Tanzania Waterpoint Classification Model

Overview

In this notebook I attempt to build a supervised ternary classification model to predict the operational status of waterpoints in Tanzania. To do so, I analyze data on waterpoints and then build three machine learning models using chosen features from the dataset. Each model is evaluated and optimized.

Target Audience/Business Problem

Here I sought to build a model to predict waterpoint status and unlock insights that would be useful to the Tanzanian government or party interested in the maintenance/repair of waterpoints. By using a machine learning model to categorize waterpoints by operational status, time and resources could be theoretically better allocated. Waterpoints which need maintenance / repair could be prioritized without a visit to each.

Objective: Build a supervised classification model which can predict the operational status of a waterpoint belonging to one of three categories:

- Functional
- Non-functional
- Functional but needing repairs

Required Packages

In [88]:

```
import pandas as pd
import numpy as np

from matplotlib import pyplot as plt
import seaborn as sns

import pickle
import time
import folium
import math

from imblearn.pipeline import Pipeline as imbpipeline
from imblearn.over_sampling import SMOTE

from sklearn import svm
from sklearn.feature_selection import RFECV
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
from sklearn.model_selection import train_test_split, cross_val_score, KFold, GridSearc
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import classification_report, log_loss,\
accuracy_score, confusion_matrix, plot_confusion_matrix, make_scorer, mean_squared_erro
```

```
from sklearn.cluster import KMeans

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

I. Explore data / EDA Part 1

My first step was doing EDA on the data. This included creating a pandas dataframe from the csv files included, and then exploring it descriptively and visually to better understand it.

Ultimately this led to a better understanding of which columns to drop before model prototyping, and which to include.

In [89]:
```
training_values = pd.read_csv('tanzania_training_values.csv')
training_labels = pd.read_csv('tanzania_training_labels.csv')
df = training_values.merge(training_labels, on='id')
df.head()
```

Out[89]:

	id	amount_tsh	date_recorded	funder	gps_height	installer	longitude	latitude	wpt_name
0	69572	6000.0	2011-03-14	Roman	1390	Roman	34.938093	-9.856322	none
1	8776	0.0	2013-03-06	Grumeti	1399	GRUMETI	34.698766	-2.147466	Zahanati
2	34310	25.0	2013-02-25	Lottery Club	686	World vision	37.460664	-3.821329	Kwa Mahundi
3	67743	0.0	2013-01-28	Unicef	263	UNICEF	38.486161	-11.155298	Zahanati Ya Nanyumbu
4	19728	0.0	2011-07-13	Action In A	0	Artisan	31.130847	-1.825359	Shuleni

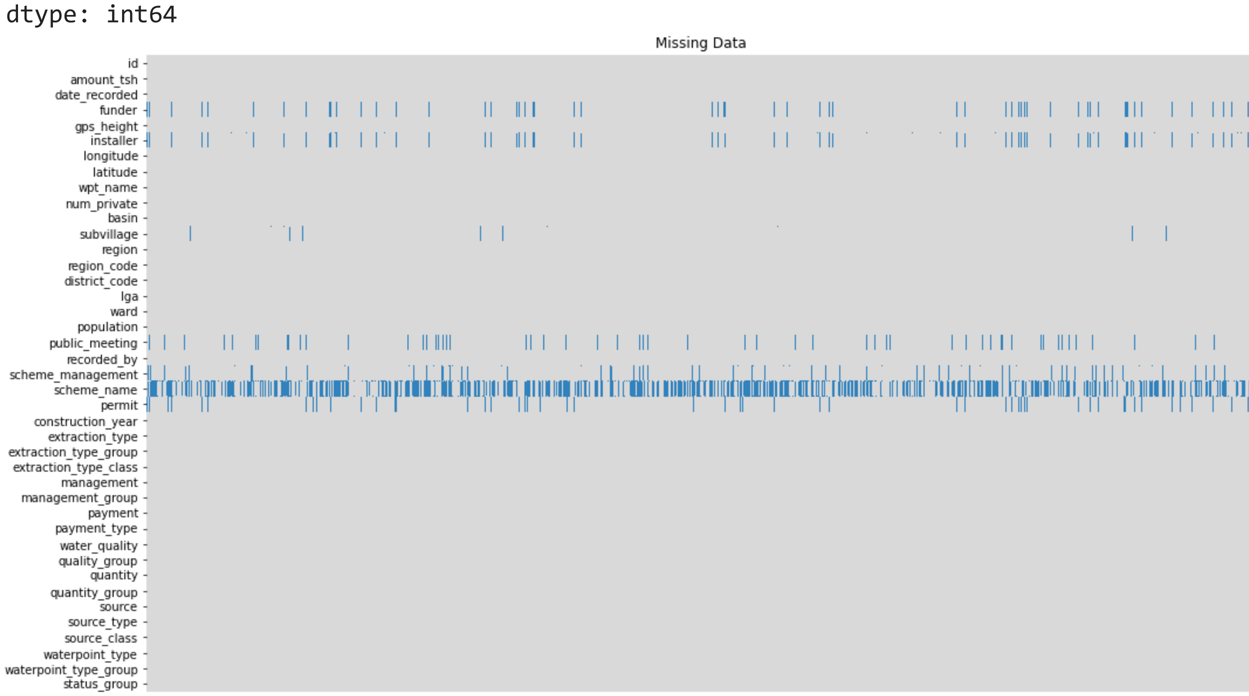
5 rows × 41 columns

Missing Values

In [7]:
```
# view null values
print("There are {} duplicates".format(df.duplicated().sum()))
print("\nSummary of null values:")
print(df.isna().sum())

plt.figure(figsize=(17,10))
sns.heatmap(df.isnull().transpose(), xticklabels = False, cbar = False, cmap = 'tab20c_
plt.title('Missing Data')
plt.show()
```

```
There are 0 duplicates

Summary of null values:
id                           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
status_group                 0
dtype: int64
```

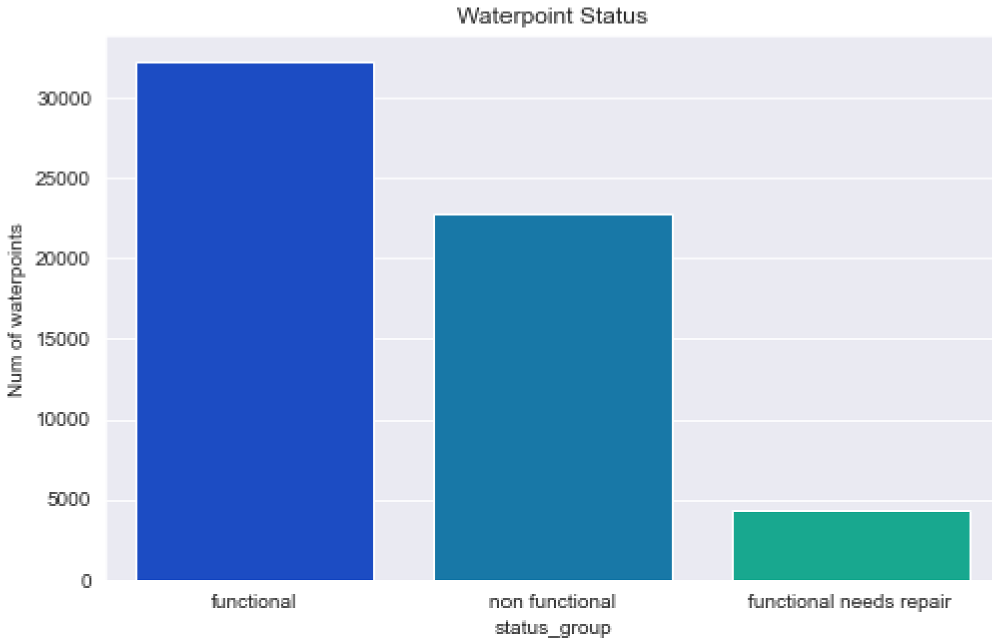


Plot of missing (null) values in the dataset. `scheme_name` had > 28,000 null values and so was determined at this step to be excluded from further analysis. Several other features had > 3K missing null values.

Labels Analysis

In [29]:
```
print(training_labels.status_group.value_counts(normalize=True))
plt.figure(figsize=(8,5))
sns.set_style('darkgrid')
sns.countplot(df.status_group, alpha=1, palette='winter')
plt.title('Waterpoint Status')
plt.ylabel('Num of waterpoints')
plt.show()
```

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



Only about 7% of the labels in the dataset belonged to the 'functional needs repair' group. In order to build a model to better include this class in predictions, the use of resampling is used below (Section 3).

EDA Geographical Data

EDA of `latitude` and `longitude` as gps coordinates.

In [90]:
```
def plot_lat_long(df):
    m = folium.Map(width=550, height=350, location=[df.latitude.median(), df.longitude.

    functional = df[df.status_group == 'functional']
    repair = df[df.status_group == 'functional needs repair']
    non_functional = df[df.status_group == 'non functional']

    functional_fg = folium.FeatureGroup(name='Functional')
    repair_fg = folium.FeatureGroup(name='Functional Needs Repair')
    non_functional_fg = folium.FeatureGroup(name='Non Functional')
```

```
    # functional
    for lat, long in zip(functional.latitude, functional.longitude):
        loc = [lat,long]
        folium.Circle(location=loc, color = 'red', radius=5, opacity=.4, tooltip=f'lat:

    # functional needs repair
    for lat, long in zip(repair.latitude, repair.longitude):
        loc = [lat,long]
        folium.Circle(location=loc, color = 'blue', radius=5, opacity=.4, tooltip=f'lat

    # non functional
    for lat, long in zip(non_functional.latitude, non_functional.longitude):
        loc = [lat,long]
        folium.Circle(location=loc, color = 'yellow', radius=5, opacity=.4, tooltip=f'l

    m.add_child(functional_fg)
    m.add_child(repair_fg)
    m.add_child(non_functional_fg)

    # turn on layer control
    m.add_child(folium.map.LayerControl())

    display(m)
```

In [91]:
```
plot_lat_long(df)
```

Make this Notebook Trusted to load map: File -> Trust Notebook

Zooming out revealed that while the majority of waterpoints were in Tanzania, some were recorded with a latitude and longitude that was clearly not. Hovering over the point showed that all of those waterpoints placed in the ocean were located at the same latitude and longitude which made it easy to identify them:

In [10]:
```
print(f'Number of incorrectly placed waterpoints: {len(df[df.longitude == 0])}')
```

```
Number of incorrectly placed waterpoints: 1812
```

EDA Numeric Variables

EDA on `population` , `amount_tsh` , `gps_height` , `construction_year`

- `amount_tsh` : Total static head (amount water available to waterpoint)
- `gps_height` : Altitude of the well
- `population` : Population around the well
- `construction_year` - Year the waterpoint was constructed
- `num_private` - not described

In [287…

```
print(df.num_private.unique())
print(df.num_private.nunique())
```

```
[   0   39    5   45    6    3  698   32   15    7   25  102    1   93
   14   34  120   17  213   47    8   41   80  141   20   35  131    4
   22   11   87   61   65  136    2  180   38   62    9   16   23   42
   24   12  668  672   58  150  280  160   50 1776   30   27   10   94
   26  450  240  755   60  111  300   55 1402]
65
```

Since `num_private` was not described it is unclear how to include this in the data. There were 65 unique numbers, but it is unclear if they are ordinal, continuous or categorical. Due to this, `num_private` was dropped.

In [8]:

```
print(len(df[df.construction_year == 0]))
print(len(df[df.construction_year != 0]))
```

```
20709
38691
```

In [214…

```
df.construction_year.unique()
```

Out[214…

```
array([1999, 2010, 2009, 1986,    0, 2011, 1987, 1991, 1978, 1992, 2008,
       1974, 2000, 2002, 2004, 1972, 2003, 1980, 2007, 1973, 1985, 1970,
       1995, 2006, 1962, 2005, 1997, 2012, 1996, 1977, 1983, 1984, 1990,
       1982, 1976, 1988, 1989, 1975, 1960, 1961, 1998, 1963, 1971, 1994,
       1968, 1993, 2001, 1979, 1967, 2013, 1969, 1981, 1964, 1966, 1965],
      dtype=int64)
```

There were a signgicant number of missing values in `construction_year` (represented as '0').

In [61]:

```
print(df.amount_tsh.describe())
print('\nmean amount_tsh for top 50% of data: {}'.format(df.sort_values(by='amount_tsh'
print(f'\nPercentage of data greater than the amount_tsh mean: {len(df[df.amount_tsh >
```

```
count     59400.000000
mean        317.650385
std        2997.574558
min           0.000000
25%           0.000000
50%           0.000000
75%          20.000000
max      350000.000000
Name: amount_tsh, dtype: float64

mean amount_tsh for top 50% of data: 635.3007693602694

Percentage of data greater than the amount_tsh mean: 13.360269360269362
```

- `amount_tsh` had a very large range (0 - ~30,000) in values.

- Roughly 70% of the records indicate a value of 0 for `amount_tsh`. Only 180 records above 10,000.

Due to the large range visualization wasn't useful.

View differences amongst the four different status groups.

In [280…

```
# drop records with a year of 0, as these are unknown values
df[df.construction_year > 0].groupby('status_group').construction_year.median().reset_i
```

Out[280…

	status_group	construction year median
0	functional	2003
1	functional needs repair	1998
2	non functional	1994

In [198…

```
df.groupby('status_group').population.mean().reset_index()
```

Out[198…

	status_group	population
0	functional	187.553303
1	functional needs repair	175.102154
2	non functional	170.016430

In [15]:

```
df.groupby('status_group').amount_tsh.mean().reset_index()
```

Out[15]:

	status_group	amount_tsh
0	functional	461.798235
1	functional needs repair	267.071577
2	non functional	123.481230

In [16]:

```
df.groupby('status_group').gps_height.median().reset_index()
```

Out[16]:

	status_group	gps_height
0	functional	550
1	functional needs repair	385
2	non functional	293

At first glance it doesn't appear that there is much of a difference between status groups for `population`, but that there are more significant differences among `amount_tsh`, `gps_height` and `construction_year`.

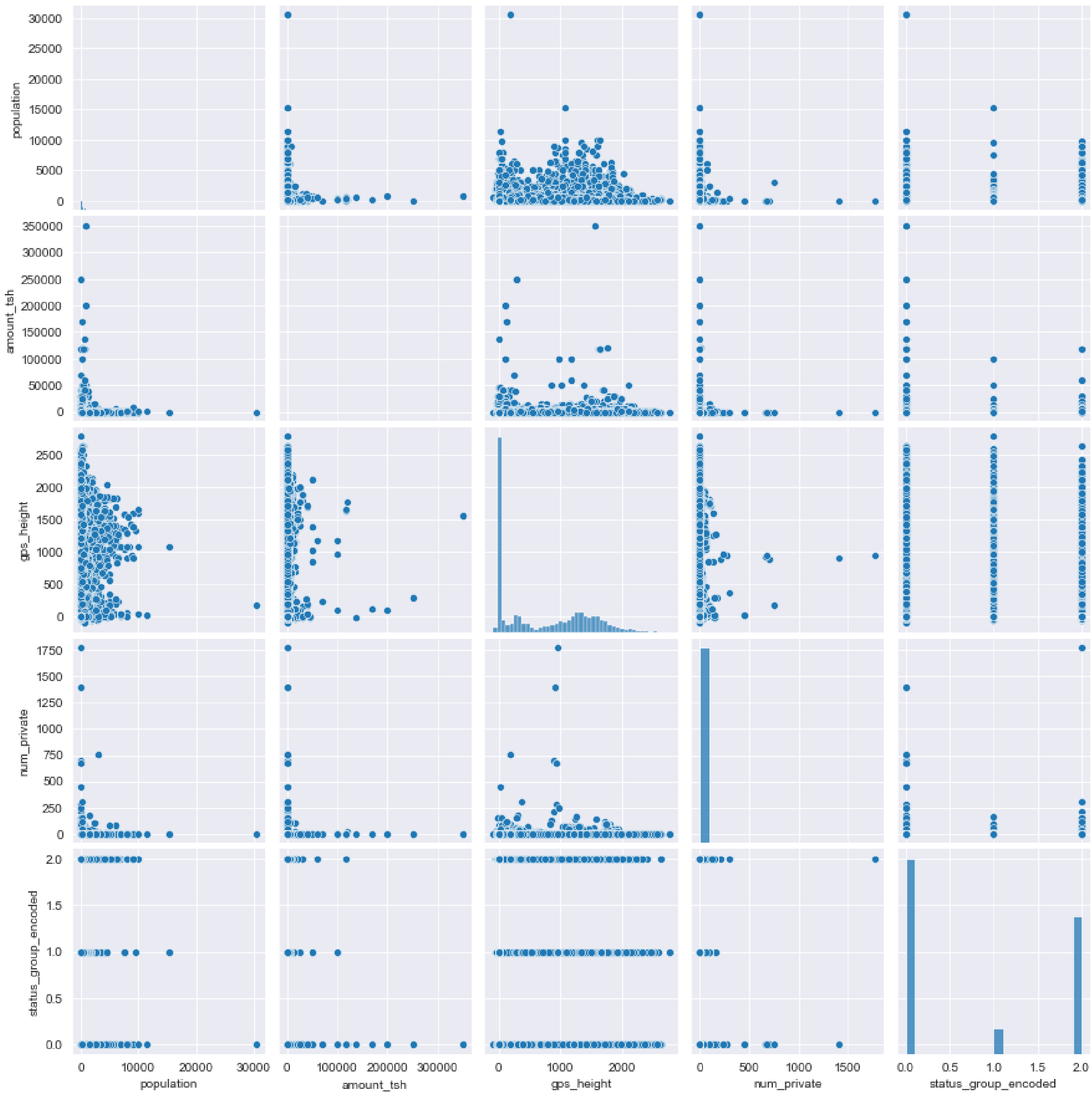
Visual Analysis of numeric variables

Labels need to be mapped to numbers in order to prep them for plotting and further analysis.

In [7]:
```python
def map_labels(x):
    if x == 'functional':
        return 0
    elif x == 'functional needs repair':
        return 1
    else:
        return 2
df['status_group_encoded'] = df.status_group.apply(lambda x: map_labels(x))
```

In [32]:
```python
sns.pairplot(df[['population','amount_tsh','gps_height','num_private','status_group_enc
```

Out[32]:
```
<seaborn.axisgrid.PairGrid at 0x264023ebbb0>
```

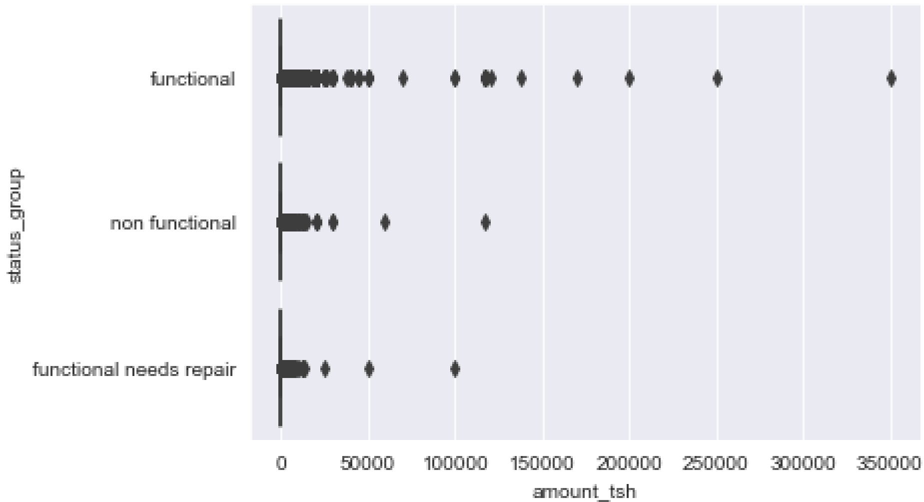


While `population` and `gps_height` don't seem to differ by group, it looks like the highest values for `amount_tsh` are likely to come from functional water wells.

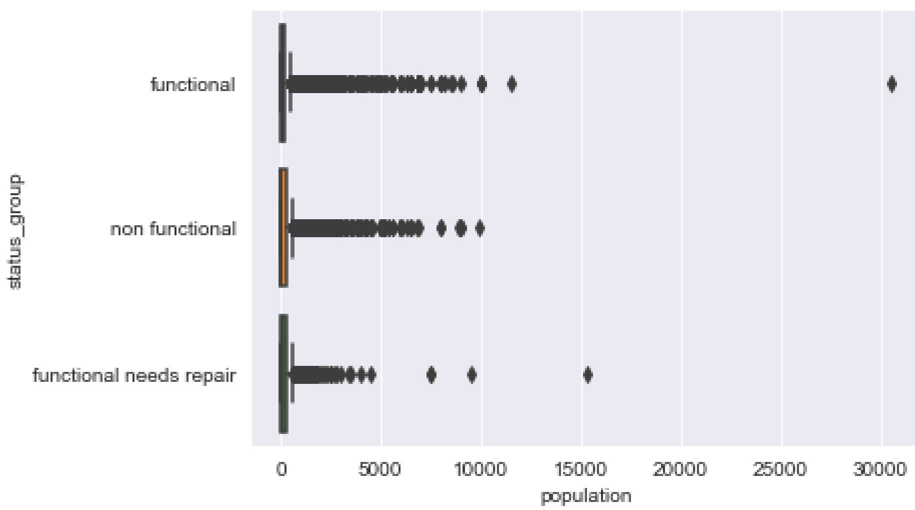
Histograms and Boxplots

Plot histograms and boxplots for each of the three continuous variables to determine if there were any relationships between each and status group.

In [278...
```
sns.set_style('darkgrid')
sns.boxplot(x='amount_tsh', y='status_group', data=df)
plt.show()
```



In [299...
```
sns.boxplot(x='population', y='status_group', data=df)
plt.show()
```

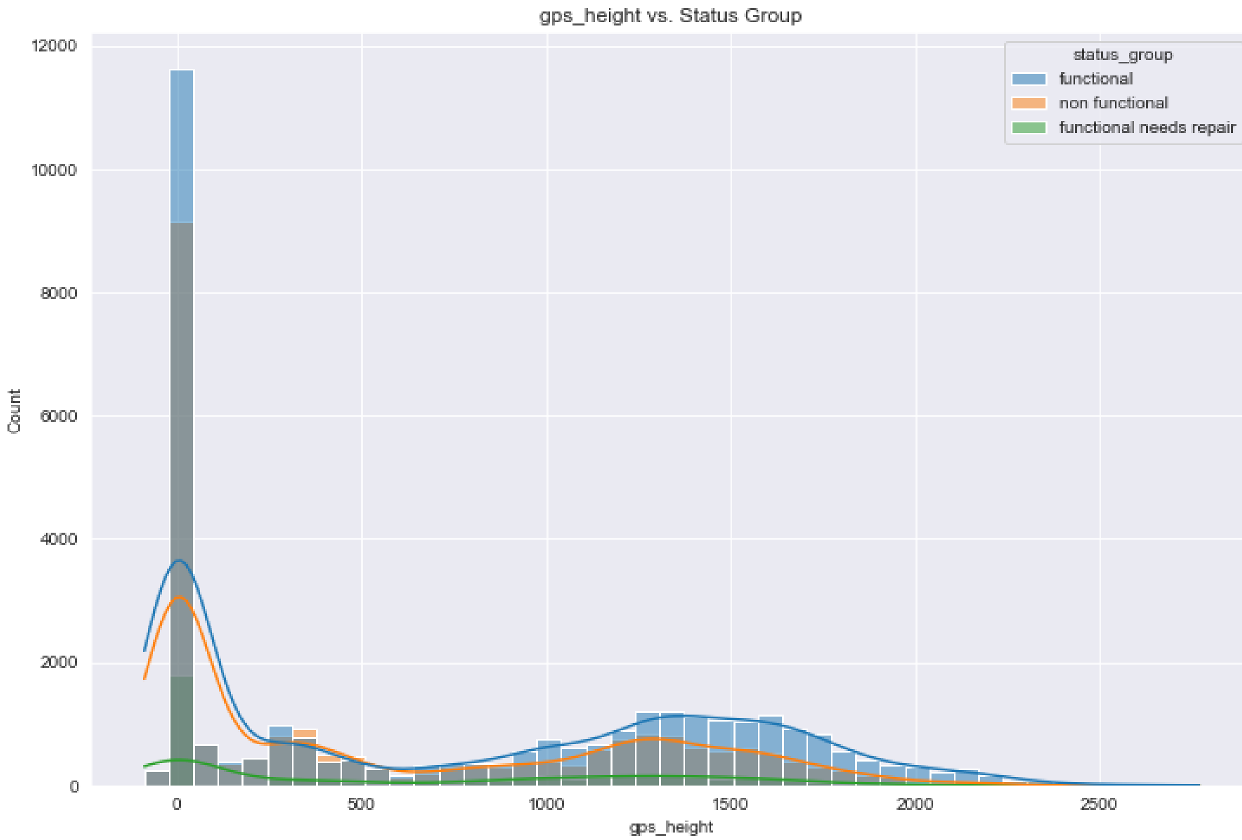


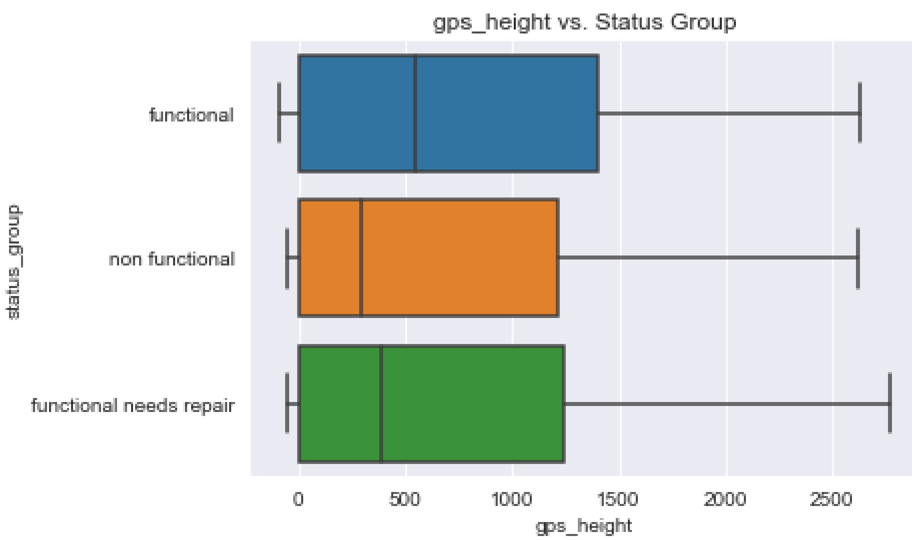
In [312...
```
len(df[df.population > 10000])
```

Out[312... 3

In [304...
```
def plot_continuous(col, df):
    fig, ax = plt.subplots(figsize=(12,8))
    sns.set_style('darkgrid')
    sns.histplot(x=col, hue='status_group', data=df, kde=True)
    plt.title('{} vs. Status Group'.format(col))
    plt.show()
    sns.boxplot(x=col, y='status_group', data=df)
    plt.title('{} vs. Status Group'.format(col))
    plt.show()
```

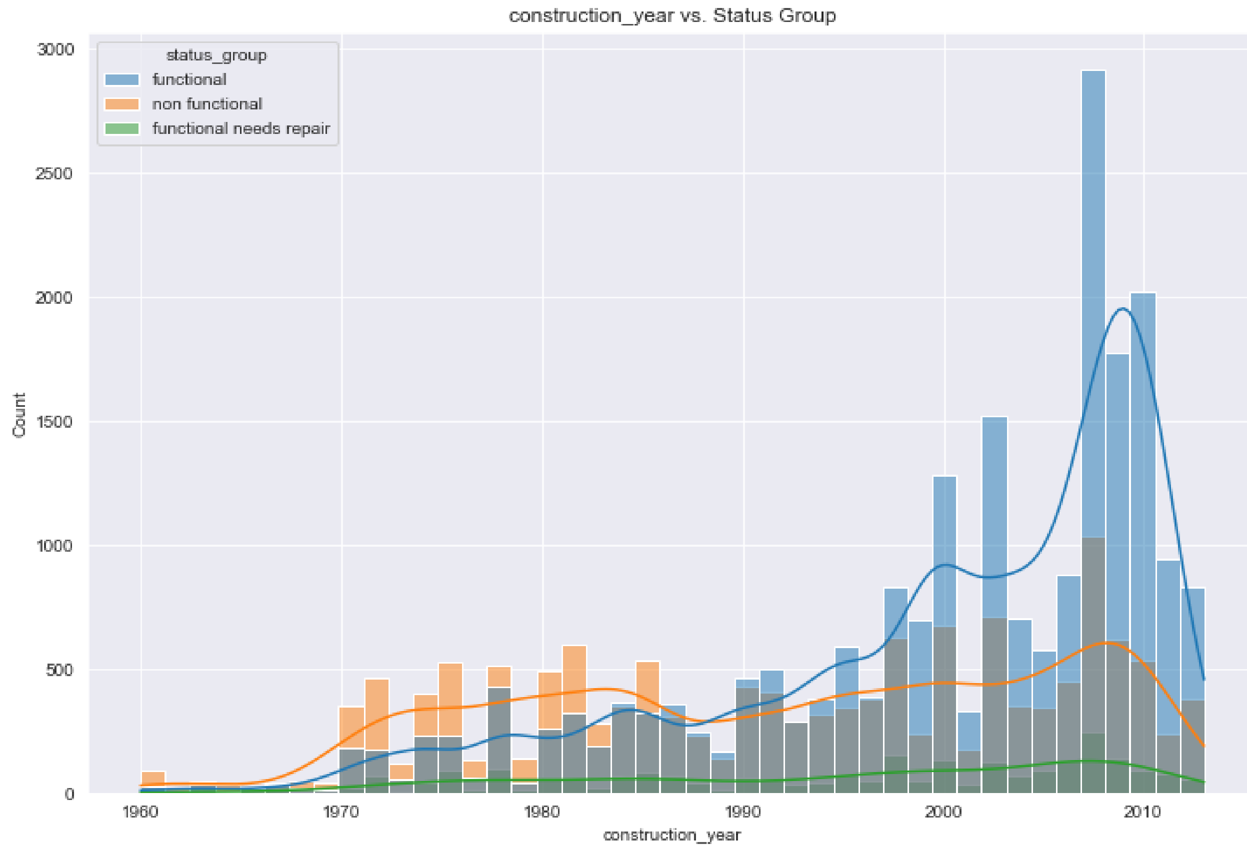
In [200...
```
plot_continuous('gps_height', df)
```

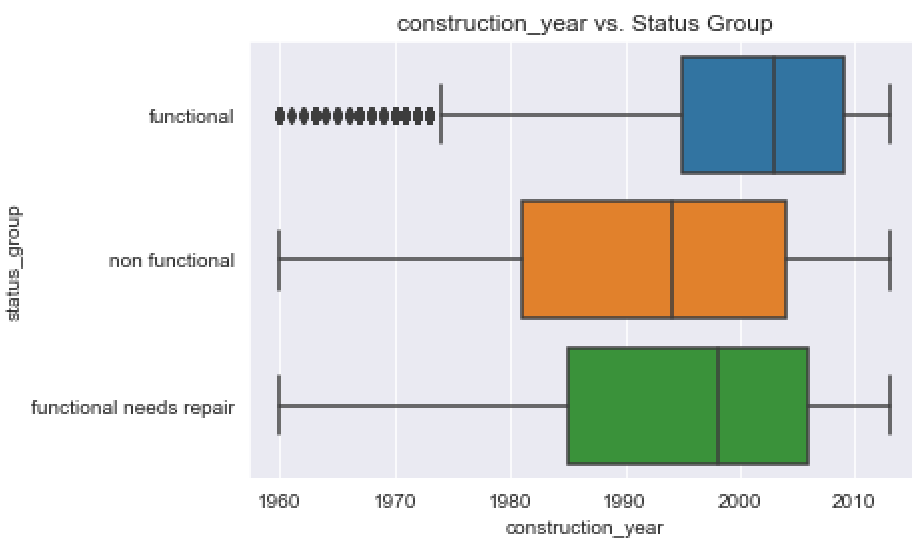




In [191…

```
plot_continuous('construction_year', df[df.construction_year > 0])
```





Interpretation: Distributions between status groups seem to be fairly similar for population and gps height. As seen in the boxplots and above, there does seem to be a material difference in mean for `gps_height` as well as `construction_year`.

EDA Categorical Variables

Most of the variables in the dataset were categorical. This involved looking at numeric information regarding the features as well as exploring relationships with status group visually.

- `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

Visual EDA

During this section, categorical variables were inspected visually. First, they were grouped together by type. Many groupings reference the same general information (ie: `waterpoint_type` and `waterpoint_type_group`).

In [377…

```
# only group the features with less than 50 unique values
pd.DataFrame(df.drop(['amount_tsh','num_private','latitude','longitude','id','gps_heigh
```

Out[377...

```
wpt_name                  False
subvillage                False
scheme_name               False
installer                 False
ward                      False
funder                    False
lga                       False
construction_year         False
region_code                True
region                     True
district_code              True
extraction_type            True
extraction_type_group      True
scheme_management          True
management                 True
source                     True
basin                      True
water_quality              True
payment                    True
waterpoint_type            True
source_type                True
payment_type               True
extraction_type_class      True
quality_group              True
waterpoint_type_group      True
management_group           True
quantity                   True
quantity_group             True
source_class               True
status_group               True
public_meeting             True
permit                     True
recorded_by                True
Name: 0, dtype: bool
```

In [424...

```python
# group categorical variables together
location = ['basin','region','region_code','district_code']
others = ['public_meeting', 'permit']
who = ['scheme_management']
extraction = ['extraction_type','extraction_type_group','extraction_type_class']
management = ['management','management_group']
payment = ['payment','payment_type']
quality = ['water_quality','quality_group']
quantity = ['quantity','quantity_group']
source = ['source','source_type','source_class']
waterpoint_type = ['waterpoint_type','waterpoint_type_group']
cat_vars = location + others + who + extraction + management + payment + quality + quan
len(cat_vars)
```

Out[424... 23

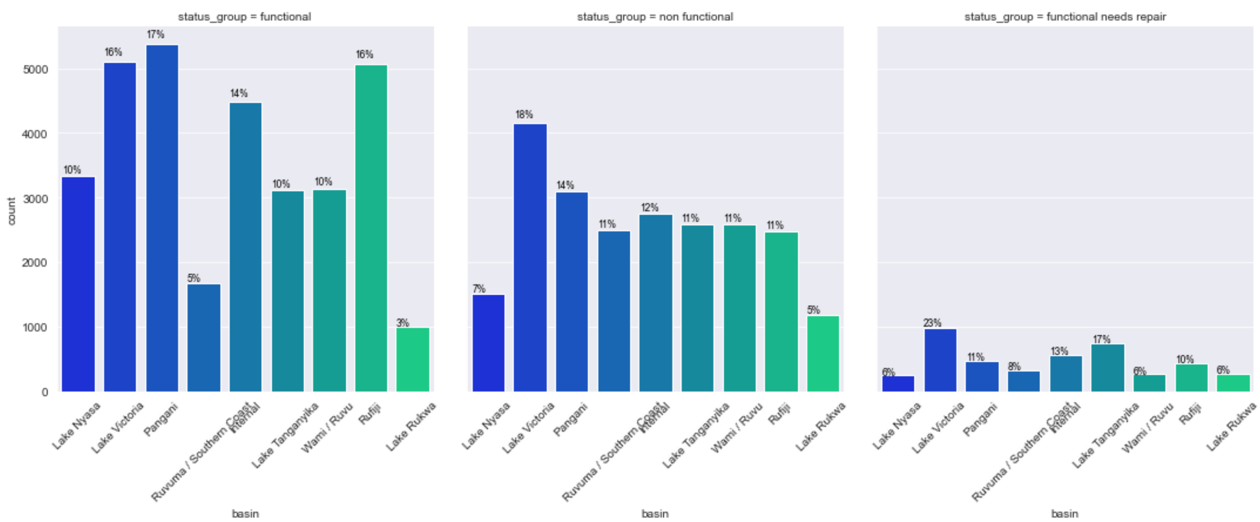
In [245...

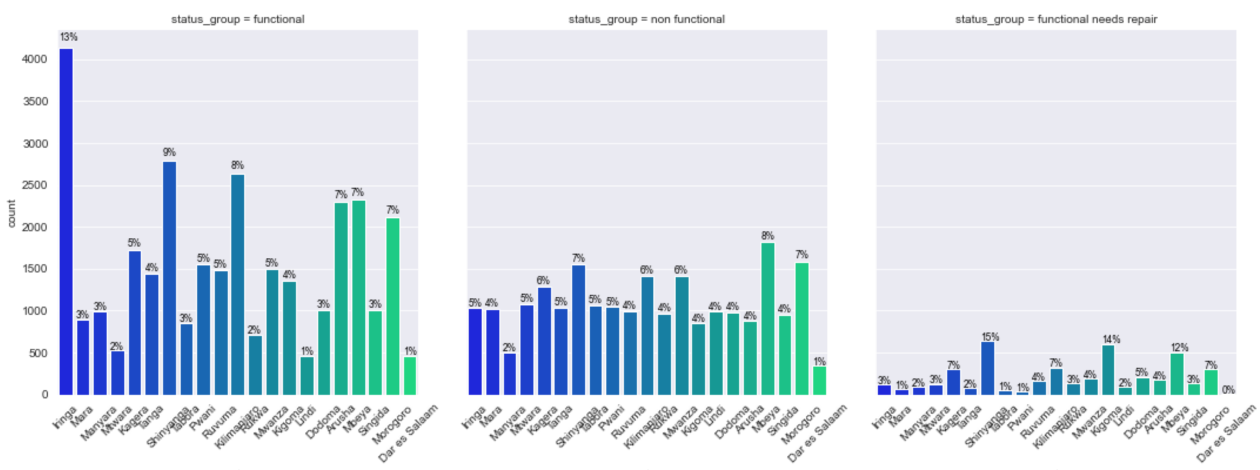
```python
"""
Plot each feature into a set of subplots
Each subplot answers the question:
For each status group, how many of each category is represented in the data?

Determine if there is a relationship between the feature and status group
for each plot, every category annotated with % representation of that category
"""
def count_plot_by_group(col):
    sns.set_style("darkgrid")
    temp = sns.catplot(x=col, kind='count', col='status_group', data=df, height=5, pale
```

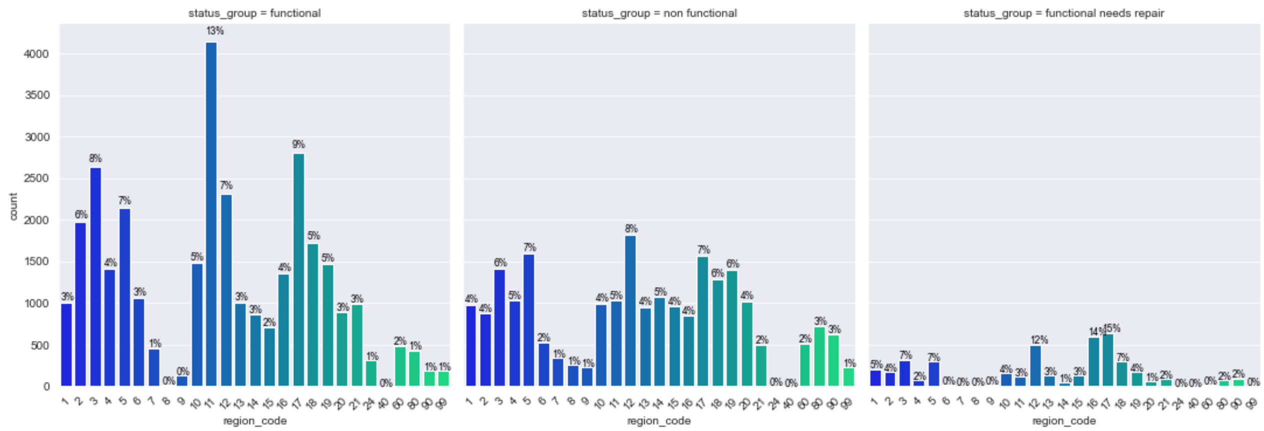
```
    temp.set_xticklabels(rotation = 45)

    for current_plot in range(df.status_group.nunique()):
        ax = temp.facet_axis(0,current_plot)
        for p in ax.patches:
            group = ax.title.get_text().split(" = ")[1]
            total = len(df[df.status_group == group])
            if np.isnan(p.get_height()):
                height = 0
            else:
                height = p.get_height()
            ax.text(p.get_x()+.015,
                    height*1.02,
                    '{:0.0f}%'.format(height / total * 100),
                    color='black',
                    rotation='horizontal',size='small')
    plt.show()
```

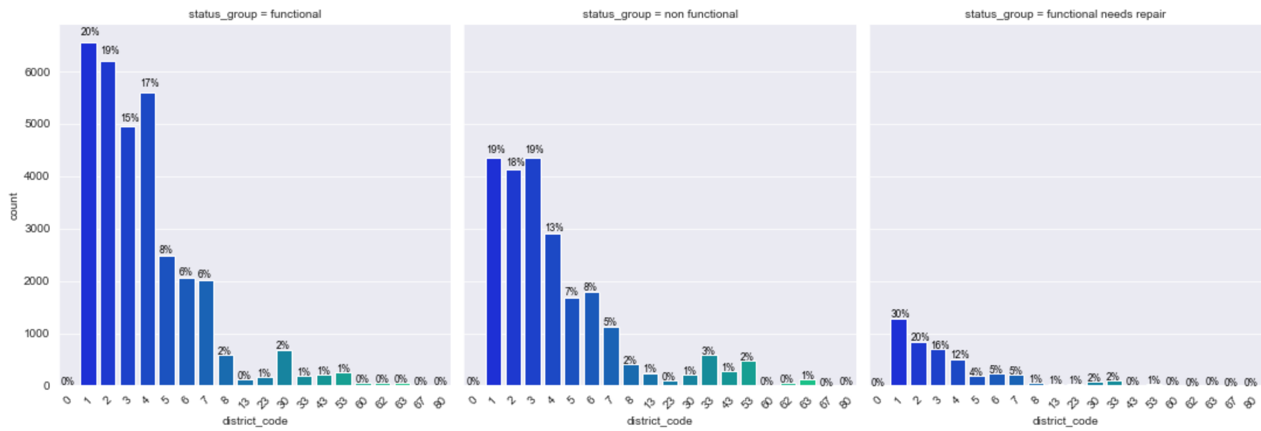
In [425…

```
for col in location:
    count_plot_by_group(col)
```









Conclusion: The distributions for `district_code` look to be most similar, meaning no significant relationship between `district_code` and `status_group` seems present in the histograms. There does seem to a difference for `basin`, `region` and `region_code`.

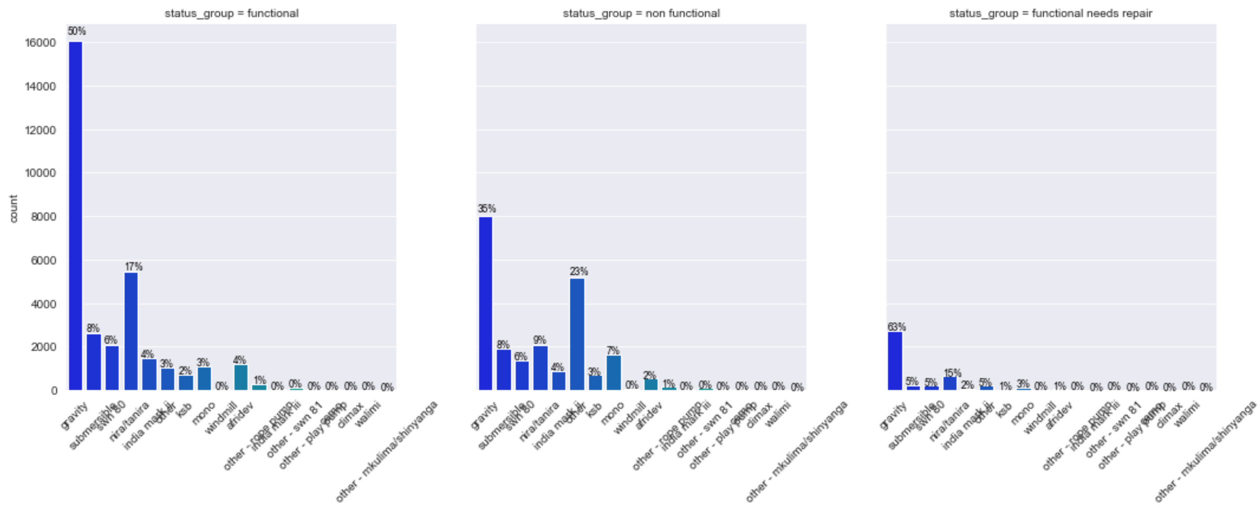
- Drop: `region_code` (too many unique values), `district_code` (no differences observed)

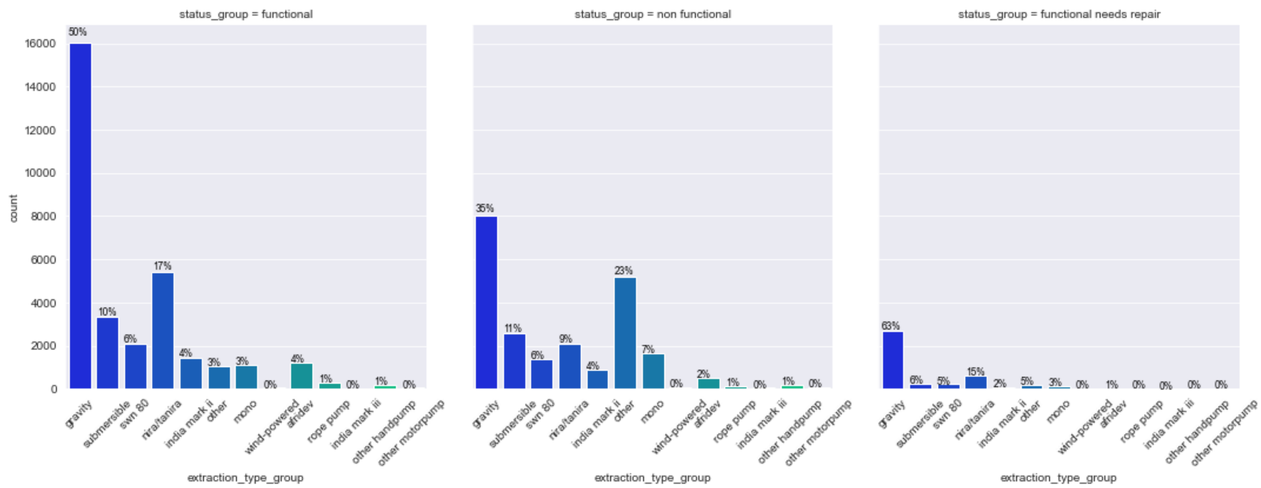
In [90]:
```
# regions with the most non functional waterpoints
nf_region_count = df[df.status_group == 'non functional'].groupby('region').id.count().
nf_region_count
```

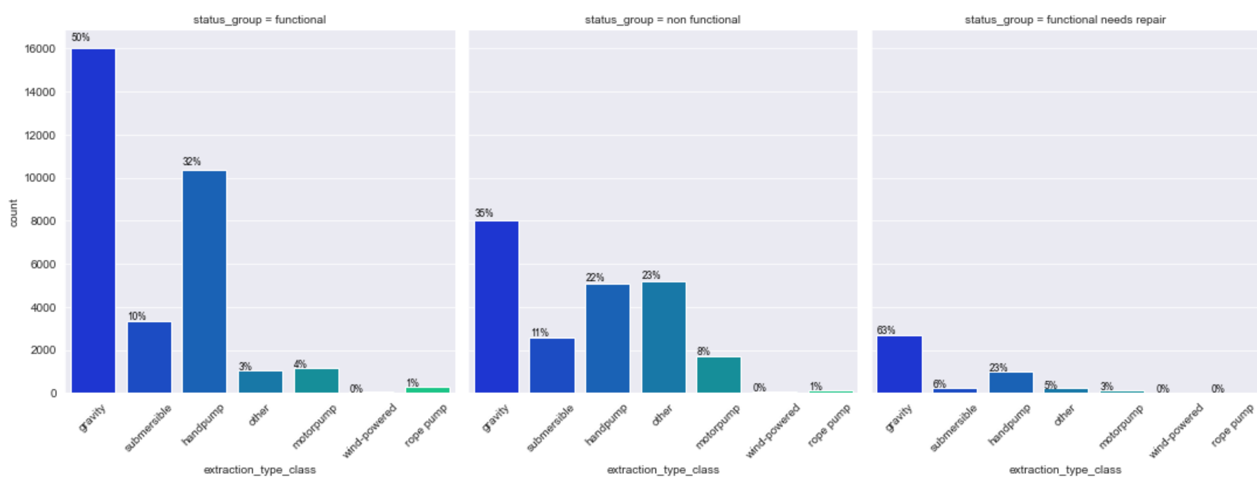
Out[90]:

	region	count
10	Mbeya	1816
11	Morogoro	1587
17	Shinyanga	1558
6	Kilimanjaro	1417
13	Mwanza	1417

In [41]:
```
for col in extraction:
    count_plot_by_group(col)
```



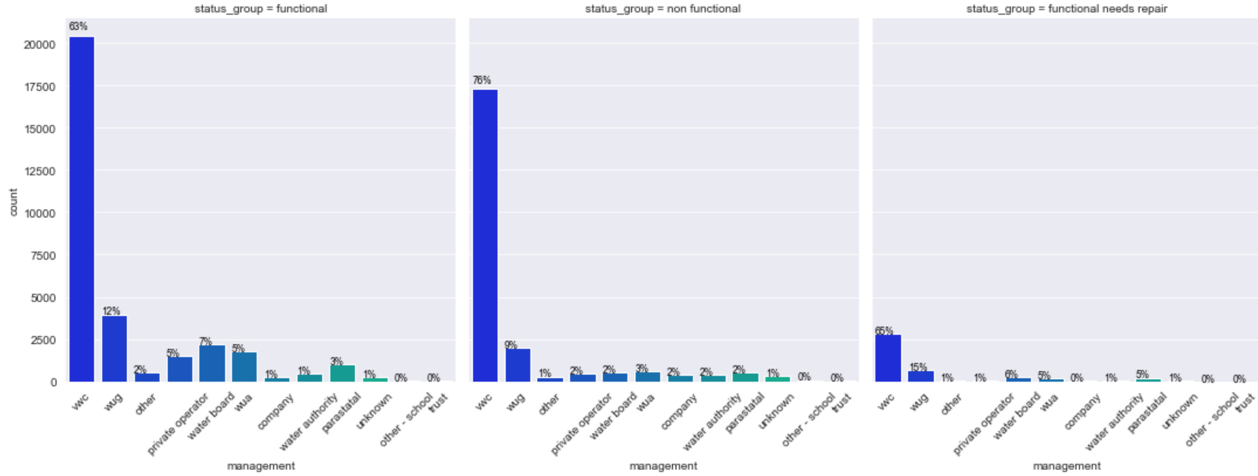


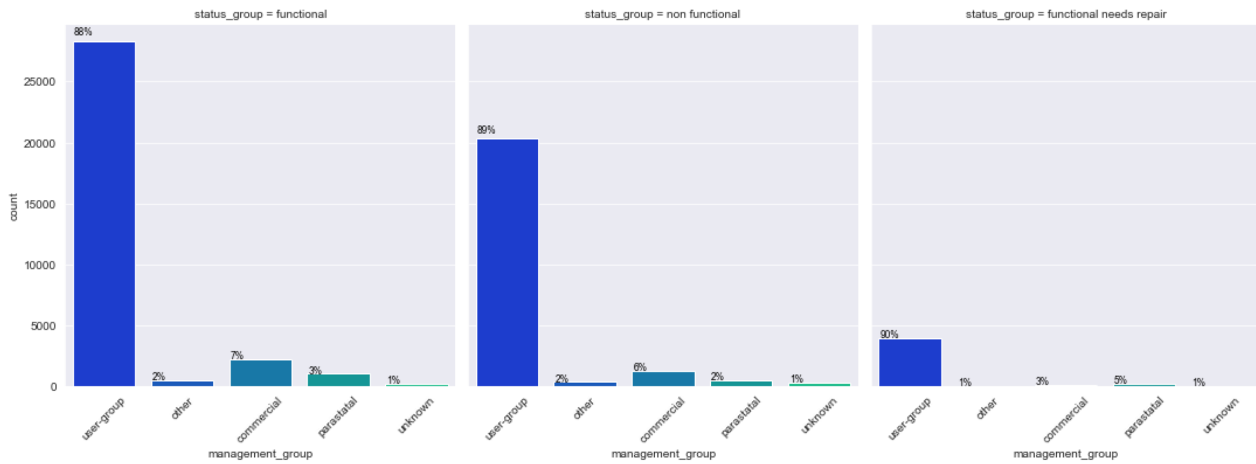


Some differences between status groups for the three above. More significant differences observed for `extraction_type_class` , notably under both 'gravity' and 'other' categories.

- Drop: `extraction_type_group` (redundant), `extraction_type_class` (redundant)

In [42]:
```
for col in management:
    count_plot_by_group(col)
```

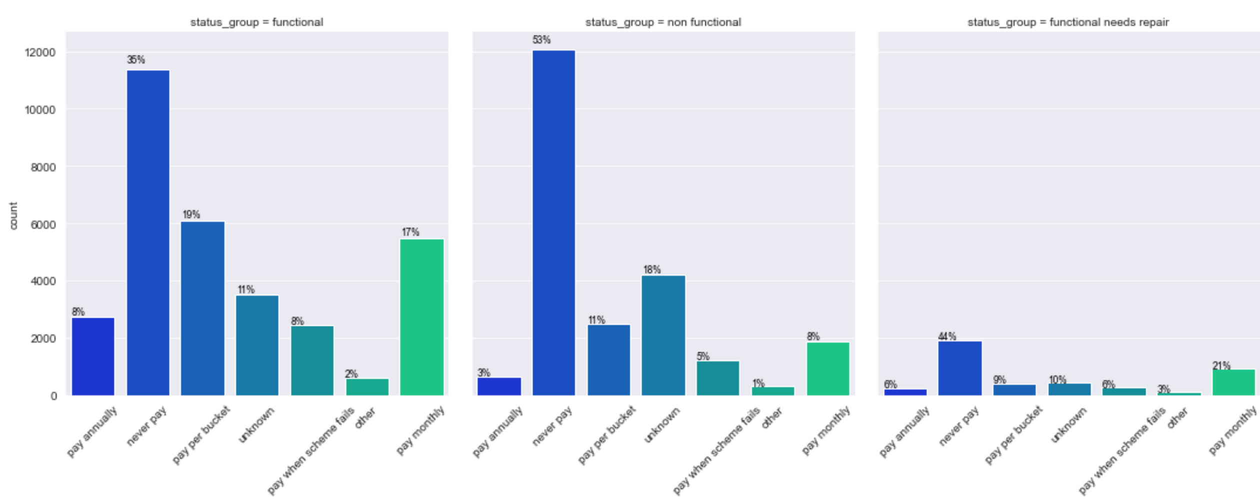


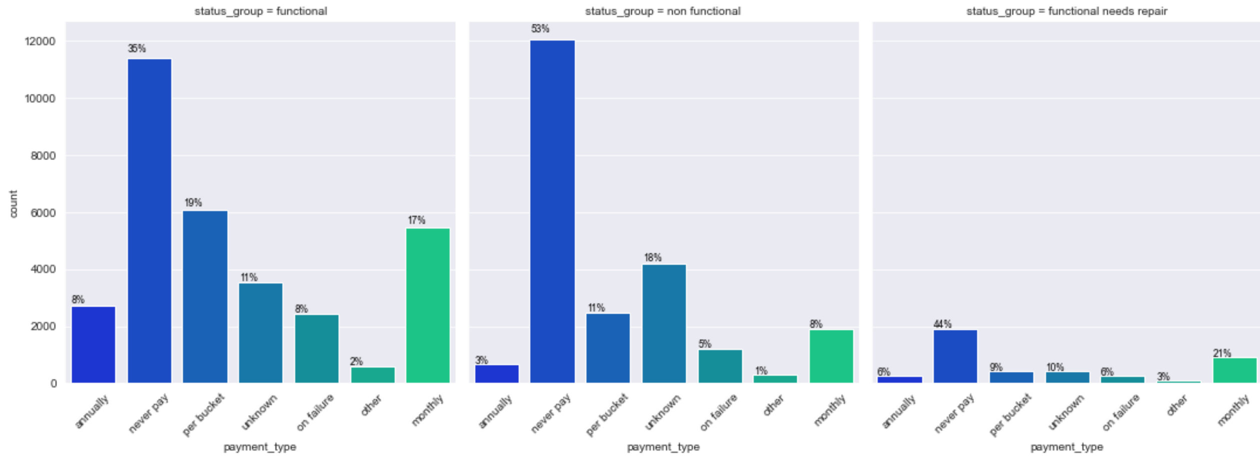


No relationship observed for `management_group` . There does appear to be a slight difference between status groups for `management` .

- Dropped: `management_group` (no differences observed)

In [43]:
```
for col in payment:
    count_plot_by_group(col)
```

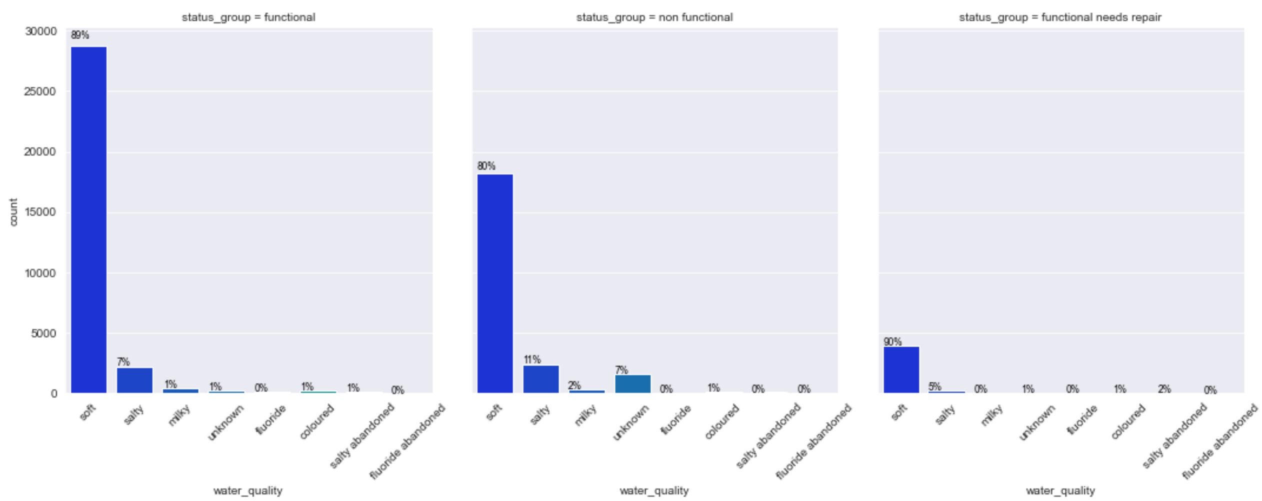


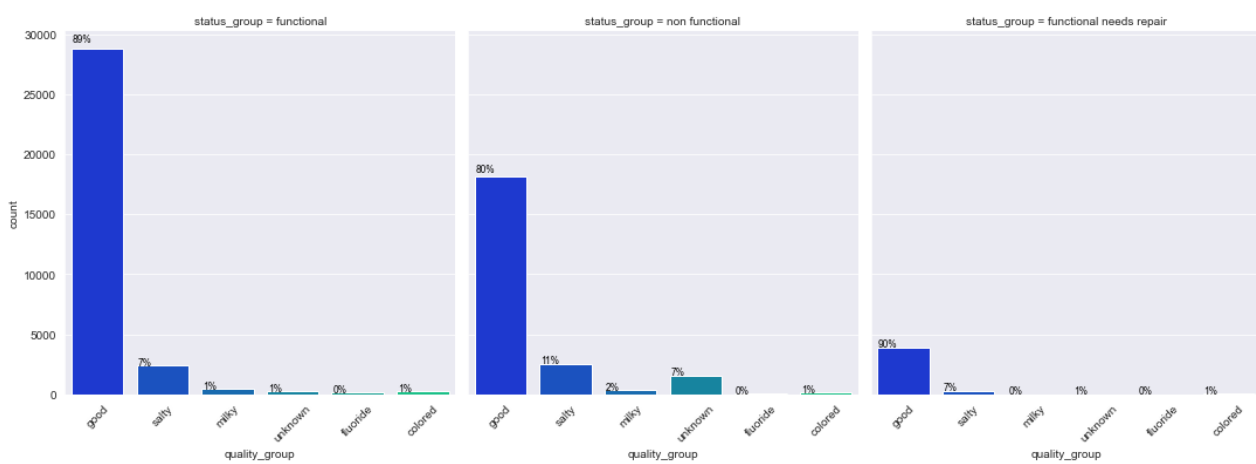


Some differences noticed in the distributions for `payment_type` and `payment`. Distributions look identical. 1 of the columns can be dropped.

- Drop: `payment` (redundant)

In [45]:
```
for col in quality:
    count_plot_by_group(col)
```



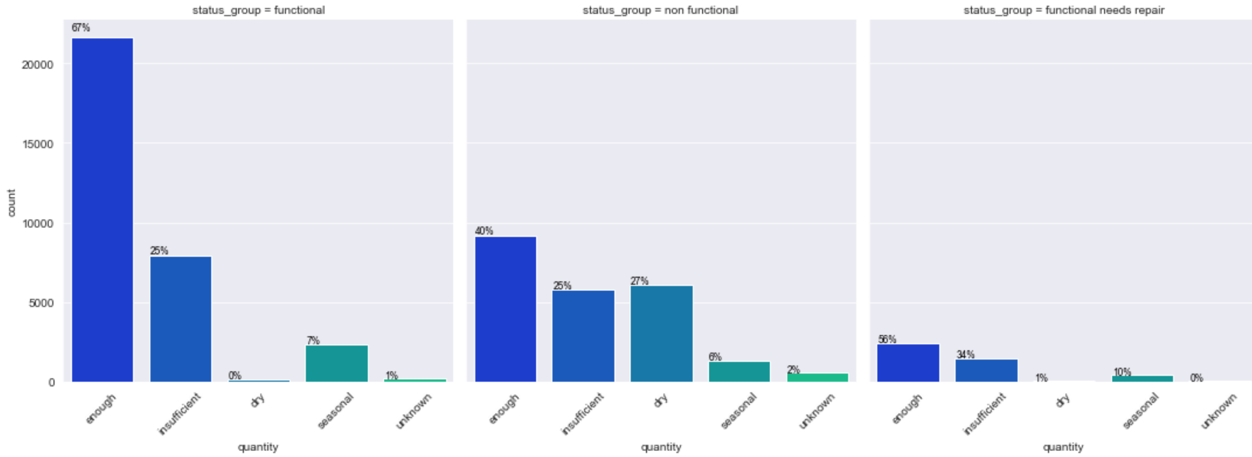


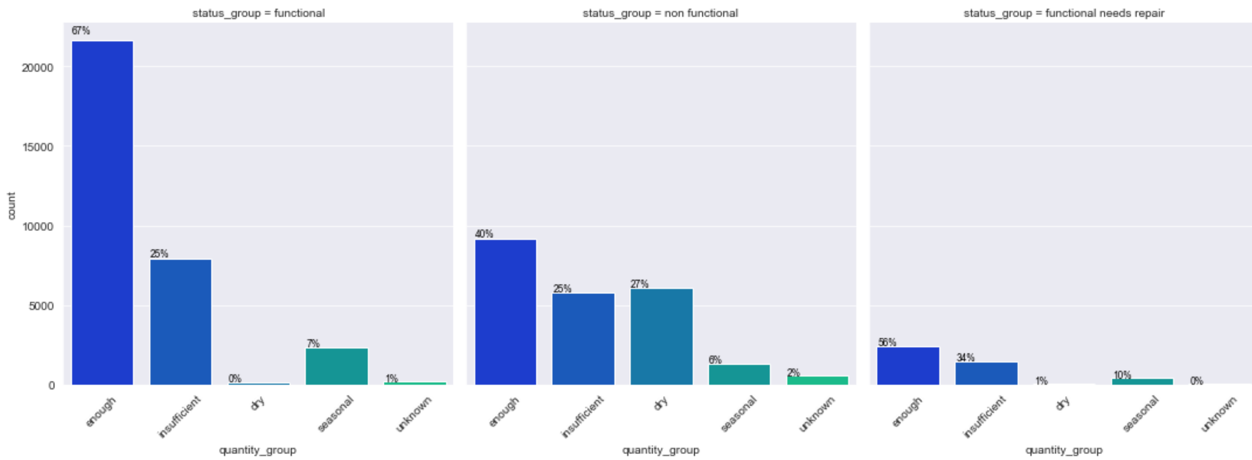
Some differences observed between differernt status groups. Distributions for `water_quality` and `quality_group` look very similar. More dimensions for `water_quality`

- Dropped: `quality_group` (redundant)

In [46]:

```python
for col in quantity:
    count_plot_by_group(col)
```



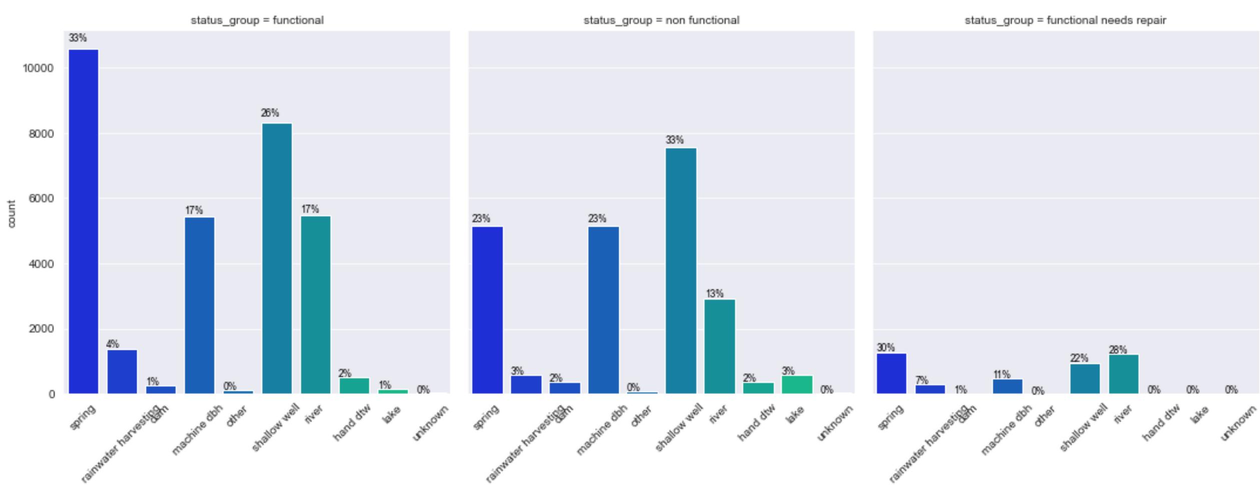


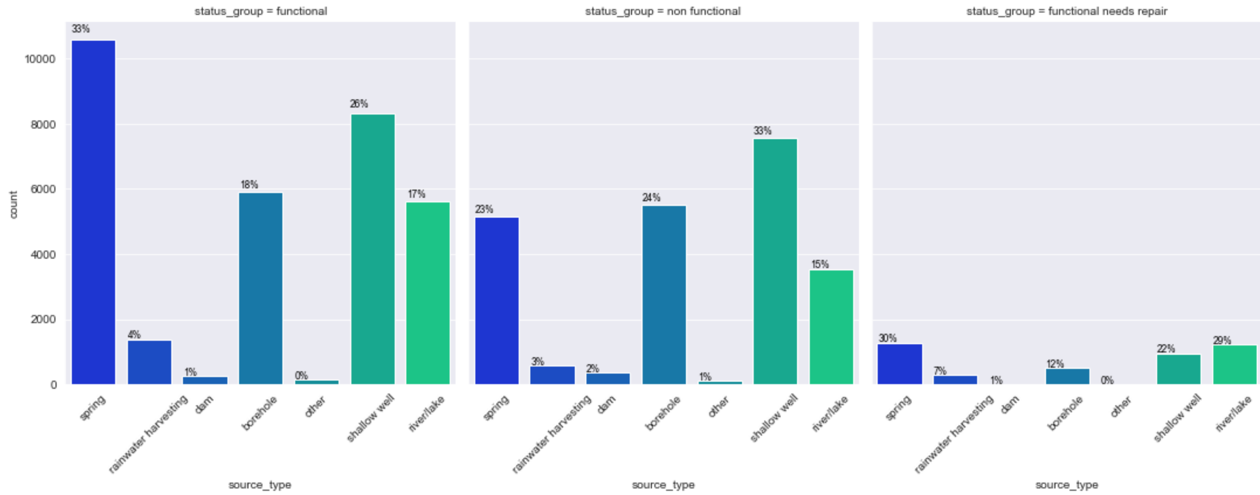
There appears to be some relationship between `quantity` / `quantity_group` and `status_group` : namely, dry wells are more likely to be in the non functional group. Distributions look identical - 1 of the columns can be dropped.

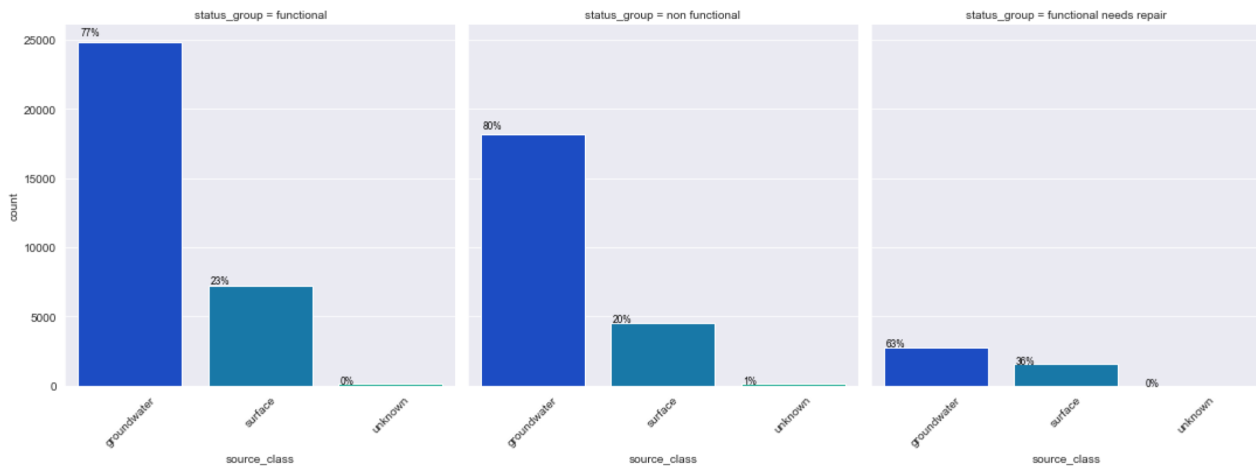
- Drop: `quantity` (redundant)

In [48]:

```python
for col in source:
    count_plot_by_group(col)
```

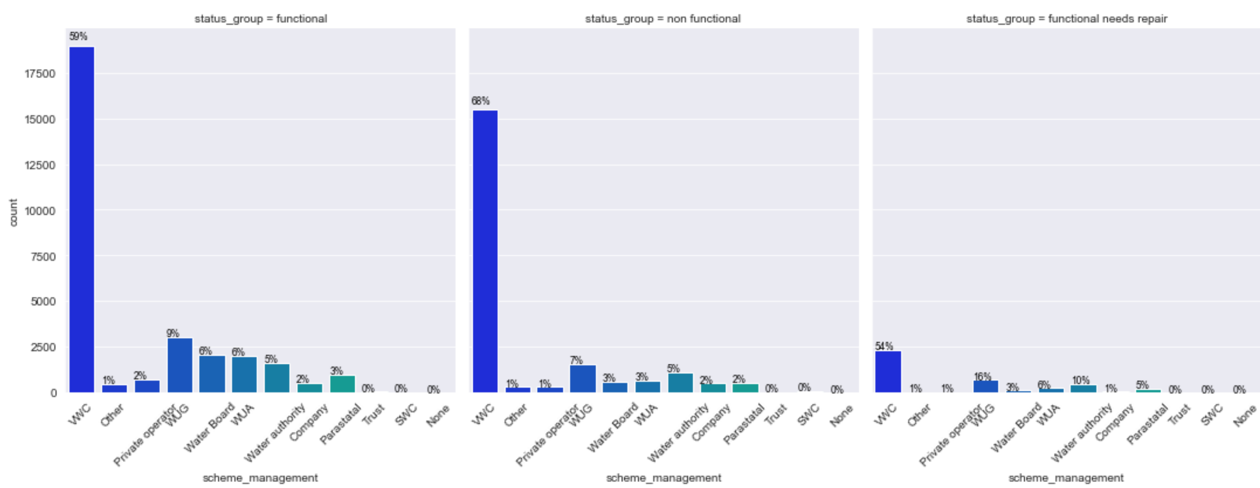






In [49]:

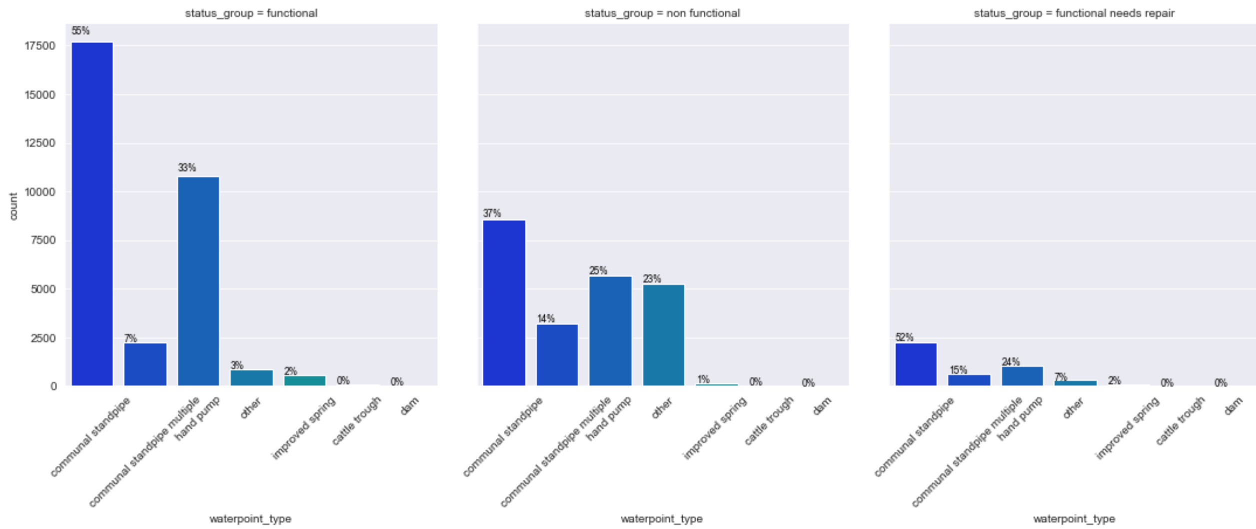
```
for col in who:
    count_plot_by_group(col)
```

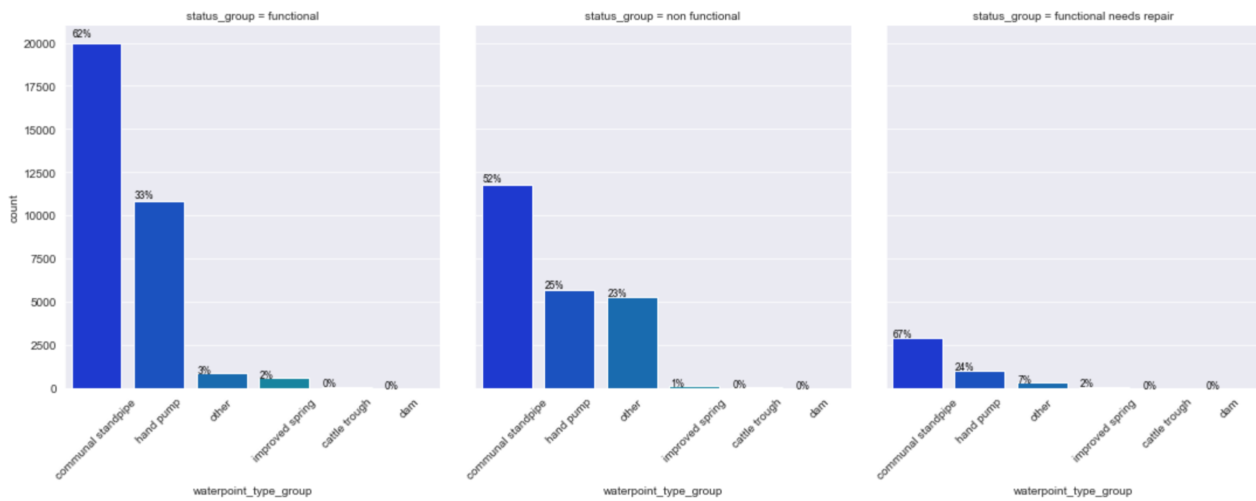


No relationship oberserved for `scheme_management`. Distributions seem to be roughly similar among status groups. Given this, and number of missing values, concluded this feature can be removed.

In [50]:

```
for col in waterpoint_type:
    count_plot_by_group(col)
```





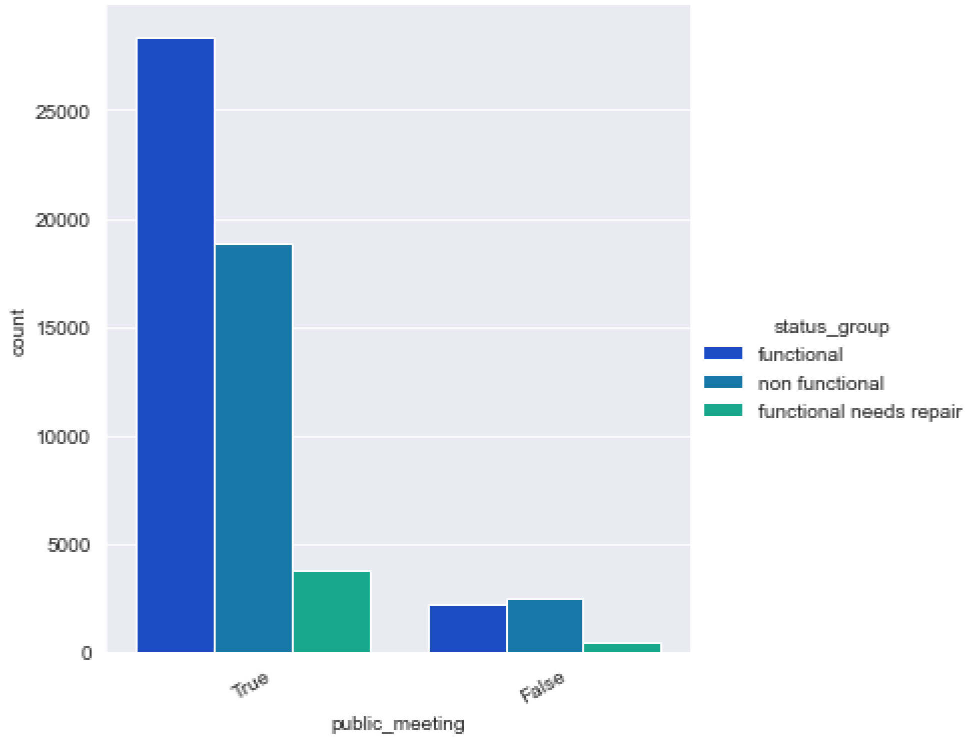
Some differences between status groups for the above and more significant differences observed for `waterpoint_type_group` , notably under both 'communical standpipe' and 'other' categories.

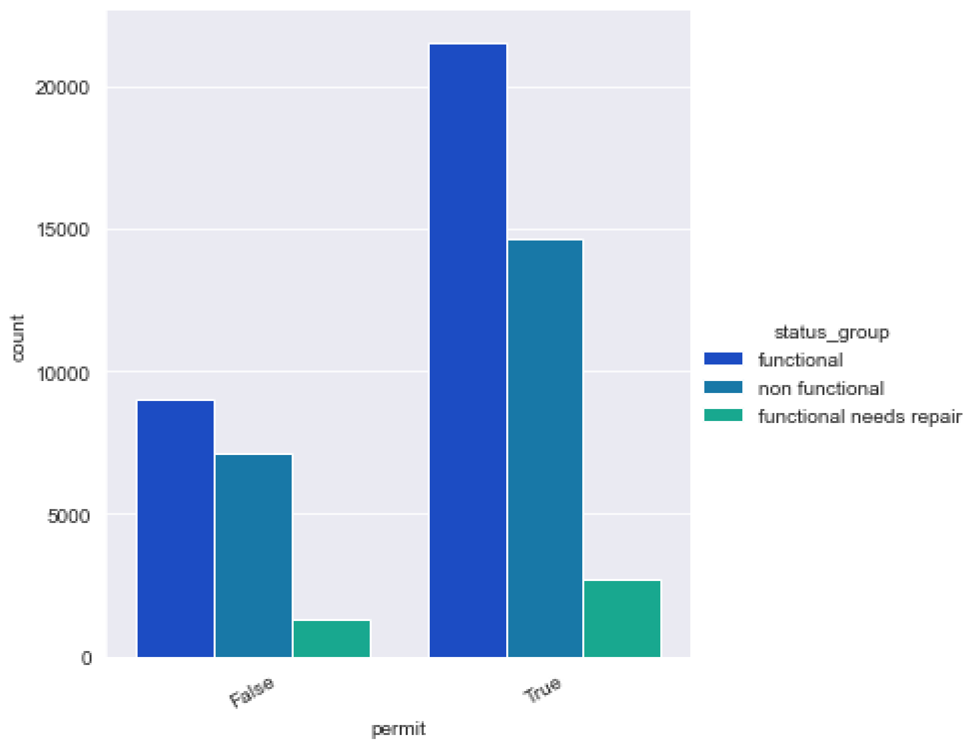
In summary, the following columns were dropped based on this EDA: `management_group` , `region_code` , `district_code` , `scheme_management` , `extraction_type_class` , `extraction_type_group` , `payment` , `quantity` , `source_class` , `source` , `quality_group` , `waterpoint_type_group`

Boolean Variables

- `permit`
- `public_meeting`

In [40]:
```
for col in others:
    ax = sns.catplot(x=col, kind='count', hue='status_group', data=df, height=5, palett
    ax.set_xticklabels(rotation = 30)
    plt.show()
```





There does appear to be a difference in distribution for `public_meeting` - namely that when `public_meeting` is false, most waterpoints are 'non functional'. Some differences within `permit` observed. When false, there appear to be a more equal number of non functional and functional waterpoints.

Others

- `funder`
- `installer`

In [4]:
```
print(df.installer.isna().sum())
print(df.funder.isna().sum())
print(len(df[df.installer == '0']))
print(len(df[df.funder == '0']))

df_2 = df.copy()
df_2 = df_2.fillna(value={'funder':'unknown', 'installer':'unknown'})
df_2.funder = df_2.funder.apply(lambda x: 'unknown' if x == '0' else x)
df_2.installer = df_2.installer.apply(lambda x: 'unknown' if x == '0' else x)
df_2.installer.isna().sum()
```

```
3655
3635
777
777
```

Out[4]: 0

In [5]:
```
# only include installer with 1000 or more waterpoints
installer_count = df_2.groupby('installer').id.count().reset_index().sort_values(by='id
installer_count = installer_count[installer_count['count'] >= 1000]
installer_count
```

Out[5]:

	installer	count
389	DWE	17402
2130	unknown	4433
570	Government	1825
1473	RWE	1206
296	Commu	1060
336	DANIDA	1050

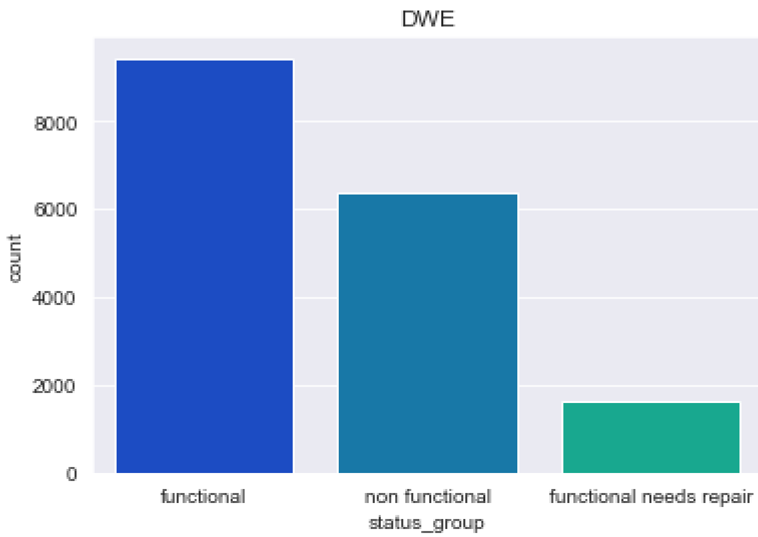
In [6]:
```
# only include funders with 1000 or more waterpoints
funder_count = df_2.groupby('funder').id.count().reset_index().sort_values(by='id', asc
funder_count = funder_count[funder_count['count'] >= 1000]
funder_count
```

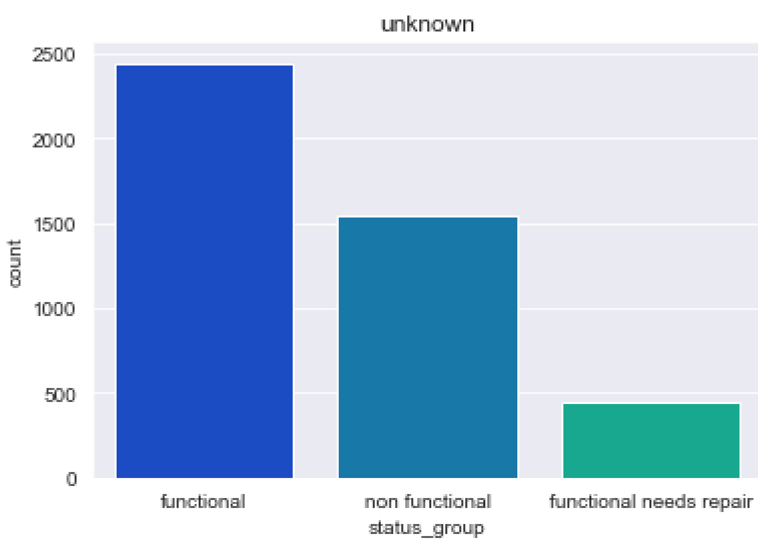
Out[6]:

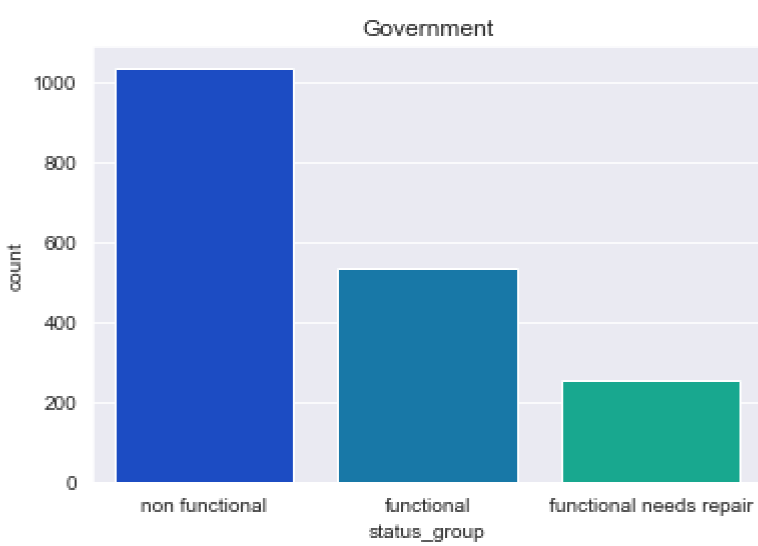
	funder	count
455	Government Of Tanzania	9084
1896	unknown	4412
260	Danida	3114
512	Hesawa	2202
1415	Rwssp	1374
1864	World Bank	1349
726	Kkkt	1287
1866	World Vision	1246
1740	Unicef	1057

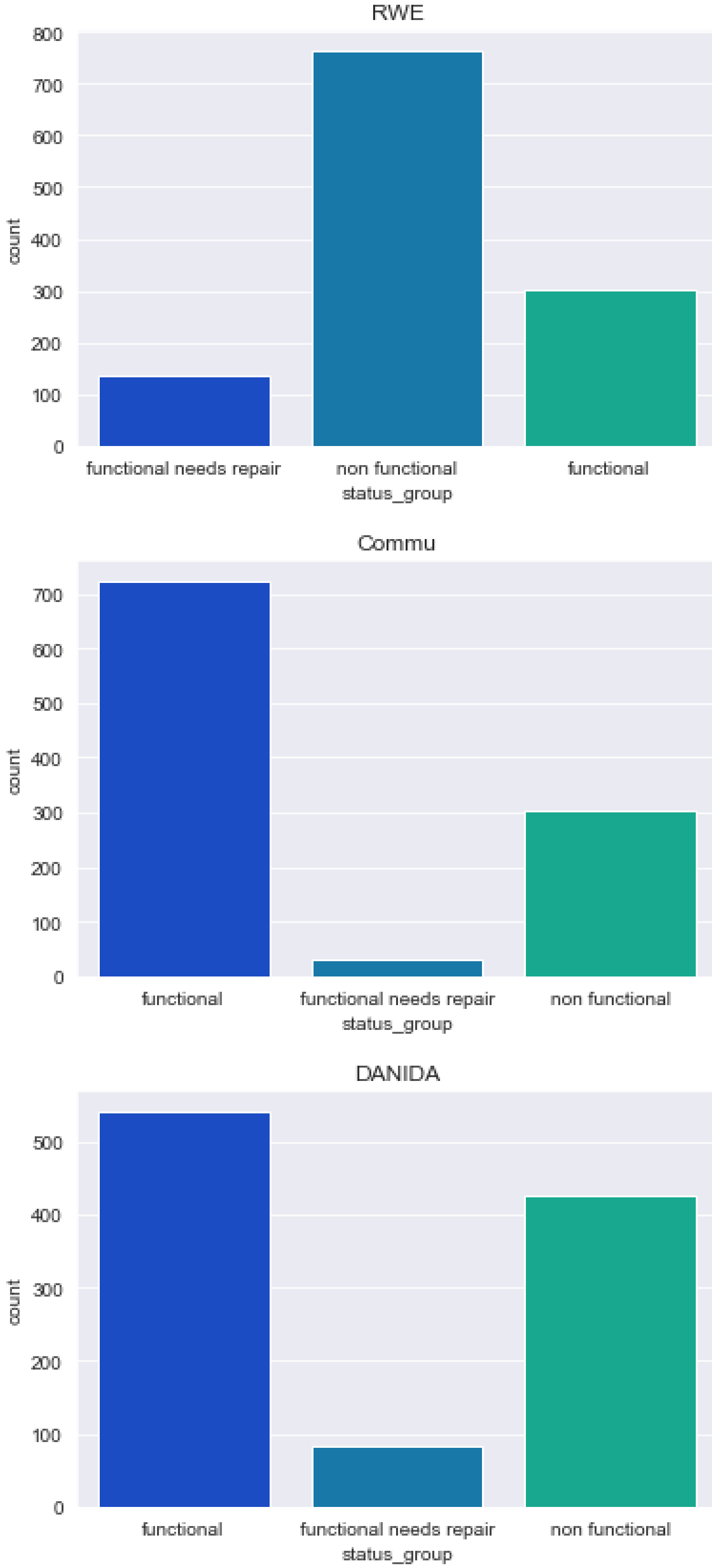
In [8]:

```
# fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(16,10))
sns.set_style('darkgrid')
for installer in installer_count.installer:
    sns.countplot(df_2[df_2.installer == installer].status_group, palette='winter')
    plt.title(installer)
    plt.show()
```







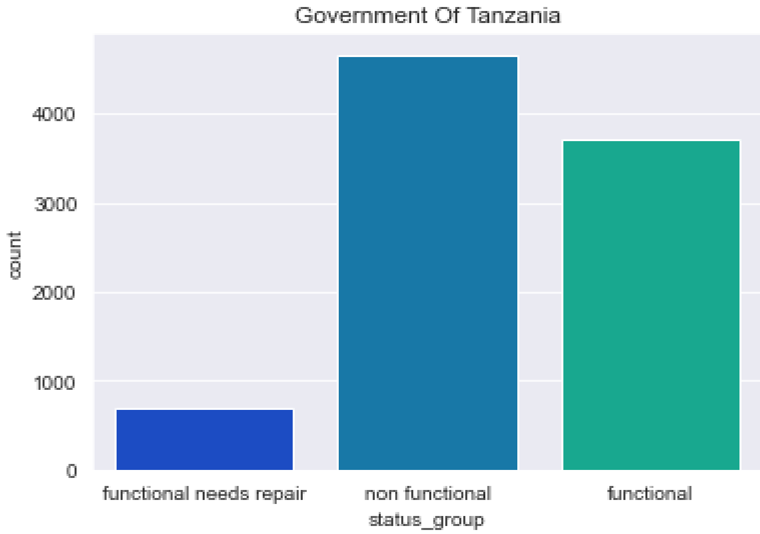


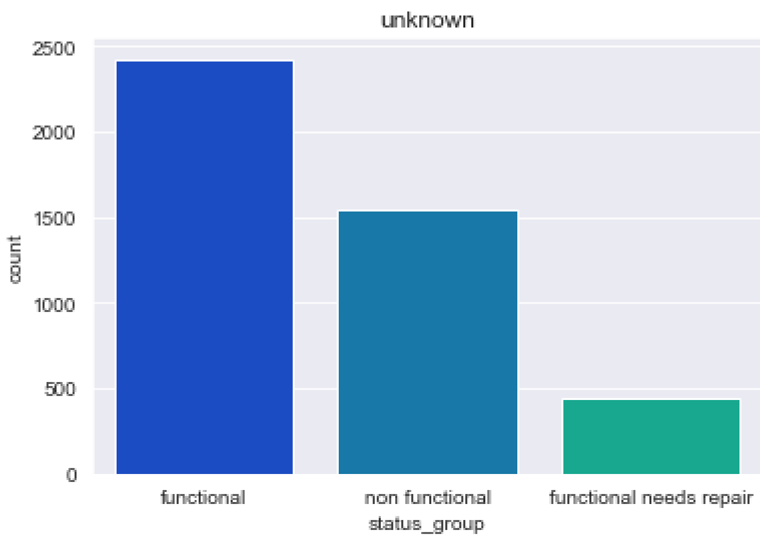
In [10]:
```
df[df.installer=='RWE'].status_group.value_counts(normalize=True)
```

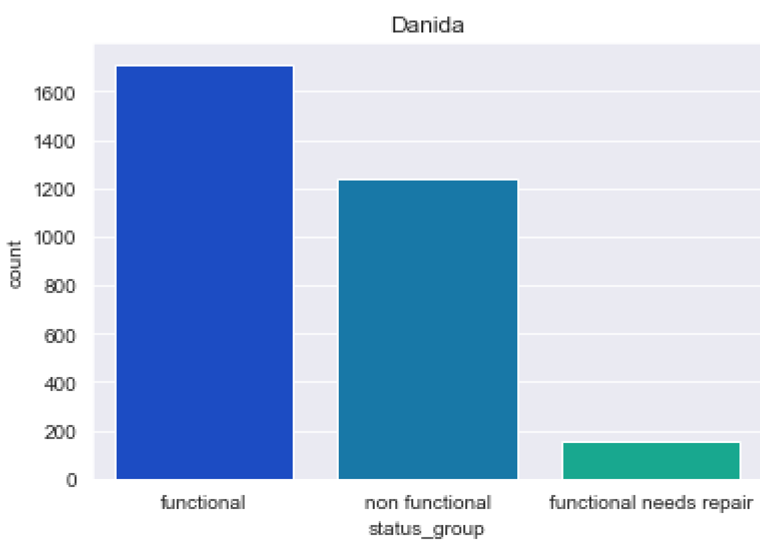
Out[10]:
```
non functional          0.634328
functional              0.252073
functional needs repair    0.113599
Name: status_group, dtype: float64
```

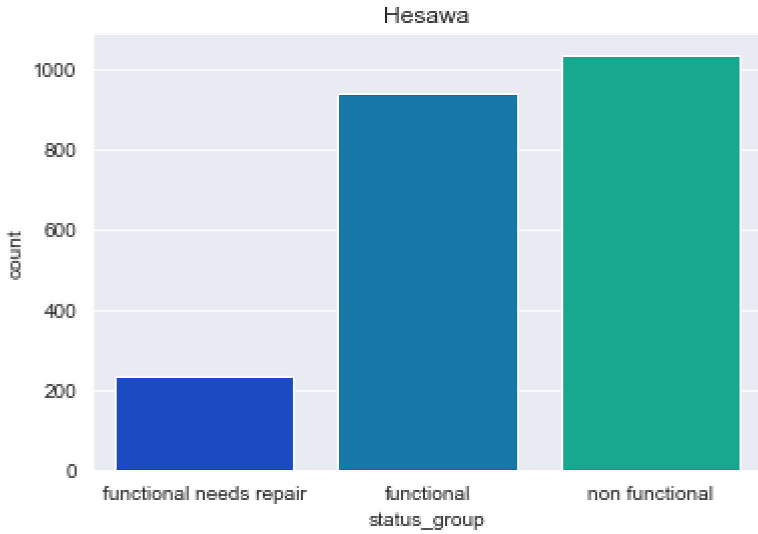
In [9]:
```
# fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(16,10))
```

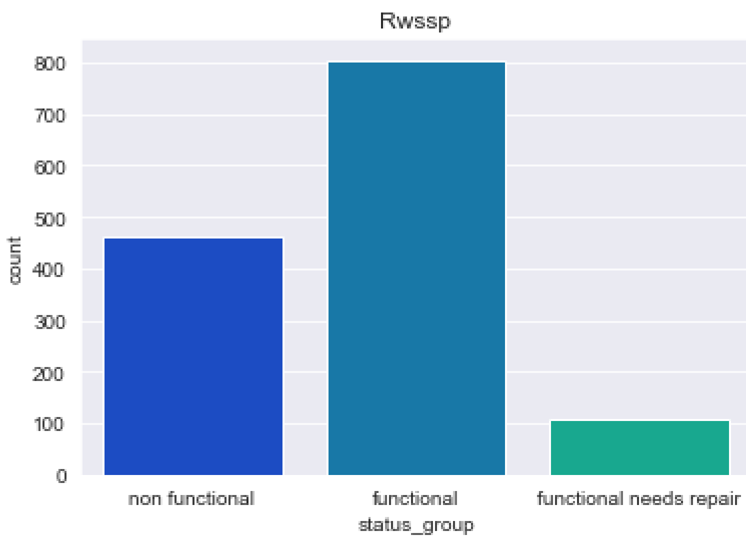
```
for funder in funder_count.funder:
    sns.countplot(df_2[df_2.funder == funder].status_group, palette = 'winter')
    plt.title(funder)
    plt.show()
```

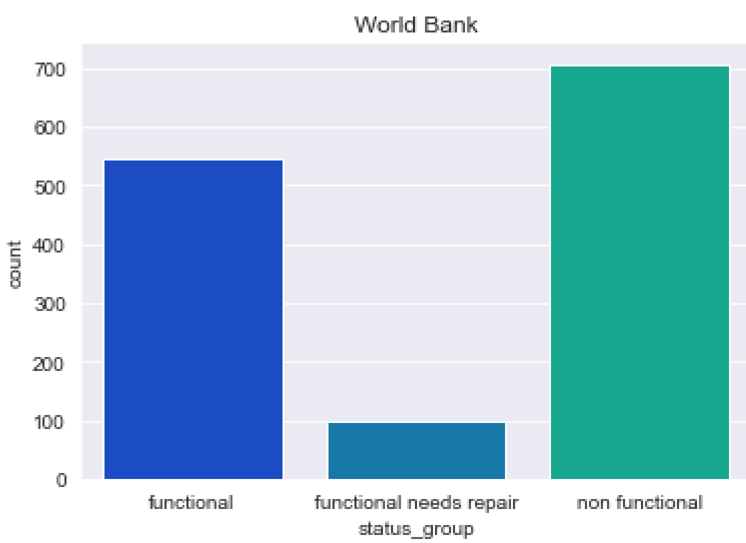


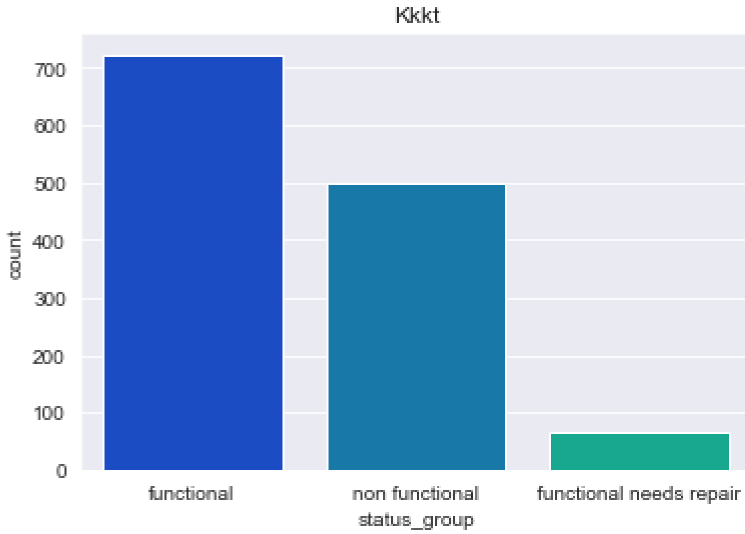


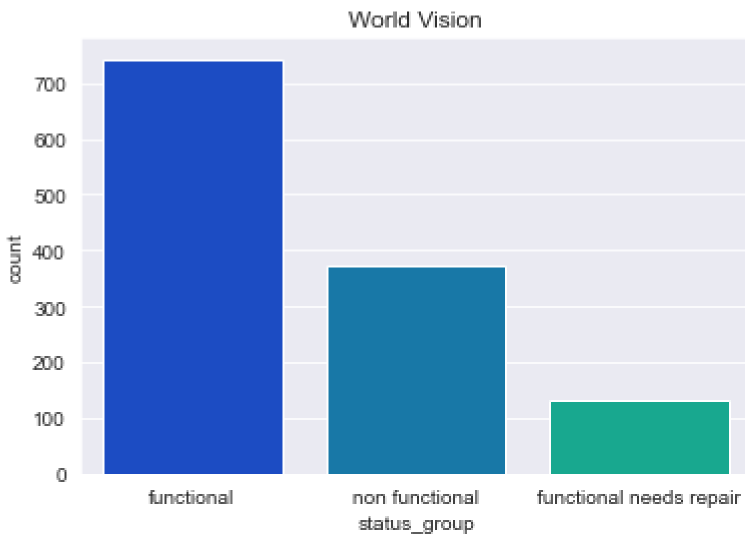


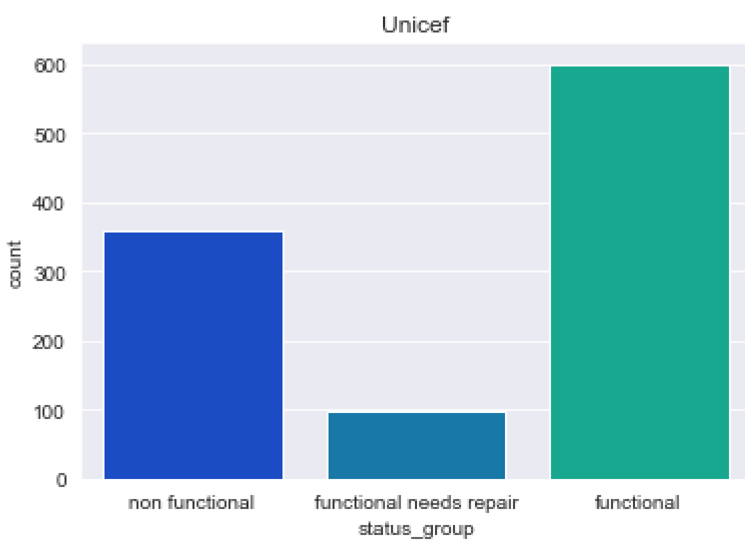

Hesawa
count
1000
800
600
400
200
0
functional needs repair
functional
non functional
status_group


Rwssp
count
800
700
600
500
400
300
200
100
0
non functional
functional
functional needs repair
status_group


World Bank
count
700
600
500
400
300
200
100
0
functional
functional needs repair
non functional
status_group







Findings

- When looking at `installer` **RWE**, **Commu** and **DANIDA** all appear to have a different distribution for `status_group` than the overall dataset.
- For `funder`, **Government of Tanzania**, **Hesawa**, **Rwssp**, **World Bank** and **Unicef** appear to have a different distribution for `status_group` than the overall dataset.

2. Data Preparation / Preprocessing

During this step, using insights unlocked from EDA, I cleaned and preprocessed the data to get it ready for use in models. This included:

- replacing null values
- feature engineering
- dropping columns
- splitting the data for model training and testing.

Missing Values and Outliers

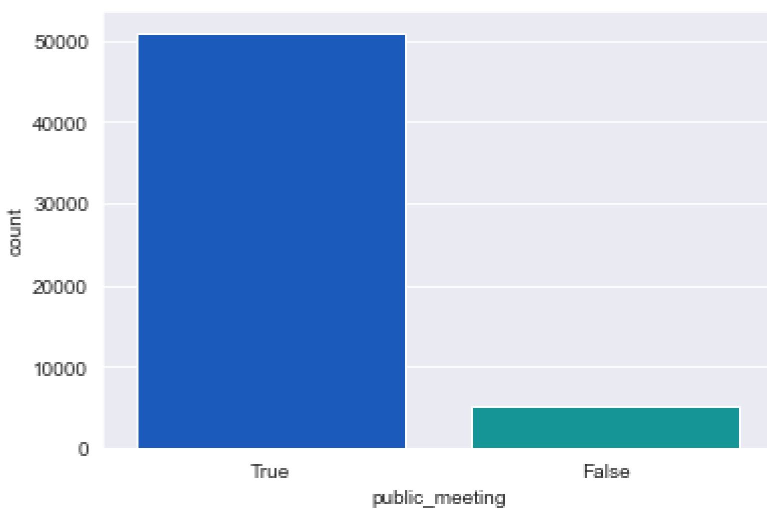
In this section, I explore adding in missing values for two boolean variables `permit` and `public_meeting` as well as removing outliers from the `latitude / longitude` columns.

Variables: `permit` and `public_meeting`

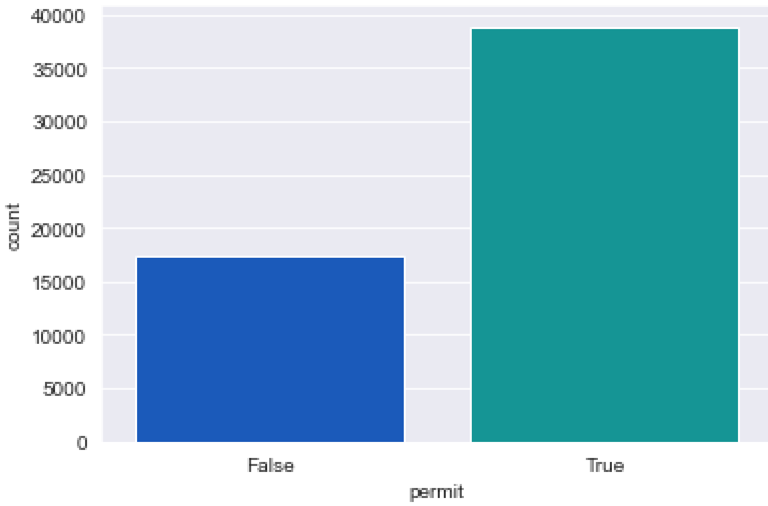
In [167…
```
# view the number of missing values for each column
df[['permit','public_meeting']].isnull().sum()
```

Out[167…
```
permit            3056
public_meeting    3334
dtype: int64
```

In [168…
```
# visualize the proportion of True and False for each variable
sns.set_style('darkgrid')

sns.countplot(df.public_meeting,  palette='winter')
plt.show()
print(df.public_meeting.value_counts(normalize=True))

sns.countplot(df.permit,  palette='winter')
plt.show()
print(df.permit.value_counts(normalize=True))
```




```
True     0.909838
False    0.090162
Name: public_meeting, dtype: float64
```




```
True     0.68955
False    0.31045
Name: permit, dtype: float64
```

In [154…

```
# replace the null values for permit
isnull = df.permit.isnull()
sample = df.permit.dropna().sample(isnull.sum(), replace=True, random_state=123).values
df.loc[isnull,'permit'] = sample

# replace the null values for public_meeting
isnull = df.public_meeting.isnull()
sample = df.public_meeting.dropna().sample(isnull.sum(), replace=True, random_state=123
df.loc[isnull,'public_meeting'] = sample

# check to see if there are any null values remaining
df[['permit','public_meeting']].isnull().sum()
```

In [171…

```
# visualize the results again
print(df.public_meeting.value_counts(normalize=True))
print(df.permit.value_counts(normalize=True))
```

```
True     0.910455
False    0.089545
Name: public_meeting, dtype: float64
True     0.689242
False    0.310758
Name: permit, dtype: float64
```

Variables: `latitude` and `longitude`

From the map visualization above, it was clear that I needed to move or remove the waterpoints which were not located in Tanzania. Because there were a significant amount of waterpoints which were incorrectly placed (~1800) I decided not to drop those records, but to instead place them at the median latitude and longitude for those groups.

In [ ]:

```
for status in list(df.status_group.unique()):
    lat = df[(df.status_group == status) & (df.longitude != 0)].latitude.median()
    long = df[(df.status_group == status) & (df.longitude != 0)].longitude.median()
    df['latitude'] = np.where((df.status_group == status) & (df.longitude==0), lat, df.
    df['longitude'] = np.where((df.status_group == status) & (df.longitude==0), long, d

plot_lat_long(df)
```

Feature Engineering

In this section, I explore the variables to prepare for feature engineering. Methods to add new columns are done in the 'define methods' section below.

Variable: `construction_year`

`construction_year` included a high number of missing values, labeled as '0' in the data. There seemed to be average differences in when a waterpoint was constructed based on status group. `construction_year` was broken into 4 categories:

- unknown: `construction_year` = 0
- old: `construction_year` > 0 <= 1994
- mid: `construction_year` > 1994 < 2003
- new: `construction_year` >= 2003

In [209…

```
df[df.construction_year > 0].groupby('status_group').construction_year.median().reset_i
```

Out[209…

	status_group	construction_year
0	functional	2003
1	functional needs repair	1998
2	non functional	1994

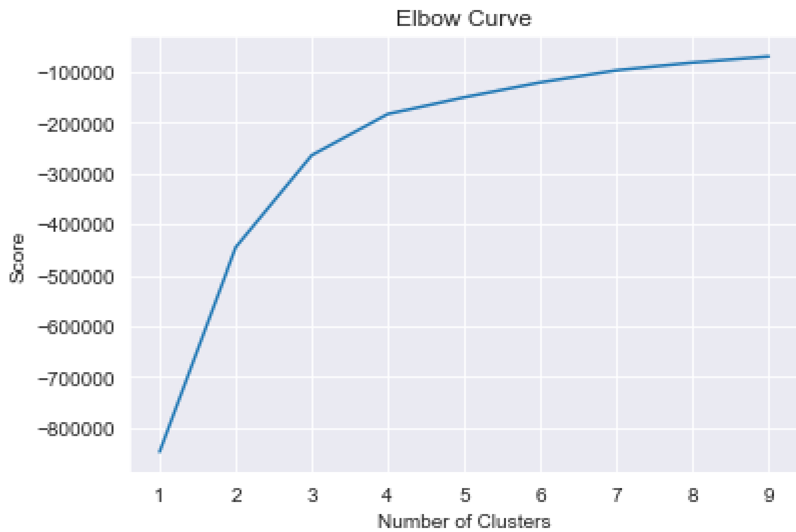
Variables: `latitude and longitude`

Now that the outliers were moved - rather than use the raw latitude and longitude, I decided to use KMeans clustering to group the waterpoints into different areas. Because we don't know what those 'clusters' are beforehand, an unsupervised learning technique was required herer.

I used the elbow method to first validate the number of clusters. For each cluster, SSE was calculated. As the number of clusters increase, error decreases but improvements will decline at a certain optimal point.

In [159…

```
# map the lat and long to x and y coordinates
K_clusters = range(1,10)
kmeans = [KMeans(n_clusters=i) for i in K_clusters]
X = df[['latitude','longitude']]
score = [kmeans[i].fit(X).score(X) for i in range(len(kmeans))]

# Visualize
plt.plot(K_clusters, score)
plt.xlabel('Number of Clusters')
plt.ylabel('Score')
plt.title('Elbow Curve')
plt.show()
```



The score levels off after 3.5/4, indicating that there will be minimal benefit from going above 4 clusters.

In [160…

```
kmeans = KMeans(n_clusters = 4, init ='k-means++') # use 4 clusters from above
# fit to calculate clustering
kmeans.fit(df[['latitude','longitude']])
centers = kmeans.cluster_centers_ # coord of cluster centers for plotting
centers
```

Out[160…

	id	amount_tsh	date_recorded	funder	gps_height	installer	longitude	latitude	wpt_name
0	69572	6000.0	2011-03-14	Roman	1390	Roman	34.938093	-9.856322	none
1	8776	0.0	2013-03-06	Grumeti	1399	GRUMETI	34.698766	-2.147466	Zahanati
2	34310	25.0	2013-02-25	Lottery Club	686	World vision	37.460664	-3.821329	Kwa Mahundi
3	67743	0.0	2013-01-28	Unicef	263	UNICEF	38.486161	-11.155298	Zahanati Ya Nanyumbu
4	19728	0.0	2011-07-13	Action In A	0	Artisan	31.130847	-1.825359	Shuleni

5 rows × 42 columns

Variables: `funder` and `installer`

In [435…

```
# use values from EDA above, installers and funders which have a relationship with stat
# create new boollean columns: true if installers and funders from these lists
installers = ['RWE','Commu','DANIDA']
funders = ['Government Of Tanzania', 'Hesawa', 'Rwssp', 'World Bank', 'Unicef']

df['installer_bool'] = df.installer.apply(lambda x: True if x in installers else False)
df['funder_bool'] = df.funder.apply(lambda x: True if x in funders else False)
```

Putting it all together

Data prepared for model training and fitting using the techniques and analysis from Step 2. To summarize again here briefly:

Dropped columns

26 columns were dropped. Some were irrelevant to `status_group` such as `id`, and others had too many unique values to be encoded (ie: `wpt_name`).

Engineered Features

Three features were engineered:

- construction_year_label
- cluster_label
- installer_bool
- funder_bool

After this, the data is split into training and test sets, and the categorical variables are one hot encoded so they are ready for model training and fitting.

Define methods

- Take the EDA and feature engineering work from above and encapsulate in methods for reproducability, and organziation

In [21]:

```
"""
input: values and labels
output: 2 dataframes, X and y
outliers removed
"""
def prep_data(values, labels):
    df = values.merge(labels, on='id') # merge the data and labels
    # print('Original columns: {}'.format(df.columns))

    # latitude and longitude - remove outliers (waterpoints located at 0 longitude in t
    for status in list(df.status_group.unique()):
        lat_median = df[df.longitude != 0].latitude.median()
        long_median = df[df.longitude != 0].longitude.median()
        df['latitude'] = np.where((df.longitude==0), lat_median, df.latitude)
        df['longitude'] = np.where((df.longitude==0), long_median, df.longitude)

    # convert cat columns into objects
    for col in df:
        if df[col].dtype == object:
            df[col] = df[col].astype('category')

    # fill in missing values
    # replace the null values for permit
    isnull = df.permit.isnull()
    sample = df.permit.dropna().sample(isnull.sum(), replace=True, random_state=123).va
    df.loc[isnull,'permit'] = sample

    # replace the null values for public_meeting
```

```
    isnull = df.public_meeting.isnull()
    sample = df.public_meeting.dropna().sample(isnull.sum(), replace=True, random_state
    df.loc[isnull,'public_meeting'] = sample

    # separate into X, y
    X = df.drop('status_group', axis=1)
    y = df.status_group

    return X,y

"""
input: X with non numeric cols converted to category
output: dataframe with columns dropped ready for splitting
"""
def engineer_features(df):

    # construction_year
    def construction_year_code(x):
        if x == 0:
            return 'unknown'
        elif x <= 1994:
            return 'old'
        elif x < 2003:
            return 'mid'
        else:
            return 'new'
    df['construction_year_label'] = df.construction_year.apply(lambda x: construction_y

    # latitude / longitude
    # using kmeans create 4 clusters, grouping the waterpoints together
    # cluster column
    kmeans = KMeans(n_clusters = 4, init ='k-means++', random_state=123) # use 4 cluste
    # fit to calculate clustering
    kmeans.fit(df[['latitude','longitude']])
    # create new column with cluster labels
    df['cluster_label'] = kmeans.fit_predict(df[['latitude','longitude']])

    # installer and funder
    installers = ['RWE','Commu','DANIDA']
    funders = ['Government Of Tanzania', 'Hesawa', 'Rwssp', 'World Bank', 'Unicef']

    df['installer_bool'] = df.installer.apply(lambda x: True if x in installers else Fa
    df['funder_bool'] = df.funder.apply(lambda x: True if x in funders else False)

    return df

"""
after prepping, label encode the target data into numbers
one hot encode categorical columns
split the data into training and test sets
return split data
"""
def encode_split_data(X,y,numeric_cols=[]):
    # assign each status group a number
    le = LabelEncoder()
    y = le.fit_transform(y)

    # split before applying any preprocessing
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_st
```

```python
    # if numeric columns exist, separate columns
    if len(numeric_cols) > 0:
        numeric_cols = numeric_cols
        cat_cols = X.drop(numeric_cols,axis=1).columns
    else:
        cat_cols = X.columns

    # one hot encode the cat columns
    ohe = OneHotEncoder(handle_unknown='ignore', sparse=False) #drop=first

    X_train_ohe = pd.DataFrame(ohe.fit_transform(X_train[cat_cols]), columns=ohe.get_fe
    X_test_ohe = pd.DataFrame(ohe.transform(X_test[cat_cols]), columns=ohe.get_feature_
    X_train_ohe.index= X_train.index
    X_test_ohe.index= X_test.index

    # add continuous data to categorical data if numeric columns exist
    if len(numeric_cols) > 0:
        X_train = pd.concat([X_train[numeric_cols], X_train_ohe], axis=1)
        X_test = pd.concat([X_test[numeric_cols], X_test_ohe], axis=1)
    else:
        X_train = X_train_ohe
        X_test = X_test_ohe

    # create unsplit X and y for cross_val_score
    X_ohe = pd.DataFrame(ohe.fit_transform(X[cat_cols]), columns=ohe.get_feature_names_
    X_all = pd.concat([X[numeric_cols], X_ohe], axis=1)
    y_all = y

    print(f'Number of columns after encoding: {len(X_train.columns)}')

    return X_train, X_test, y_train, y_test, X_all, y_all
```

In [28]:

```python
# start from scratch
training_values = pd.read_csv('tanzania_training_values.csv')
training_labels = pd.read_csv('tanzania_training_labels.csv')

to_drop_numeric = ['id','date_recorded','construction_year','longitude','latitude','num

to_drop_cat = ['funder','installer','wpt_name','subvillage','lga','ward','scheme_name',
               'management_group','region_code','district_code','scheme_management',
               'extraction_type_class','extraction_type_group','payment','quantity',
               'source_class','source','quality_group','waterpoint_type_group'] #'permi

cols_to_drop = to_drop_numeric + to_drop_cat
print(f'{len(cols_to_drop)} columns were dropped.\n')

X,y = prep_data(training_values, training_labels)
X = engineer_features(X) # add new columns
X = X.drop(cols_to_drop, axis=1)

print(f'Columns to keep:{X.columns}')
numeric_cols =['gps_height','amount_tsh','population']

# encode and split data
X_train, X_test, y_train, y_test, X_all, y_all = encode_split_data(X,y,numeric_cols)

# keep track of final models for comparison
model_dict = {}
```

```
# for evaluating model fitting
kf = KFold(n_splits=5, random_state=42, shuffle=True)

# map the labels to numeric counterparts for use later
le = LabelEncoder().fit(y)
status_group_dict = {index:label for index,label in enumerate(le.classes_)}
status_group_dict
```

```
26 columns were dropped.

Columns to keep:Index(['amount_tsh', 'gps_height', 'basin', 'region', 'population',
       'public_meeting', 'permit', 'extraction_type', 'management',
       'payment_type', 'water_quality', 'quantity_group', 'source_type',
       'waterpoint_type', 'construction_year_label', 'cluster_label',
       'installer_bool', 'funder_bool'],
      dtype='object')
Number of columns after encoding: 115
```

Out[28]:
```
{0: 'functional', 1: 'functional needs repair', 2: 'non functional'}
```

3. Model Prototyping

During this step, I take the split data and actual fit and train various classifier models. I fit and train three types of ML models below:

- Logistic Regression
- Random Forests
- XGBoost

For each type of model, I first fit a baseline model, and then I tune hyperparameters to attempt to achieve optimal results.

A note on metrics

Here I define an 'in need' waterpoint as belonging to 'non functional' or 'functional needs repair' groups.

In terms of metrics, I am most concerned with overall model performance, and how well each model does on in-need groups.

From the perspective of the 'in need' waterpoints, I was looking to minimize **false negatives** (missing an 'in need' waterpoint), which would come at the expense of an increase in **false positives** (saying a waterpoint is 'in need' when it isn't). In other words, I was looking to maximize **recall** scores for the minority classes. I sought to optimize recall for these classes during tuning and through the use of oversampling, which comes at the expense of precision.

Define methods

Below are the methods used throughout the current step

In [22]:
```
"""
Evaluate results of a classification model.
```

```
Output:
Accuracy scores for train and test data
Number of false positives
Number of false negatives
classifcation report
plotted confusion matrix
"""
def display_results(model, X_train, X_test, y_train, y_test):
    labels=['functional','functional needs repair', 'non functional']
    y_hat_test = model.predict(X_test)
    y_hat_train = model.predict(X_train)

    matrix = confusion_matrix(y_test, y_hat_test)
    # from perspective of non functional and needs repair
    fp = matrix[0][1] + matrix[0][2] # predicted non functional or needs repair even th
    fn = matrix[1][0] + matrix[2][0] # predicted functional even though it should be no
    tp_f = matrix[0][0]
    tp_nr = matrix[1][1]
    tp_nf = matrix[2][2]

    print(f"\nTraining Accuracy: {accuracy_score(y_train, y_hat_train) :.2%}\n")
    print(f"Testing Accuracy: {accuracy_score(y_test, y_hat_test) :.2%}\n")
    print(f'False positives: {fp}\n')
    print(f'False negatives: {fn}\n' )
    print(f'Total true positives for minority classes: {tp_nr + tp_nf}\n')
    print(classification_report(y_test, y_hat_test, target_names=labels))
    plot_confusion_matrix(model, X_test, y_test, xticks_rotation=45, display_labels=lab
    plt.grid(False)

"""
After evaluating model, add the results to a dictionary so that it can later be compare
"""
def add_model_dict(model, name, y_true, y_pred, cv_score):
    params = model.get_params()
    labels=['functional','functional needs repair', 'non functional']
    report = classification_report(y_test, y_pred, target_names=labels, output_dict=Tru
    accuracy = report['accuracy']*100
    functional_precision = report['functional']['precision']
    functional_recall = report['functional']['recall']
    repair_precision = report['functional needs repair']['precision']
    repair_recall = report['functional needs repair']['recall']
    nf_precision = report['non functional']['precision']
    nf_recall = report['non functional']['recall']
    sum_recall = repair_recall + nf_recall

    matrix = confusion_matrix(y_test, y_pred)
    fp = matrix[0][1] + matrix[0][2] # predicted non functional or needs repair even th
    # optimize against
    fn = matrix[1][0] + matrix[2][0] # predicted functional even though it should be no
    tp_f = matrix[0][0]
    tp_nr = matrix[1][1]
    tp_nf = matrix[2][2]

    if name in model_dict.keys():
        print('model already found in dictionary.')
        return
    else:
        model_dict[name] = dict(model=model,
                                parms=params,
```

```
                                  overall_accuracy=accuracy,
                                  fn=fn,
                                  cv_score=cv_score,
                                  functional_precision=functional_precision,
                                  functional_recall=functional_recall,
                                  needs_repair_precision=repair_precision,
                                  needs_repair_recall=repair_recall,
                                  nf_precision=nf_precision,
                                  nf_recall=nf_recall,
                                  sum_recall=sum_recall)

        # return a dataframe with updated information
        print('model added to dictionary.')
        return

def reset_model_dict():
    model_dict = {}

"""
for a given model, return the time it takes to fit the model and make predictions
"""
def training_time(model):
    start = time.time()
    model.fit(X_train, y_train)
    stop = time.time()
    train_time = round((stop-start),2)
    return train_time

"""plot feature importances for RF and XGBoost models"""
def plot_feature_importances(model, num_features=10):
    feature_df = pd.DataFrame(list(zip(xgb_final['classifier'].feature_importances_, X_
    feature_df = feature_df.iloc[:num_features,:]
    n_features = len(feature_df)
    sns.set_style('darkgrid')
    plt.figure(figsize=(12,8))
    sns.barplot(x=feature_df['coef'], y=list(range(n_features)),orient='h', palette='wi
    plt.yticks(np.arange(n_features), feature_df['feature'])
    plt.xlabel('Feature importance')
    plt.ylabel('Feature')
    plt.show()

"""After a grid search or randomized search, look through a top list of models,
and further assess based on effectiveness in minimizing false negatives for in need wat
(higher recall scores for the minority classes)
Rerank models """
def get_best_clf(df, classifier):

    # set up lists for dataframe
    models=[]
    recall_scores =[]
    cv_scores=[]

    for index, row in df.iterrows():
        params = row['params']
        param_dict={}
        for k, v in params.items():
            new_k = k.split('classifier__')[1]
            param_dict[new_k] = v

        clf = classifier
```

```
#         print(param_dict)
        clf.set_params(**param_dict)
        pipe = imbpipeline(steps=[['smote',SMOTE(random_state=42)],
                                  ['classifier',clf]]).fit(X_train, y_train)

        y_hat_test = pipe.predict(X_test)
        testing_accuracy = accuracy_score(y_test, y_hat_test)

        # get the sum of the recall scores for the minority classes
        labels=['functional','functional needs repair', 'non functional']
        report = classification_report(y_test, y_hat_test, target_names=labels,output_d
        sum_recall = report['functional needs repair']['recall'] + report['non function

        # get the cross validation score using all data, not just train data
        cv_score = cross_val_score(pipe, X_all, y_all, cv=kf)
        cv_score = np.mean(cv_score)

        models.append(pipe)
        recall_scores.append(sum_recall)
        cv_scores.append(cv_score)

    new_df = pd.DataFrame(list(zip(models,recall_scores,cv_scores)), columns=['model','
    new_df = new_df.sort_values(by=['recall','cv_score'],ascending=False)
    top_model = new_df.sort_values(by=['recall','cv_score'],ascending=False).iloc[0].mo
    return new_df, top_model
```

Load models from files

Below are the saved models for loading

In [35]:
```
# Load fitted GridSearch objects from files

# LOGISTIC REGRESSION
with open("models/lr_baseline.pickle", 'rb') as file:
    lr_baseline = pickle.load(file)

with open("models/lr_baseline_cv_score.pickle", 'rb') as file:
    lr_baseline_cv_score = pickle.load(file)

with open("models/lr_grid_smote.pickle", 'rb') as file:
    lr_grid_smote = pickle.load(file)

with open("models/lr_grid_no_smote.pickle", 'rb') as file:
    lr_grid_no_smote = pickle.load(file)


# RANDOM FORESTS
with open("models/rf_baseline_model.pickle", 'rb') as file:
    rf_baseline = pickle.load(file)

with open("models/rf_baseline_cv_score.pickle", 'rb') as file:
    rf_baseline_cv_score = pickle.load(file)

with open("models/rf_random_grid.pickle",'rb') as file:
    rf_random_grid = pickle.load(file)

with open("models/rf_grid_smote.pickle", 'rb') as file:
    rf_grid_smote = pickle.load(file)
```

```
with open("models/best_rf.pickle", 'rb') as file:
    best_rf = pickle.load(file)


# XGBoost
with open("models/xgb_baseline.pickle",'rb') as file:
    xgb_baseline = pickle.load(file)

with open("models/xgb_baseline_cv_score.pickle",'rb') as file:
    xgb_baseline_cv_score = pickle.load(file)

# Load fitted object from files
with open("models/xgb_random_grid.pickle",'rb') as file:
    xgb_random_grid = pickle.load(file)

# load the fitted objects from files
with open("models/xgb_grid_smote.pickle",'rb') as file:
    xgb_grid_smote = pickle.load(file)

with open("models/best_xgb.pickle",'rb') as file:
    best_xgb = pickle.load(file)
```

Logistic Regression

Models:

1. Baseline Model
2. Hyperparameter Tuning with GridSearchCV
3. Hyperparameter Tuning with GridSearchCV and SMOTE

Model 1: Baseline Model

*Using **pickle**, I stored the fitted models in files so that they can be easily loaded when running the notebook. Model training can be lengthy, especially when conducting a randomized search or grid search.*

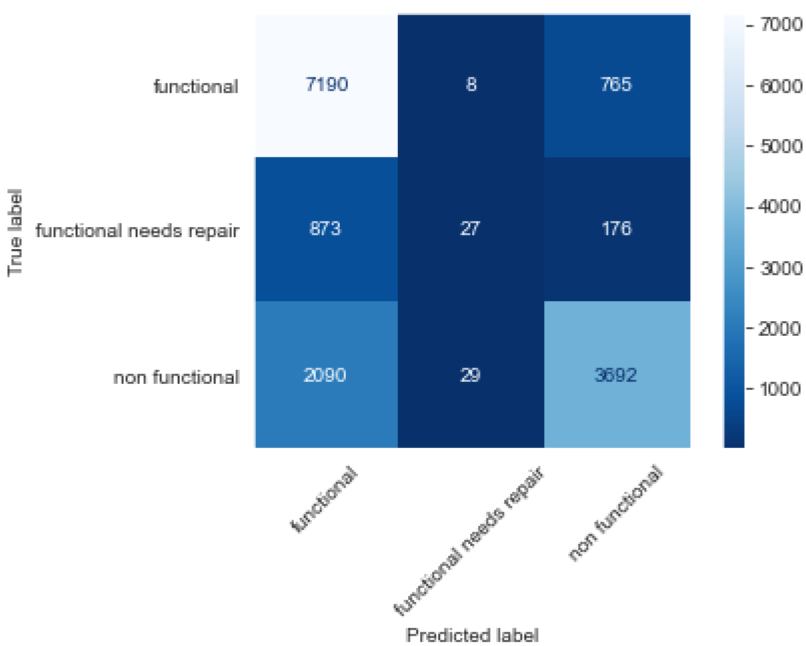
In [18]:
```
lr_baseline = LogisticRegression(random_state=42, max_iter=2000, multi_class="multinomi
lr_baseline.fit(X_train, y_train)

# Load from files
lr_baseline_cv_score = np.mean(cross_val_score(lr_baseline, X_all, y_all, cv=kf))
print(f'Mean Cross Validation Score for an Multinomial Logisitc Regression Model (No Tu

with open('models/lr_baseline.pickle', 'wb') as f:
    pickle.dump(lr_baseline, f)

with open('models/lr_baseline_cv_score.pickle', 'wb') as f:
    pickle.dump(lr_baseline_cv_score, f)
```

In [22]:
```
y_pred = lr_baseline.predict(X_test)
add_model_dict(lr_baseline, 'baseline_logreg', y_test, y_pred, lr_baseline_cv_score)
```

```
model added to dictionary.
```

In [164…
```
print('Baseline Logistic Regression Model')
display_results(lr_baseline, X_train, X_test, y_train, y_test)
```

```
Baseline Logistic Regression Model

Training Accuracy: 73.49%

Testing Accuracy: 73.46%

False positives: 773

False negatives: 2963

Total true positives for minority classes: 3719

                         precision    recall  f1-score   support

             functional       0.71      0.90      0.79      7963
functional needs repair       0.42      0.03      0.05      1076
         non functional       0.80      0.64      0.71      5811

               accuracy                           0.73     14850
              macro avg       0.64      0.52      0.52     14850
           weighted avg       0.72      0.73      0.71     14850
```



Model 2: Hyperparameter Tuning

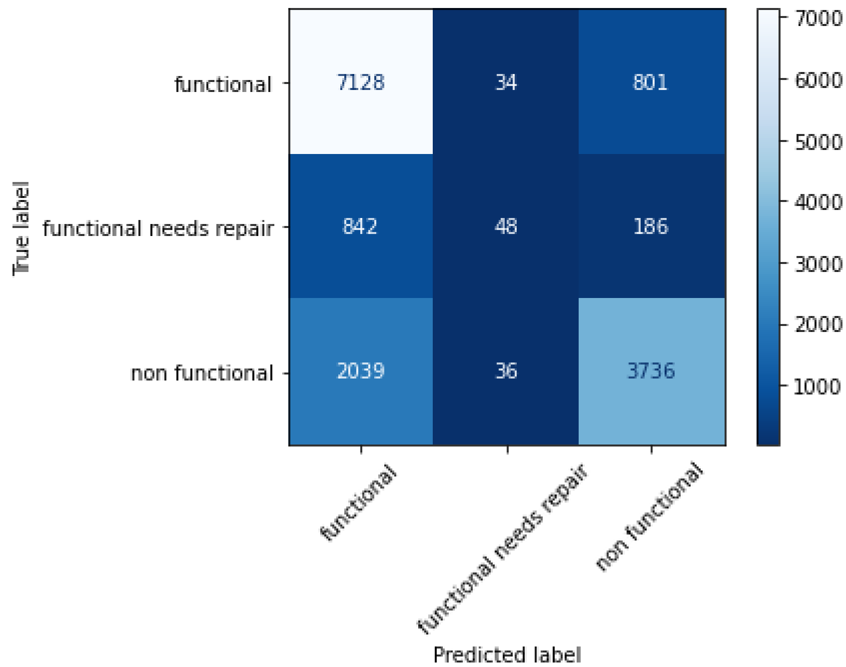
In [67]:
```
# set params for search
logreg_grid_params = {'classifier__max_iter':[100,1000,2000],
                      'classifier__multi_class':['multinomial'],
                      'classifier__C':[1,1e16],
                      'classifier__class_weight':[None,'balanced'],
                      'classifier__solver':['lbfgs','sag','saga','newton-cg']}

# set up pipeline with SMOTE. Do this in pipeline so that smote is applied to every fol
pipeline_smote = imbpipeline(steps=[['smote',SMOTE(random_state=42, n_jobs=-1)],
                                    ['scaler',StandardScaler()],
                                    ['classifier',LogisticRegression()]])

# set up pipeline without SMOTE for comparison. Do this in pipeline so that smote is ap
pipeline = imbpipeline(steps=[['scaler',StandardScaler()],
                              ['classifier',LogisticRegression()]])
```

```python
# set up the gridsearch object
lr_grid_smote = GridSearchCV(estimator=pipeline_smote,
                             param_grid=logreg_grid_params,
                             cv=kf, verbose=1)

lr_grid_no_smote = GridSearchCV(estimator=pipeline,
                                param_grid=logreg_grid_params,
                                cv=kf, verbose=1)

# fit the grid searches
lr_grid_smote = lr_grid_smote.fit(X_train, y_train)
lr_grid_no_smote = lr_grid_no_smote.fit(X_train, y_train)

# save the models for easy loading on notebook restart
with open('models/lr_grid_smote.pickle', 'wb') as f:
    pickle.dump(lr_grid_smote, f)

with open('models/lr_grid_no_smote.pickle', 'wb') as f:
    pickle.dump(lr_grid_no_smote, f)
```

```
Fitting 5 folds for each of 48 candidates, totalling 240 fits
Fitting 5 folds for each of 48 candidates, totalling 240 fits
```

In [22]:
```python
print('Logistic Regression Model 2 (no SMOTE)')
display_results(lr_grid_no_smote.best_estimator_, X_train, X_test, y_train, y_test)
```

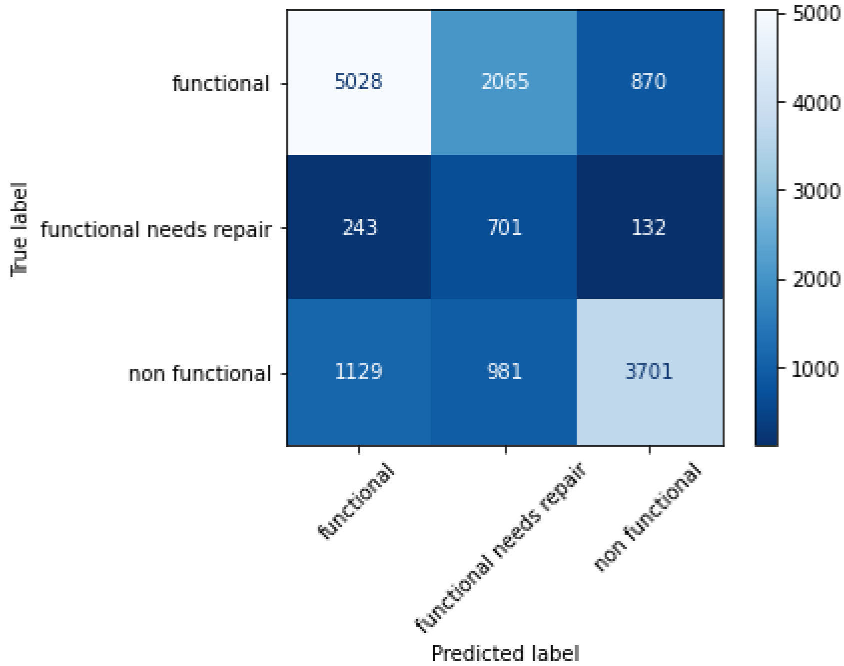
```
Logistic Regression Model 2 (no SMOTE)

Training Accuracy: 74.00%

Testing Accuracy: 73.48%

False positives: 835

False negatives: 2881

Total true positives for minority classes: 3784

                         precision    recall  f1-score   support

             functional       0.71      0.90      0.79      7963
functional needs repair       0.41      0.04      0.08      1076
         non functional       0.79      0.64      0.71      5811

               accuracy                           0.73     14850
              macro avg       0.64      0.53      0.53     14850
           weighted avg       0.72      0.73      0.71     14850
```



In [25]:

```
print('Logistic Regression Model 2 (with SMOTE)')
display_results(lr_grid_smote.best_estimator_, X_train, X_test, y_train, y_test)
```

```
Logistic Regression Model 2 (with SMOTE)

Training Accuracy: 63.24%

Testing Accuracy: 63.50%

False positives: 2935

False negatives: 1372

Total true positives for minority classes: 4402

                         precision    recall  f1-score   support

             functional       0.79      0.63      0.70      7963
functional needs repair       0.19      0.65      0.29      1076
         non functional       0.79      0.64      0.70      5811

               accuracy                           0.64     14850
              macro avg       0.59      0.64      0.56     14850
           weighted avg       0.74      0.64      0.67     14850
```



In [70]:
```
print('CV Score: {}'.format(lr_grid_smote.best_score_))
print('\nBest params: {}'.format(lr_grid_smote.best_params_))
```

```
CV Score: 0.6300785634118967

Best params: {'classifier__C': 1, 'classifier__class_weight': None, 'classifier__max_ite
r': 100, 'classifier__multi_class': 'multinomial', 'classifier__solver': 'lbfgs'}
```

Interpretation

The solvers were different for the baseline model compared with the top results from the grid searches ('lbfgs' for the baseline model versus 'newton-cg' and 'saga'). All models favored a lower C value indicating stronger regularization led to better outcomes here.

Using oversampling led to better recall results for the minority classes, most significantly for the 'functional needs repair' category which is least represented in the labels data. Recall for the 'functional needs repair' category increased from .05 to .61.

However, due to oversampling, overall model accuracy went down from 72.81% (Model 1) to 63.55% (Model 2). Precision suffered as well.

In this case, because we are optimizing for higher recall scores (against false negatives) in the minority classes, the second model was stronger.

Wrap up

In [26]:
```
# add the final model to the dictionary for comparison later
logreg_final = lr_grid_smote.best_estimator_
logreg_final_cv = lr_grid_smote.best_score_
y_pred = logreg_final.predict(X_test)
add_model_dict(logreg_final, 'logreg_final', y_test, y_pred, logreg_final_cv)
```

```
model added to dictionary.
```

Random Forests

Models:

1. Baseline Model
2. Hyperparameter Tuning with RandomizedSearchCV (with SMOTE)
3. Hyperparameter Tuning with GridSearchCV (with SMOTE)

Model 1: Baseline Model

In [148…

```
rf_baseline = RandomForestClassifier()
rf_baseline.fit(X_train, y_train)
rf_baseline_cv_score = np.mean(cross_val_score(rf_baseline, X_all, y_all, cv=kf))
print(f'Mean Cross Validation Score for a Random Forest Classifier (No Tuning): {rf_bas

with open('models/rf_baseline_model.pickle', 'wb') as f:
    pickle.dump(rf_baseline, f)

with open('models/rf_baseline_cv_score.pickle', 'wb') as f:
    pickle.dump(rf_baseline_cv_score, f)
```

```
Mean Cross Validation Score for a Random Forest Classifier (No Tuning):  78.32%
```

In [29]:

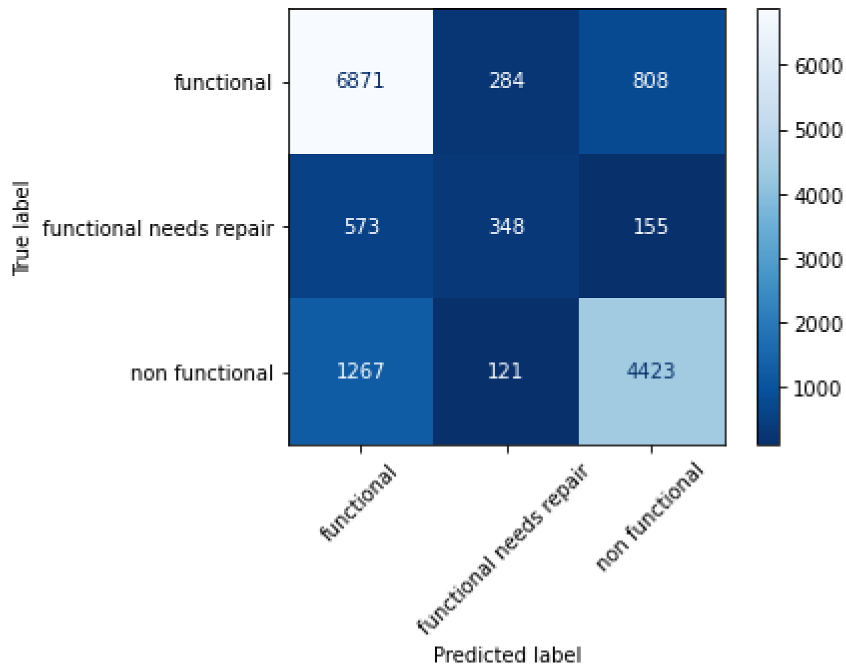
```
y_pred = rf_baseline.predict(X_test)
add_model_dict(rf_baseline, 'rf_baseline', y_test, y_pred, rf_baseline_cv_score)
```

```
model added to dictionary.
```

In [17]:

```
print('Baseline Random Forests Model')
display_results(rf_baseline, X_train, X_test, y_train, y_test)
```

```
Baseline Random Forests Model

Training Accuracy: 94.27%

Testing Accuracy: 78.40%

False positives: 1092

False negatives: 1840

Total true positives for minority classes: 4771

                         precision    recall  f1-score   support

             functional       0.79      0.86      0.82      7963
functional needs repair       0.46      0.32      0.38      1076
         non functional       0.82      0.76      0.79      5811

               accuracy                           0.78     14850
              macro avg       0.69      0.65      0.66     14850
           weighted avg       0.78      0.78      0.78     14850
```



Interpretation:

Testing accuracy (76.41%) is an improvement over the baseline logistic regression model (70.61%). Large difference in training and testing accuracy indicates the model may be overfitting.

Model 2: RandomizedSearchCV

Due to the fact that this is a large dataset that will incur longer fitting times, a randomized search was used to first refine hyperparameter ranges before doing a gridsearch.

In [20]:
```
# randomized grid search 1
# use SMOTE
start = time.time()

# define the hyperparameter ranges
estimators = np.arange(100,700,50)
max_depth = np.arange(10,110,10)
criterion = ['gini','entropy']
min_samples_leaf = np.arange(1,5,1)
min_samples_split = np.arange(1,10,1)
random_grid = dict(classifier__n_estimators=estimators,
                   classifier__max_depth = max_depth,
                   classifier__min_samples_leaf = min_samples_leaf,
                   classifier__min_samples_split = min_samples_split,
                   classifier__criterion=criterion)

pipeline_smote = imbpipeline(steps=[['smote',SMOTE(random_state=42)],
                              ['classifier',RandomForestClassifier()]])
# set up the object and fit
rf_random_grid = RandomizedSearchCV(estimator=pipeline_smote,
                                    param_distributions = random_grid,
                                    n_iter = 100, cv=kf, random_state=123)

rf_random_grid = rf_random_grid.fit(X_train, y_train)

stop = time.time()
```

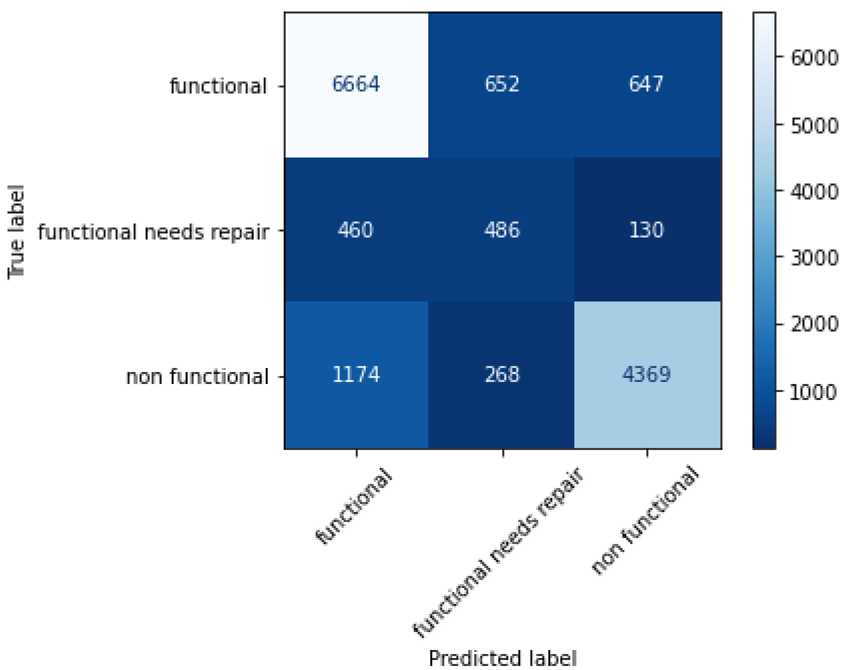
```python
print('time it took: {}'.format(round((stop-start),2)/3600))

# save the fitted object as a file for ease of access
with open('models/rf_random_grid.pickle','wb') as f:
    pickle.dump(rf_random_grid, f)
```

```
time it took: 5.314538888888889
```

In [19]:

```python
print('Random Forests Model 2 (RandomizedSearchCV)')
display_results(rf_random_grid.best_estimator_, X_train, X_test, y_train, y_test)
```

```
Random Forests Model 2 (RandomizedSearchCV)

Training Accuracy: 84.19%

Testing Accuracy: 77.57%

False positives: 1299

False negatives: 1634

Total true positives for minority classes: 4855

                         precision    recall  f1-score   support

             functional       0.80      0.84      0.82      7963
functional needs repair       0.35      0.45      0.39      1076
         non functional       0.85      0.75      0.80      5811

               accuracy                           0.78     14850
              macro avg       0.67      0.68      0.67     14850
           weighted avg       0.79      0.78      0.78     14850
```



In [34]:

```python
print('RF Randomized Search CV Score: {}'.format(rf_random_grid.best_score_))
print('RF Randomized Search Best params: {}'.format(rf_random_grid.best_params_))
```

```
RF Randomized Search CV Score: 0.7661728395061729
RF Randomized Search Best params: {'classifier__n_estimators': 500, 'classifier__min_sam
ples_split': 3, 'classifier__min_samples_leaf': 2, 'classifier__max_depth': 90, 'classif
ier__criterion': 'gini'}
```

In [9]:
```
rf_random_grid_results = pd.DataFrame(rf_random_grid.cv_results_)
rf_random_grid_results.sort_values(by='rank_test_score', ascending=True).head().iloc[:,
```

Out[9]:

	param_classifier__n_estimators	param_classifier__min_samples_split	param_classifier__min_samples_leaf
14	650	2	2
32	300	3	2
62	400	9	1
96	500	8	1
9	650	6	1

Notes/Interpretation:

- The top 5 models from the search had either 650 or 300 estimators. Variance among the other parameters.
- Conducting the randomized search incurred signficant time (4.8 hours).
- The difference between training and testing scores grew smaller compared to the baseline model produced from the search, indicating less overfitting.
- Improvements seen in testing accuracy in the randomized search model (76.17%) versus the baseline random forests model (74.70%).

Model 3: GridSearchCV

Using parameters from the randomized search, apply further tuning with a grid search.

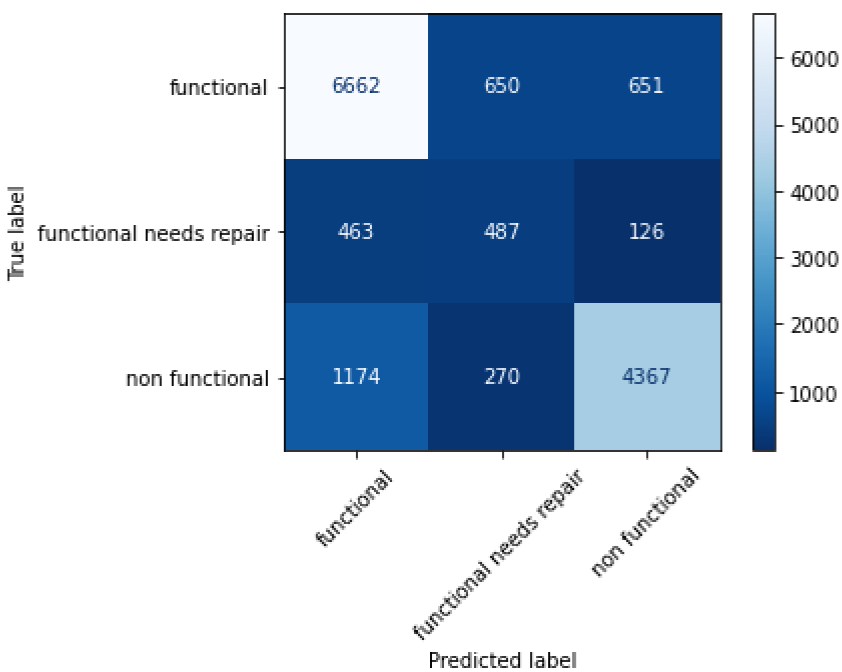
In [310…
```
start = time.time()
# set params for search
estimators = [275, 300, 325, 350]
max_depth = [60,70,80,90]
criterion = ['gini','entropy']
min_samples_leaf = [2]
min_samples_split = [2,3,4,5]

param_grid_final = dict(classifier__n_estimators=estimators,
                        classifier__max_depth = max_depth,
                        classifier__min_samples_leaf = min_samples_leaf,
                        classifier__min_samples_split = min_samples_split,
                        classifier__criterion=criterion)

# set up pipelines, 1 with smote and 1 without. Do this in pipeline so that smote is ap
pipeline_smote = imbpipeline(steps=[['smote',SMOTE(random_state=42)],
                              ['classifier',RandomForestClassifier()]])

# set up the gridsearch object
rf_grid_smote = GridSearchCV(estimator=pipeline_smote,
                             param_grid = param_grid_final,
                             cv=kf, verbose=1)
```

```
# fit the grid searches
rf_grid_smote = rf_grid_smote.fit(X_train, y_train)

stop = time.time()
print('time it took: {}'.format(round((stop-start),2)/3600))

# save the models for easy loading on notebook restart
with open('models/rf_grid_smote.pickle','wb') as f:
    pickle.dump(rf_grid_smote, f)
```

```
Fitting 5 folds for each of 128 candidates, totalling 640 fits
time it took: 5.894383333333333
```

In [32]:
```
print('Random Forests Model 3 (GridSearchCV/SMOTE)')
display_results(rf_grid_smote.best_estimator_.named_steps['classifier'], X_train, X_tes
```

```
Random Forests Model 3 (GridSearchCV/SMOTE)

Training Accuracy: 83.94%

Testing Accuracy: 77.55%

False positives: 1301

False negatives: 1637

Total true positives for minority classes: 4854

                         precision    recall  f1-score   support

             functional       0.80      0.84      0.82      7963
functional needs repair       0.35      0.45      0.39      1076
         non functional       0.85      0.75      0.80      5811

               accuracy                           0.78     14850
              macro avg       0.67      0.68      0.67     14850
           weighted avg       0.79      0.78      0.78     14850
```



In [22]:
```
print('\nCV Score: {}'.format(rf_grid_smote.best_score_))
print('\nBest params: {}'.format(rf_grid_smote.best_params_))
```

```
CV Score: 0.7698540965207631

Best params: {'classifier__criterion': 'gini', 'classifier__max_depth': 80, 'classifier_
_min_samples_leaf': 2, 'classifier__min_samples_split': 5, 'classifier__n_estimators': 5
00}
```

In [17]:

```
rf_grid_smote_df = pd.DataFrame(rf_grid_smote.cv_results_)
rf_grid_smote_df.sort_values(by='rank_test_score', ascending=True).iloc[:,4:]
best_rf  = get_best_clf(rf_grid_smote_df.head(10), RandomForestClassifier())

with open('models/best_rf.pickle','wb') as f:
    pickle.dump(best_rf, f)
```

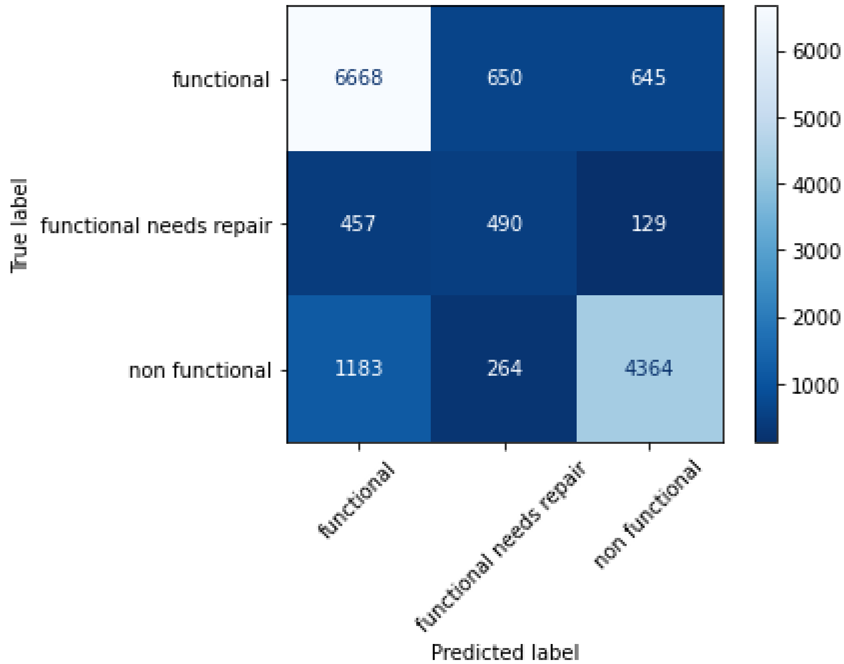
In [33]:

```
best_rf_df, best_rf_model = best_rf[0], best_rf[1]
```

In [32]:

```
print(f'Final Random Forests Model CV Score: {best_rf_df.iloc[0].cv_score}')
print('Final Random Forests Model')
display_results(best_rf_model, X_train, X_test, y_train, y_test)
```

```
Final Random Forests Model CV Score: 0.7737710437710438
Final Random Forests Model

Training Accuracy: 84.20%

Testing Accuracy: 77.59%

False positives: 1295

False negatives: 1640

Total true positives for minority classes: 4854

                         precision    recall  f1-score   support

             functional       0.80      0.84      0.82      7963
functional needs repair       0.35      0.46      0.40      1076
         non functional       0.85      0.75      0.80      5811

               accuracy                           0.78     14850
              macro avg       0.67      0.68      0.67     14850
           weighted avg       0.79      0.78      0.78     14850
```



Interpretation:

Model 3 (GridSearchCV) had a stronger cross validation score than the baseline model (76% vs 74%). Model 3 had stronger recall scores for the 'functional' and 'functional needs repair' classes indicating it made more overall correct classifications for those categories. Overall testing accuracy was also stronger than for baseline model.

Overall, when comparing models fitted with oversampled data, the Random Forests model appears to be a stronger fit than the logistic regression one (.75 vs .62 cross validation score).

In [34]:

```
# add the model to the dictionary for comparison later
rf_final = best_rf_model
rf_final_cv = best_rf_df.iloc[0].cv_score
y_pred = rf_final.predict(X_test)
add_model_dict(rf_final, 'rf_final', y_test, y_pred, rf_final_cv)
```

```
model added to dictionary.
```

XG Boost

Models:

1. Baseline Model
2. Hyperparameter Tuning with RandomizedSearchCV (with SMOTE)
3. Hyperparameter Tuning with GridSearchCV (with SMOTE)

Model 1: Baseline

In [147…

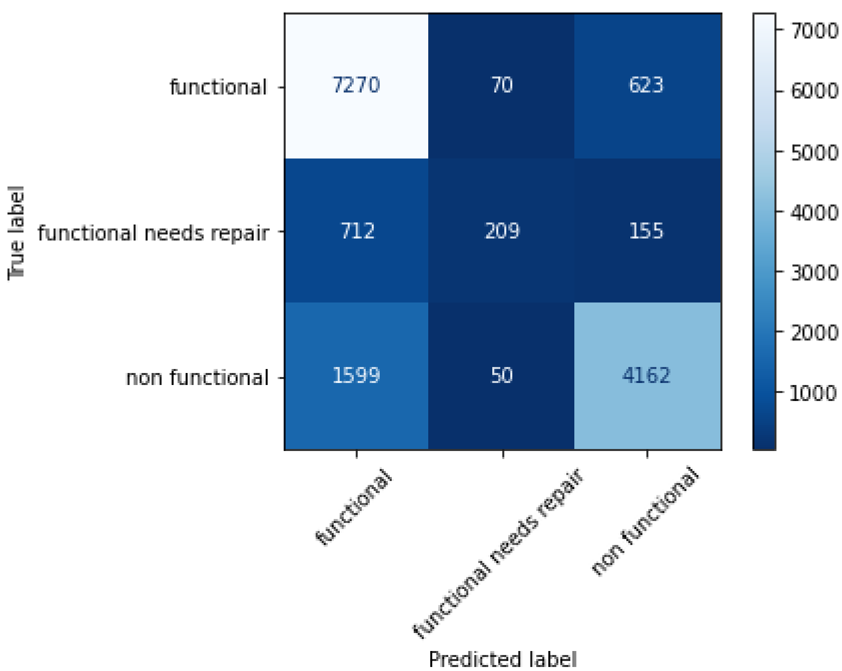
```
# Fit a baseline xgboost classifier model
xgb_baseline = XGBClassifier(eval_metric = 'merror')
xgb_baseline.fit(X_train, y_train)

# Get baseline cross validation score for XGBoost Classifier
xgb_baseline_cv_score = np.mean(cross_val_score(xgb_baseline, X_all, y_all, cv=kf))
```

```python
print(f'Mean Cross Validation Score for an XGBoost Classifier (No Tuning): {xgb_baselin

with open('models/xgb_baseline.pickle','wb') as f:
    pickle.dump(xgb_baseline, f)

with open('models/xgb_baseline_cv_score.pickle','wb') as f:
    pickle.dump(xgb_baseline_cv_score, f)
```

```
Mean Cross Validation Score for an XGBoost Classifier (No Tuning):  78.50%
```

In [27]:
```python
display_results(xgb_baseline, X_train, X_test, y_train, y_test)
```

```
Training Accuracy: 81.59%

Testing Accuracy: 78.39%

False positives: 693

False negatives: 2311

Total true positives for minority classes: 4371

                         precision    recall  f1-score   support

            functional       0.76      0.91      0.83      7963
functional needs repair       0.64      0.19      0.30      1076
        non functional       0.84      0.72      0.77      5811

              accuracy                           0.78     14850
             macro avg       0.75      0.61      0.63     14850
          weighted avg       0.78      0.78      0.77     14850
```



Interpretation: Baseline XGBoost classifier model performs better on overall accuracy than the logisitic regression and random forests baseline models. Minority class recall is still weak. Hyperparameter tuning and oversampling method performed as a next step to address this.

- XGBoost baseline cross validation score: **77.54%**
- Random Forests cross validation score: **75.63%**

- Logistic regression cross validation score: **72.81%**

In [36]:
```
# Add baseline model to dictionary
y_pred = xgb_baseline.predict(X_test)
add_model_dict(xgb_baseline, 'xgb_baseline', y_test, y_pred, xgb_baseline_cv_score)
```

```
model added to dictionary.
```

Model 2: RandomizedSearchCV

A randomized search was used to first refine hyperparameter ranges before doing a gridsearch.

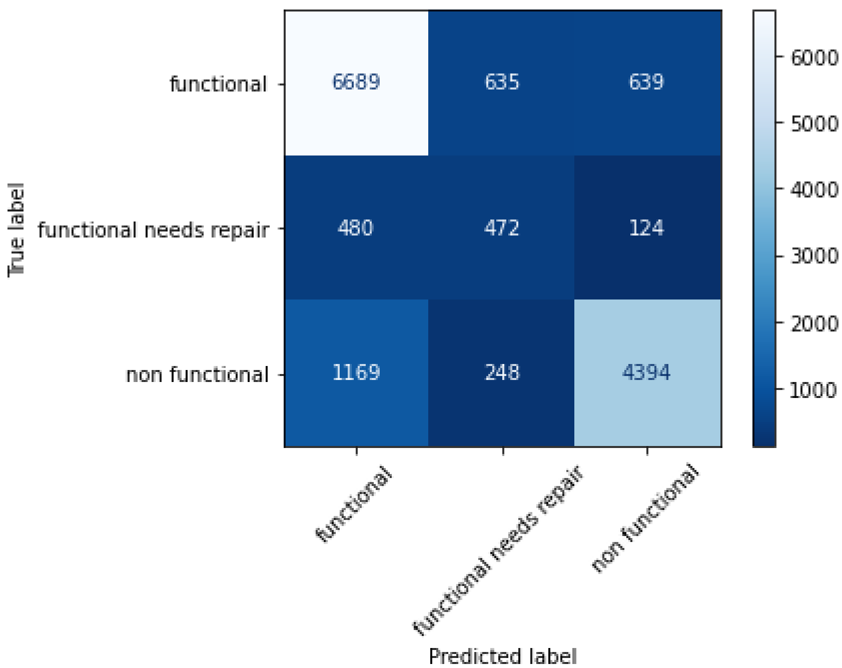
In [13]:
```
start = time.time()

#set params for search
random_grid = {'classifier__max_depth':np.arange(5,20,1),
               'classifier__min_child_weight':np.arange(1,9,1), # between 0 and 1
               'classifier__learning_rate':[.0001,.001,.01,.05,.1,.2,.3,.4],
               'classifier__colsample_bylevel':np.arange(0.3,1.1,.1),
               'classifier__colsample_bytree':np.arange(.3,1.1,.1),
               'classifier__n_estimators':np.arange(100,600,25) # default 100, number o
              }

pipe = imbpipeline(steps=[['smote',SMOTE(random_state=42)],
                          ['classifier',XGBClassifier(eval_metric = 'merror')]])


# set up the randomizedsearchcv object
xgb_random_grid = RandomizedSearchCV(estimator=pipe,
                                     param_distributions=random_grid,
                                     scoring='accuracy',
                                     n_iter=200,
                                     cv=3, verbose=1)
# fit the object
xgb_random_grid.fit(X_train, y_train)

stop = time.time()
print('time it took: {} hours.'.format(round((stop-start)/3600,2)))

# with open('models/xgb_random_grid.pickle','wb') as f:
#     pickle.dump(xgb_random_grid, f)
```

```
Fitting 3 folds for each of 200 candidates, totalling 600 fits
time it took: 19.39 hours.
```

In [30]:
```
print('XGBoost Model 2 (RandomizedSearchCV/SMOTE)')
display_results(xgb_random_grid.best_estimator_.named_steps['classifier'], X_train, X_t
```

```
XGBoost Model 2 (RandomizedSearchCV/SMOTE)

Training Accuracy: 85.16%

Testing Accuracy: 77.81%

False positives: 1274

False negatives: 1649

Total true positives for minority classes: 4866

              precision    recall  f1-score   support
```

```
              functional       0.80      0.84      0.82      7963
functional needs repair       0.35      0.44      0.39      1076
          non functional       0.85      0.76      0.80      5811

                accuracy                           0.78     14850
               macro avg       0.67      0.68      0.67     14850
            weighted avg       0.79      0.78      0.78     14850
```



In [53]:
```
score = np.mean(cross_val_score(xgb_random_grid.best_estimator_.named_steps['classifier
print(f'Mean Cross Validation Score for an XGBoost Classifier (RandomizedSearchCV): {sc
print('RF RandomizedSearchCV Best params: {}'.format(xgb_random_grid.best_params_))
```

```
Mean Cross Validation Score for an XGBoost Classifier (RandomizedSearchCV):  79.34%
RF RandomizedSearchCV Best params: {'classifier__n_estimators': 500, 'classifier__min_ch
ild_weight': 3, 'classifier__max_depth': 19, 'classifier__learning_rate': 0.01, 'classif
ier__colsample_bytree': 0.7000000000000002, 'classifier__colsample_bylevel': 0.4}
```

In [31]:
```
xgb_random_grid_df = pd.DataFrame(xgb_random_grid.cv_results_).sort_values(by='rank_tes
xgb_random_grid_df.head().iloc[:,4:]
```

Out[31]:

	param_classifier__n_estimators	param_classifier__min_child_weight	param_classifier__max_depth	para
34	500	3	19	
73	150	3	16	
16	550	4	18	
197	425	3	17	
10	425	8	17	

Model 3: GridSearchCV

Using the parameters established above, a grid search was conducted to further optimize model results.

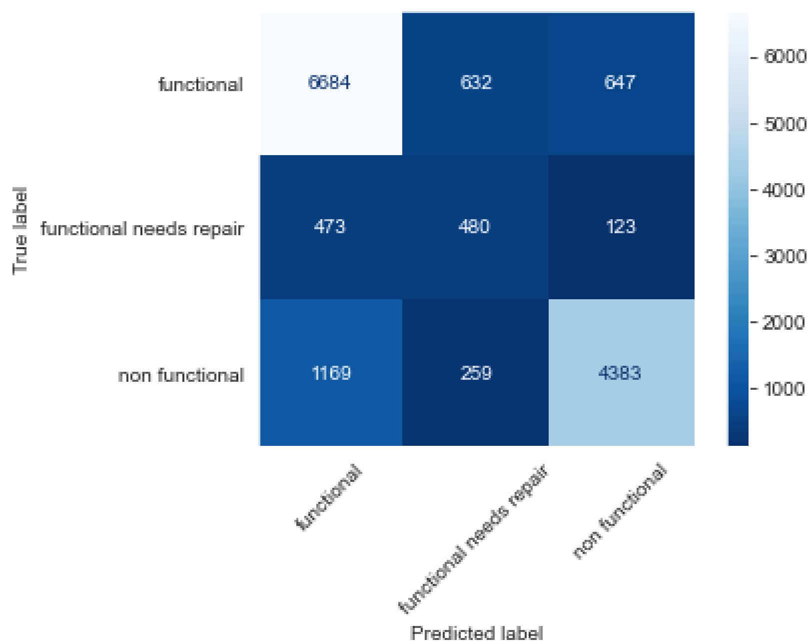
In [71]:
```
# if not xg_grid:
# set params for search using the randomized search params as guide

start = time.time()

xg_grid_params = {'classifier__n_estimators': np.arange(400,500,25),
                  'classifier__max_depth': [16,17,18],
                  'classifier__min_child_weight': [6,7,8],
                  'classifier__learning_rate':[0.025,0.05],
                  'classifier__colsample_bytree': [0.6,0.7],
                  'classifier__colsample_bylevel': [0,4,0.5]}

# Set up smote in pipeline so that smote is applied to every fold
pipe_smote = imbpipeline(steps=[['smote',SMOTE(random_state=42)],
                                ['classifier',XGBClassifier(eval_metric = 'merror')]])

xgb_grid_smote = GridSearchCV(estimator=pipe_smote,
                              param_grid=xg_grid_params,
                              scoring='accuracy',
                              cv=3,
                              verbose=1)

# fit the grid searches
xgb_grid_smote = xgb_grid_smote.fit(X_train, y_train)

stop = time.time()
print('time it took: {} hours'.format(round((stop-start),2)/3600))

# # save the models for easy loading on notebook restart
# with open("models/xgb_grid_smote.pickle",'wb') as f:
#     pickle.dump(xgb_grid_smote, f)
```

```
Fitting 3 folds for each of 432 candidates, totalling 1296 fits
time it took: 24.435986111111113 hours
```

In [30]:
```
# with funder and installer engineered features
print('XGBoost Model 3 (GridSearchCV/SMOTE)')
display_results(xgb_grid_smote.best_estimator_.named_steps['classifier'], X_train, X_te
```

```
XGBoost Model 3 (GridSearchCV/SMOTE)

Training Accuracy: 84.49%

Testing Accuracy: 77.76%

False positives: 1279

False negatives: 1642

Total true positives for minority classes: 4863

                         precision    recall  f1-score   support

             functional       0.80      0.84      0.82      7963
functional needs repair       0.35      0.45      0.39      1076
         non functional       0.85      0.75      0.80      5811
```

```
    accuracy                           0.78     14850
   macro avg       0.67      0.68      0.67     14850
weighted avg       0.79      0.78      0.78     14850
```



Final Step:

Take the top 25 models from the GridSearchCV and rank them based on recall scores and cross validation using the entire dataset.

In [23]:

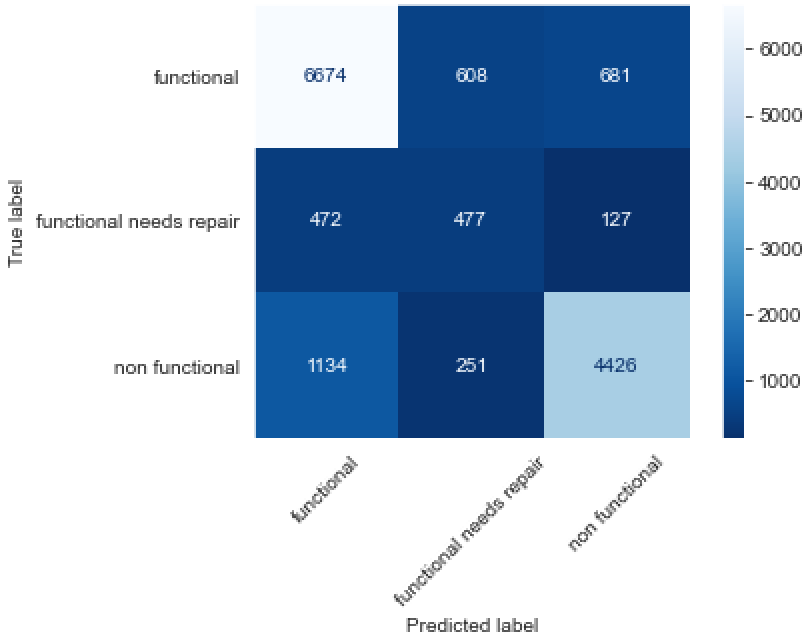
```
xgb_grid_smote_df = pd.DataFrame(xgb_grid_smote.cv_results_).sort_values(by='rank_test_
xgb_grid_smote_df.head().iloc[:,4:]
best_xgb = get_best_clf(xgb_grid_smote_df.head(25), XGBClassifier(eval_metric='merror')

best_xgb_df, best_xgb_model  = best_xgb[0], best_xgb[1]
```

In [34]:

```
best_xgb_model  = best_xgb[1]

print(f'Final XGBoost Model CV Score: {best_xgb_df.iloc[0].cv_score}')
print('\nFinal XGBoost Model')
display_results(best_xgb_model, X_train, X_test, y_train, y_test)
```

```
Final XGBoost Model CV Score: 0.7763299663299664

Final XGBoost Model

Training Accuracy: 86.53%

Testing Accuracy: 77.96%

False positives: 1289

False negatives: 1606

Total true positives for minority classes: 4903

                         precision    recall  f1-score   support

             functional       0.81      0.84      0.82      7963
```

```
functional needs repair       0.36      0.44      0.40      1076
         non functional       0.85      0.76      0.80      5811

               accuracy                           0.78     14850
              macro avg       0.67      0.68      0.67     14850
           weighted avg       0.79      0.78      0.78     14850
```



Interpretation:

- When compared to the baseline model, the final XGBoost model had a slightly weaker cross validation score (77.63% vs 77.98%) and a slightly weaker testing score (77.96% for Model 3 vs 78.39% for Baseline model).
- Model 3 showed significant improvements for recall scores on the minority classes ('functional' and 'functional needs repair') indicating it made more overall correct classifications for those categories.
- Overall, when comparing models fitted with oversampled data, the XGBoost model appears to be a stronger fit than both the logistic regression and random forest models.

In [33]:

```
# add the models to dictionary
xgb_final = best_xgb_model
xgb_final_cv = best_xgb_df.cv_score.iloc[0]
y_pred = xgb_final.predict(X_test)
add_model_dict(xgb_final, 'xgb_final', y_test, y_pred, xgb_final_cv)

# with open("models/xgb_final.pickle",'wb') as f:
#     pickle.dump(xgb_final, f)
```

```
model added to dictionary.
```

XGBoost Feature Analysis

In [51]:

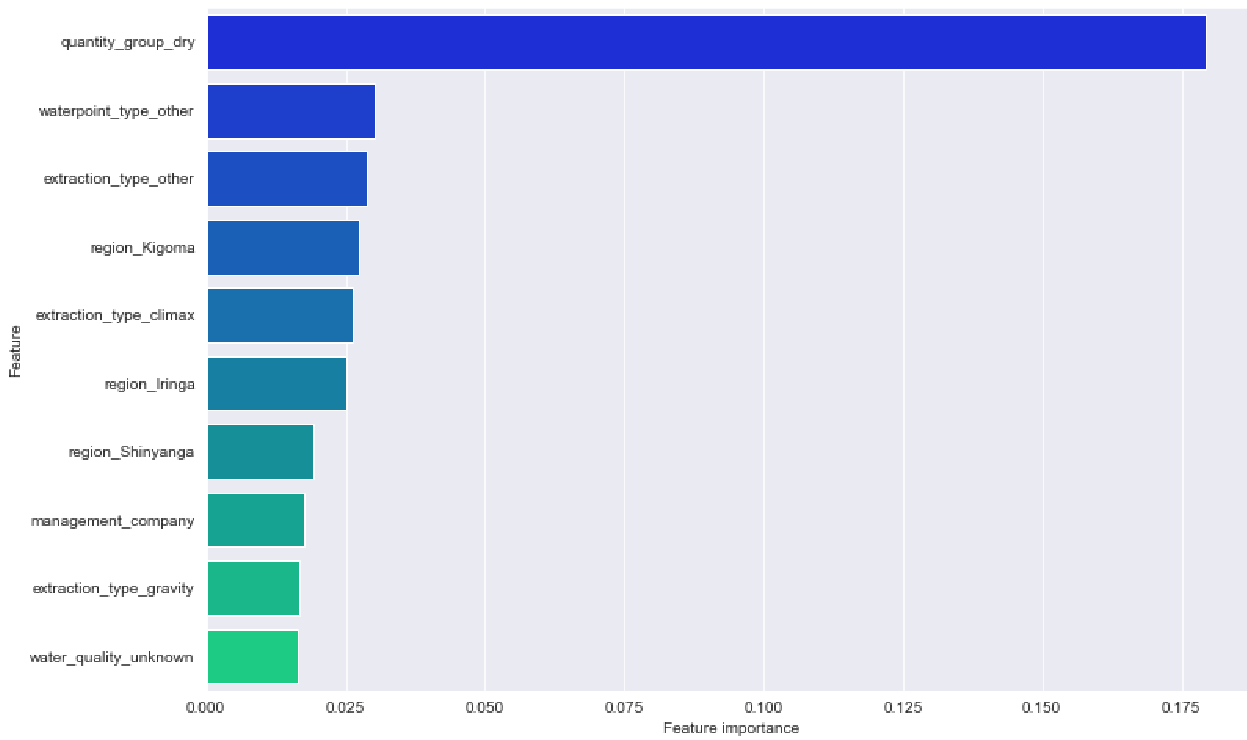
```
pd.DataFrame(list(zip(xgb_final['classifier'].feature_importances_, X_train.columns.val
```

Out[51]:

	coef	feature
82	0.179458	quantity_group_dry
100	0.030228	waterpoint_type_other
46	0.028768	extraction_type_other
17	0.027482	region_Kigoma
39	0.026262	extraction_type_climax
15	0.025015	region_Iringa
29	0.019310	region_Shinyanga
55	0.017545	management_company
40	0.016552	extraction_type_gravity
81	0.016393	water_quality_unknown

In [101…

```
plot_feature_importances(xgb_final['classifier'])
```

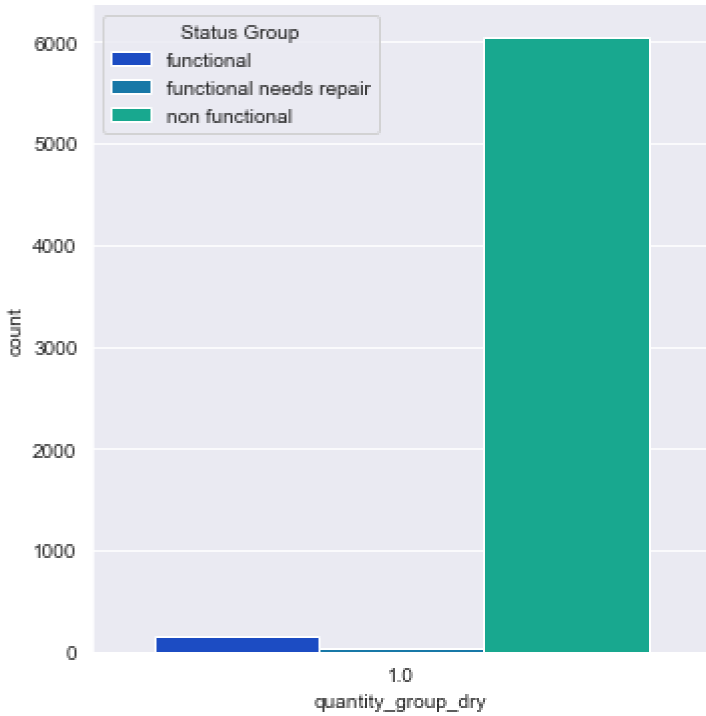


Further analysis on top features

quantity_group_dry

In [56]:

```
# plot the quantity group dry feature by status group
sns.set_style('darkgrid')
temp = pd.concat([X_all, pd.DataFrame(y_all, columns=['status_group'])], axis=1)
sns.catplot(x='quantity_group_dry', kind='count', hue='status_group',
            data=temp[temp.quantity_group_dry == 1], height=5, palette='winter', legend
plt.legend(title='Status Group',labels=['functional','functional needs repair','non fun
plt.show()
```



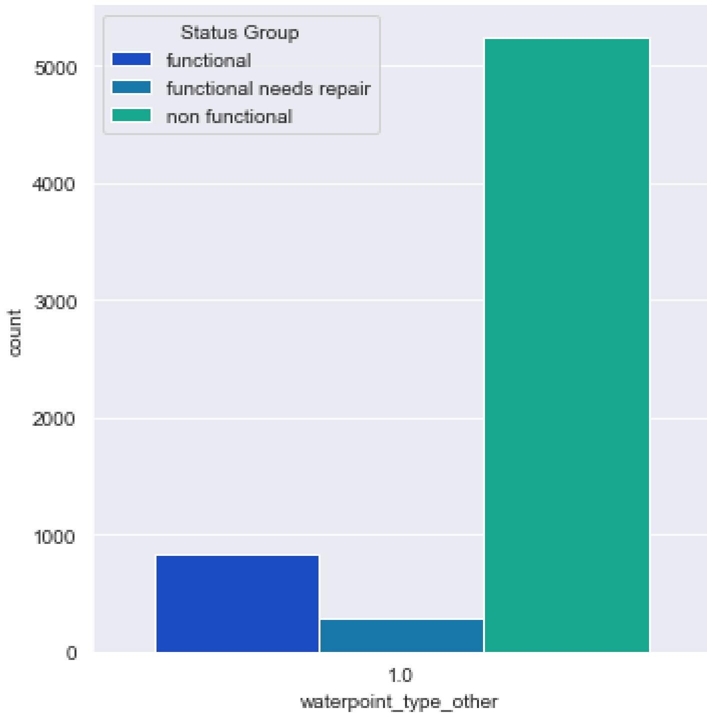
In [69]:
```
print(status_group_dict)

temp[temp.quantity_group_dry == 1].status_group.value_counts(normalize=True)
```

```
{0: 'functional', 1: 'functional needs repair', 2: 'non functional'}
```

Out[69]:
```
2    0.968940
0    0.025136
1    0.005924
Name: status_group, dtype: float64
```

waterpoint_type_other

In [71]:
```
# plot the quantity group dry feature by status group
sns.set_style('darkgrid')
temp = pd.concat([X_all, pd.DataFrame(y_all, columns=['status_group'])], axis=1)
sns.catplot(x='waterpoint_type_other', kind='count', hue='status_group',
            data=temp[temp.waterpoint_type_other == 1], height=5, palette='winter', leg
plt.legend(title='Status Group',labels=['functional','functional needs repair','non fun
plt.show()
```



In [68]:
```
temp[temp.waterpoint_type_other == 1].status_group.value_counts(normalize=True)
```

Out[68]:
```
2    0.822414
0    0.131661
1    0.045925
Name: status_group, dtype: float64
```

4. Results Analysis

During this step, I look at the data for each of the models chosen from above, alongside their baseline counterparts in order to make a final determination on classification model. The three algorithms used:

- Logistic Regression
- Random Forests
- XGBoost

In [40]:
```
results_df = pd.DataFrame.from_dict(model_dict,orient='index').reset_index()
results_df = results_df.rename(columns={'index':'model_name'})
results_df['train_time'] = results_df.model.apply(lambda x: training_time(x))
```

In [41]:
```
results_df.head()
```

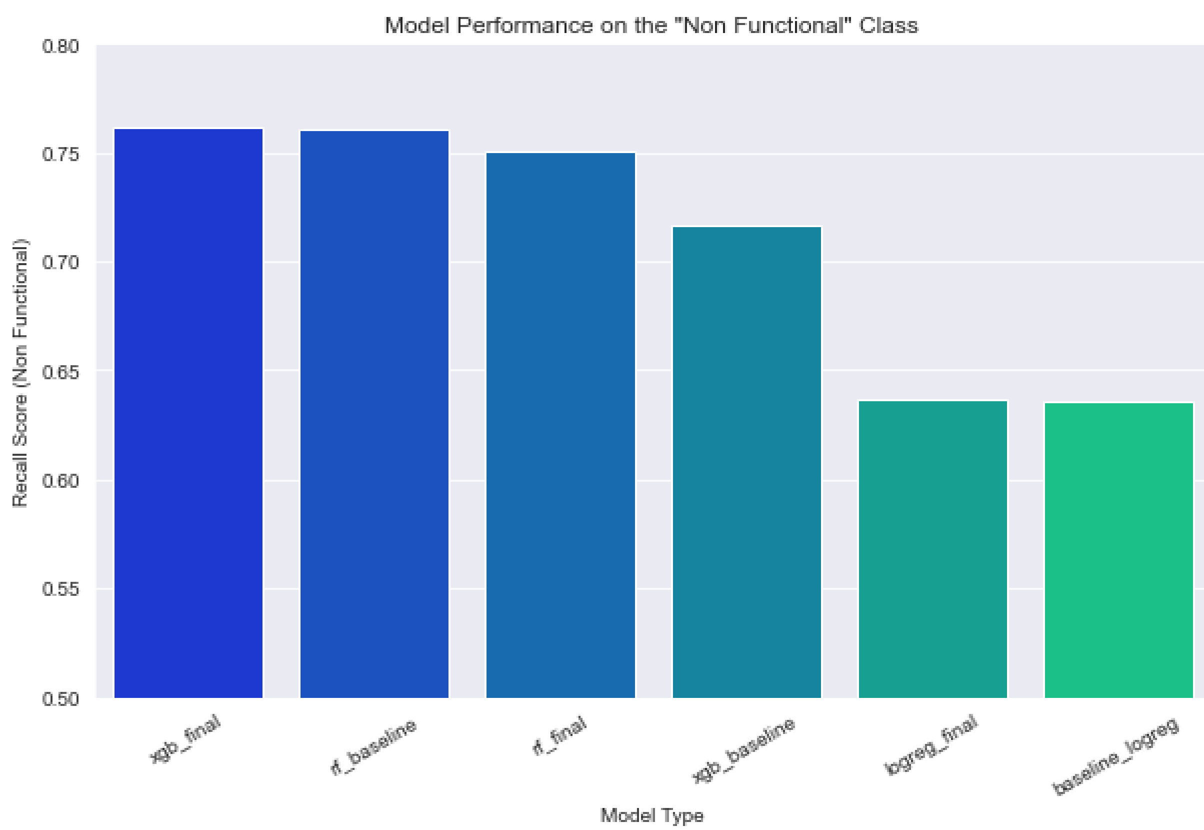
Out[41]:

	model_name	model	parms	overall_accuracy	fn	cv_scc
0	baseline_logreg	LogisticRegression(max_iter=2000, multi_class=...	{'C': 1.0, 'class_weight': None, 'dual': False...	73.461279	2963	0.7353

	model_name	model	parms	overall_accuracy	fn	cv_scc
1	logreg_final	(SMOTE(n_jobs=-1, random_state=42), StandardSc...	{'memory': None, 'steps': [('smote', SMOTE(n_j...	63.501684	1372	0.6300
2	rf_baseline	(DecisionTreeClassifier(max_features='auto', r...	{'bootstrap': True, 'ccp_alpha': 0.0, 'class_w...	78.397306	1840	0.7832
3	rf_final	(SMOTE(random_state=42), (DecisionTreeClassifi...	{'memory': None, 'steps': [('smote', SMOTE(ran...	77.589226	1640	0.7737
4	xgb_baseline	XGBClassifier(base_score=0.5, booster='gbtree'...	{'objective': 'multi:softprob', 'base_score': ...	78.390572	2311	0.7850

In [46]:

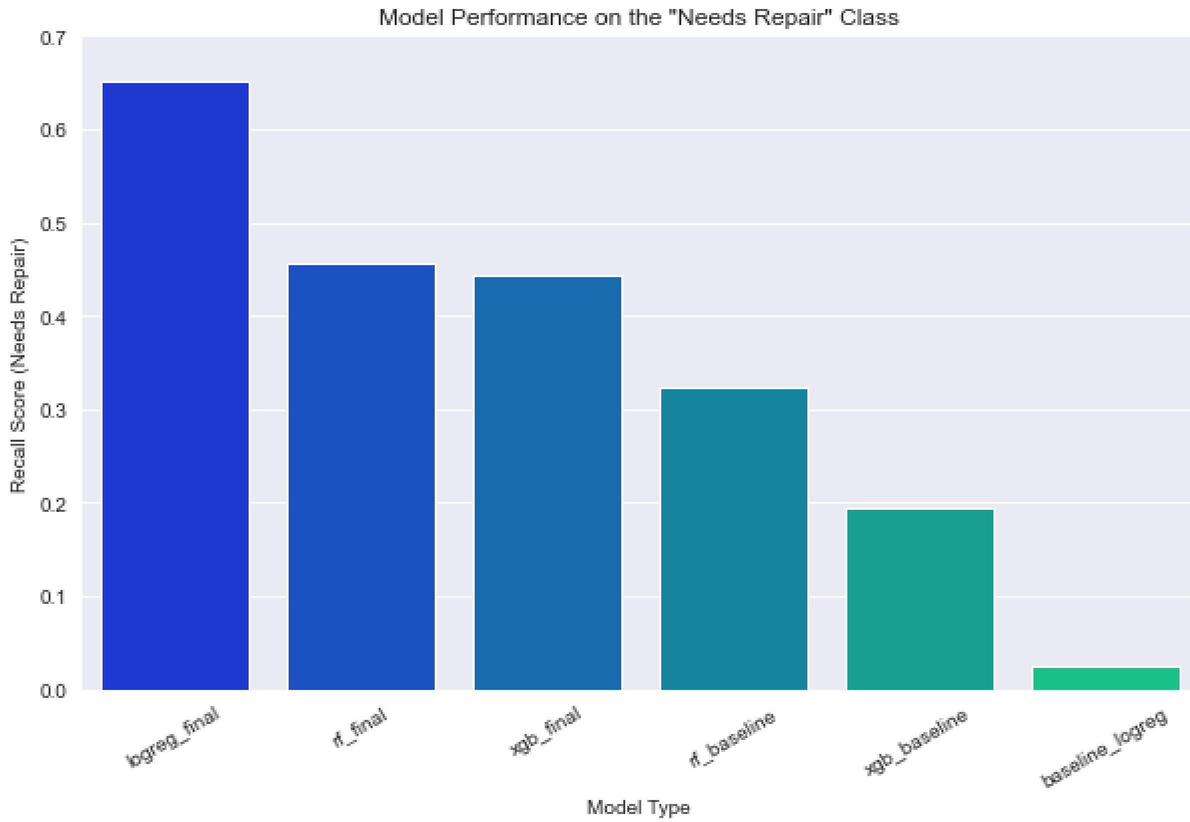
```
plt.figure(figsize=(10,6))
sns.set_style('darkgrid')
ax = sns.barplot(x='model_name', y='nf_recall',data=results_df.sort_values(by='nf_recal
ax.set_title('Model Performance on the "Non Functional" Class')
ax.set_ylabel('Recall Score (Non Functional)')
ax.set_xlabel('Model Type')
ax.set_xticklabels(ax.get_xticklabels(),rotation = 30)
ax.set_ylim(0.5,0.8)
plt.show()
```



In [44]:

```
plt.figure(figsize=(10,6))
```

```
sns.set_style('darkgrid')
ax = sns.barplot(x='model_name', y='needs_repair_recall',data=results_df.sort_values(by
ax.set_title('Model Performance on the "Needs Repair" Class')
ax.set_ylabel('Recall Score (Needs Repair)')
ax.set_xlabel('Model Type')
ax.set_xticklabels(ax.get_xticklabels(),rotation = 30)
ax.set_ylim(0,0.7)
plt.show()
```



Here you can see the effects of the SMOTE oversampling technique on each of the baseline models. An increase in recall here corresponds to finding more of the minority class ('functional needs repair'). The model say 'yes' more frequently to this class, at the expense of more false positives (incorrectly classifying a waterwell as this class).

Due to using oversampling, the precision doesn't go up with the final model. Recall and precision are inversely related and the current graph is almost an opposite to the graph above.

This corresponds to the fact that there are more false positives. The model is more likely to 'guess' that a waterwell is from the 'functional needs repair' class than without oversampling, but those guesses are also incorrect.

The final xgboost model precision goes up, and this may be due to effective hyperparameter tuning.

In [47]:
```
results_df.sort_values(by='overall_accuracy', ascending=False)[['model_name','cv_score'
```

Out[47]:

	model_name	cv_score
2	rf_baseline	0.783215
4	xgb_baseline	0.785017

	model_name	cv_score
5	xgb_final	0.776330
3	rf_final	0.773771
0	baseline_logreg	0.735370
1	logreg_final	0.630079

Although the final logistic regression model had the best recall score for our minority class, it scored very low on overall accuracy, which is one reason it wasn't chosen as the strongest model. XGBoost outperformed other models on overall testing accuracy, as well as on the minority class 'functional needs repair' and the number of false negatives (missing waterpoints that need to be repaired/replaced).

Model Comparison (Radar Chart)

In [48]:

```
results_df[results_df.model_name.apply(lambda x: 'baseline' not in x)]
```

Out[48]:

	model_name	model	parms	overall_accuracy	fn	cv_score	functional_prec
1	logreg_final	(SMOTE(n_jobs=-1, random_state=42), StandardSc...	{'memory': None, 'steps': [('smote', SMOTE(n_j...	63.501684	1372	0.630079	0.78
3	rf_final	(SMOTE(random_state=42), (DecisionTreeClassifi...	{'memory': None, 'steps': [('smote', SMOTE(ran...	77.589226	1640	0.773771	0.80
5	xgb_final	(SMOTE(random_state=42), XGBClassifier(base_sc...	{'memory': None, 'steps': [('smote', SMOTE(ran...	77.959596	1606	0.776330	0.80

In [49]:

```
# normalize the columns for the radar chart
results_df['train_time_normalized_inverse'] = 1- ((results_df['train_time'] - np.min(re
                                          (np.max(results_df.train_time) - np.min(resul

results_df['cv_score_normalized'] = ((results_df['cv_score'] - np.min(results_df.cv_sco
                                        (np.max(results_df.cv_score) - np.min(results_d

results_df['needs_repair_recall_normalized'] = ((results_df['needs_repair_recall'] - np
                                          (np.max(results_df.needs_repair_recall) - np.

results_df['nf_recall_normalized'] = ((results_df['nf_recall'] - np.min(results_df.nf_r
                                          (np.max(results_df.nf_recall) - np.min(result

results_df['nf_precision_normalized'] = ((results_df['nf_precision'] - np.min(results_d
                                          (np.max(results_df.nf_precision) - np.min(res
```

```
results_df['needs_repair_precision_normalized'] = ((results_df['needs_repair_precision'
                                                     (np.max(results_df.needs_repair_pre

# drop the baseline models
results_df_plot = results_df[results_df.model_name.apply(lambda x: 'baseline' not in x)

print(list(results_df_plot.model_name))

# create the radar chart
import plotly.graph_objects as go
import plotly.offline as pyo

radar_df = results_df_plot[['cv_score_normalized',
                            'needs_repair_recall_normalized',
                            'nf_recall_normalized',
                            'needs_repair_precision_normalized',
                            'nf_precision_normalized',
                            'train_time_normalized_inverse']]

model_names = list(results_df_plot.model_name) #list(results_df_plot.model_name)

categories = ['Cross Validation Score',
              'Recall (Needs Repair)',
              'Recall (Non Functional)',
              'Precision (Functional Needs Repair)',
              'Precision (Non Functional)',
              'Train Time (Inverse)']

categories = [*categories, categories[0]]

models=[]
for i in range(len(model_names)):
    temp = list(radar_df.iloc[i].values)
    temp = [*temp, temp[0]]
    models.append(temp)

fig = go.Figure(
    data=[
        go.Scatterpolar(r=models[0], theta=categories, fill='toself', name= model_names
        go.Scatterpolar(r=models[1], theta=categories, fill='toself',  name= model_name
        go.Scatterpolar(r=models[2], theta=categories, fill='toself', name= model_names
#         go.Scatterpolar(r=models[3], theta=categories, name= model_names[3]),
#         go.Scatterpolar(r=models[4], theta=categories, name= model_names[4]),
#         go.Scatterpolar(r=models[5], theta=categories, name= model_names[5])
    ],
    layout=go.Layout(
        title=go.layout.Title(text='Model Comparison'),
        polar={'radialaxis':{'visible':True}},
        showlegend=True
    )
)
pyo.plot(fig)
```

```
['logreg_final', 'rf_final', 'xgb_final']
```

Out[49]:
```
'temp-plot.html'
```

5. Feature Selection

Source

Taking the final XGBoost model chosen, I decided to do feature selection with Recursive Feature Elimination to determine whether the model could be optimized even further by using RFE to decide on which features to keep, as opposed to EDA used above. To do so I took the following steps:

1. Took the original data, and only dropped the columns with missing values. After one-hot-encoding, this left 344 columns compared to 105 columns originally.
2. Conduct an RFE grid search using the RFECV package from sklearn and the XGBoost model with the parameters from above. The output gave me an final list and number of features.
3. Visualize the results and compare to the XGBoost model above.

In [43]:

```
with open('models/rfecv.pickle','rb') as f:
    rfecv = pickle.load(f)

with open("models/xgb_final_rfe.pickle",'rb') as f:
    xgb_final_rfe = pickle.load(f)
```

In [174…

```
# start from scratch
training_values = pd.read_csv('tanzania_training_values.csv')
training_labels = pd.read_csv('tanzania_training_labels.csv')

to_drop_numeric = ['id','date_recorded','construction_year','longitude','latitude','amo

to_drop_cat = ['funder','installer','wpt_name','subvillage','ward','scheme_name','recor
               'scheme_management','permit']

cols_to_drop = to_drop_numeric + to_drop_cat
print(f'{len(cols_to_drop)} columns were dropped')

X,y = prep_data(training_values, training_labels)
X = engineer_features(X) # add new columns
X = X.drop(cols_to_drop, axis=1)

print(f'Columns to keep:{X.columns}')
numeric_cols =['gps_height','num_private','population']

# encode and split data
X_train_fe, X_test_fe, y_train_fe, y_test_fe, X_all_fe, y_all_fe = encode_split_data(X,

# keep track of final models for comparison
model_dict = {}

# for evaluating model fitting
kf = KFold(n_splits=5, random_state=42, shuffle=True)

print(len(X_train_fe.columns))
```

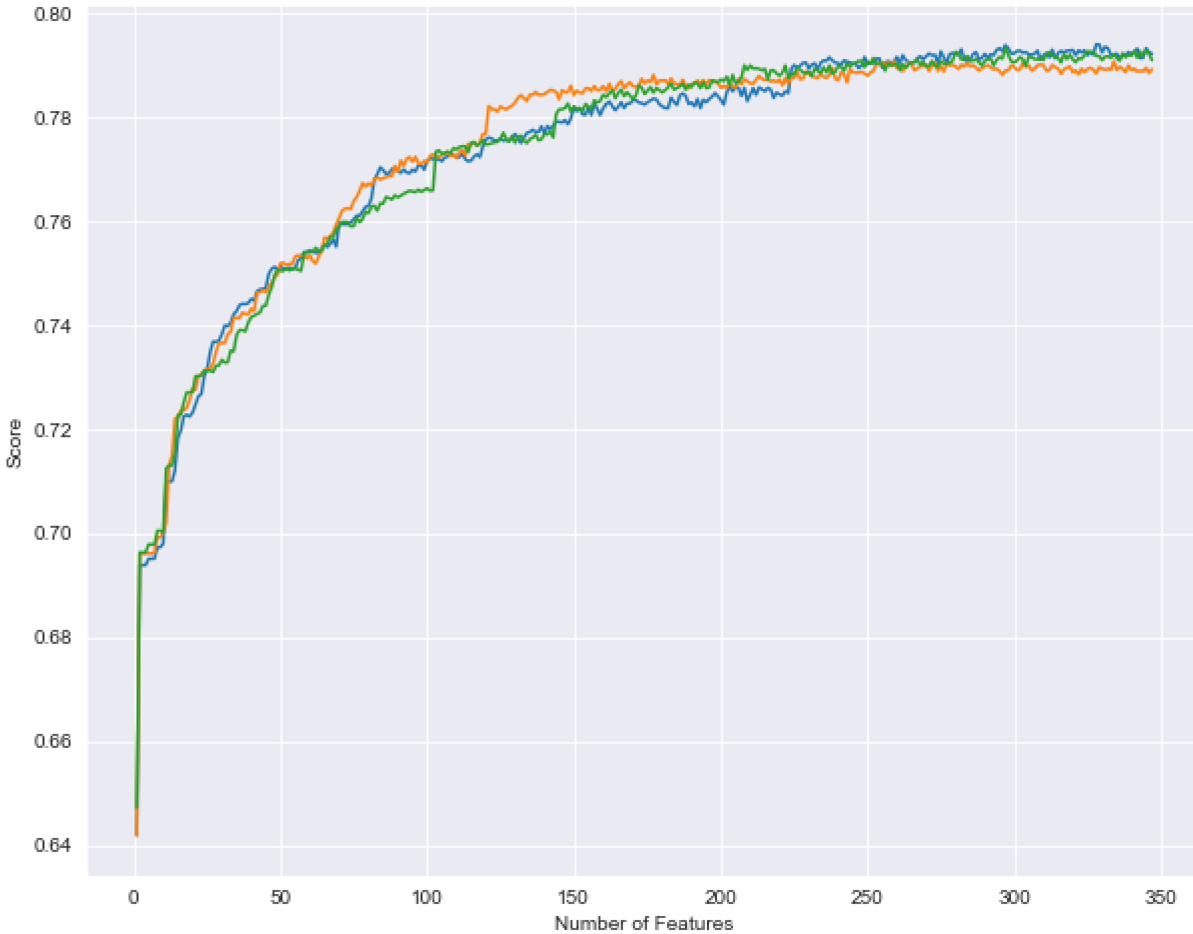
```
15 columns were dropped
Columns to keep:Index(['gps_height', 'num_private', 'basin', 'region', 'region_code',
       'district_code', 'lga', 'population', 'public_meeting',
       'extraction_type', 'extraction_type_group', 'extraction_type_class',
       'management', 'management_group', 'payment', 'payment_type',
       'water_quality', 'quality_group', 'quantity', 'quantity_group',
       'source', 'source_type', 'source_class', 'waterpoint_type',
       'waterpoint_type_group', 'construction_year_label', 'cluster_label',
```

```
       'installer_bool', 'funder_bool'],
      dtype='object')
Number of columns after encoding: 347
347
```

In [43]:
```python
# using the optimal model found in the last step conduct a RFECV
xgb_rfe = xgb_grid_smote.best_estimator_.named_steps['classifier']

rfecv = RFECV(estimator=xgb_rfe,
              step=1,
              cv=3,
              scoring='accuracy',
              min_features_to_select=1,
              verbose=0)

rfecv.fit(X_train_fe, y_train_fe)

print(f'Optimal number of features: {rfecv.n_features_}')

# Filter X_train and X_test using the columns selected by RFECV
X_train_rfe = X_train_fe.loc[:,rfecv.support_]
X_test_rfe = X_test_fe.loc[:,rfecv.support_]

# save the columns for later use
rfe_cols = X_train_rfe.columns

with open("models/rfecv.pickle",'wb') as f:
    pickle.dump(rfecv, f)

with open("models/rfe_cols.pickle",'wb') as f:
    pickle.dump(rfe_cols, f)
```

In [177…
```python
# df_features = pd.DataFrame(columns=['feature','support','ranking'])
# for i in range(X_train_rfe.shape[1]):
#     row = {'feature':i, 'feature_name':X_train_rfe.columns[i], 'support':rfecv.suppor
#     df_features = df_features.append(row, ignore_index=True)
# df_features.sort_values(by='ranking')
```

In [52]:
```python
plt.figure(figsize=(10,8))
plt.plot(range(1, len(rfecv.grid_scores_)+1), rfecv.grid_scores_)
plt.xlabel('Number of Features')
plt.ylabel('Score')
plt.show()
```



In [172...

```
xgb_final_rfe = xgb_final.fit(X_train_rfe, y_train_fe)
```

In [179...

```
# with open("models/xgb_final_rfe.pickle",'wb') as f:
#     pickle.dump(xgb_final_rfe, f)
```

In [175...

```
xgb_final_rfe_cv_score = np.mean(cross_val_score(xgb_final_rfe, X_all_fe, y_all_fe, cv=
print(f'Mean Cross Validation Score for an XGBoost Classifier (with RFE): {xgb_final_rf
```

```
Mean Cross Validation Score for an XGBoost Classifier (with RFE):  78.39%
```

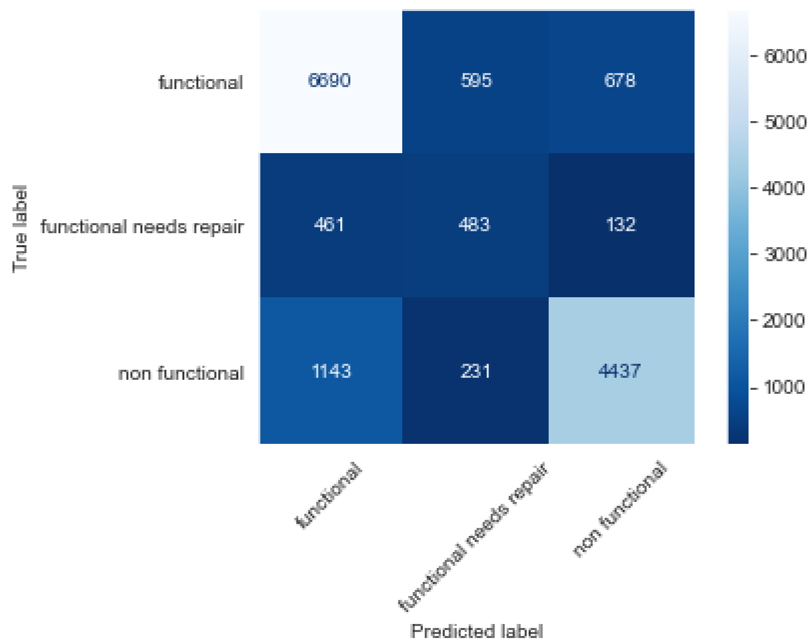
In [176...

```
# xgb_rfe = xgb_grid_rfe.best_estimator_.named_steps['classifier']
print('XGBoost Model with Recursive Feature Elimination\n')
display_results(xgb_final_rfe, X_train_rfe, X_test_rfe, y_train_fe, y_test_fe)
```

```
XGBoost Model with Recursive Feature Elimination


Training Accuracy: 85.52%

Testing Accuracy: 78.18%

False positives: 1273

False negatives: 1604

Total true positives for minority classes: 4920
```

	precision	recall	f1-score	support
functional	0.81	0.84	0.82	7963

```
functional needs repair       0.37      0.45      0.41      1076
         non functional       0.85      0.76      0.80      5811

               accuracy                           0.78     14850
              macro avg       0.67      0.68      0.68     14850
           weighted avg       0.79      0.78      0.78     14850
```



Considerations

- This model does outperform the final model chosen above (78.4% vs. 77.6%), although at the expense of increasing dimensionality (312 vs. 109 features).
- Results indicated a stronger model, but the resulting increase in dimensions means longer fitting and prediction times. This should be considered.

6. Predictions

Here I define a method for making predictions with the chosen model. This method takes in data with unknown labels, and assigns a label ('functional', 'non functional', 'functional needs repair') to each of the rows in the data.

In [45]:
```
data = pd.read_csv('tanzania_test_values.csv')
```

In [46]:
```
with open("models/rfe_cols.pickle",'rb') as f:
    rfe_cols = pickle.load(f)

with open('models/df_final.pickle', 'rb') as f:
    df_final = pickle.load(f)
```

In [47]:
```
"""
take in testing data (as dataframe) about well(s) and make prediction(s) about status
return a dataframe, each row containing an id and features for a well and a correspondi
"""

to_drop_numeric = ['id','date_recorded','construction_year','longitude','latitude','amo
```

```
to_drop_cat = ['funder','installer','wpt_name','subvillage','ward','scheme_name','recor
               'scheme_management','permit']

cols_to_drop = to_drop_numeric + to_drop_cat

# save columns for final dataframe
ids = data['id']
funders = data['funder']
installers=data['installer']
construction_years = data['construction_year']

# convert cat columns into objects
for col in data:
    if data[col].dtype == object:
        data[col] = data[col].astype('category')

# REMOVE OUTLIERS
# latitude and longitude - remove outliers (waterpoints located at 0 longitude in the o
lat = data[data.longitude != 0].latitude.median()
long = data[data.longitude != 0].longitude.median()
data['latitude'] = np.where((data.longitude==0), lat, data.latitude)
data['longitude'] = np.where((data.longitude==0), long, data.longitude)

lat_lon = data[['latitude','longitude']]

# MISSING VALUES
# replace the null values for permit
isnull = data.permit.isnull()
sample = data.permit.dropna().sample(isnull.sum(), replace=True, random_state=123).valu
data.loc[isnull,'permit'] = sample

# replace the null values for public_meeting
isnull = data.public_meeting.isnull()
sample = data.public_meeting.dropna().sample(isnull.sum(), replace=True, random_state=1
data.loc[isnull,'public_meeting'] = sample

# FEATURE ENGINEERING
data = engineer_features(data) # add new columns

# DROP COLUMNS
data = data.drop(cols_to_drop, axis=1)

# CREATE DATAFRAME
numeric_cols =['gps_height','num_private','population']
cat_cols = data.drop(numeric_cols,axis=1).columns # get cat cols
rfe_col_filter = rfe_cols

# save copy of dataframe for reading later
df_final = pd.concat([data[cat_cols],data[numeric_cols]],axis=1)

# for predictions
# one hot encode
ohe = OneHotEncoder()
ohe = OneHotEncoder(handle_unknown='ignore', sparse=False) #drop=first
data_ohe = pd.DataFrame(ohe.fit_transform(data[cat_cols]), columns=ohe.get_feature_name

# combine one hot encoded columns and numeric columns
# only include columns from RFE
data_final = pd.concat([data_ohe, data[numeric_cols]], axis=1)
```

```python
data_final = data_final[rfe_col_filter]

# MAKE PREDICTIONS
# make preds
preds = pd.DataFrame(xgb_final_rfe.predict(data_final), columns=['status_group_enc'])
preds['status_group'] = preds.status_group_enc.apply(lambda x: status_group_dict[x])

# append predictions to dataframe without encoding
df_final = pd.concat([ids, df_final, preds, lat_lon, funders, installers, construction_
df_final.head()

# with open('models/df_final.pickle', 'wb') as f:
#     pickle.dump(df_final, f)
```

In [48]:

```python
df_final.head()
```

Out[48]:

	id	basin	region	region_code	district_code	lga	public_meeting	extraction_type	extra
0	50785	Internal	Manyara	21	3	Mbulu	True	other	
1	51630	Pangani	Arusha	2	2	Arusha Rural	True	gravity	
2	17168	Internal	Singida	13	2	Singida Rural	True	other	
3	45559	Ruvuma / Southern Coast	Lindi	80	43	Liwale	True	other	
4	49871	Ruvuma / Southern Coast	Ruvuma	10	3	Mbinga	True	gravity	

5 rows × 37 columns

In [87]:

```python
# regions with the most non functional waterpoints
# nf_region_count_final = df_final[df_final.status_group == 'non functional'].groupby('
# nf_region_count_final
```

In [50]:

```python
# regions with the most non functional or needs repair waterpoints
in_need = df_final[(df_final.status_group == 'non functional') | (df_final.status_group
nf_repair_region_count_final = in_need.groupby('region').id.count().reset_index().sort_
nf_repair_region_count_final
```

Out[50]:

	region	count
10	Mbeya	636
17	Shinyanga	579
4	Kagera	451
11	Morogoro	426
13	Mwanza	389

The regions with the most non functional waterpoitns are also the regions with the most non functional and needs repair waterpoints.

In [86]:
```
in_need_region = df_final[df_final.region.apply(lambda x: x  in nf_repair_region_count_
plot_lat_long(in_need_region)
```

Make this Notebook Trusted to load map: File -> Trust Notebook

In [51]:
```
df_final[df_final.quantity_group == 'dry'].status_group.value_counts(normalize=True)
```

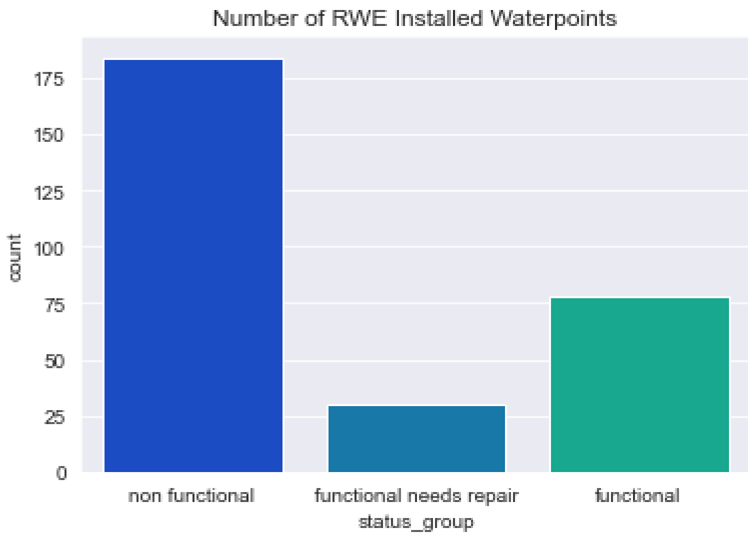
Out[51]:
```
non functional             0.997396
functional needs repair    0.001953
functional                 0.000651
Name: status_group, dtype: float64
```

In [52]:
```
in_need_installer = in_need.groupby('installer').id.count().reset_index().sort_values(b
in_need_installer
```

Out[52]:

	installer	count
120	DWE	1713
178	Government	307
407	RWE	214
243	KKKT	117
104	DANIDA	112

In [76]:
```
sns.countplot(df_final[df_final.installer == 'RWE'].status_group, palette='winter')
plt.title('Number of RWE Installed Waterpoints')
plt.show()
```

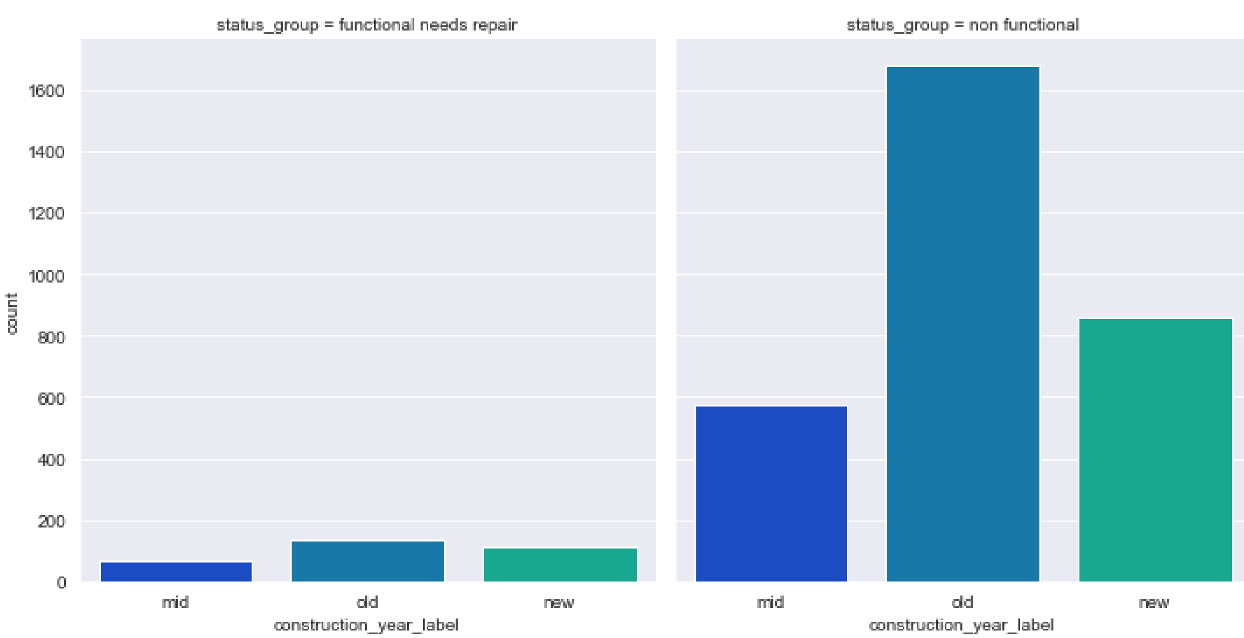


In [67]:
```
# # show the number of non functional and needing repair waterpoints, grouped by funder
# in_need_funder = in_need.groupby('funder').id.count().reset_index().sort_values(by='i
# in_need_funder
```

In [66]:
```
# # show the number of non functional and needing repair waterpoints, grouped by year c
# in_need_years = in_need.groupby('construction_year_label').id.count().reset_index().s
# in_need_years
```

In [63]:
```
sns.catplot(x='construction_year_label', kind='count', col='status_group', data=in_need
```

Out[63]:
```
<seaborn.axisgrid.FacetGrid at 0x26d049bd160>
```



In [65]:
```
in_need[(in_need.construction_year_label != 'unknown') & (in_need.status_group == 'non
```

Out[65]:
```
old    0.539698
new    0.276438
mid    0.183864
Name: construction_year_label, dtype: float64
```

Conclusions

Next Steps

- Other machine learning algorithms: KNN, Naive Bayes, and Support Vector Machines are missing from the above trials. Due to the size of the data, training time should be considered.
- More feature selection techniques: Due to the larger number of features present in the dataset, utilizing other methods to refine the feature list could improve model performance and efficiency.
- Metrics: As mentioned above, the final models chosen in this analysis were based on optimizing for recall of the minority classes. Other metrics, like f-1 score and precision can be prioritized in future studies. In addition other resampling techniques could be experimented with to see if this positively affects results.
- More feature engineering: Conducting more EDA and experimenting with other ways of creating new features could yield more positive outcomes.
- Further investigation: Digging deeper into some of the insights. For example, it seems like waterpoints around Lake Victoria have more issues. Why? What makes waterpoints installed by certain parties more likely to have issues?

In [ ]:

```

```