Loop through 3 dataframes and plot point for each coordinate
5 for coord in lat_long_nf[::5]:
6     folium.CircleMarker(location=[coord[0], coord[1]], opacity=0.6, color='red').add_to(this_map)
7 for coord in lat_long_f[::5]:
8     folium.CircleMarker(location=[coord[0], coord[1]], opacity=0.6, color='blue').add_to(this_map)
9 for coord in lat_long_fnr[::5]:
10    folium.CircleMarker(location=[coord[0], coord[1]], opacity=0.6, color='green').add_to(this_map)
11
12
13 #Set the zoom to fit our bounds
14 this_map.fit_bounds(this_map.get_bounds())
15
16 this_map
```

Out[56]:



As we saw above, there is a high rate of non functional waterpoints in the southeast corner of Tanzania in Mtwara and Lindi, as well as up north in Mara, and the southwest in Rukwa. We can see the cluster of high functional wells in Iringa, Shinyanga, Kilimanjaro, and Arusha. There is a cluster of functional but need repair waterpoints in Kigoma.

### 5.3 Change status\_group to integer format

```
In [57]: 1 # Change status_group/target values to numeric values
          2 df[ 'status' ] = df.status_group.map({ "non functional":0, "functional ne
          3 df.head()
```

Out[57]:

	status_group	amount_tsh	gps_height	installer	longitude	latitude	basin	region	population
0	functional	6000.0	1390	OTHER	34.938093	-9.856322	Lake Nyasa	Iringa	100000
1	functional	0.0	1399	OTHER	34.698766	-2.147466	Lake Victoria	Mara	100000
2	functional	25.0	686	World Vision	37.460664	-3.821329	Pangani	Manyara	100000
3	non functional	0.0	263	OTHER	38.486161	-11.155298	Ruvuma / Southern Coast	Mtwara	100000
4	functional	0.0	0	OTHER	31.130847	-1.825359	Lake Victoria	Kagera	100000

```
In [58]: 1 df = df.drop( 'status_group' , axis=1)
```

```
In [59]: 1 df.shape
```

Out[59]: (53309, 18)

## 6 Modeling

### 6.1 Data Preprocessing

Following we will create our dummy variables for our categorical columns and perform train test split to prepare for modeling.

In [60]: 1 df.info()

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 53309 entries, 0 to 59399
Data columns (total 18 columns):
 #   Column           Non-Null Count  Dtype  
---  --  
 0   amount_tsh        53309 non-null   float64 
 1   gps_height        53309 non-null   int64  
 2   installer         53309 non-null   object  
 3   longitude          53309 non-null   float64 
 4   latitude           53309 non-null   float64 
 5   basin              53309 non-null   object  
 6   region              53309 non-null   object  
 7   population         53309 non-null   int64  
 8   permit              53309 non-null   int64  
 9   construction_year  53309 non-null   int64  
 10  extraction_type_class 53309 non-null   object  
 11  management          53309 non-null   object  
 12  payment_type        53309 non-null   object  
 13  water_quality       53309 non-null   object  
 14  quantity             53309 non-null   object  
 15  source_type          53309 non-null   object  
 16  waterpoint_type     53309 non-null   object  
 17  status               53309 non-null   int64  
dtypes: float64(3), int64(5), object(10)
memory usage: 10.2+ MB
```

### 6.1.1 Create dummies

In [61]: 1 *# Create lists of categorical and continuous columns*  
 2 cat\_col = ['installer', 'basin', 'region', 'extraction\_type\_class', 'management',  
 3 'quantity', 'source\_type', 'waterpoint\_type']  
 4 cont\_col = ['amount\_tsh', 'gps\_height', 'longitude', 'latitude', 'population']

In [62]: 1 *#Create dummies*  
 2 dummy\_df = pd.get\_dummies(df, columns=cat\_col, drop\_first=True)  
 3  
 4 dummy\_df.shape

Out[62]: (53309, 102)

### 6.1.2 Separate target and perform train test split

In [63]: 1 y = dummy\_df['status']  
 2 X = dummy\_df.drop(['status'], axis=1)  
 3 X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0)

### 6.1.3 Model Statistics Function

Accuracy will be our main metric used to track model performance, but we will calculate recall,

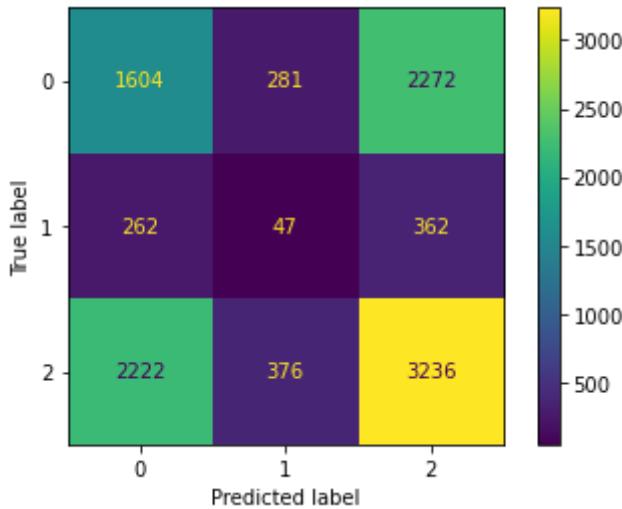
precision, cross validation, and f1 score to provide more detail.

```
In [64]: 1 # function to track model metrics and plot confusion matrix
2
3 def model_score(model, X, y_pred, y_true):
4     acc = accuracy_score(y_true, y_pred)
5     rec = recall_score(y_true, y_pred, average='weighted')
6     f1 = f1_score(y_true, y_pred, average='weighted')
7     prec = precision_score(y_true, y_pred, average='weighted')
8     cross = cross_val_score(estimator=model, X=X, y=y_true, cv=10).mean()
9
10    print(f' The accuracy score is: {round(acc, 4)}')
11    print(f' The recall score is: {round(rec, 4)}')
12    print(f' The F1 score is: {round(f1, 4)}')
13    print(f' The precision score is: {round(prec, 4)}')
14    print(f' The cross validation score of accuracy is: {round(cross, 4)}')
15
16    #Confusion matrix
17    return plot_confusion_matrix(model, X, y_true);
```

## 6.2 Dummy Classifier Model

```
In [65]: 1 dummy = DummyClassifier(random_state=42)
2 dummy.fit(X_train, y_train)
3 print(dummy.score(X_test, y_test))
4
5 plot_confusion_matrix(dummy, X_test, y_test);
```

0.4583567810917276

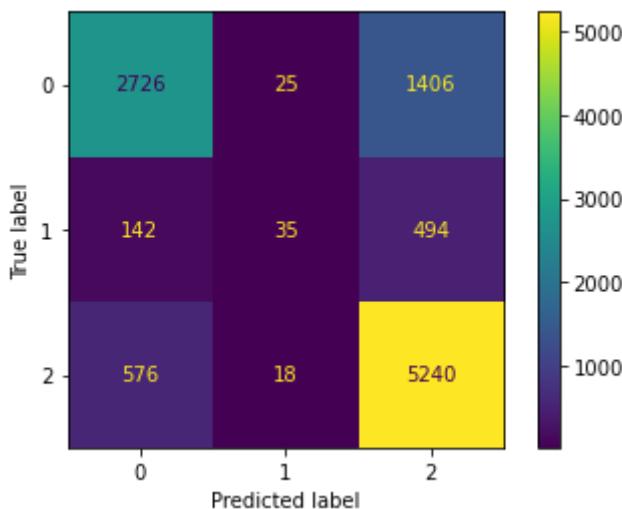


Our baseline dummy model performed very poorly with an accuracy score of 46%. Our data is heavily imbalanced, which explains how our ternary model performed close to 50%.

## 6.3 Logistic Regression

```
In [66]: # Make pipe
    pipe_lr = Pipeline([('ss', StandardScaler()),
                         ('lr', LogisticRegression())))
    # Fit and predict
    pipe_lr.fit(X_train, y_train)
    test_preds = pipe_lr.predict(X_test)
    lr_score = model_score(pipe_lr, X_test, test_preds, y_test)
```

Test data model score:  
The accuracy score is: 0.7504  
The recall score is: 0.7504  
The F1 score is: 0.7275  
The precision score is: 0.7384  
The cross validation score of accuracy is: 0.7491



Our logistic regression model is improved to 75% accuracy over the dummy model. This model struggled to predict wells that were functional but needed repairs, likely due to class imbalances.

## 6.4 K Nearest Neighbors

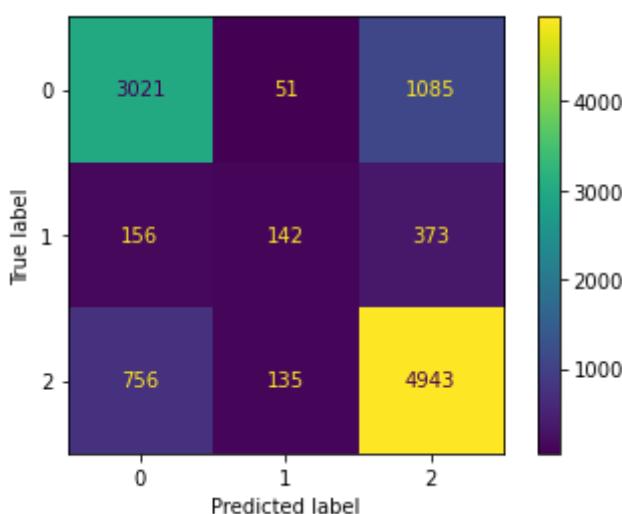
Below I will run GridSearch with my Pipeline to create a K Nearest Neighbors model. I ran GridSearch to find the best parameters, and have then commented out the code to save computing time while still showing the process. The same process is repeated for all following models of running GridSearch and commenting out code.

```
In [67]: 1 # GridSearch
2 knn = KNeighborsClassifier()
3 grid = {
4     'n_neighbors' : [5, 10, 15, 20, 25, 40]
5 }
6
7 #knn_grid_search = GridSearchCV(knn, grid, cv=3)
8 #knn_grid_search.fit(X_train, y_train)
9
10 #knn_grid_search.best_params_
```

```
In [68]: 1 # Narrow down parameters for 2nd gridsearch
2 knn = KNeighborsClassifier()
3 grid = {
4     'n_neighbors' : [7, 8, 9, 10, 11, 12, 13]
5 }
6
7 #knn_grid_search = GridSearchCV(knn, grid, cv=3)
8 #knn_grid_search.fit(X_train, y_train)
9
10 #knn_grid_search.best_params_
```

```
In [69]: 1 # Make pipe
2 pipe_knn = Pipeline([('ss', StandardScaler()),
3                      ('knn', KNeighborsClassifier(n_neighbors=9))])
4 #Fit and predict
5 pipe_knn.fit(X_train, y_train)
6 test_preds = pipe_knn.predict(X_test)
7
8 print("Test data model score:")
9 knn_score = model_score(pipe_knn, X_test, test_preds, y_test)
```

Test data model score:  
The accuracy score is: 0.7603  
The recall score is: 0.7603  
The F1 score is: 0.7512  
The precision score is: 0.7493  
The cross validation score of accuracy is: 0.7136



```
In [70]: 1 # Check for training and test sets
2 # Predict on training and test sets
3 training_preds = pipe_knn.predict(X_train)
4 test_preds = pipe_knn.predict(X_test)
5
6 # Accuracy of training and test sets
7 training_accuracy = accuracy_score(y_train, training_preds)
8 test_accuracy = accuracy_score(y_test, test_preds)
9
10 print('Training Accuracy: {:.4}%'.format(training_accuracy * 100))
11 print('Validation accuracy: {:.4}%'.format(test_accuracy * 100))
```

Training Accuracy: 80.23%  
 Validation accuracy: 76.03%

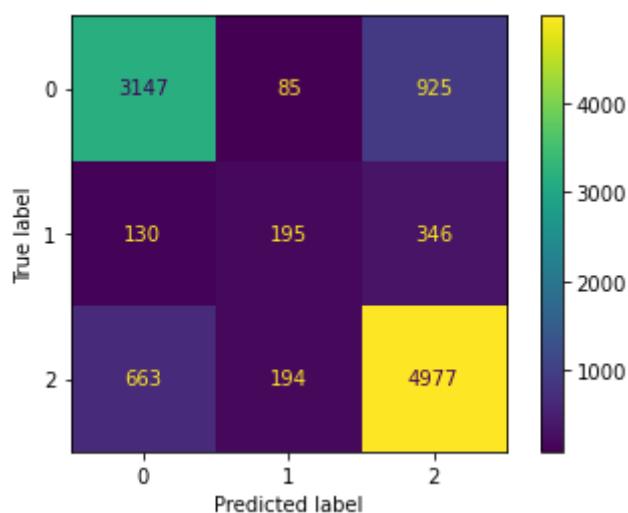
The K Nearest Neighbors model outperformed the Logistic Regression model. Number of neighbors was hypertuned by running and GridSearch and optimal parameters were put into our pipe. Our K Nearest Neighbors model is not overfitting as the accuracy of training and test sets are 80.23% and 76.03%, respectively.

## 6.5 Decision Tree Model

```
In [71]: 1 # GridSearch commented out to show process
2 dt = DecisionTreeClassifier()
3 dt_grid = {
4     'criterion' : ['entropy', 'gini'],
5     'max_depth': [10, 20, 30, 40, 50, 60, None],
6     'min_samples_split' : [1, 2, 5, 10, 20, 30],
7     'min_impurity_decrease' : [0.0, 0.1, 0.2, 0.3, 0.4, 0.5],
8     'min_impurity_split' : [None, 0.1, 0.2, 0.3, 0.4, 0.5],
9 }
10
11 #dt_tree = GridSearchCV(estimator=dt, param_grid=dt_grid, cv=5)
12 #dt_tree.fit(X_train, y_train)
13
14 #print(f'Best parameters are {dt_tree.best_params_}')
15 #print(f'Best score {dt_tree.best_score_}') #0.768565112
16 #print(f'Best estimator score {dt_tree.best_estimator_.score(X_test, )}
```

```
In [72]:  
1 # Make pipe  
2 pipe_dt = Pipeline([('ss', StandardScaler()),  
3                     ('dt', DecisionTreeClassifier(criterion='gini', ma  
4                                         min_samples_split=2  
5 #Fit and Predict  
6 pipe_dt.fit(X_train, y_train)  
7 test_preds = pipe_dt.predict(X_test)  
8  
9 print("Test data model score:")  
10 dt_score = model_score(pipe_dt, X_test, test_preds, y_test)
```

Test data model score:  
The accuracy score is: 0.7802  
The recall score is: 0.7802  
The F1 score is: 0.7753  
The precision score is: 0.7732  
The cross validation score of accuracy is: 0.7402



```
In [73]: 1 # Predict on training and test sets
2 training_preds = pipe_dt.predict(X_train)
3 test_preds = pipe_dt.predict(X_test)
4
5 # Accuracy of training and test sets
6 training_accuracy = accuracy_score(y_train, training_preds)
7 test_accuracy = accuracy_score(y_test, test_preds)
8
9 print('Training Accuracy: {:.4}%'.format(training_accuracy * 100))
10 print('Validation accuracy: {:.4}%'.format(test_accuracy * 100))
```

Training Accuracy: 89.31%  
Validation accuracy: 78.02%

Our decision tree model once again improved our test accuracy scores to 78%, but the model is highly overfitting with training accuracy at 89%.

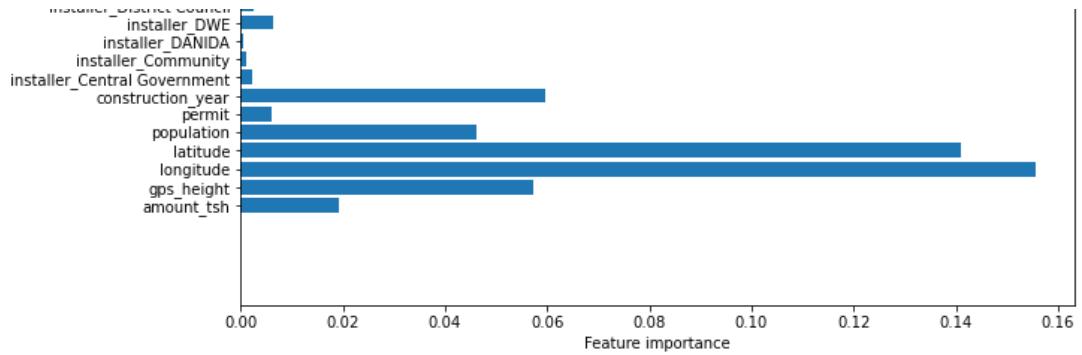
#### ▼ 6.5.1 Function to plot feature importances

```
In [74]: 1 def plot_feature_importances(model):
2     n_features = X_train.shape[1]
3     plt.figure(figsize=(10,25))
4     plt.barh(range(n_features), model.feature_importances_, align='center')
5     plt.yticks(np.arange(n_features), X_train.columns.values)
6     plt.xlabel('Feature importance')
7     plt.ylabel('Feature')
```

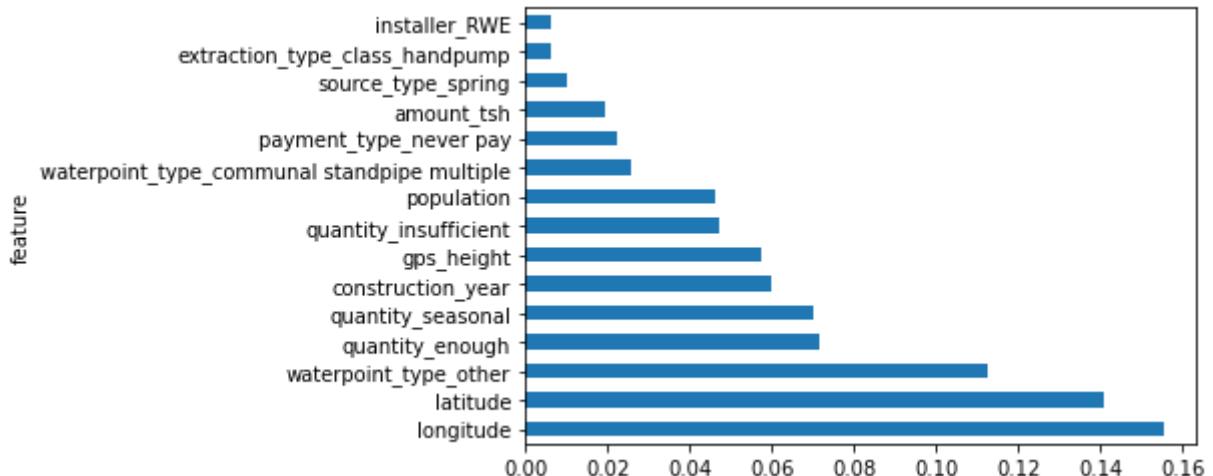
#### ▼ 6.5.2 Decision Tree Feature Importances

```
In [75]: ▾ 1 # Instantiate and fit a DecisionTreeClassifier with optimal parameter
2 tree_clf = DecisionTreeClassifier(criterion='gini', max_depth=20, min_
3 tree_clf.fit(X_train, y_train)
4
5 plot_feature_importances(tree_clf)
```





```
In [76]: # Top features
1 feature_importances=pd.DataFrame(columns=['feature','importance'])
2
3
4 feature_importances['feature']= X_train.columns
5
6 feature_importances['importance']=tree_clf.feature_importances_
7
8 feature_importances= feature_importances.set_index('feature')
9
10 feature_importances['importance'].sort_values(ascending = False).head
```



Our decision tree model showed longitude, latitude, waterpoint type other, and enough quantity being the most important features.

## 6.6 Random Forests

In [77]:

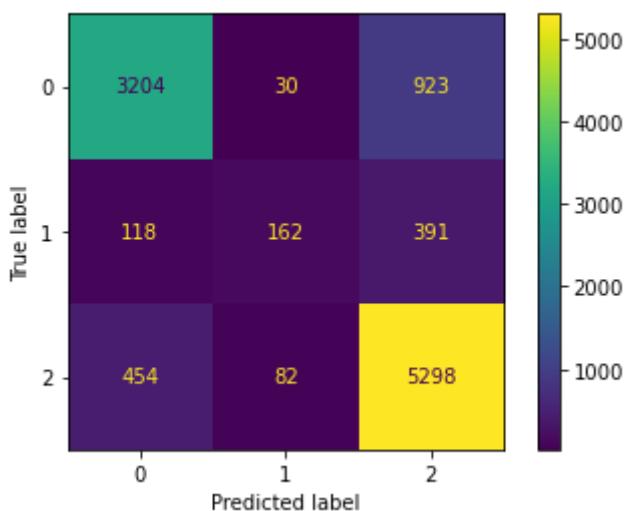
```
1 #Instantiate RandomForestClassifier
2 forest = RandomForestClassifier(n_estimators=100, max_depth= 5)
3 forest.fit(X_train, y_train)
4
5 #scores on folds
6 scores = cross_val_score(estimator=forest, X=X_train, y=y_train, cv=5)
7 print(np.mean(scores))
8
9 #scores on on test
10 score = forest.score(X_test, y_test)
11 print(score)
12
13 # grid search on random forest
14
15 #grid = {
16 #    'criterion' : ['entropy', 'gini'],
17 #    'max_depth': [5,10,15,20, None],
18 #    'min_impurity_decrease' : [0.0, 0.1, 0.2, 0.3, 0.4, 0.5], #values
19 #    'min_impurity_split' : [None, 0.1, 0.2, 0.3, 0.4, 0.5], #values
20 #}
21 #gs_forest = GridSearchCV(estimator=forest, param_grid=grid, cv=5)
22 #gs_forest.fit(X_train, y_train)
23
24 #print(f'Best parameters are {gs_forest.best_params_}')
25 #print(f'Best score {gs_forest.best_score_}')
26 #print(f'Best estimator score {gs_forest.best_estimator_.score(X_test)}
```

0.6940701175783002

0.70727818420559

```
In [78]: 1 # Make pipeline with tuned hyperparameters
  2 pipe_rf = Pipeline([('ss', StandardScaler()),
  3                      ('RF', RandomForestClassifier(bootstrap=True, criti
  4 # Fit and predict
  5 pipe_rf.fit(X_train, y_train)
  6 test_preds = pipe_rf.predict(X_test)
  7
  8 # Print metrics
  9 print("Test data model score:")
10 rf_score = model_score(pipe_rf, X_test, test_preds, y_test)
```

Test data model score:  
The accuracy score is: 0.8126  
The recall score is: 0.8126  
The F1 score is: 0.8024  
The precision score is: 0.8065  
The cross validation score of accuracy is: 0.7794



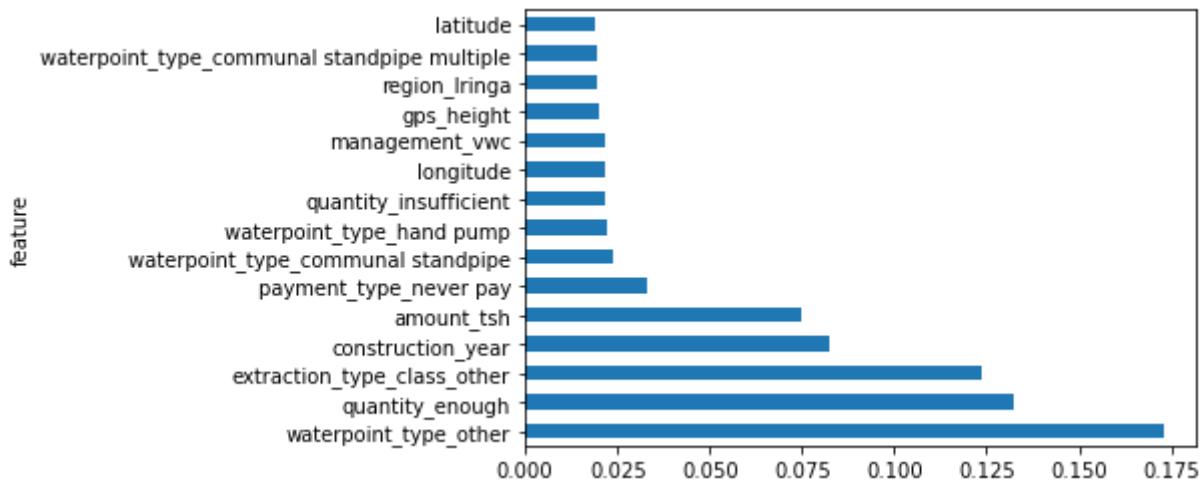
```
In [79]: 1 # Predict on training and test sets
  2 training_preds = pipe_rf.predict(X_train)
  3 test_preds = pipe_rf.predict(X_test)
  4
  5 # Accuracy of training and test sets
  6 training_accuracy = accuracy_score(y_train, training_preds)
  7 test_accuracy = accuracy_score(y_test, test_preds)
  8
  9 print('Training Accuracy: {:.4}%'.format(training_accuracy * 100))
10 print('Validation accuracy: {:.4}%'.format(test_accuracy * 100))
```

Training Accuracy: 93.02%  
Validation accuracy: 81.26%

Upon running GridSearch with our Random Forests Pipeline, we have once again improved from our baseline accuracy to 81.22% testing accuracy. The model is still overfitting the training data, as the training accuracy is 93.35%, but this is our best performing model so far.

### 6.6.1 Random Forests Feature Importances

```
In [80]: # Top features
1 feature_importances=pd.DataFrame(columns=['feature','importance'])
2
3
4 feature_importances['feature']= X_train.columns
5
6 feature_importances['importance']=forest.feature_importances_
7
8 feature_importances= feature_importances.set_index('feature')
9
10 feature_importances['importance'].sort_values(ascending = False).head
```

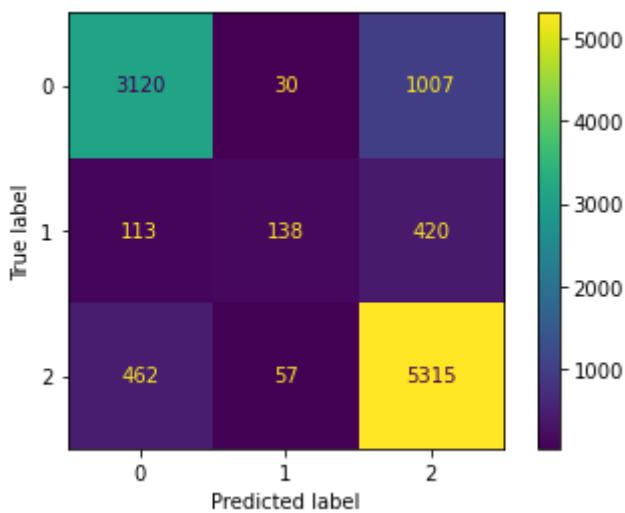


Our random forests model show waterpoint\_type other, enough quantity, extraction\_type\_class\_other, and construction\_year being the most important features to the model.

## 6.7 XG Boost

```
In [81]: 1 # Instantiate XGBClassifier
2 xgb = XGBClassifier()
3
4 # Fit XGBClassifier
5 xgb.fit(X_train, y_train)
6
7 print("Test data model score:")
8 xgb_model_score = model_score(xgb, X_test, test_preds, y_test)
```

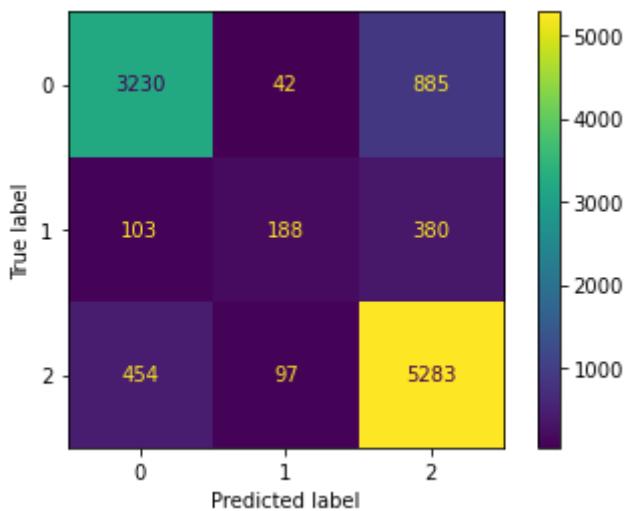
Test data model score:  
The accuracy score is: 0.8126  
The recall score is: 0.8126  
The F1 score is: 0.8024  
The precision score is: 0.8065  
The cross validation score of accuracy is: 0.7808



```
In [82]: 1 # Gridsearch commented out
2 # xgb = XGBClassifier()
3 # grid = {
4 #     'learning_rate': [0.1, 0.2],
5 #     'max_depth': [4, 6, 8, 10],
6 #     'xmin_child_weight': [1, 2],
7 #     'subsample': [0.5, 0.7],
8 #     'n_estimators': [100, 250, 500],
9 # }
10
11 #gs_xgb = GridSearchCV(estimator=xgb, param_grid=grid, cv=5)
12 #gs_xgb.fit(X_train, y_train)
13
14 #print(f'Best parameters are {gs_xgb.best_params_}')
15 #print(f'Best score {gs_xgb.best_score_}')
16 #print(f'Best estimator score {gs_xgb.best_estimator_.score(X_test, y_
```

```
In [83]:  
  1 # Instantiate xg Boost Classifier pipeline with tuned hyperparameters  
  2 pipe_xgb = Pipeline([('ss', StandardScaler()),  
  3                         ('xgb', XGBClassifier(learning_rate=0.1, max_depth=6,  
  4                                     n_estimators=250, subsample=0.7))])  
  5  
  6 pipe_xgb.fit(X_train, y_train)  
  7 test_preds = pipe_xgb.predict(X_test)  
  8  
  9 print("Test data model score:")  
10 dt_score = model_score(pipe_xgb, X_test, test_preds, y_test)
```

Test data model score:  
The accuracy score is: 0.8161  
The recall score is: 0.8161  
The F1 score is: 0.8077  
The precision score is: 0.8102  
The cross validation score of accuracy is: 0.7787



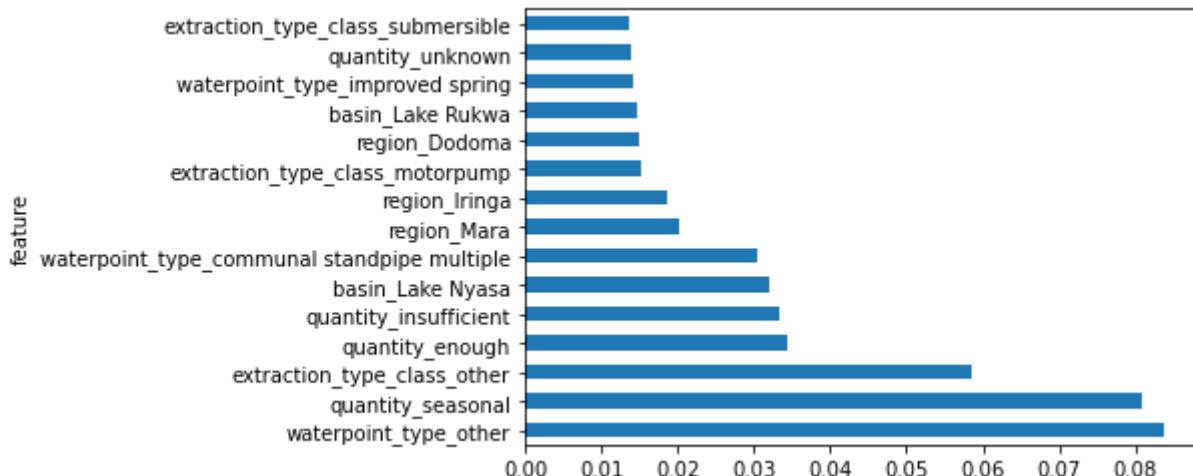
```
In [84]:  
  1 # Predict on training and test sets  
  2 training_preds = pipe_xgb.predict(X_train)  
  3 test_preds = pipe_xgb.predict(X_test)  
  4  
  5 # Accuracy of training and test sets  
  6 training_accuracy = accuracy_score(y_train, training_preds)  
  7 test_accuracy = accuracy_score(y_test, test_preds)  
  8  
  9 print('Training Accuracy: {:.4}%'.format(training_accuracy * 100))  
10 print('Validation accuracy: {:.4}%'.format(test_accuracy * 100))
```

Training Accuracy: 92.41%  
Validation accuracy: 81.61%

Our best performing model ended up being the XG Boost model with tuned hyperparameters, although the random forests model was not far behind with 81.22% testing accuracy. The model has overfitted the training data, but the testing accuracy is overperforming any other model at 81.61%.

### ▼ 6.7.1 XGB Feature Importances

```
In [85]: 1 feature_importances=pd.DataFrame(columns=['feature','importance'])
2
3 feature_importances['feature']=x_train.columns
4
5 feature_importances['importance']=xgb.feature_importances_
6
7 feature_importances= feature_importances.set_index('feature')
8
9 feature_importances['importance'].sort_values(ascending = False).head
```



Our XG Boost model shows the most important features to be waterpoint\_type\_other, quantity\_seasonal, extraction\_type\_class\_other, and quantity\_enough.

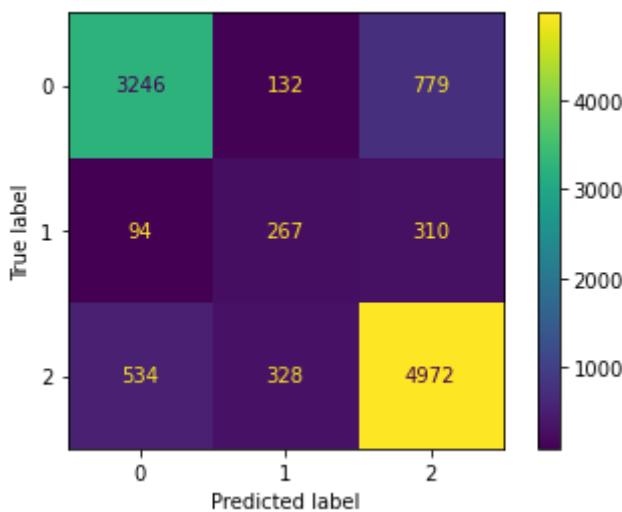
### ▼ 6.7.2 SMOTE and XGBoost

```
In [86]: 1 # Apply SMOTE oversampling
2
3 oversample = SMOTE(sampling_strategy = 'auto', n_jobs = -1)
4
5 X_train_smote, y_train_smote = oversample.fit_resample(X_train, y_train)
6
7 print(y_train.value_counts())
8 print(y_train_smote.value_counts())
```

```
2    23192
0    16672
1    2783
Name: status, dtype: int64
2    23192
1    23192
0    23192
Name: status, dtype: int64
```

```
In [87]:  
  1 # Instantiate XGB pipeline with SMOTE preprocessing  
  2 xgb_smote = Pipeline([('ss', StandardScaler()),  
  3                         ('RF', XGBClassifier(learning_rate=0.1, max_depth=5,  
  4                                     n_estimators=250, subsample=0.7))])  
  5 xgb_smote.fit(X_train_smote, y_train_smote)  
  6 test_preds = xgb_smote.predict(X_test)  
  7  
  8 print("Test data model score:")  
  9 dt_score = model_score(xgb_smote, X_test, test_preds, y_test)
```

Test data model score:  
The accuracy score is: 0.7958  
The recall score is: 0.7958  
The F1 score is: 0.7966  
The precision score is: 0.7987  
The cross validation score of accuracy is: 0.7787



```
In [88]:  
  1 # Predict on training and test sets  
  2 training_preds = xgb_smote.predict(X_train)  
  3 test_preds = xgb_smote.predict(X_test)  
  4  
  5 # Accuracy of training and test sets  
  6 training_accuracy = accuracy_score(y_train, training_preds)  
  7 test_accuracy = accuracy_score(y_test, test_preds)  
  8  
  9 print('Training Accuracy: {:.4}%'.format(training_accuracy * 100))  
 10 print('Validation accuracy: {:.4}%'.format(test_accuracy * 100))
```

Training Accuracy: 91.61%  
Validation accuracy: 79.58%

Our XG Boost model with SMOTE preprocessing gave us worse accuracy and more overfitting than the non SMOTE XG Boost model. The XG Boost model is our final model, yeilding 81.61% accuracy on test data.

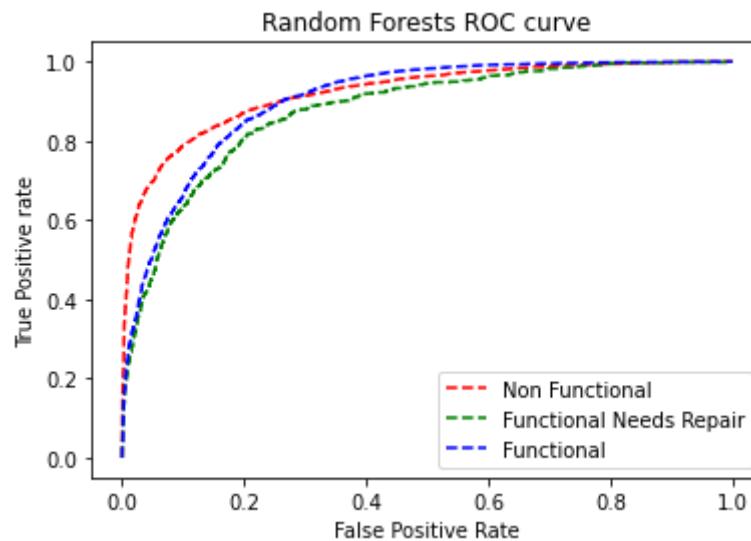
## 6.8 ROC AUC Analysis

```
In [89]:  
1 # Binarize for multiclass problem  
2 y_multi_test = label_binarize(y_test, classes=[0, 1, 2])  
3  
4 # Predict probability for each model  
5 dt_model_pred_proba = pipe_dt.predict_proba(X_test)  
6 rf_model_pred_proba = pipe_rf.predict_proba(X_test)  
7 knn_model_pred_proba = pipe_knn.predict_proba(X_test)  
8 log_model_pred_proba = pipe_lr.predict_proba(X_test)  
9 dummy_model_pred_proba = dummy.predict_proba(X_test)  
10 xgb_model_pred_proba = pipe_xgb.predict_proba(X_test)  
11  
12 # Calculate AUC ROC score using predicted probability for each model  
13 dt_test = roc_auc_score(y_multi_test, dt_model_pred_proba, multi_class='auto')  
14 rf_auc = roc_auc_score(y_multi_test, rf_model_pred_proba, multi_class='auto')  
15 knn_auc = roc_auc_score(y_multi_test, knn_model_pred_proba, multi_class='auto')  
16 log_model_auc = roc_auc_score(y_multi_test, log_model_pred_proba, multi_class='auto')  
17 dummy_auc = roc_auc_score(y_multi_test, dummy_model_pred_proba, multi_class='auto')  
18 xgb_auc = roc_auc_score(y_multi_test, xgb_model_pred_proba, multi_class='auto')  
19  
20 print(f'The AUC score for our baseline model is: {round(dummy_auc, 4)}')  
21 print(f'The AUC score for our decision tree model is: {round(dt_test, 4)}')  
22 print(f'The AUC score for our random forest model is: {round(rf_auc, 4)}')  
23 print(f'The AUC score for our k nearest neighbor model is: {round(knn_auc, 4)}')  
24 print(f'The AUC score for our logistic regression model is: {round(log_model_auc, 4)}')  
25 print(f'The AUC score for our XG Boost model is: {round(xgb_auc, 4)}')
```

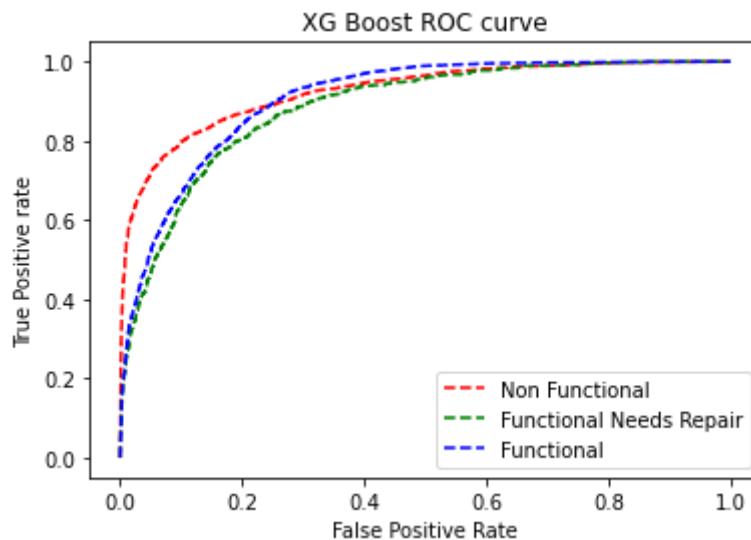
The AUC score for our baseline model is: 0.5029  
The AUC score for our decision tree model is: 0.7698  
The AUC score for our random forest model is: 0.8996  
The AUC score for our k nearest neighbor model is: 0.8503  
The AUC score for our logistic regression model is: 0.8315  
The AUC score for our XG Boost model is: 0.905

In [90]:

```
1 fig, ax = plt.subplots()
2
3 # Create dictionary
4 n_class = 3
5
6 fpr = {}
7 tpr = {}
8 thresh = {}
9
10 # Iterate through each class to create roc_curve
11 for i in range(n_class):
12     fpr[i], tpr[i], thresh[i] = roc_curve(y_test, rf_model_pred_proba
13
14 # plotting
15 plt.plot(fpr[0], tpr[0], linestyle='--', color='red', label='Non Functional')
16 plt.plot(fpr[1], tpr[1], linestyle='--', color='green', label='Functional Needs Repair')
17 plt.plot(fpr[2], tpr[2], linestyle='--', color='blue', label='Functional')
18 plt.title('Random Forests ROC curve')
19 plt.xlabel('False Positive Rate')
20 plt.ylabel('True Positive rate')
21 plt.legend(loc='best')
22 plt.savefig('./images/roc_rf.png');
```



```
In [91]: # Iterate through each class to create roc_curve
for i in range(n_class):
    fpr[i], tpr[i], thresh[i] = roc_curve(y_test, xgb_model_pred_proba)
# plotting
plt.plot(fpr[0], tpr[0], linestyle='--', color='red', label='Non Functional')
plt.plot(fpr[1], tpr[1], linestyle='--', color='green', label='Functional Needs Repair')
plt.plot(fpr[2], tpr[2], linestyle='--', color='blue', label='Functional')
plt.title('XG Boost ROC curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive rate')
plt.legend(loc='best')
plt.savefig('./images/roc_xgb.png');
```



Before performing the ROC AUC curves analysis, our best performing model was our XG Boost model, followed closely by our Random Forest model. The AUC of each model showed the same results again with XG Boost (AUC = 90.5%) outperforming all other models, followed closely by our Random Forests (AUC = 89.9%) model.

## ▼ 7 Conclusions

XG Boost was our top performing model, although Random Forests was not far behind. The poor performance of the K Nearest Neighbors, Decision Tree, and Logistic Regression models indicate that the data is not easily separable. Our XG Boost model performs with an 81.61% testing accuracy. The XG Boost model also boasted the highest AUC scores at 90.5%.

Based on my findings, I am confident to partner with the Tanzanian government to help solve their water crisis by predicting water pump failure. As we illustrated above, there is a high rate of non functional waterpoints in the southeast corner of Tanzania in Mtwara and Lindi, as well as up north in Mara, and the southwest in Rukwa. These areas need immediate attention as the situations here are critical. There are a high number of functional wells in Iringa, Shinyanga, Kilimanjaro, and Arusha. There is a cluster of functional but need repair waterpoints in Kigoma, these should be addressed to prevent failure which can be more expensive to repair. Several of our models showed

one of its most important features to be quantity enough for the waterpoint. There are over 8,000 waterpoints that have enough water in them but are non functional. These are a high priority to address as well since there is water present.