0.1 Project #3:

- Student name: Milena Afeworki
- Student pace: full time
- Scheduled project review date/time: 07/09/2021 @ 10:15 PT
- Instructor name: Abhineet Kulkarni
- Blog post URL:

```
In [ ]:
```

1 Tanzanian Water Well Functionality Classification

Tanzania, as a developing country, struggles with providing clean water to its population of over 57,000,000. There are many waterpoints already established in the country, but some are in need of repair while others have failed altogether. Using data from Taarifa and the Tanzanian Ministry of Water, we need to predict which pumps are functional, which need some repairs, and which don't work at all. A smart understanding of which waterpoints will fail can improve maintenance operations and ensure that clean, potable water is available to communities across Tanzania.

1.1 The Business Problem

This project aims to use the data to anticipate when a well needs repair or maintenance, ideally before it breaks and disrupts the local water supply. Failing to identify nonfunctional water supply lines could lead villagers to suffer in many ways, including traveling long distances to other water sources resulting in increased time and effort to fetch water, and being exposed to different health-related issues that come with poor water quality. Accordingly, though it is important to build a model that will accurately classify the wells, it is crucial that our model's tolerance to errors of misclassifying wells, especially the 'nonfunctional' and 'needs repair' groups, is as low as possible.

1.2 Data Understanding

- **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

In [1]:
```
# import all the necessary libraries
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.model_selection import train_test_split, cross_val_score

from sklearn.metrics import confusion_matrix, classification_report
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
# sns.set_style('whitegrid')

from sklearn.metrics import plot_confusion_matrix

from xgboost import XGBClassifier
```
executed in 1.32s, finished 08:10:04 2021-07-11

1.3 Obtain data

Let's first import the data and take a look at the info to see if we need to do some data cleaning.

In [2]:

```
# Load data set
df = pd.read_csv('training_set_values.csv')
df.head()
```

executed in 335ms, finished 08:10:04 2021-07-11

Out[2]:

	id	amount_tsh	date_recorded	funder	gps_height	installer	longitude	latitude	wpt_name	num_private	...	payment_ty
0	69572	6000.0	2011-03-14	Roman	1390	Roman	34.938093	-9.856322	none	0	...	annu
1	8776	0.0	2013-03-06	Grumeti	1399	GRUMETI	34.698766	-2.147466	Zahanati	0	...	never p
2	34310	25.0	2013-02-25	Lottery Club	686	World vision	37.460664	-3.821329	Kwa Mahundi	0	...	per buc
3	67743	0.0	2013-01-28	Unicef	263	UNICEF	38.486161	-11.155298	Zahanati Ya Nanyumbu	0	...	never p
4	19728	0.0	2011-07-13	Action In A	0	Artisan	31.130847	-1.825359	Shuleni	0	...	never p

5 rows × 40 columns

In [3]:
```
df.info()
```

executed in 78ms, finished 08:10:04 2021-07-11

```
 22  permit                 56344 non-null  object
 23  construction_year      59400 non-null  int64
 24  extraction_type        59400 non-null  object
 25  extraction_type_group  59400 non-null  object
 26  extraction_type_class  59400 non-null  object
 27  management             59400 non-null  object
 28  management_group       59400 non-null  object
 29  payment                59400 non-null  object
 30  payment_type           59400 non-null  object
 31  water_quality          59400 non-null  object
 32  quality_group          59400 non-null  object
 33  quantity               59400 non-null  object

 34  quantity_group         59400 non-null  object
 35  source                 59400 non-null  object
 36  source_type            59400 non-null  object
 37  source_class           59400 non-null  object
 38  waterpoint_type        59400 non-null  object
 39  waterpoint_type_group  59400 non-null  object
dtypes: float64(3), int64(7), object(30)
```

In [4]:
```
# check for NaNs
df.isna().sum()
```
executed in 79ms, finished 08:10:04 2021-07-11

Out[4]:
```
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
dtype: int64
```

1.4 Scrubbing the data

1.4.1 Cleaning based on info

Key observations from here:

1. Dealing with missing values:

- funder = 3635
- installer = 3655
- subvillage = 371
- public_meeting = 3334
- scheme_managment = 3877
- scheme_name = 28166
- permit = 3056

2. Dealing with date_recorded data type.

3. Dealing with outliers.

1.4.2 Dealing with the missing values.

Funder

In [5]:
```
df.funder.value_counts(normalize=True)
```
executed in 30ms, finished 08:10:04 2021-07-11

Out[5]:
```
Government Of Tanzania    0.162898
Danida                    0.055841
Hesawa                    0.039487
Rwssp                     0.024639
World Bank                0.024191
                            ...   
Kanisa La Neema           0.000018
Tcrst                     0.000018
Raramataki                0.000018
Tasae                     0.000018
Africaone Ltd             0.000018
Name: funder, Length: 1897, dtype: float64
```

In [6]:
```
# Replace NaNs in 'funder' by 'other'
df['funder'] = df['funder'].replace(np.nan, 'other')
```
executed in 12ms, finished 08:10:04 2021-07-11

In [7]:
```
df.drop(columns=['scheme_name', 'subvillage', 'public_meeting',
                 'num_private', 'permit'], axis=1, inplace = True)
```
executed in 31ms, finished 08:10:04 2021-07-11

Scheme managment

In [8]:
```
df.scheme_management.value_counts()
```
executed in 13ms, finished 08:10:04 2021-07-11

Out[8]:
```
VWC                 36793
WUG                  5206
Water authority      3153
WUA                  2883
Water Board          2748
Parastatal           1680
Private operator     1063
Company              1061
Other                 766
SWC                    97
Trust                  72
None                    1
Name: scheme_management, dtype: int64
```

In [9]:
```
# Replace NaNs in 'scheme_managment' by 'other'
df['scheme_management'] = df['scheme_management'].replace(np.nan,'other')
```
executed in 14ms, finished 08:10:04 2021-07-11

We might need this feature as it might give a better glance at which organization is responsible for the managment of a water well project scheme.

Installer

In [10]:
```
# Replace NaNs in 'installer' by 'other'
df['installer'] = df['installer'].replace(np.nan,'other')
```
executed in 14ms, finished 08:10:04 2021-07-11

In [ ]:

In [ ]:

1.4.3 Dealing with date_recorded

Tanzania has a rainy/wet season from December to May and a dry season from July to October. But as seen from the data not all the wells recieve their water source from rainfall so the season may not be of importance to us. Also the date recorded doesn't really signify age of the well so we are going to drop it all together.

In [11]:

```
df.drop(columns=['date_recorded'], axis=1, inplace = True)
```

executed in 31ms, finished 08:10:04 2021-07-11

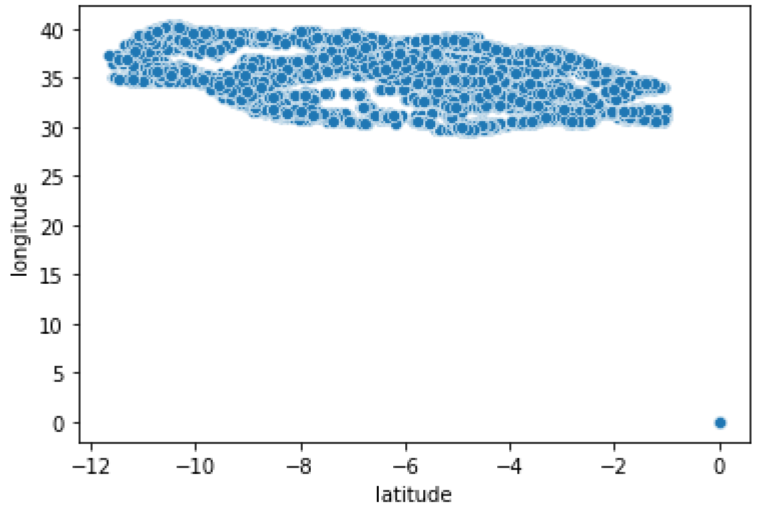
1.4.4 Dealing with outliers

Lets take a look at the location of those wells on the map and explore for any misplaced data or anything that looks weird.

In [12]:

```
sns.scatterplot('latitude', 'longitude', data=df);
```

executed in 263ms, finished 08:10:05 2021-07-11



Looking at the scatter plot of the locational coordinates, we notice an outlier with a 0' 0' latitude and longitude which really doesn't make sense since these points are far off Tanzania. In this next step lets see how many of our data have these coordinates and drop them accordingly.

In [13]:
```
test = df.loc[df['longitude']==0, 'latitude'].value_counts()
test
```
executed in 14ms, finished 08:10:05 2021-07-11

Out[13]:
```
-2.000000e-08    1812
Name: latitude, dtype: int64
```

In [14]:
```
to_drop = df.loc[(df['longitude']==0) & (df['latitude'] == -2.000000e-08)]
to_drop.shape
```
executed in 14ms, finished 08:10:05 2021-07-11

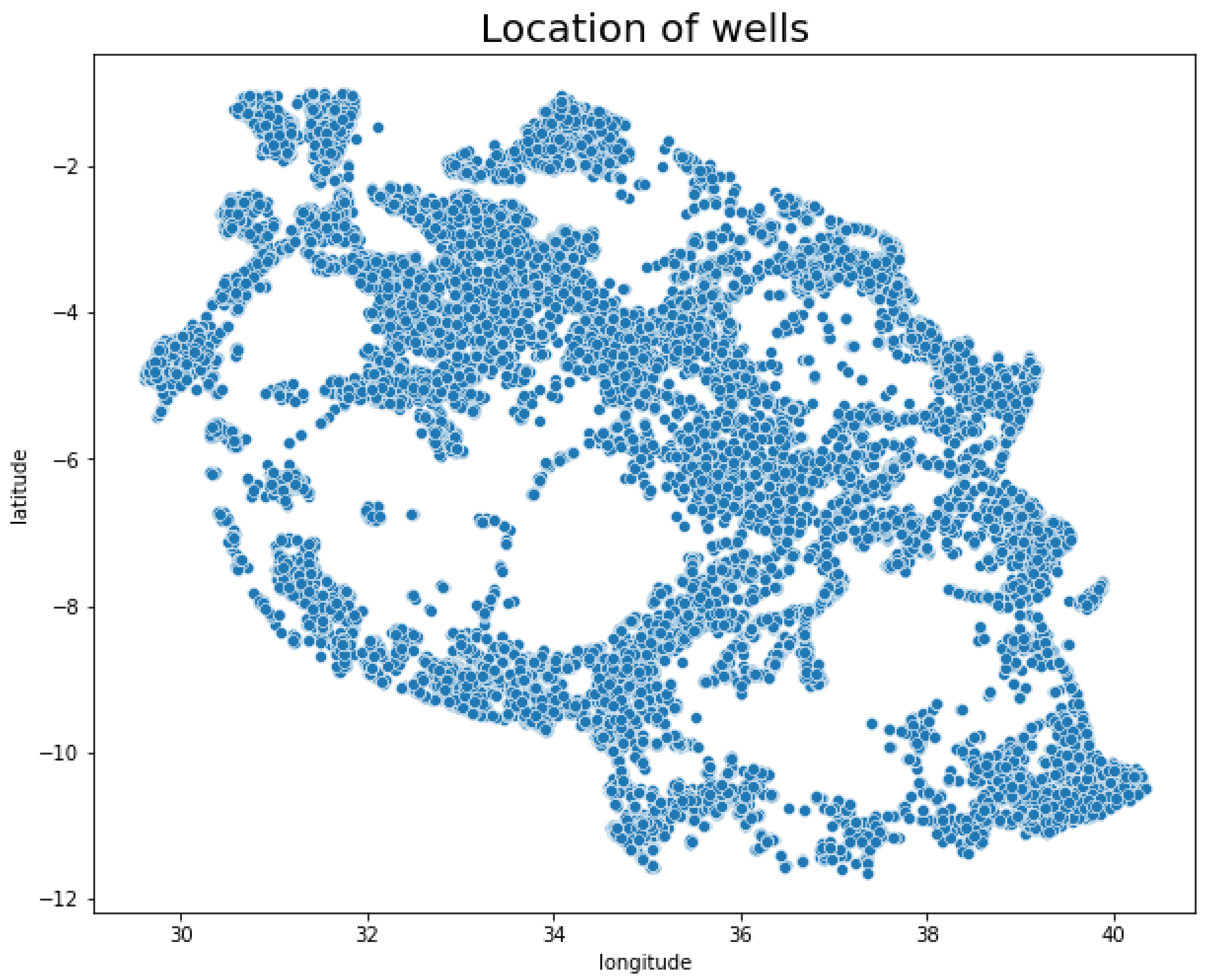
Out[14]:
```
(1812, 34)
```

In [15]:
```
df.drop(to_drop.index, axis=0, inplace=True)
```
executed in 31ms, finished 08:10:05 2021-07-11

In [16]:

```
#plot longitude and latitude of Tanzania
fig = plt.figure(figsize=(10,8))
sns.scatterplot('longitude', 'latitude',  data=df)
plt.title('Location of wells', fontsize=20);
```

executed in 268ms, finished 08:10:05 2021-07-11



Now this looks much better. Check one last time to see if we have any missing values.

In [17]:
```
df.info()
```
executed in 77ms, finished 08:10:05 2021-07-11

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 57588 entries, 0 to 59399
Data columns (total 34 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   id                     57588 non-null  int64  
 1   amount_tsh             57588 non-null  float64
 2   funder                 57588 non-null  object 
 3   gps_height             57588 non-null  int64  
 4   installer              57588 non-null  object 
 5   longitude              57588 non-null  float64
 6   latitude               57588 non-null  float64
 7   wpt_name               57588 non-null  object 
 8   basin                  57588 non-null  object 
 9   region                 57588 non-null  object 
 10  region_code            57588 non-null  int64  
 11  district_code          57588 non-null  int64  
 12  lga                    57588 non-null  object 
 13  ward                   57588 non-null  object 
 14  population             57588 non-null  int64  
 15  recorded_by            57588 non-null  object 
 16  scheme_management      57588 non-null  object 
 17  construction_year      57588 non-null  int64  
 18  extraction_type        57588 non-null  object 
 19  extraction_type_group  57588 non-null  object 
 20  extraction_type_class  57588 non-null  object 
 21  management             57588 non-null  object 
 22  management_group       57588 non-null  object 
 23  payment                57588 non-null  object 
 24  payment_type           57588 non-null  object 
 25  water_quality          57588 non-null  object 
 26  quality_group          57588 non-null  object 
 27  quantity               57588 non-null  object 
 28  quantity_group         57588 non-null  object 
 29  source                 57588 non-null  object 
 30  source_type            57588 non-null  object 
 31  source_class           57588 non-null  object 
 32  waterpoint_type        57588 non-null  object 
 33  waterpoint_type_group  57588 non-null  object 
```

```
dtypes: float64(3), int64(6), object(25)
memory usage: 15.4+ MB
```

1.5 Explore

Now that our data is clean we will move on to the next step and merge the two tables to their corresponding ids inorder to label them according to their functionality.

1.5.1 Merging Labels to the well ids

In [18]:

```
# Read csv file
df1 = pd.read_csv('training_set_labels.csv')
df1.head()
```

executed in 29ms, finished 08:10:05 2021-07-11

Out[18]:

	id	status_group
0	69572	functional
1	8776	functional
2	34310	functional
3	67743	non functional
4	19728	functional

In [19]:

```
# merge the two tables
df2 = pd.merge(df, df1, how = 'inner', left_on = ['id'], right_on = ['id'])
df2.shape
```

executed in 139ms, finished 08:10:05 2021-07-11

Out[19]:

```
(57588, 35)
```

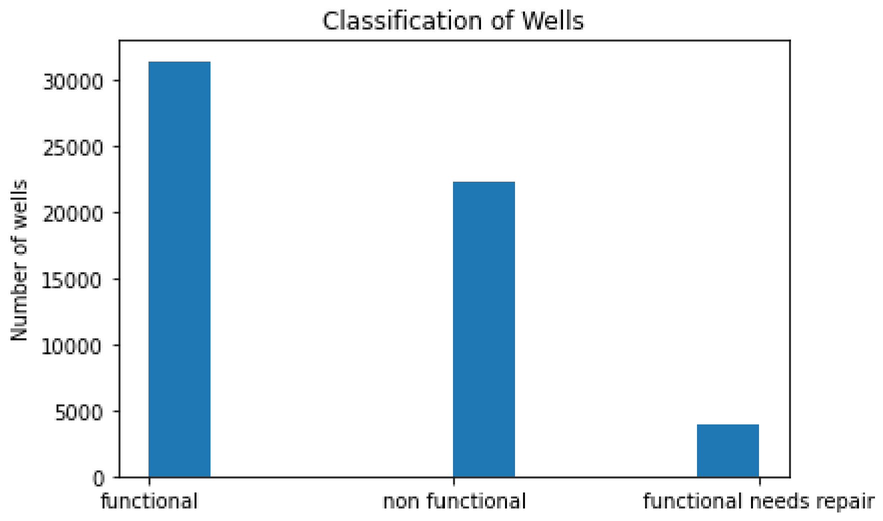
In [20]:
```
df2.status_group.value_counts()
```
executed in 15ms, finished 08:10:05 2021-07-11

Out[20]:
```
functional                 31389
non functional             22268
functional needs repair     3931
Name: status_group, dtype: int64
```

1.5.2 Visualization of features in relation to functionality

In [21]:
```
#plot number of wells according to functionality
plt.hist('status_group', data=df2);
plt.title('Classification of Wells')
plt.ylabel('Number of wells')
```
executed in 141ms, finished 08:10:06 2021-07-11

Out[21]:
```
Text(0, 0.5, 'Number of wells')
```



In [22]:
```
df2.status_group.value_counts(normalize=True)
```
executed in 15ms, finished 08:10:06 2021-07-11

Out[22]:
```
functional                 0.545061
non functional             0.386678
functional needs repair    0.068261
Name: status_group, dtype: float64
```

We do see a class inbalance in the status group with 54.5% functional, 38.67% non functional, 6.82% functional needs repair.

In [23]:
```
# plot wells on map with respect to water point height
plt.figure(figsize=(12,10))

plt.scatter(x='longitude', y='latitude', c='gps_height' , data=df2, s=10, cmap='twilight')
plt.colorbar().set_label('GPS Height')
plt.xlabel('Longitude', fontsize=15)
plt.ylabel('Latitude', fontsize=15)
plt.title('Location of Wells and their GPS height', fontsize=20)

plt.show()
```

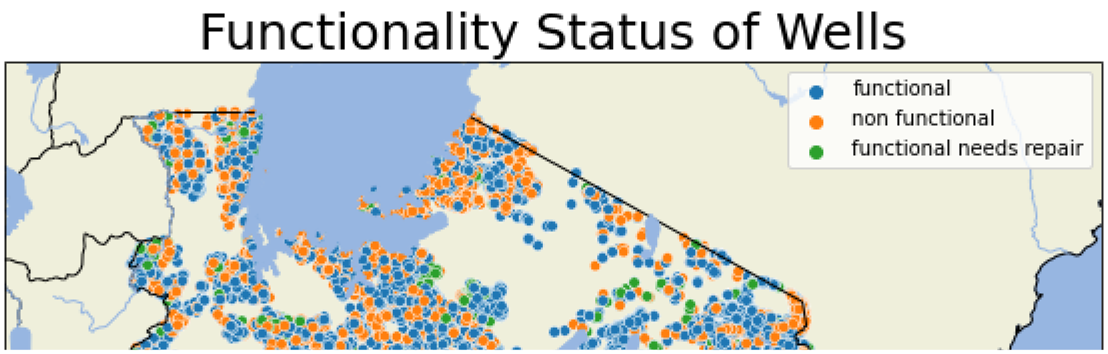
executed in 798ms, finished 08:10:06 2021-07-11

Expectedly, wells will low GPS height of their water point seem to be clustered around 'non functional' or 'functional needs repair' classes. We will take a look at those in the actual map below and try to see the relationship.

In [24]:

```
import cartopy.crs as ccrs
import cartopy.feature as cfeature


plt.figure(figsize=(14,10))

# Creates the map
ca_map = plt.axes(projection=ccrs.PlateCarree())

ca_map.add_feature(cfeature.LAND)
ca_map.add_feature(cfeature.OCEAN)
ca_map.add_feature(cfeature.COASTLINE)
ca_map.add_feature(cfeature.BORDERS, linestyle='-')
ca_map.add_feature(cfeature.LAKES)
ca_map.add_feature(cfeature.RIVERS, linestyle='-' )
# ca_map.add_feature(cfeature.STATES.with_scale('10m'))


# Plots the data onto map
sns.scatterplot(df2['longitude'], df2['latitude'],
            s=30,
            hue=df2['status_group'],
            transform=ccrs.PlateCarree())

# Plot labels
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)
plt.title('Functionality Status of Wells', fontsize=25)
plt.legend()
plt.show()
```

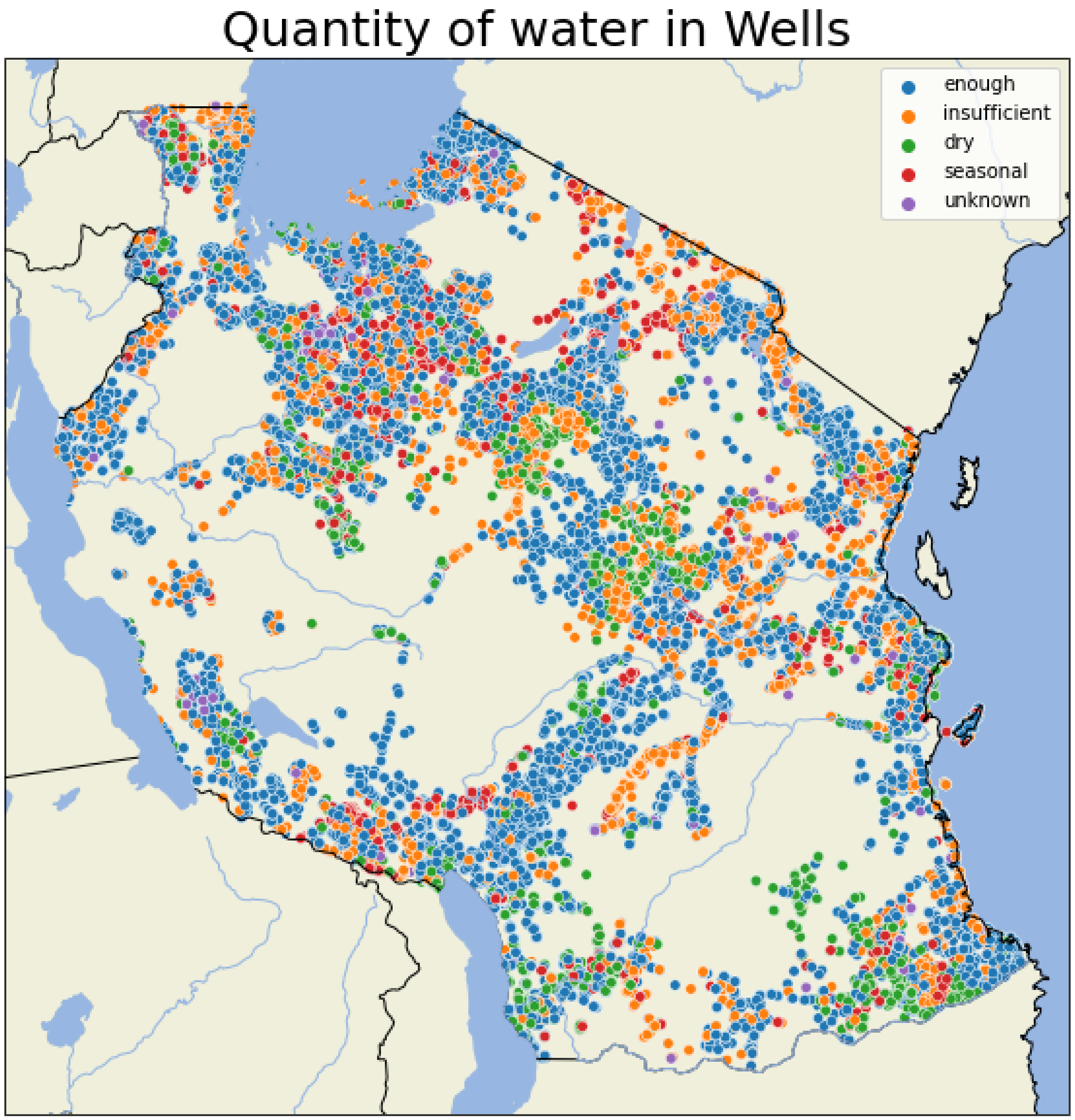
executed in 7.44s, finished 08:10:14 2021-07-11


Functionality Status of Wells
functional
non functional
functional needs repair

In [25]:

```python
import cartopy.crs as ccrs
import cartopy.feature as cfeature


plt.figure(figsize=(14,10))

# Creates the map
ca_map = plt.axes(projection=ccrs.PlateCarree())

ca_map.add_feature(cfeature.LAND)
ca_map.add_feature(cfeature.OCEAN)
ca_map.add_feature(cfeature.COASTLINE)
ca_map.add_feature(cfeature.BORDERS, linestyle='-')
ca_map.add_feature(cfeature.LAKES)
ca_map.add_feature(cfeature.RIVERS)
# ca_map.add_feature(cfeature.STATES.with_scale('10m'))


# Plots the data onto map
sns.scatterplot(df2['longitude'], df2['latitude'],
            s=30,
            hue=df2['quantity'],
            transform=ccrs.PlateCarree())

# Plot labels
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)
plt.title('Quantity of water in Wells', fontsize=25)
plt.legend()
plt.show()
```

executed in 1.65s, finished 08:10:15 2021-07-11


Quantity of water in Wells
● enough
● insufficient
● dry
● seasonal
● unknown

In [26]:
```
df.amount_tsh.value_counts(normalize=True)
```
executed in 14ms, finished 08:10:15 2021-07-11

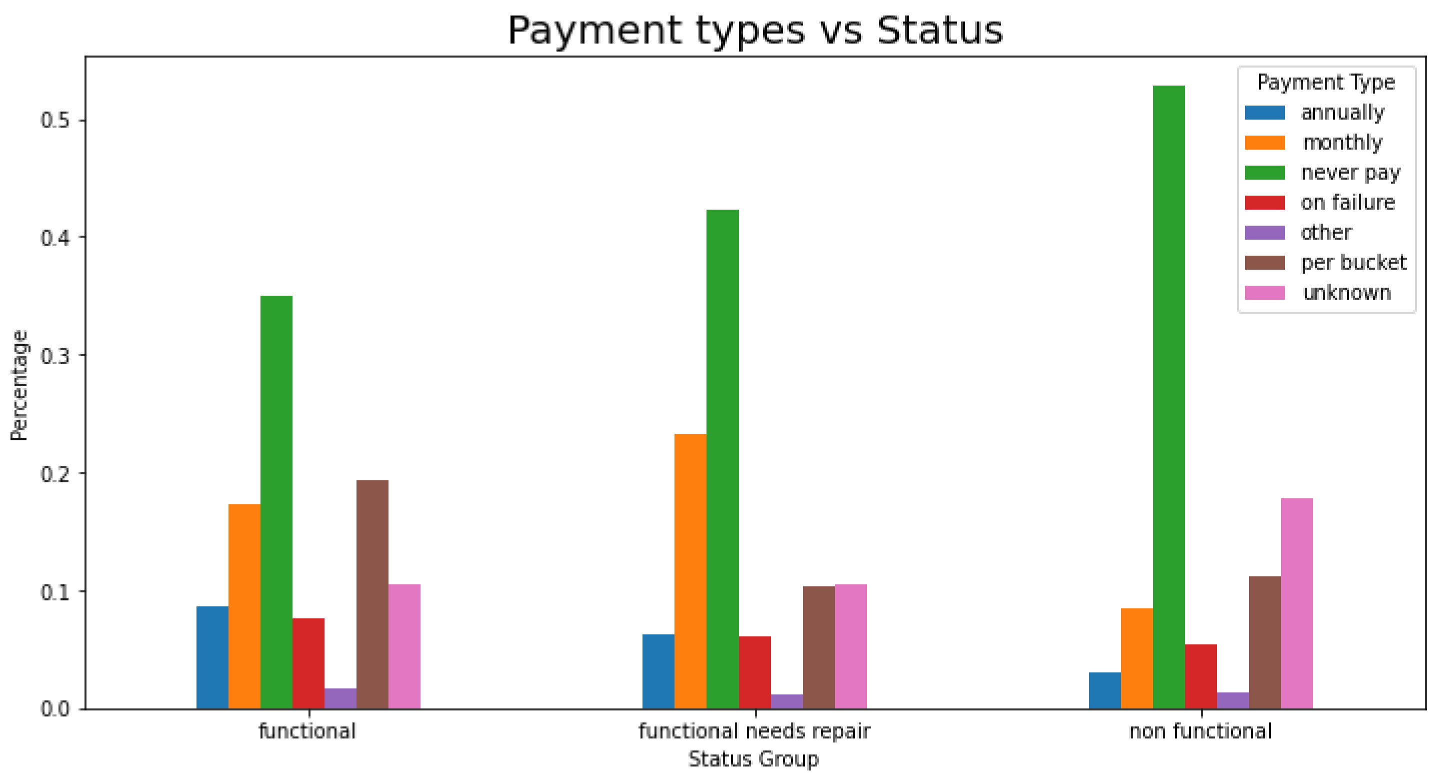
Out[26]:
```
0.0         0.691585
500.0       0.053865
50.0        0.042926
1000.0      0.025839
20.0        0.025405
              ...
8500.0      0.000017
6300.0      0.000017
220.0       0.000017
138000.0    0.000017
12.0        0.000017
Name: amount_tsh, Length: 98, dtype: float64
```

Majority of the wells (69%) have a total static head of 0, which could signify that they are dry. Since this data was recorded at a specific date though, this could mean that those wells are non functional or possibly seasonal and happened to be dry at the time of inspection.

In [27]:
```
payment = df2.groupby('status_group')['payment_type'].value_counts(normalize = True).unstack()
```
executed in 30ms, finished 08:10:15 2021-07-11

In [28]:
```
payment.plot.bar(figsize = (12, 6))
plt.xticks(rotation = 0)
plt.title('Payment types vs Status', fontsize=20)
plt.legend(title = 'Payment Type')
plt.xlabel('Status Group')
plt.ylabel('Percentage')
```
executed in 286ms, finished 08:10:16 2021-07-11

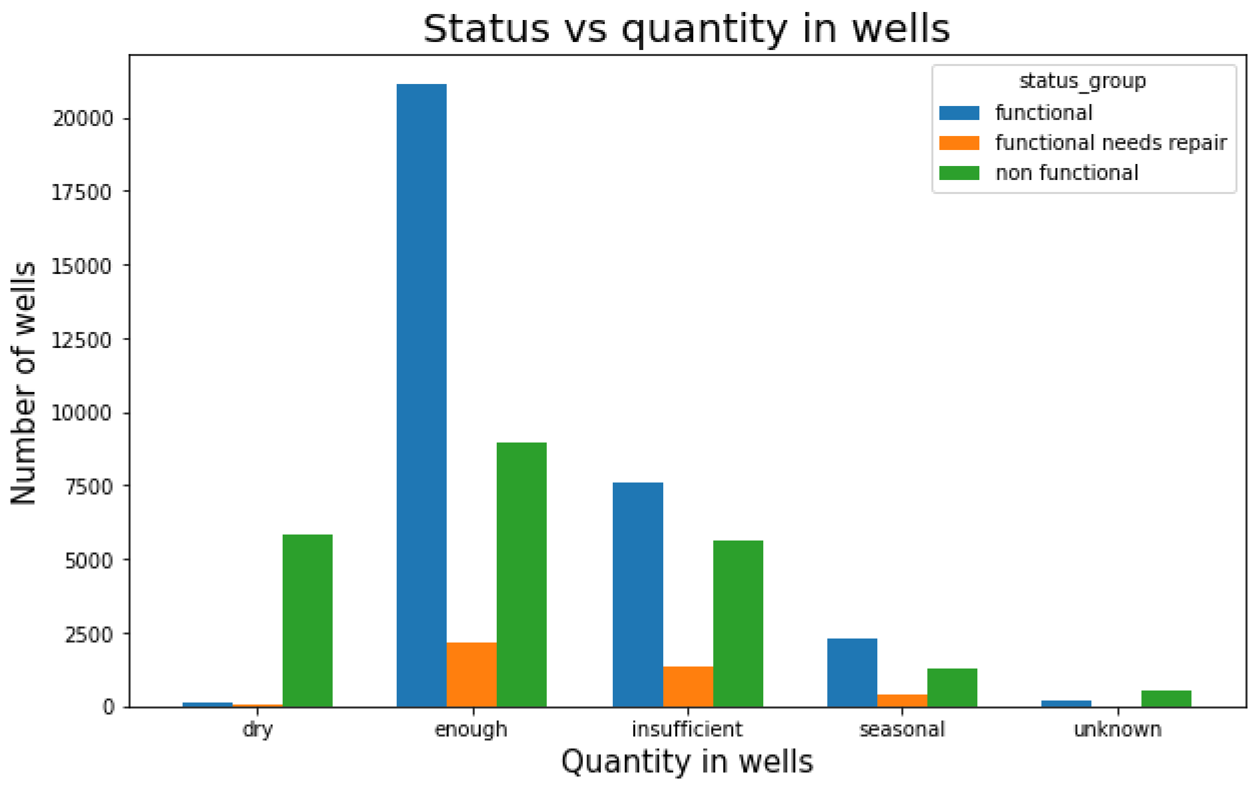
Out[28]:
```
Text(0, 0.5, 'Percentage')
```



Let's keep plotting visualizations of different features with respect to status group of wells to understand the trend of the functionality of the those wells

In [29]:
```
quantity_df = df2.groupby('quantity')['status_group'].value_counts().unstack()
```
executed in 29ms, finished 08:10:16 2021-07-11

In [30]:
```
quantity_df.plot.bar(figsize = (10, 6), width=0.7)
plt.title('Status vs quantity in wells', fontsize=20)
plt.xlabel('Quantity in wells', fontsize=15)
plt.ylabel('Number of wells', fontsize=15)
plt.xticks(rotation = 0);
```
executed in 283ms, finished 08:10:16 2021-07-11



In [31]:
```
df2.quantity.value_counts(normalize=True)
```
executed in 12ms, finished 08:10:16 2021-07-11

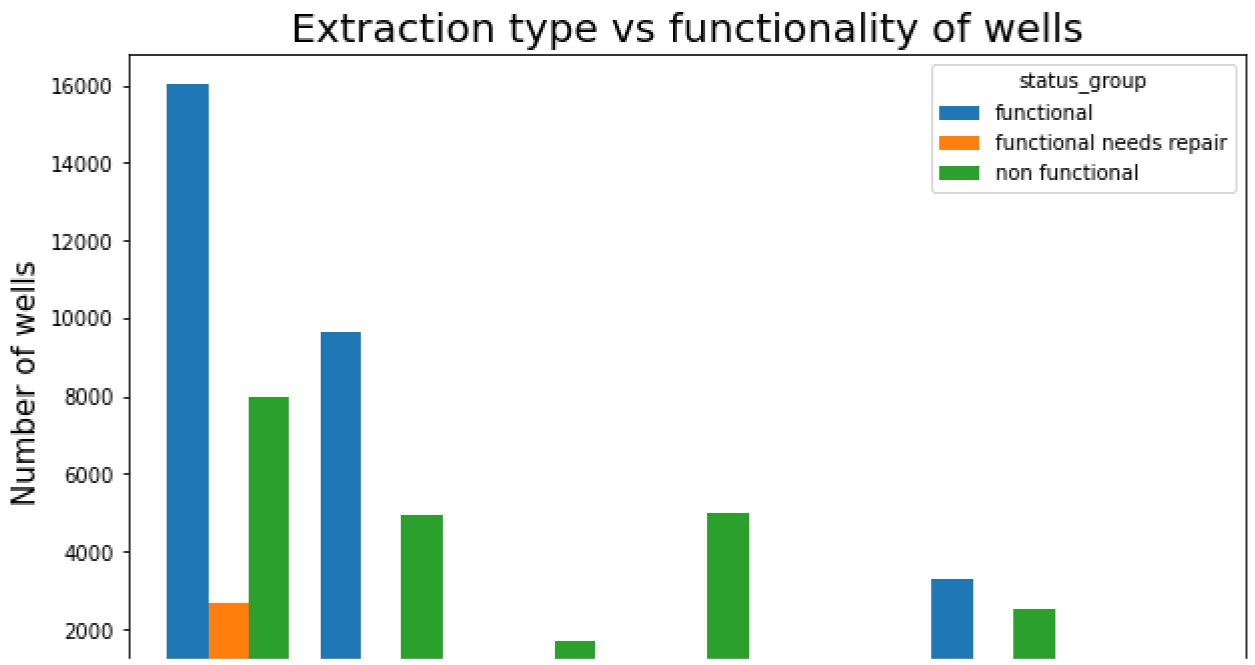
Out[31]:
```
enough          0.560186
insufficient    0.252900
dry             0.104015
seasonal        0.069476
unknown         0.013423
Name: quantity, dtype: float64
```

In [32]:
```
quantity_df = df2.groupby('water_quality')['status_group'].value_counts().unstack()
```
executed in 30ms, finished 08:10:16 2021-07-11

In [33]:
```
quantity_df.plot.bar(figsize = (12, 6), width=0.7)
plt.title('Status vs Quality of wells')
plt.xlabel('Quality of wells')
plt.ylabel('Number of wells')
plt.xticks(rotation = 0);
```
executed in 253ms, finished 08:10:16 2021-07-11

In [34]:
```
quantity_df = df2.groupby('extraction_type_class')['status_group'].value_counts().unstack()
```
executed in 29ms, finished 08:10:16 2021-07-11

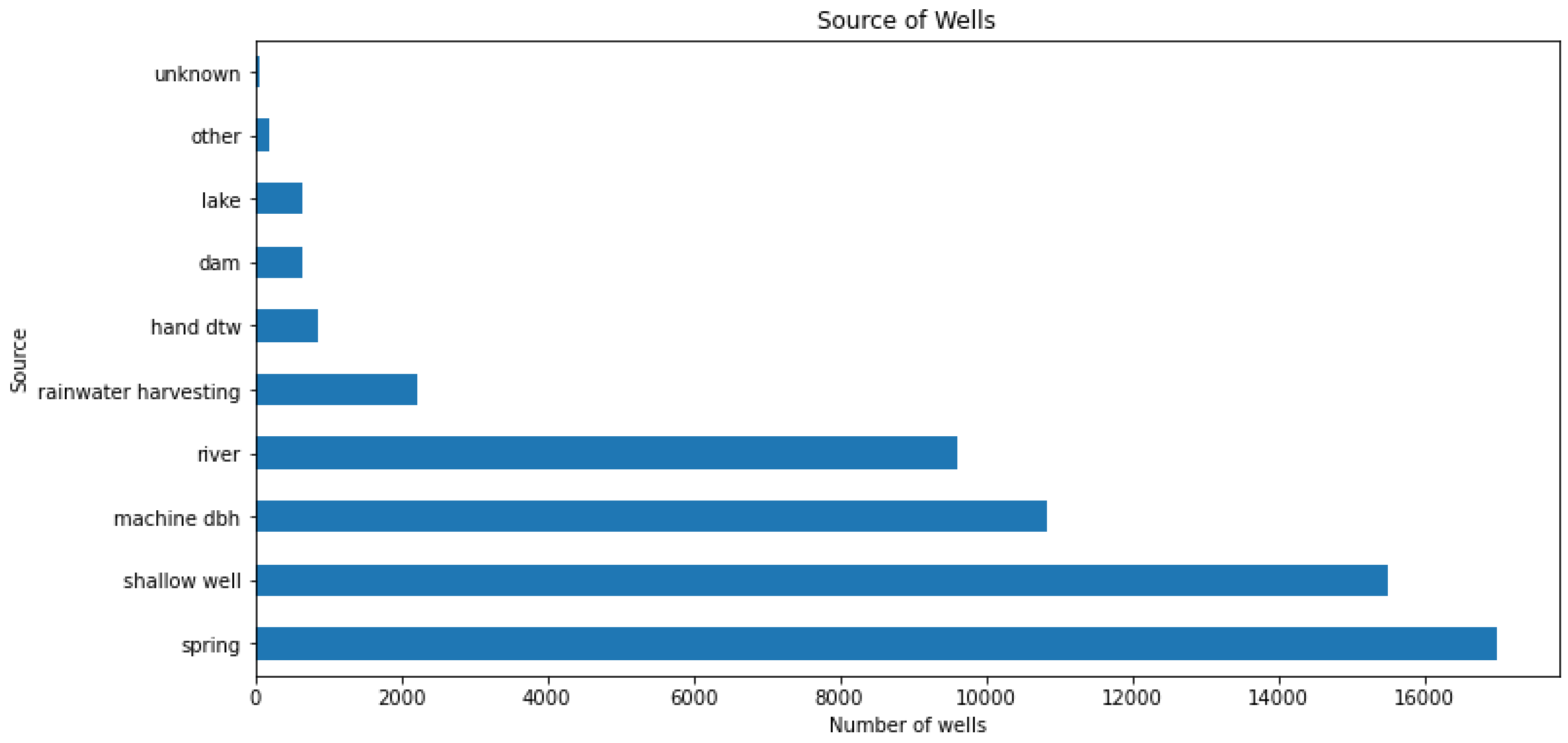
In [35]:
```
quantity_df.plot.bar(figsize = (10, 6), width=0.8)

plt.title('Extraction type vs functionality of wells', fontsize=(20))
plt.xlabel('Extraction Type', fontsize=15)
plt.ylabel('Number of wells', fontsize=15)
plt.xticks(rotation = 0);
```
executed in 252ms, finished 08:10:17 2021-07-11



In [36]:
```
df2.source.value_counts().plot.barh(figsize = (12, 6))
plt.title('Source of Wells')
plt.ylabel('Source')
plt.xlabel('Number of wells')
```

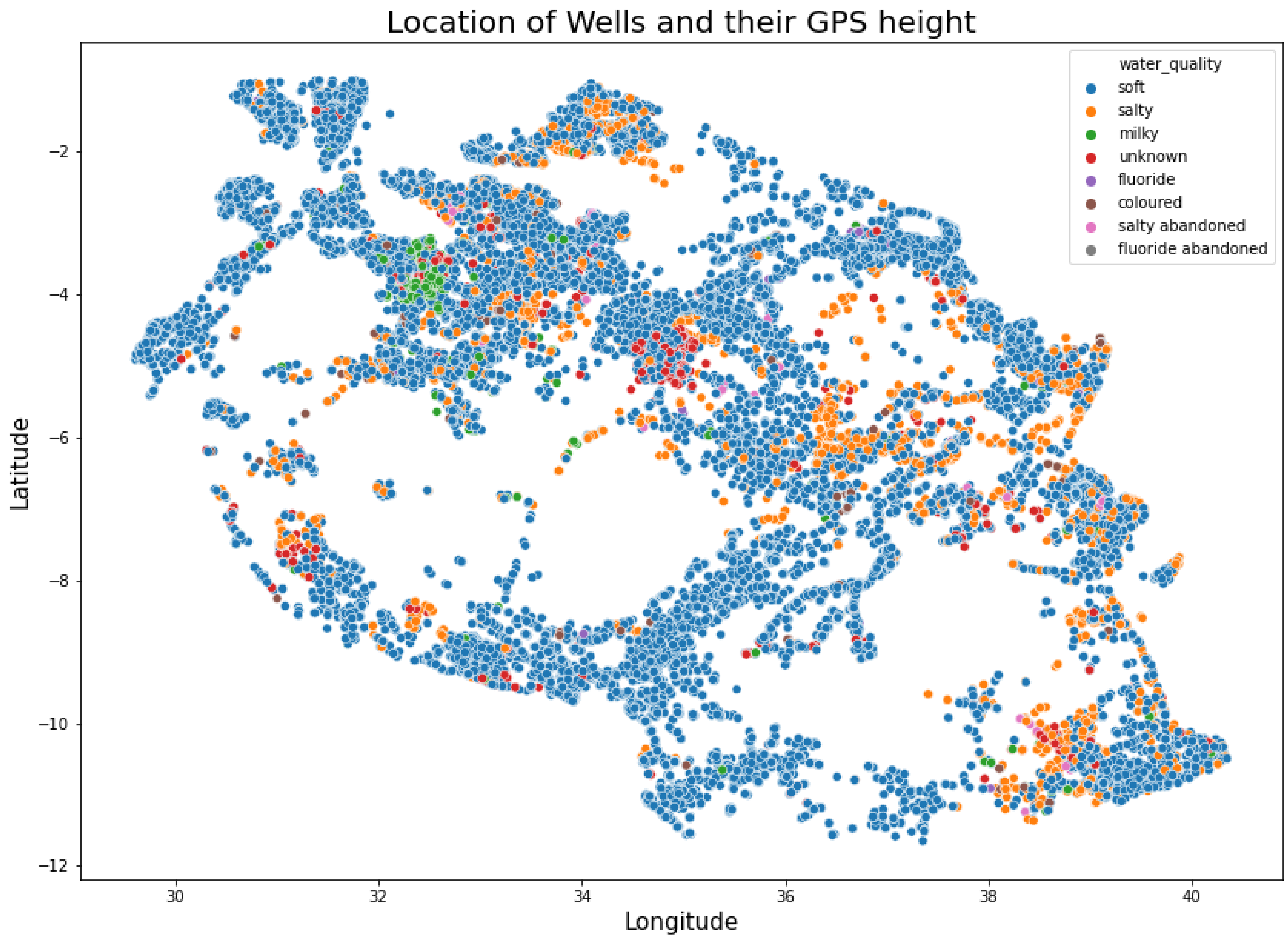
executed in 218ms, finished 08:10:17 2021-07-11

Out[36]:
```
Text(0.5, 0, 'Number of wells')
```



In [37]:
```
plt.figure(figsize=(14,10))
sns.scatterplot(x='longitude', y='latitude', hue='water_quality' , data=df2)

plt.xlabel('Longitude', fontsize=15)
plt.ylabel('Latitude', fontsize=15)
plt.title('Location of Wells and their GPS height', fontsize=20)

plt.show()
```

executed in 1.44s, finished 08:10:18 2021-07-11



1.5.3 Construction year

Construction year is an important feature for our modeling but we do have a large range of years which will be a huge number of columns when creating dummies. We will categorize the years in such a way that they will be parts of a 10 year time period to reduce the classes of this feature. Another point worth mentioning is that more than 30% of our data has no record of the construction year. Since this is an important feature and losing that much amount of data is not worth the risk we will take the option of replacing those values with random choice from the rest of the data.

In [38]:
```
df2.construction_year.value_counts(normalize=True)
```
executed in 14ms, finished 08:10:18 2021-07-11

Out[38]:
```
0       0.328141
2010    0.045930
2008    0.045374
2009    0.043985
2000    0.036310
2007    0.027558
2006    0.025544
2003    0.022331
2011    0.021810
2004    0.019501
2012    0.018823
2002    0.018667
1978    0.018007
1995    0.017608
2005    0.017556
1999    0.017000
1998    0.016774
1990    0.016566
1985    0.016410
1980    0.014083
1996    0.014083
1984    0.013527
1982    0.012919
1994    0.012815
1972    0.012294
1974    0.011739
1997    0.011183
1992    0.011113
1993    0.010558
2001    0.009377
1988    0.009047
1983    0.008474
1975    0.007588
1986    0.007536
1976    0.007189
1970    0.007137
1991    0.005626
1989    0.005487
1987    0.005244
```

```
1981    0.004133
1977    0.003508
1979    0.003334
1973    0.003195
2013    0.003056
1971    0.002518
1960    0.001771
1967    0.001528
1963    0.001476
1968    0.001337
1969    0.001025
1964    0.000695
1962    0.000521
1961    0.000365
1965    0.000330
1966    0.000295
Name: construction_year, dtype: float64
```

In [39]:

```
df2['construction_year'] = df2['construction_year'].replace(0, np.nan)
```

executed in 14ms, finished 08:10:18 2021-07-11

In [40]:

```
#replace missing values with random choice
s = df2.construction_year.value_counts(normalize=True)
df2['const_year'] = df2['construction_year']
df2.loc[df2.construction_year.isna(), 'const_year'] = np.random.choice(s.index, p=s.values, size=df2.constru
```

executed in 46ms, finished 08:10:18 2021-07-11

In [41]:
```
df2['const_year'].value_counts(normalize = True)
```
executed in 19ms, finished 08:10:18 2021-07-11

Out[41]:
```
2010.0    0.069077
2008.0    0.067549
2009.0    0.064823
2000.0    0.054716
2007.0    0.041311
2006.0    0.038428
2003.0    0.032906
2011.0    0.032472
2004.0    0.028756
2002.0    0.028079
2012.0    0.027870
1978.0    0.027106
2005.0    0.026117
1995.0    0.025682
1998.0    0.025526
1999.0    0.025023
1985.0    0.024693
1990.0    0.024363
1980.0    0.020768
```

In [42]:
```
df2.drop('construction_year', axis=1, inplace=True)
```
executed in 30ms, finished 08:10:18 2021-07-11

Now that we will have replaced the missing values, next step would be to create bins for the years. The typical life expectancy of a water well is supposedly 65-100 years, and the life expectency of a water well pump is 10-15 years. We will bin the construction year column in such a way and then take a look at the visualizations.

In [43]:
```
norm = df2.const_year.value_counts()
```
executed in 13ms, finished 08:10:18 2021-07-11

In [44]:
```
norm = df2.groupby('const_year')['status_group'].value_counts().unstack()
```
executed in 29ms, finished 08:10:18 2021-07-11

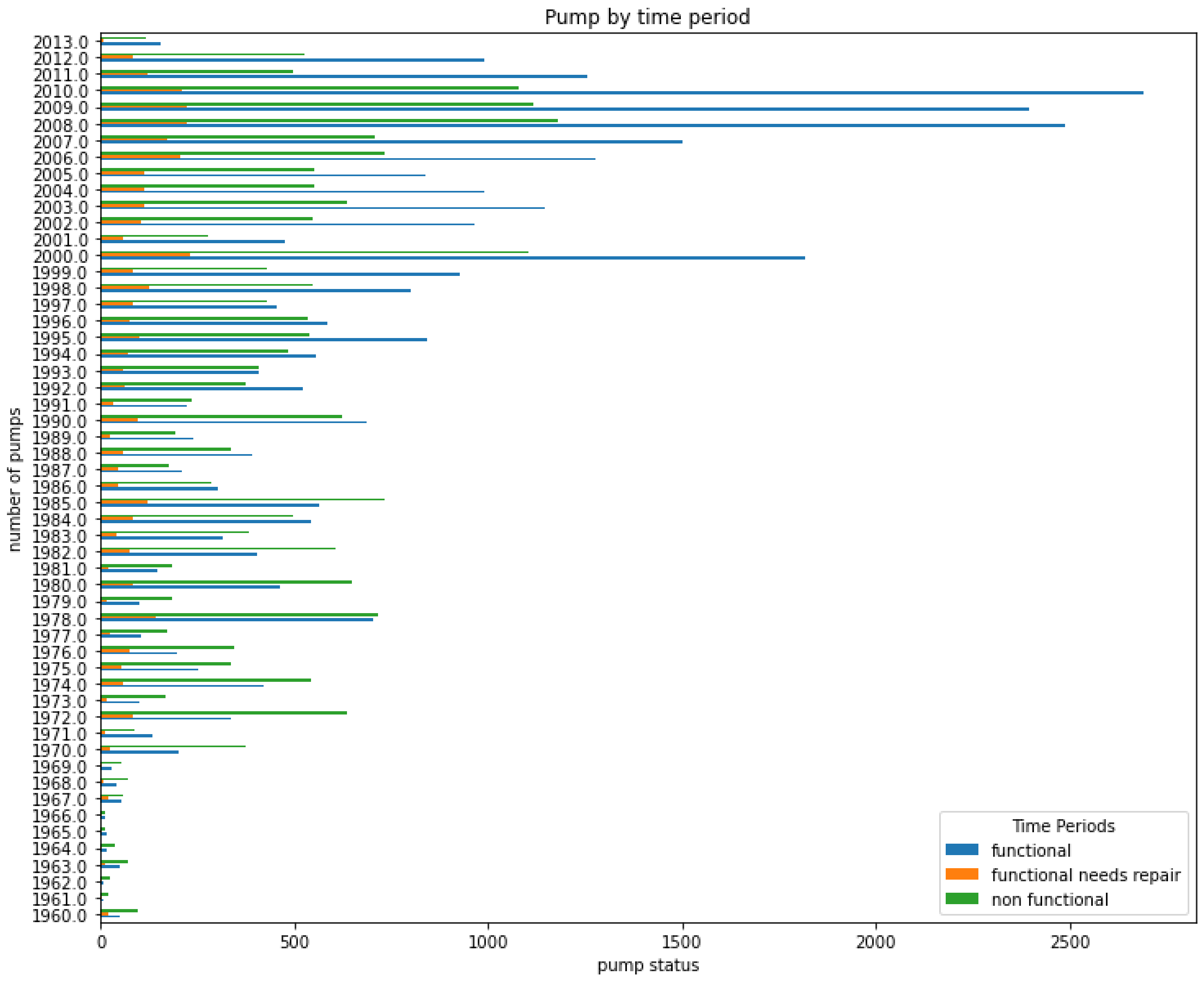
In [45]:

```
norm.plot.barh(figsize = (12, 10))
plt.title('Pump by time period')
plt.xlabel('pump status')
plt.ylabel('number of pumps')
plt.xticks(rotation = 0)
plt.legend(title = 'Time Periods')
```

executed in 1.50s, finished 08:10:20 2021-07-11

Out[45]:

```
<matplotlib.legend.Legend at 0x1ef8b060fd0>
```


Pump by time period
number of pumps
2013.0
2012.0
2011.0
2010.0
2009.0
2008.0
2007.0
2006.0
2005.0
2004.0
2003.0
2002.0
2001.0
2000.0
1999.0
1998.0
1997.0
1996.0
1995.0
1994.0
1993.0
1992.0
1991.0
1990.0
1989.0
1988.0
1987.0
1986.0
1985.0
1984.0
1983.0
1982.0
1981.0
1980.0
1979.0
1978.0
1977.0
1976.0
1975.0
1974.0
1973.0
1972.0
1971.0
1970.0
1969.0
1968.0
1967.0
1966.0
1965.0
1964.0
1963.0
1962.0
1961.0
1960.0
0
500
1000
1500
2000
2500
pump status
Time Periods
functional
functional needs repair
non functional

In [46]:
```
bins = [1960, 1970, 1980, 1990, 2000, 2010, 2015]
periods = ['1960-1969', '1970-1979', '1980-1989', '1990-1999', '2000-2009', '2010-2015']

df2['construction_periods'] = pd.cut(df2['const_year'], bins ,
                                     labels = periods, right = False)
```
executed in 13ms, finished 08:10:20 2021-07-11

In [47]:
```
df2.head()
```
executed in 28ms, finished 08:10:20 2021-07-11

Out[47]:

	id	amount_tsh	funder	gps_height	installer	longitude	latitude	wpt_name	basin	region	...	quantity	quantity_
0	69572	6000.0	Roman	1390	Roman	34.938093	-9.856322	none	Lake Nyasa	Iringa	...	enough	e
1	8776	0.0	Grumeti	1399	GRUMETI	34.698766	-2.147466	Zahanati	Lake Victoria	Mara	...	insufficient	insu
2	34310	25.0	Lottery Club	686	World vision	37.460664	-3.821329	Kwa Mahundi	Pangani	Manyara	...	enough	e
3	67743	0.0	Unicef	263	UNICEF	38.486161	-11.155298	Zahanati Ya Nanyumbu	Ruvuma / Southern Coast	Mtwara	...	dry	
4	19728	0.0	Action In A	0	Artisan	31.130847	-1.825359	Shuleni	Lake Victoria	Kagera	...	seasonal	se

5 rows × 36 columns

In [48]:

```
df2.construction_periods.value_counts()
```

executed in 15ms, finished 08:10:20 2021-07-11

Out[48]:

```
2000-2009    22849
1990-1999    11384
1980-1989     8223
2010-2015     7730
1970-1979     6603
1960-1969      799
Name: construction_periods, dtype: int64
```

In [49]:

```
time_periods = df2.groupby('status_group')['construction_periods'].value_counts().unstack()
time_periods.head()
```

executed in 31ms, finished 08:10:20 2021-07-11

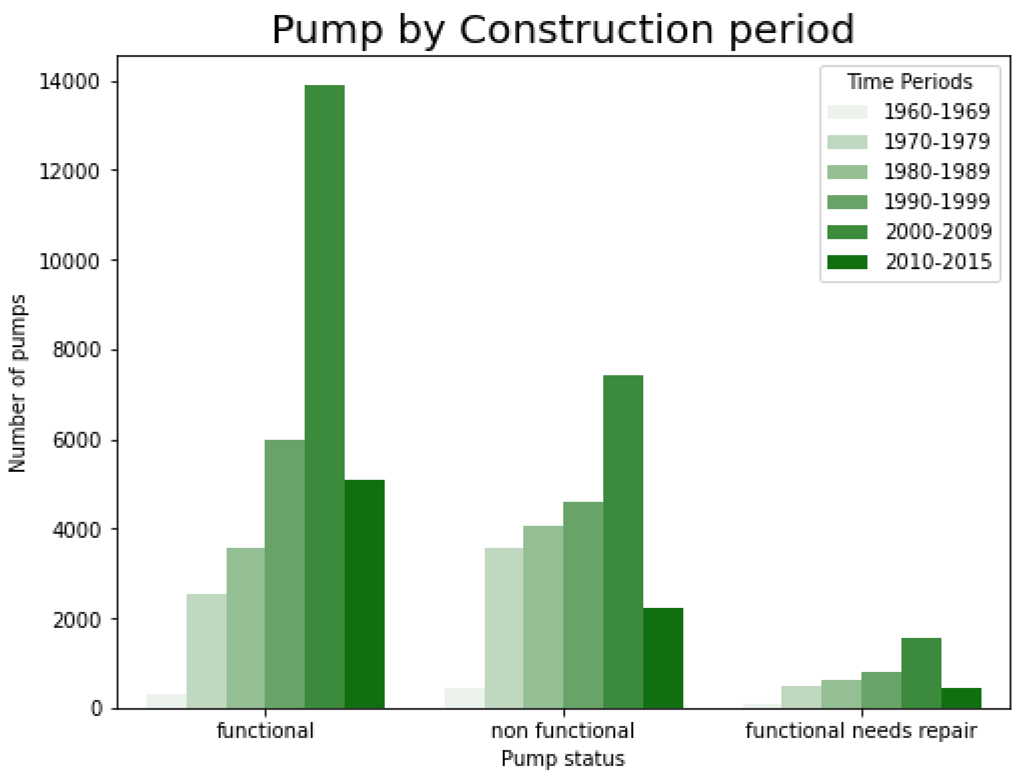
Out[49]:

construction_periods	1960-1969	1970-1979	1980-1989	1990-1999	2000-2009	2010-2015
status_group						
functional	283	2542	3579	6002	13893	5090
functional needs repair	69	503	600	784	1553	422
non functional	447	3558	4044	4598	7403	2218

In [50]:
```python
# sns.set_style('darkgrid')
plt.figure(figsize=(8,6))
sns.countplot(data=df2, x='status_group', hue='construction_periods', color='Green')

plt.title('Pump by Construction period', fontsize=20)
plt.xlabel('Pump status')
plt.ylabel('Number of pumps')
plt.xticks(rotation = 0)
plt.legend(title = 'Time Periods')
```

executed in 311ms, finished 08:10:20 2021-07-11

Out[50]:
```
<matplotlib.legend.Legend at 0x1ef8b0429a0>
```



In [51]:
```
df2.population.value_counts(normalize=True)
```
executed in 14ms, finished 08:10:20 2021-07-11

Out[51]:
```
0       0.339810
1       0.121987
200     0.033688
150     0.032854
250     0.029190
          ...   
3241    0.000017
1960    0.000017
1685    0.000017
2248    0.000017
1439    0.000017
Name: population, Length: 1049, dtype: float64
```

The population column has more than 33% of its data to be 0 values and 12% of only 1 value. This information doesn't seem correct at all, better drop that feature. Some of the features are repetetive so we will just use the one with the most data and drop all the other similar features.

In [52]:
```
df2.drop(columns=['wpt_name', 'recorded_by', 'lga', 'ward',
                  'extraction_type', 'extraction_type_group', 'population',
                  'management', 'payment_type', 'water_quality', 'source',
                  'source_class', 'waterpoint_type', 'quantity_group',
                  'region_code', 'const_year', 'district_code'], axis=1, inplace=True)
```
executed in 15ms, finished 08:10:20 2021-07-11

In [53]:
```
df2.shape
```
executed in 15ms, finished 08:10:20 2021-07-11

Out[53]:
```
(57588, 19)
```

In [54]:
```
# Replace target values - there are three classes
df2 = df2.replace({'status_group': {'functional' : 1,
                                    'non functional' : 0,
                                    'functional needs repair' : 2}})
# Check to see that it worked
df2.iloc[15:20]
```
executed in 62ms, finished 08:10:21 2021-07-11

Out[54]:

	id	amount_tsh	funder	gps_height	installer	longitude	latitude	basin	region	scheme_management	extraction_ty
15	61848	0.0	Rudep	1645	DWE	31.444121	-8.274962	Lake Tanganyika	Rukwa	VWC	h
16	48451	500.0	Unicef	1703	DWE	34.642439	-9.106185	Rufiji	Iringa	WUA	
17	58155	0.0	Unicef	1656	DWE	34.569266	-9.085515	Rufiji	Iringa	WUA	
18	34169	0.0	Hesawa	1162	DWE	32.920154	-1.947868	Lake Victoria	Mwanza	other	
19	18274	500.0	Danida	1763	Danid	34.508967	-9.894412	Lake Nyasa	Iringa	VWC	

1.5.4 Creating Dummies

In [55]:
```
target = ['status_group']

categorical = ['funder', 'installer', 'basin', 'region',
               'scheme_management', 'extraction_type_class', 'management_group', 'payment',
               'quality_group', 'quantity', 'source_type', 'waterpoint_type_group',
               'construction_periods']


continuous = ['amount_tsh', 'gps_height', 'longitude', 'latitude']
```
executed in 14ms, finished 08:10:21 2021-07-11

In [56]:
```
# print number of classes in each category
for col in categorical:
    print(col, df2[col].value_counts().count())
```
executed in 92ms, finished 08:10:21 2021-07-11

```
funder 1859
installer 2114
basin 9
region 21
scheme_management 13
extraction_type_class 7
management_group 5
payment 7
quality_group 6
quantity 5
source_type 7
waterpoint_type_group 6
construction_periods 6
```

In [57]:
```
categories_to_remove = {}
for col in categorical:
    df_tmp = pd.DataFrame(df2[col].value_counts(normalize=True))
    other_categories = list(df_tmp.loc[df_tmp[col]<0.01].index)
    df2[col] = df2[col].map(lambda x: 'other' if x in other_categories else x)
    categories_to_remove[col] = other_categories
```
executed in 1.46s, finished 08:10:22 2021-07-11

In [58]:
```
# checking if our features of less than 1% are replaced
for col in categorical:
    print(df2[col].value_counts(normalize=True), '\n', df2[col].value_counts(normalize=True).count(), '\n')
```
executed in 155ms, finished 08:10:22 2021-07-11

```
other                      0.527593
Government Of Tanzania     0.153539
Danida                     0.054074
Hesawa                     0.033236
World Bank                 0.023356
Kkkt                       0.022348
World Vision               0.021254
Rwssp                      0.020612
Unicef                     0.017972
District Council           0.014638
Tasaf                      0.014482
Dhv                        0.014395
Private Individual         0.014309
0                          0.013492
Norad                      0.013284
Germany Republi            0.010592
Tcrs                       0.010454
Ministry Of Water          0.010245
Water                      0.010124
```

In [59]:
```
df2.drop(target , axis=1).columns
```
executed in 15ms, finished 08:10:22 2021-07-11

Out[59]:
```
Index(['id', 'amount_tsh', 'funder', 'gps_height', 'installer', 'longitude',
       'latitude', 'basin', 'region', 'scheme_management',
       'extraction_type_class', 'management_group', 'payment', 'quality_group',
       'quantity', 'source_type', 'waterpoint_type_group',
       'construction_periods'],
      dtype='object')
```

In [60]:
```
df_dummies = pd.get_dummies(df2.drop(target, axis=1))
df_dummies.head().sort_values(by='id')
```
executed in 111ms, finished 08:10:22 2021-07-11

Out[60]:

	id	amount_tsh	gps_height	longitude	latitude	funder_0	funder_Danida	funder_Dhv	funder_District Council	funder_Germany Republi	...
1	8776	0.0	1399	34.698766	-2.147466	0	0	0	0	0	...
4	19728	0.0	0	31.130847	-1.825359	0	0	0	0	0	...
2	34310	25.0	686	37.460664	-3.821329	0	0	0	0	0	...
3	67743	0.0	263	38.486161	-11.155298	0	0	0	0	0	...
0	69572	6000.0	1390	34.938093	-9.856322	0	0	0	0	0	...

5 rows × 119 columns

In [61]:
```
df_dummies.shape
```
executed in 15ms, finished 08:10:22 2021-07-11

Out[61]:
```
(57588, 119)
```

The next step would be to concatinate the Target features with the dummies.

In [62]:
```
df_dummies['status_group'] = df2['status_group'].values
```
executed in 14ms, finished 08:10:22 2021-07-11

In [63]:
```
df_dummies.shape
```
executed in 15ms, finished 08:10:22 2021-07-11

Out[63]:
```
(57588, 120)
```

1.6 Modeling

Here, we will run some Models using the classification algorithms of KNN, Random Forest and XGBoost. First we will run baseline models in each method and then move on to tunning and optimizing those models to increase performance and metric scores. I will use F1-score as my deciding metric, but precision and recall will let us know what values we're having trouble classifying, and where I can improve.

In [64]:
```
# assign variables for features and target
X = df_dummies.drop('status_group', axis = 1)
y = df_dummies['status_group']
```
executed in 30ms, finished 08:10:22 2021-07-11

In [65]:
```
# split into test and train data sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state = 42)
```
executed in 47ms, finished 08:10:22 2021-07-11

In [66]:
```
len(y_test)
```
executed in 14ms, finished 08:10:23 2021-07-11

Out[66]:
```
11518
```

In [67]:
```
from sklearn.metrics import precision_score, recall_score, accuracy_score, f1_score

models = []
def get_metrics(y_test, X_test, model):
    labels = y_test.to_numpy()
    preds = model.predict(X_test)

    metrics = {}
    metrics['accuracy'] = accuracy_score(labels, preds)
    metrics['f1'] = f1_score(labels, preds, average='macro')
    metrics['precision'] = precision_score(labels, preds, average='macro')
    metrics['recall'] = recall_score(labels, preds, average='macro')

    return metrics
```
executed in 13ms, finished 08:10:23 2021-07-11

1.7 KNN

The KNN model is simple to fit, but time-consuming to predict on, especially on this large dataset. It also has relatively few hyperparameters to tune, so it may not improve much.

1.7.1 Baseline Model

In [68]:
```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier()
knn.fit(X_train, y_train);

metrics = get_metrics(y_test, X_test, knn)
metrics['name'] = 'K Nearest Neighbors Baseline'
models.append(metrics)
```
executed in 4.05s, finished 08:10:27 2021-07-11

In [69]:
```
print(classification_report(y_train, knn.predict(X_train),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```
executed in 5.17s, finished 08:10:32 2021-07-11

```
               precision    recall  f1-score   support

nonfunctional       0.61      0.65      0.63     17842
   functional       0.71      0.77      0.74     25088
 needs repair       0.61      0.04      0.07      3140

     accuracy                           0.67     46070
    macro avg       0.65      0.48      0.48     46070
 weighted avg       0.67      0.67      0.65     46070
```

In [70]:
```
print(confusion_matrix(y_test, knn.predict(X_test)))
print(classification_report(y_test, knn.predict(X_test)))
```
executed in 2.72s, finished 08:10:34 2021-07-11

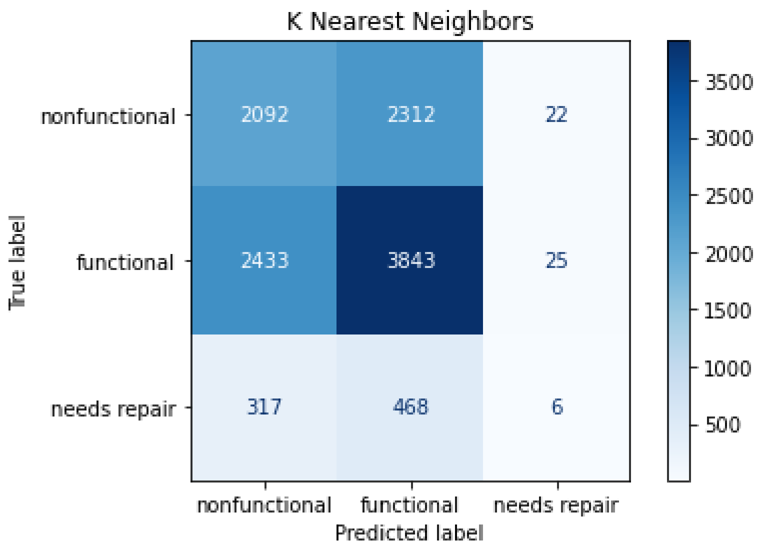
```
[[2092 2312   22]
 [2433 3843   25]
 [ 317  468    6]]
              precision    recall  f1-score   support

           0       0.43      0.47      0.45      4426
           1       0.58      0.61      0.59      6301
           2       0.11      0.01      0.01       791

    accuracy                           0.52     11518
   macro avg       0.38      0.36      0.35     11518
weighted avg       0.49      0.52      0.50     11518
```

1.7.2 Confusion matrix

In [71]:
```
plt.figure(figsize=(6,5))
plot_confusion_matrix(knn, X_test, y_test,
                      display_labels=['nonfunctional', 'functional',
                                      'needs repair'], values_format='d',
                                      cmap=plt.cm.Blues)
plt.title('K Nearest Neighbors')
plt.tight_layout()

plt.show()
```

executed in 1.56s, finished 08:10:36 2021-07-11

```
<Figure size 432x360 with 0 Axes>
```



In this KNN our model was able to capture 47% of the non functional, 61% of the functional, and only 1% of the functional needs repair.

1.7.3 Standardizing

First scale the data. We scale the data after splitting the train and test data to avoid data leakage.

In [72]:
```
# Import StandardScaler
from sklearn.preprocessing import StandardScaler


# Instantiate StandardScaler
scaler = StandardScaler()

# Transform the training and test sets
scaled_data_train = scaler.fit_transform(X_train)
scaled_data_test = scaler.transform(X_test)

# Convert into a DataFrame
scaled_df_train = pd.DataFrame(scaled_data_train, columns=X.columns)
scaled_df_train.head()
```

executed in 173ms, finished 08:10:36 2021-07-11

Out[72]:

	id	amount_tsh	gps_height	longitude	latitude	funder_0	funder_Danida	funder_Dhv	funder_District Council	funder_Germany Republi
0	1.273710	2.393683	1.172595	0.057318	-1.376628	-0.11822	-0.240045	-0.120373	-0.122854	-0.102501
1	-1.251616	-0.104170	1.035817	-0.205919	0.330106	-0.11822	-0.240045	-0.120373	-0.122854	-0.102501
2	-0.368808	-0.104170	0.474307	0.603923	0.441211	-0.11822	-0.240045	-0.120373	-0.122854	-0.102501
3	-1.426384	-0.104170	-0.994257	-1.321809	1.661788	-0.11822	-0.240045	-0.120373	-0.122854	-0.102501
4	0.708950	-0.104170	0.838569	-1.917170	0.502559	-0.11822	-0.240045	-0.120373	-0.122854	-0.102501

5 rows × 119 columns

Now that you've preprocessed the data it's time to train a KNN classifier and validate its accuracy.

In [73]:
```
# Import KNeighborsClassifier
from sklearn.neighbors import KNeighborsClassifier

# Instantiate KNeighborsClassifier
knn = KNeighborsClassifier()
# Fit the classifier
knn.fit(scaled_data_train, y_train);


# Predict on the test set

metrics = get_metrics(y_test, scaled_data_test, knn)
metrics['name'] = 'K Nearest Neighbors scaled'
models.append(metrics)
```
executed in 1m 2.34s, finished 08:11:39 2021-07-11

1.7.4 Optimazing k-value

Let's first create a function to iterate over a range of K-values to find out the best value for the optimum f1-score. Then pass that value for the first round of the GridSearchCV and take note of the result. In the second round of the GridSearchCV, we will try and narrow down the values around the successful ones already found in the first pass.

1.7.5 Grid search CV

In [74]:
```
def find_best_k(X_train, y_train, X_test, y_test, min_k=1, max_k=25):
    best_k = 0
    best_score = 0.0
    for k in range(min_k, max_k+1, 2):
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)
        preds = knn.predict(X_test)
        f1 = f1_score(y_test, preds, average='weighted')
        if f1 > best_score:
            best_k = k
            best_score = f1

    print("Best Value for k: {}".format(best_k))
    print("F1-Score: {}".format(best_score))
```

executed in 13ms, finished 08:11:39 2021-07-11

In [75]:
```
find_best_k(scaled_data_train, y_train, scaled_data_test, y_test)
```

executed in 14m 9s, finished 08:25:48 2021-07-11

```
Best Value for k: 7
F1-Score: 0.7433163911145209
```

In [76]:
```
from sklearn.model_selection import GridSearchCV
```

executed in 14ms, finished 08:25:48 2021-07-11

In [77]:
```
param_grid = {
    'n_neighbors': [1, 5, 9], # default 5
    'weights': ['uniform', 'distance'], # default 'uniform'
    'leaf_size': [30, 40], # default 30
    'p': [1, 2] # default 2
}

grid_search = GridSearchCV(knn, param_grid, cv=3, scoring='f1_macro')
grid_search.fit(scaled_data_train, y_train)
```
executed in 1h 1m 45s, finished 09:27:33 2021-07-11

Out[77]:
```
GridSearchCV(cv=3, error_score=nan,
             estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30,
                                            metric='minkowski',
                                            metric_params=None, n_jobs=None,
                                            n_neighbors=5, p=2,
                                            weights='uniform'),
             iid='deprecated', n_jobs=None,
             param_grid={'leaf_size': [30, 40], 'n_neighbors': [1, 5, 9],
                         'p': [1, 2], 'weights': ['uniform', 'distance']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='f1_macro', verbose=0)
```

In [78]:
```
grid_search.best_params_
```
executed in 14ms, finished 09:27:33 2021-07-11

Out[78]:
```
{'leaf_size': 30, 'n_neighbors': 9, 'p': 1, 'weights': 'distance'}
```

In [79]:
```
knn_tuned = KNeighborsClassifier(n_neighbors=9, weights='distance', leaf_size=30, p=1)
knn_tuned.fit(scaled_data_train, y_train)
get_metrics(y_test, scaled_data_test, knn_tuned)
metrics['name'] = 'K Nearest Neighbors tuned'
models.append(metrics)
```
executed in 1m 2.70s, finished 09:28:36 2021-07-11

In [80]:
```
print(classification_report(y_train, knn_tuned.predict(scaled_data_train),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```

executed in 3m 55s, finished 09:32:30 2021-07-11

```
               precision    recall  f1-score   support

nonfunctional       1.00      1.00      1.00     17842
   functional       1.00      1.00      1.00     25088
 needs repair       1.00      1.00      1.00      3140

     accuracy                           1.00     46070
    macro avg       1.00      1.00      1.00     46070
 weighted avg       1.00      1.00      1.00     46070
```

In [81]:
```
print(confusion_matrix(y_test, knn_tuned.predict(scaled_data_test)))
print(classification_report(y_test, knn_tuned.predict(scaled_data_test),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```

executed in 1m 58.9s, finished 09:34:29 2021-07-11

```
[[3149 1175  102]
 [ 764 5339  198]
 [ 160  427  204]]
               precision    recall  f1-score   support

nonfunctional       0.77      0.71      0.74      4426
   functional       0.77      0.85      0.81      6301
 needs repair       0.40      0.26      0.32       791

     accuracy                           0.75     11518
    macro avg       0.65      0.61      0.62     11518
 weighted avg       0.75      0.75      0.75     11518
```

In [82]:
```
# Second round of GridSearchCV selection of parameters
param_grid = {
    'n_neighbors': [8, 9, 10], # default 5
#     'leaf_size': [30] # default 30
}
knn = KNeighborsClassifier()
grid_search = GridSearchCV(knn, param_grid, cv=3, scoring='f1_macro')
grid_search.fit(scaled_data_train, y_train)
```
executed in 8m 47s, finished 09:43:16 2021-07-11

Out[82]:
```
GridSearchCV(cv=3, error_score=nan,
             estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30,
                                            metric='minkowski',
                                            metric_params=None, n_jobs=None,
                                            n_neighbors=5, p=2,
                                            weights='uniform'),
             iid='deprecated', n_jobs=None,
             param_grid={'n_neighbors': [8, 9, 10]}, pre_dispatch='2*n_jobs',
             refit=True, return_train_score=False, scoring='f1_macro',
             verbose=0)
```

In [83]:
```
grid_search.best_params_
```
executed in 13ms, finished 09:43:16 2021-07-11

Out[83]:
```
{'n_neighbors': 9}
```

Since we don't see any change in the parameters selection we are going to keep our first trial and proceed to the next step.

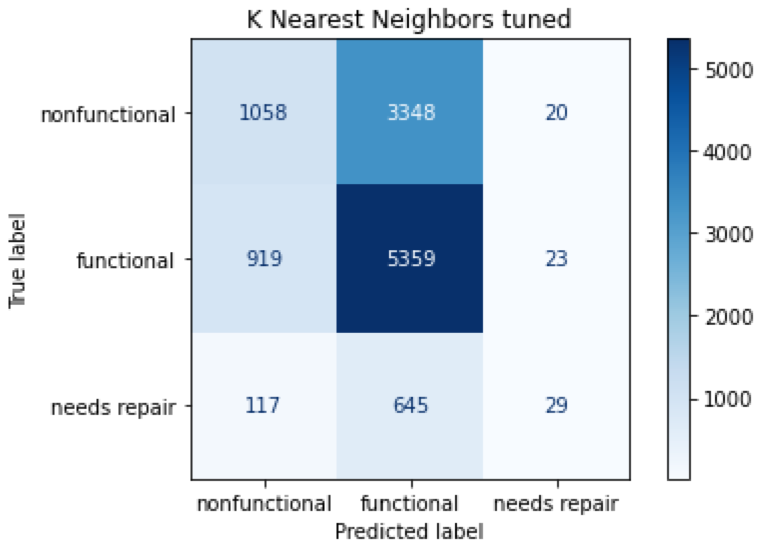
In [84]:
```
print(confusion_matrix(y_test, knn_tuned.predict(scaled_data_test)))
print(classification_report(y_test, knn_tuned.predict(scaled_data_test),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```
executed in 1m 56.9s, finished 09:45:13 2021-07-11

```
[[3149 1175  102]
 [ 764 5339  198]
 [ 160  427  204]]
               precision    recall  f1-score   support

nonfunctional       0.77      0.71      0.74      4426
   functional       0.77      0.85      0.81      6301
 needs repair       0.40      0.26      0.32       791

     accuracy                           0.75     11518
    macro avg       0.65      0.61      0.62     11518
 weighted avg       0.75      0.75      0.75     11518
```

1.7.6 Confusion Matrix

In [85]:
```
from sklearn.metrics import plot_confusion_matrix
```
executed in 13ms, finished 09:45:13 2021-07-11

In [86]:
```
plot_confusion_matrix(knn_tuned, X_test, y_test,
                      display_labels=['nonfunctional','functional','needs repair'],
                                       values_format='d', cmap=plt.cm.Blues)
plt.title('K Nearest Neighbors tuned')
plt.tight_layout()

plt.show()
```
executed in 1m 29.5s, finished 09:46:43 2021-07-11



From the confusion matrix we can conclude that using the Grid search CV tunning method our model was able to predict:

Non functional

- 120 True positives
- (45+6) = 51 False positives
- (2300+75) = 2375 False negatives
- (3900+33+460+8) = 6947 True Negatives

Functional

- 3900 True positives
- (2300+460) = 2760 False positives
- (45+33) = 78 False negatives
- (120+75+6+8) = 209 True negatives

Needs Repair

- 8 True positives
- (75+33) = 108 False positive
- (460+6) = 466 False negative
- (120+2300+3900+45) = 6365 True Negative

Since we are more interested in less False positives especially for identifying the 'nonfunctional' and 'needs repair' wells, our KNN model seems to have done better than the baseline model. It still looks like it needs some more work done at correctly classifying the 'needs repair' class, but this could also be due to the fact that the classes 'nonfunctional' and 'needs repair' have more or less similar features affecting their functionality.

We'll try another optimizing technique and see how well our KNN model would perform.

1.7.7 Smote

In [87]:
```
from imblearn.over_sampling import SMOTE
```
executed in 203ms, finished 09:46:43 2021-07-11

In [88]:
```
# Previous original class distribution
print('Original class distribution: \n')
print(y.value_counts())
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_sample(scaled_data_train, y_train)
# Preview synthetic sample class distribution
print('-----------------------------------------')
print('Synthetic sample class distribution: \n')
print(pd.Series(y_train_resampled).value_counts())
```

executed in 41.0s, finished 09:47:24 2021-07-11

```
Original class distribution:

1    31389
0    22268
2     3931
Name: status_group, dtype: int64
-----------------------------------------
Synthetic sample class distribution:

2    25088
1    25088
0    25088
Name: status_group, dtype: int64
```

In [89]:
```
knn_smote = KNeighborsClassifier(n_neighbors=9, weights='distance',
                                 leaf_size=30, p=1)
knn_smote.fit(X_train_resampled, y_train_resampled)
get_metrics(y_test, scaled_data_test, knn_smote)
metrics['name'] = 'K Nearest Neighbors smote'
models.append(metrics)
```

executed in 1m 36.7s, finished 09:49:01 2021-07-11

In [90]:
```
print(classification_report(y_train_resampled, knn_smote.predict(X_train_resampled),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```
executed in 9m 7s, finished 09:58:07 2021-07-11

```
               precision    recall  f1-score   support

nonfunctional       1.00      1.00      1.00     25088
   functional       1.00      1.00      1.00     25088
 needs repair       1.00      1.00      1.00     25088

     accuracy                           1.00     75264
    macro avg       1.00      1.00      1.00     75264
 weighted avg       1.00      1.00      1.00     75264
```

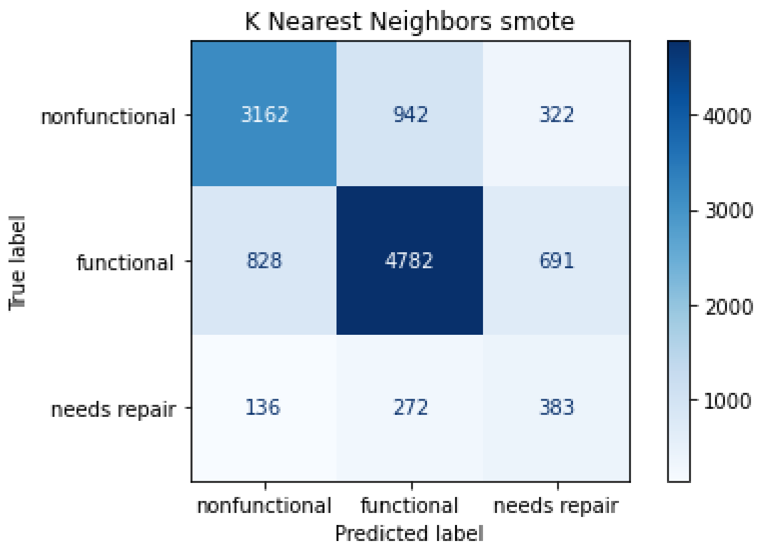
In [91]:
```
print(confusion_matrix(y_test, knn_smote.predict(scaled_data_test)))
print(classification_report(y_test, knn_smote.predict(scaled_data_test),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```
executed in 2m 57s, finished 10:01:04 2021-07-11

```
[[3162  942  322]
 [ 828 4782  691]
 [ 136  272  383]]
               precision    recall  f1-score   support

nonfunctional       0.77      0.71      0.74      4426
   functional       0.80      0.76      0.78      6301
 needs repair       0.27      0.48      0.35       791

     accuracy                           0.72     11518
    macro avg       0.61      0.65      0.62     11518
 weighted avg       0.75      0.72      0.73     11518
```

In [ ]:

1.7.8 Confusion Matrix

In [92]:
```
plot_confusion_matrix(knn_smote, scaled_data_test, y_test,
                      display_labels=['nonfunctional','functional',
                                      'needs repair'],
                      values_format='d', cmap=plt.cm.Blues)


plt.title('K Nearest Neighbors smote')
plt.tight_layout()

plt.show()
```
executed in 1m 29.7s, finished 10:02:34 2021-07-11



The K Nearest Neighbors model performed better after hyperparameter tunning in regards to Accuracy(74%) and F1 score, but didn't improve significantly with smote. The F1 score of (75%, 80%, 38%) show that there is a class imbalance and hence its effect is visible. But generally KNN also takes in incredibly long time to run on this large dataset, making it impractical to tune further. The model was generally able to capture the following percentiles of the actual wells.

K Nearest Neighbor:

1. Baseline model

Non functional (47%)
Functional (61%)
Functional needs repair (1%)

2. Gridsearch CV

   Non functional (71%)
   Functional (84%)
   Functional needs repair (27%)

3. SMOTE

   Non functional (71%)
   Functional (76%)
   Functional needs repair (41%)

1.8 Random forest

1.8.1 Baseline Model

In [93]:

```
from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier(random_state=12)
forest.fit(X_train, y_train);

metrics = get_metrics(y_test, X_test, forest)
metrics['name'] = 'Random Forest Baseline'
models.append(metrics)
```

executed in 7.50s, finished 10:02:41 2021-07-11

In [94]:
```
print(classification_report(y_train, forest.predict(X_train),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```
executed in 1.09s, finished 10:02:42 2021-07-11

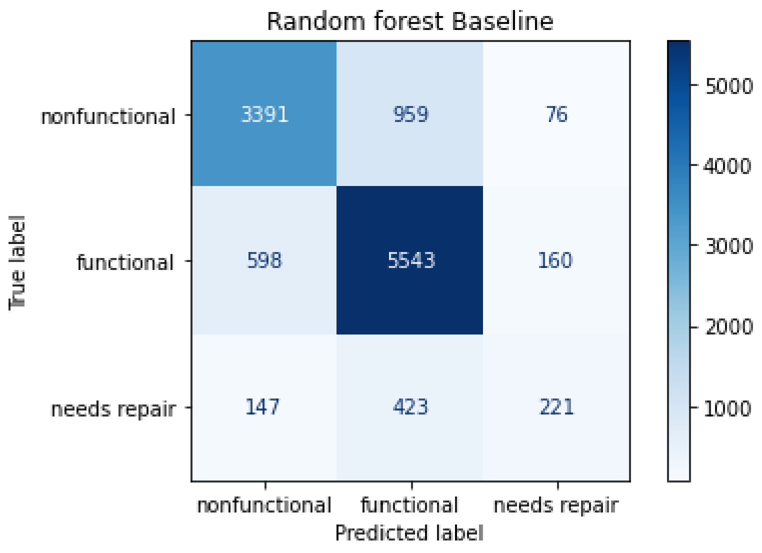
```
               precision    recall  f1-score   support

nonfunctional       1.00      1.00      1.00     17842
   functional       1.00      1.00      1.00     25088
 needs repair       1.00      1.00      1.00      3140

     accuracy                           1.00     46070
    macro avg       1.00      1.00      1.00     46070
 weighted avg       1.00      1.00      1.00     46070
```

In [95]:
```
print(confusion_matrix(y_test, forest.predict(X_test)))
print(classification_report(y_test, forest.predict(X_test),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```
executed in 602ms, finished 10:02:43 2021-07-11

```
[[3391  959   76]
 [ 598 5543  160]
 [ 147  423  221]]
               precision    recall  f1-score   support

nonfunctional       0.82      0.77      0.79      4426
   functional       0.80      0.88      0.84      6301
 needs repair       0.48      0.28      0.35       791

     accuracy                           0.79     11518
    macro avg       0.70      0.64      0.66     11518
 weighted avg       0.79      0.79      0.79     11518
```

1.8.2 Confusion matrix

In [96]:
```
plot_confusion_matrix(forest, X_test, y_test,
                     display_labels=['nonfunctional', 'functional',
                                     'needs repair'], values_format='d',
                                     cmap=plt.cm.Blues)
plt.title('Random forest Baseline')
plt.tight_layout()

plt.show()
```
executed in 506ms, finished 10:02:44 2021-07-11



1.8.3 Standardized

In [97]:

```
# Instantiate RandomForestClassifier
forest = RandomForestClassifier()
# Fit the classifier
forest.fit(scaled_data_train, y_train);


# Predict on the test set
metrics = get_metrics(y_test, scaled_data_test, forest)
metrics['name'] = 'Random forest scaled'
models.append(metrics)
```

executed in 7.39s, finished 10:02:51 2021-07-11

1.8.4 GridSearch CV

In [98]:
```
param_grid = {
    'n_estimators': [100, 200], # default 100 #boosting stages
    'max_depth': [30, 35, 40], # default None
    'max_features': [50, 60], # default 'auto': auto=sqrt(# of features)=11, None=# of features=122
#     'min_samples_split' : [20,30,40],
#     'min_samples_leaf'  : [5, 10, 15]
    # default 'auto': auto=sqrt(# of features)=11, None=# of features=122
}
forest = RandomForestClassifier()
grid_search = GridSearchCV(forest, param_grid, cv=3, scoring='f1_macro')
grid_search.fit(scaled_data_train, y_train)
```
executed in 13m 24s, finished 10:16:15 2021-07-11

Out[98]:
```
GridSearchCV(cv=3, error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                              class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              max_samples=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              n_estimators=100, n_jobs=None,
                                              oob_score=False,
                                              random_state=None, verbose=0,
                                              warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': [30, 35, 40], 'max_features': [50, 60],
                         'n_estimators': [100, 200]},
```

In [99]:
```
grid_search.best_params_
```
executed in 14ms, finished 10:16:15 2021-07-11

Out[99]:
```
{'max_depth': 35, 'max_features': 50, 'n_estimators': 200}
```

In [100]:

```
forest_tuned = RandomForestClassifier(n_estimators=200, max_depth=35,
                                      max_features=60, min_samples_leaf=35,
                                      min_samples_split=70)
forest_tuned.fit(scaled_data_train, y_train)
get_metrics(y_test, scaled_data_test, forest_tuned)
metrics['name'] = 'Random Forest tuned1'
models.append(metrics)
```

executed in 40.2s, finished 10:16:55 2021-07-11

In [101]:

```
print(confusion_matrix(y_test, forest_tuned.predict(scaled_data_test)))
print(classification_report(y_test, forest_tuned.predict(scaled_data_test),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```

executed in 653ms, finished 10:16:56 2021-07-11

```
[[3042 1370   14]
 [ 499 5778   24]
 [ 145  561   85]]
               precision    recall  f1-score   support

nonfunctional       0.83      0.69      0.75      4426
   functional       0.75      0.92      0.82      6301
 needs repair       0.69      0.11      0.19       791

     accuracy                           0.77     11518
    macro avg       0.76      0.57      0.59     11518
 weighted avg       0.77      0.77      0.75     11518
```

In [102]:
```
param_grid = {
    'n_estimators': [150, 200], # default 100
    'max_depth': [35, 45, 50], # default None
    'max_features': [55, 60, 65],
#     'min_samples_split' : 70,
#     'min_samples_leaf'  : 35
# we assume 5 would be the min sample leaf and avoid further search
}
forest = RandomForestClassifier()
grid_search = GridSearchCV(forest, param_grid, cv=3, scoring='f1_macro')
grid_search.fit(scaled_data_train, y_train)
```
executed in 24m 39s, finished 10:41:35 2021-07-11

Out[102]:
```
GridSearchCV(cv=3, error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                              class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              max_samples=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              n_estimators=100, n_jobs=None,
                                              oob_score=False,
                                              random_state=None, verbose=0,
                                              warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': [35, 45, 50],
                         'max_features': [55, 60, 65],
                         'n_estimators': [150, 200]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='f1_macro', verbose=0)
```

In [103]:
```
grid_search.best_params_
```
executed in 14ms, finished 10:41:35 2021-07-11

Out[103]:
```
{'max_depth': 45, 'max_features': 65, 'n_estimators': 150}
```

In [104]:

```
forest_tuned = RandomForestClassifier(n_estimators=200, max_depth=50,
                                      max_features=55, min_samples_leaf=35,
                                      min_samples_split=70)
forest_tuned.fit(scaled_data_train, y_train)
get_metrics(y_test, scaled_data_test, forest_tuned)
metrics['name'] = 'Random Forest tuned2'
models.append(metrics)
```

executed in 37.1s, finished 10:42:12 2021-07-11

In [105]:

```
print(confusion_matrix(y_test, forest_tuned.predict(scaled_data_test)))
print(classification_report(y_test, forest_tuned.predict(scaled_data_test),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```

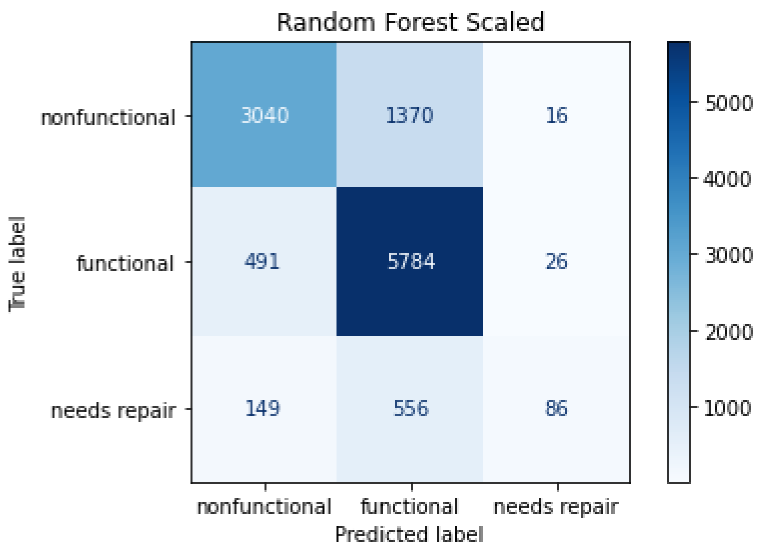
executed in 665ms, finished 10:42:13 2021-07-11

```
[[3040 1370   16]
 [ 491 5784   26]
 [ 149  556   86]]
               precision    recall  f1-score   support

nonfunctional       0.83      0.69      0.75      4426
   functional       0.75      0.92      0.83      6301
 needs repair       0.67      0.11      0.19       791

     accuracy                           0.77     11518
    macro avg       0.75      0.57      0.59     11518
 weighted avg       0.77      0.77      0.75     11518
```

1.8.5 Confusion matrix

In [106]:
```
plot_confusion_matrix(forest_tuned, scaled_data_test, y_test,
                     display_labels=['nonfunctional', 'functional',
                                     'needs repair'], values_format='d',
                      cmap=plt.cm.Blues)
plt.title('Random Forest Scaled')
plt.tight_layout()

plt.show()
```

executed in 517ms, finished 10:42:13 2021-07-11



1.8.6 Smote

In [107]:

```
forest_smote = RandomForestClassifier(n_estimators=150, max_depth=40,
                                      max_features=50, min_samples_leaf=35,
                                      min_samples_split=70)
forest_smote.fit(X_train_resampled, y_train_resampled)
get_metrics(y_test, scaled_data_test, forest_smote)
metrics['name'] = 'Random Forest smote'
models.append(metrics)
```

executed in 47.3s, finished 10:43:01 2021-07-11

In [108]:

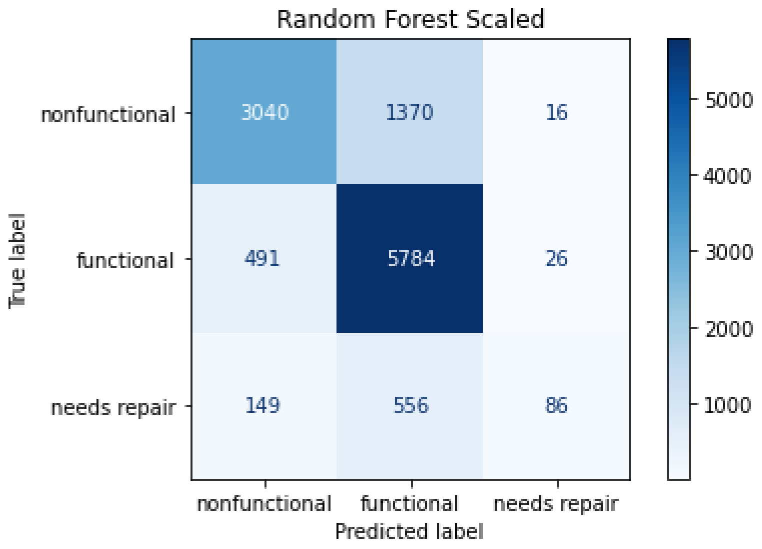
```
print(confusion_matrix(y_test, forest_smote.predict(scaled_data_test)))
print(classification_report(y_test, forest_smote.predict(scaled_data_test),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```

executed in 596ms, finished 10:43:01 2021-07-11

```
[[3082  988  356]
 [ 544 5008  749]
 [  86  260  445]]
               precision    recall  f1-score   support

nonfunctional       0.83      0.70      0.76      4426
   functional       0.80      0.79      0.80      6301
 needs repair       0.29      0.56      0.38       791

     accuracy                           0.74     11518
    macro avg       0.64      0.68      0.65     11518
 weighted avg       0.78      0.74      0.75     11518
```

1.8.7 Confusion matrix

In [109]:
```
plot_confusion_matrix(forest_tuned, scaled_data_test, y_test,
                      display_labels=['nonfunctional', 'functional',
                                      'needs repair'], values_format='d',
                      cmap=plt.cm.Blues)
plt.title('Random Forest Scaled')
plt.tight_layout()

plt.show()
```
executed in 579ms, finished 10:43:02 2021-07-11



Our Random Forest model has been able to correctly classify 5,353 data sets out of 6,837 which is 78% of our total data.

2 Feature importance

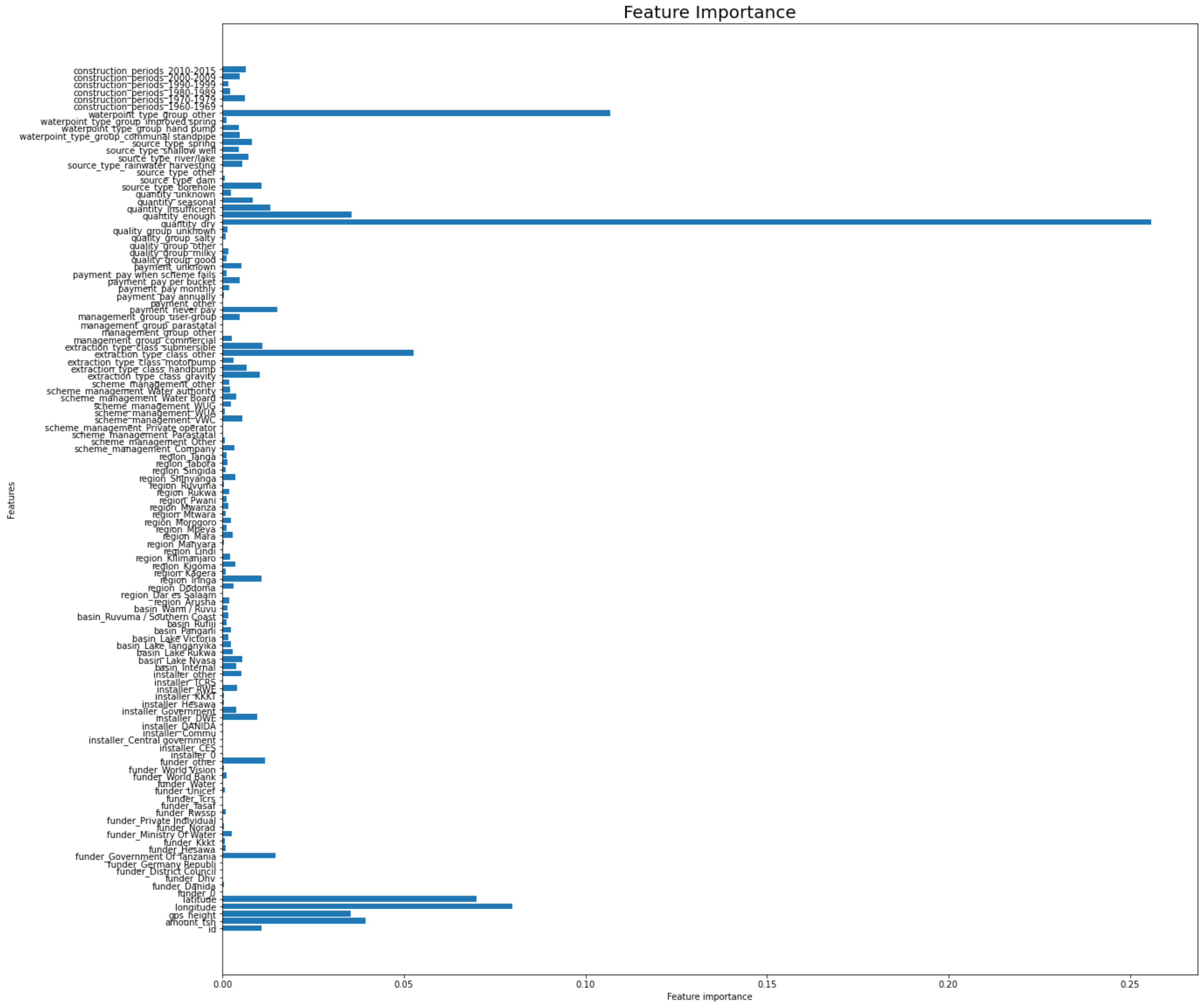
In [110]:

```
# Plot the feature importance of each feature
features = X_train.shape[1]
plt.figure(figsize=(20,20))
plt.barh(range(features), forest_tuned.feature_importances_, align='center')
plt.yticks(np.arange(features), X_train.columns.values)
plt.title('Feature Importance', fontsize=20)
plt.xlabel('Feature importance')
plt.ylabel('Features')
```

executed in 2.52s, finished 10:43:05 2021-07-11

Out[110]:

```
Text(0, 0.5, 'Features')
```


Feature Importance
construction_periods_2010-2015
construction_periods_2000-2009
construction_periods_1990-1999
construction_periods_1980-1989
construction_periods_1970-1979
construction_periods_1960-1969
waterpoint_type_group_other
waterpoint_type_group_improved spring
waterpoint_type_group_hand pump
waterpoint_type_group_communal standpipe
source_type_spring
source_type_shallow well
source_type_river/lake
source_type_rainwater harvesting
source_type_other
source_type_dam
source_type_borehole
quantity_unknown
quantity_seasonal
quantity_insufficient
quantity_enough
quantity_dry
quality_group_unknown
quality_group_salty
quality_group_other
quality_group_milky
quality_group_good
payment_unknown
payment_pay when scheme fails
payment_pay per bucket
payment_pay monthly
payment_pay annually
payment_other
payment_never pay
management_group_user-group
management_group_parastatal
management_group_other
management_group_commercial
extraction_type_class_submersible
extraction_type_class_other
extraction_type_class_motorpump
extraction_type_class_handpump
extraction_type_class_gravity
scheme_management_other
scheme_management_Water authority
scheme_management_Water Board
scheme_management_WUG
scheme_management_WUA
scheme_management_VWC
scheme_management_Private operator
scheme_management_Parastatal
scheme_management_Other
scheme_management_Company
region_Tanga
region_Tabora
region_Singida
region_Shinyanga
region_Ruvuma
region_Rukwa
region_Pwani
region_Mwanza
region_Mtwara
region_Morogoro
region_Mbeya
region_Mara
region_Manyara
region_Lindi
region_Kilimanjaro
region_Kigoma
region_Kagera
region_Iringa
region_Dodoma
region_Dar es Salaam
region_Arusha
basin_Wami / Ruvu
basin_Ruvuma / Southern Coast
basin_Rufiji
basin_Pangani
basin_Lake Victoria
basin_Lake Tanganyika
basin_Lake Rukwa
basin_Lake Nyasa
basin_Internal
installer_other
installer_TCRS
installer_RWE
installer_KKKT
installer_Hesawa
installer_Government
installer_DWE
installer_DANIDA
installer_Commu
installer_Central government
installer_CES
installer_0
funder_other
funder_World Vision
funder_World Bank
funder_Water
funder_Unicef
funder_Tcrs
funder_Tasaf
funder_Rwssp
funder_Private Individual
funder_Norad
funder_Ministry Of Water
funder_Kkkt
funder_Hesawa
funder_Government Of Tanzania
funder_Germany Republi
funder_District Council
funder_Dhv
funder_Danida
funder_0
latitude
longitude
gps_height
amount_tsh
id
Features
0.00
0.05
0.10
0.15
0.20
0.25
Feature importance

2.1 XGBoost

2.1.1 Baseline Model

In [ ]:

```
```

In [111]:

```
from sklearn.ensemble import GradientBoostingClassifier

# Instantiate XGBClassifier and fit classifier
xgb = XGBClassifier(random_state=12)
xgb.fit(X_train, y_train);

metrics = get_metrics(y_test, X_test, xgb)
metrics['name'] = 'XG Boost Baseline'
models.append(metrics)
```

executed in 11.6s, finished 10:43:16 2021-07-11

```
C:\Users\milen\anaconda3\lib\site-packages\xgboost\sklearn.py:888: UserWarning: The use of label encoder in
XGBClassifier is deprecated and will be removed in a future release. To remove this warning, do the followi
ng: 1) Pass option use_label_encoder=False when constructing XGBClassifier object; and 2) Encode your label
s (y) as integers starting with 0, i.e. 0, 1, 2, ..., [num_class - 1].
  warnings.warn(label_encoder_deprecation_msg, UserWarning)

[10:43:05] WARNING: ..\src\learner.cc:1061: Starting in XGBoost 1.3.0, the default evaluation metric used w
ith the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metric if y
ou'd like to restore the old behavior.
```

In [112]:

```
print(classification_report(y_train, xgb.predict(X_train),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```

executed in 202ms, finished 10:43:16 2021-07-11

```
               precision    recall  f1-score   support

nonfunctional       0.89      0.77      0.82     17842
   functional       0.80      0.95      0.87     25088
 needs repair       0.86      0.31      0.45      3140

     accuracy                           0.83     46070
    macro avg       0.85      0.67      0.72     46070
 weighted avg       0.84      0.83      0.82     46070
```

In [113]:

```
print(confusion_matrix(y_test, xgb.predict(X_test)))
print(classification_report(y_test, xgb.predict(X_test),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```

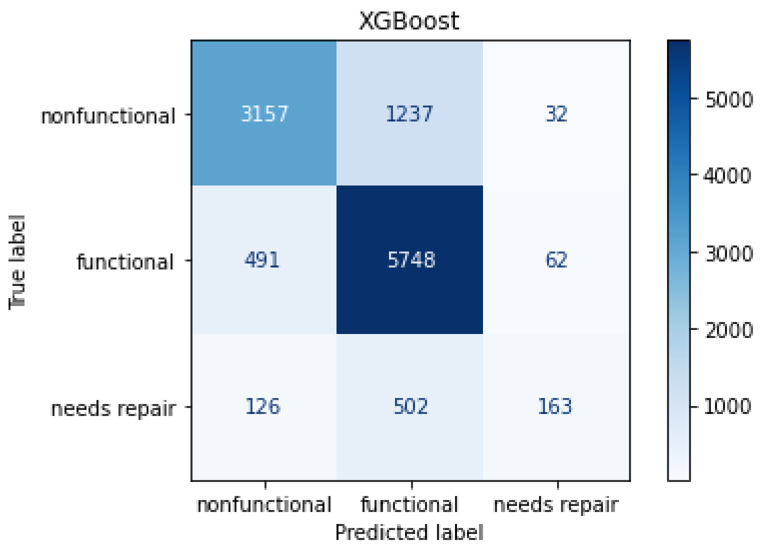
executed in 138ms, finished 10:43:17 2021-07-11

```
[[3157 1237   32]
 [ 491 5748   62]
 [ 126  502  163]]
               precision    recall  f1-score   support

nonfunctional       0.84      0.71      0.77      4426
   functional       0.77      0.91      0.83      6301
 needs repair       0.63      0.21      0.31       791

     accuracy                           0.79     11518
    macro avg       0.75      0.61      0.64     11518
 weighted avg       0.78      0.79      0.77     11518
```

2.1.2 Confusion Matrix

In [114]:
```
plot_confusion_matrix(xgb, X_test, y_test,
                      display_labels=['nonfunctional', 'functional',
                                      'needs repair'], values_format='d',
                       cmap=plt.cm.Blues)
plt.title('XGBoost')
plt.tight_layout()

plt.show()
```

executed in 301ms, finished 10:43:17 2021-07-11



2.1.3 Standardized

In [115]:
```
# Fit the classifier
xgb.fit(scaled_data_train, y_train);

# Predict on the test set
metrics = get_metrics(y_test, scaled_data_test, xgb)
metrics['name'] = 'XGBoost scaled'
models.append(metrics)
```

executed in 13.6s, finished 10:43:30 2021-07-11

```
[10:43:17] WARNING: ..\src\learner.cc:1061: Starting in XGBoost 1.3.0, the default evaluation metric used w
ith the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metric if y
ou'd like to restore the old behavior.
```

2.1.4 Grid search CV

In [116]:
```
param_grid = {
    'learning_rate': [0.05, 0.1],
    'max_depth': [6, 8, 10],
    'subsample': [0.5, 0.7],
    'n_estimators': [100, 150, 200],
}
xgb = XGBClassifier()
grid_search = GridSearchCV(xgb, param_grid, cv=None, scoring='f1_macro', n_jobs=1)
grid_search.fit(scaled_data_train, y_train)
```
executed in 1h 48m 58s, finished 12:32:28 2021-07-11

In [117]:
```
grid_search.best_params_
```
executed in 24ms, finished 12:32:28 2021-07-11

Out[117]:
```
{'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 200, 'subsample': 0.7}
```

In [118]:

```
# Instantiate XGBClassifier and fit classifier

xgb_tuned = XGBClassifier(n_estimators=200, learning_rate=0.1, max_depth=10,
                          subsample=0.7)
xgb_tuned.fit(scaled_data_train, y_train)
get_metrics(y_test, scaled_data_test, xgb_tuned)
metrics['name'] = 'XGBoost tuned1'
models.append(metrics)
```

executed in 1m 20.8s, finished 12:33:49 2021-07-11

```
[12:32:29] WARNING: ..\src\learner.cc:1061: Starting in XGBoost 1.3.0, the default evaluation metric used w
ith the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metric if y
ou'd like to restore the old behavior.
```

In [119]:

```
print(classification_report(y_train, xgb_tuned.predict(scaled_data_train),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```

executed in 463ms, finished 12:33:50 2021-07-11

```
               precision    recall  f1-score   support

nonfunctional       0.96      0.89      0.92     17842
   functional       0.89      0.98      0.94     25088
 needs repair       0.96      0.65      0.77      3140

     accuracy                           0.92     46070
    macro avg       0.94      0.84      0.88     46070
 weighted avg       0.92      0.92      0.92     46070
```

In [120]:

```
print(confusion_matrix(y_test, xgb_tuned.predict(scaled_data_test)))
print(classification_report(y_test, xgb_tuned.predict(scaled_data_test),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```

executed in 231ms, finished 12:33:50 2021-07-11

```
[[3292 1083   51]
 [ 505 5719   77]
 [ 131  473  187]]
               precision    recall  f1-score   support

nonfunctional       0.84      0.74      0.79      4426
   functional       0.79      0.91      0.84      6301
 needs repair       0.59      0.24      0.34       791

     accuracy                           0.80     11518
    macro avg       0.74      0.63      0.66     11518
 weighted avg       0.79      0.80      0.79     11518
```

In [121]:
```
param_grid = {
    'learning_rate': [0.1, 0.2],
    'max_depth': [10,15,20],
    'subsample': [0.6, 0.7],
    'n_estimators': [200, 250],
}
xgb = XGBClassifier()
grid_search = GridSearchCV(xgb, param_grid, cv=None, scoring='f1_macro', n_jobs=1)
grid_search.fit(scaled_data_train, y_train)
```
executed in 3h 29m 8s, finished 16:02:57 2021-07-11

```
d with the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metri
c if you'd like to restore the old behavior.
[12:54:10] WARNING: ..\src\learner.cc:1061: Starting in XGBoost 1.3.0, the default evaluation metric use
d with the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metri
c if you'd like to restore the old behavior.
[12:55:29] WARNING: ..\src\learner.cc:1061: Starting in XGBoost 1.3.0, the default evaluation metric use
d with the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metri
c if you'd like to restore the old behavior.
[12:56:50] WARNING: ..\src\learner.cc:1061: Starting in XGBoost 1.3.0, the default evaluation metric use
d with the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metri
c if you'd like to restore the old behavior.
[12:58:14] WARNING: ..\src\learner.cc:1061: Starting in XGBoost 1.3.0, the default evaluation metric use
d with the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metri
c if you'd like to restore the old behavior.
[12:59:59] WARNING: ..\src\learner.cc:1061: Starting in XGBoost 1.3.0, the default evaluation metric use
d with the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metri
c if you'd like to restore the old behavior.
[13:01:41] WARNING: ..\src\learner.cc:1061: Starting in XGBoost 1.3.0, the default evaluation metric use
d with the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metri
c if you'd like to restore the old behavior.
```

In [122]:
```
grid_search.best_params_
```
executed in 12ms, finished 16:02:57 2021-07-11

Out[122]:
```
{'learning_rate': 0.1, 'max_depth': 20, 'n_estimators': 200, 'subsample': 0.7}
```

In [123]:

```
# Instantiate XGBClassifier and fit classifier

xgb_tuned = XGBClassifier(n_estimators=100, learning_rate=0.1, max_depth=15,
                          subsample=0.5)
xgb_tuned.fit(scaled_data_train, y_train)
get_metrics(y_test, scaled_data_test, xgb_tuned)
metrics['name'] = 'XGBoost tuned2'
models.append(metrics)
```

executed in 53.4s, finished 16:03:51 2021-07-11

```
[16:02:58] WARNING: ..\src\learner.cc:1061: Starting in XGBoost 1.3.0, the default evaluation metric used w
ith the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metric if y
ou'd like to restore the old behavior.
```

In [124]:

```
print(confusion_matrix(y_test, xgb_tuned.predict(scaled_data_test)))
print(classification_report(y_test, xgb_tuned.predict(scaled_data_test),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```

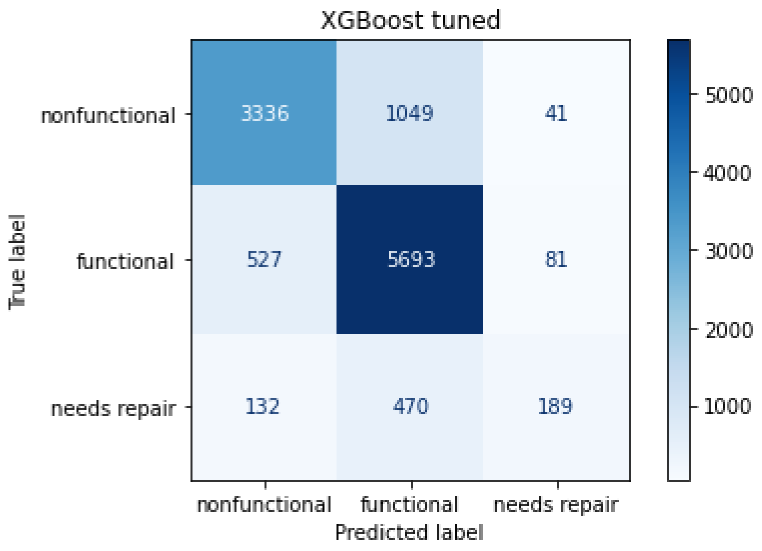
executed in 164ms, finished 16:03:51 2021-07-11

```
[[3336 1049   41]
 [ 527 5693   81]
 [ 132  470  189]]
               precision    recall  f1-score   support

nonfunctional       0.84      0.75      0.79      4426
   functional       0.79      0.90      0.84      6301
 needs repair       0.61      0.24      0.34       791

     accuracy                           0.80     11518
    macro avg       0.74      0.63      0.66     11518
 weighted avg       0.79      0.80      0.79     11518
```

2.1.5 Confusion matrix

In [125]:
```
plot_confusion_matrix(xgb_tuned, scaled_data_test, y_test,
                      display_labels=['nonfunctional', 'functional',
                                      'needs repair'], values_format='d',
                      cmap=plt.cm.Blues)
plt.title('XGBoost tuned')
plt.tight_layout()

plt.show()
```

executed in 316ms, finished 16:03:51 2021-07-11



2.1.6 Smote

In [126]:

```
# Instantiate XGBClassifier and fit classifier
xgb_smote = XGBClassifier(n_estimators=200, learning_rate=0.1, max_depth=10,
                          subsample=0.7)
xgb_smote.fit(X_train_resampled, y_train_resampled)
get_metrics(y_test, scaled_data_test, forest_smote)
metrics['name'] = 'XGBoost smote'
models.append(metrics)
```

executed in 2m 6s, finished 16:05:57 2021-07-11

```
[16:03:52] WARNING: ..\src\learner.cc:1061: Starting in XGBoost 1.3.0, the default evaluation metric used w
ith the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metric if y
ou'd like to restore the old behavior.
```

In [127]:

```
print(confusion_matrix(y_test, xgb_smote.predict(scaled_data_test)))
print(classification_report(y_test, xgb_smote.predict(scaled_data_test),
                            target_names=['nonfunctional', 'functional',
                                          'needs repair']))
```

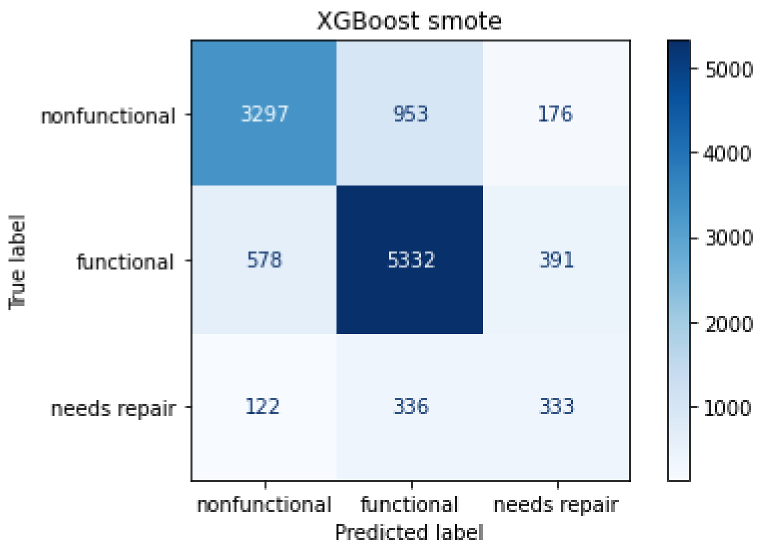
executed in 267ms, finished 16:05:57 2021-07-11

```
[[3297  953  176]
 [ 578 5332  391]
 [ 122  336  333]]
               precision    recall  f1-score   support

nonfunctional       0.82      0.74      0.78      4426
   functional       0.81      0.85      0.83      6301
 needs repair       0.37      0.42      0.39       791

     accuracy                           0.78     11518
    macro avg       0.67      0.67      0.67     11518
 weighted avg       0.78      0.78      0.78     11518
```

2.1.7 Confusion matrix

In [128]:
```
plot_confusion_matrix(xgb_smote, scaled_data_test, y_test,
                      display_labels=['nonfunctional', 'functional',
                                      'needs repair'], values_format='d',
                      cmap=plt.cm.Blues)
plt.title('XGBoost smote')
plt.tight_layout()

plt.show()
```

executed in 378ms, finished 16:05:58 2021-07-11



In [ ]:

2.2 Analysis

In [129]:
```
models_df = pd.DataFrame(models)
models_df.sort_values(by='accuracy', ascending=False)
```
executed in 30ms, finished 16:05:58 2021-07-11

Out[129]:

	accuracy	f1	precision	recall	name
5	0.797447	0.664382	0.705514	0.644198	Random Forest smote
6	0.797447	0.664382	0.705514	0.644198	Random Forest smote
7	0.797447	0.664382	0.705514	0.644198	Random Forest smote
8	0.797447	0.664382	0.705514	0.644198	Random Forest smote
4	0.794843	0.661490	0.701299	0.641750	Random Forest Baseline
9	0.787289	0.638279	0.746162	0.610530	XG Boost Baseline
10	0.786074	0.634550	0.740820	0.607812	XGBoost smote
11	0.786074	0.634550	0.740820	0.607812	XGBoost smote
12	0.786074	0.634550	0.740820	0.607812	XGBoost smote
13	0.786074	0.634550	0.740820	0.607812	XGBoost smote
1	0.751780	0.609579	0.663459	0.590658	K Nearest Neighbors smote

2.3 Conclusion & Recommendation

The overall best model was Random Forrest which performed considerably better than other models at predicting which wells were functional but need repair.(Recall score)

Random forest:

1. Baseline model

   Non functional (77%)
   Functional (87%)
   Functional needs repair (29%)

2. Gridsearch CV

Non functional (68%)
Functional (92%)
Functional needs repair (10%)

3. SMOTE

   Non functional (69%)
   Functional (79%)
   Functional needs repair (55%)

Given more time and with some more tunning it may be able to increase its performance.

2.4 Further Study

1. Identify and indicate wells that are no longer functional due to being past their life span.
2. Review which extraction types last longer in supply and quality of water.
3. Identify companies with a record of poor installation, and lack of maintenance.
4. Include date records of maintenance, accurate population size and Construction year.

```
In [ ]:
```