Exploratory Data Analysis

I start by loading the data into a pandas dataframe. First, I try to pinpoint useless features, and sort others as categorical or continuous variables.

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

In [2]:
```
df_x = pd.read_csv('data/data_x.csv', index_col='id')
df_y = pd.read_csv('data/data_y.csv', index_col='id')
df = pd.concat([df_y, df_x], axis=1)
```

In [3]:
```
df.sort_values(by='id')
```

Out[3]:

id	status_group	amount_tsh	date_recorded	funder	gps_height	installer	longitude	la
0	non functional	0.0	2012-11-13	Tasaf	0	TASAF	33.125828	-5.1
1	functional	0.0	2011-03-05	Shipo	1978	SHIPO	34.770717	-9.3
2	functional	0.0	2011-03-27	Lvia	0	LVIA	36.115056	-6.2
3	functional	10.0	2013-06-03	Germany Republi	1639	CES	37.147432	-3.1
4	non functional	0.0	2011-03-22	Cmsr	0	CMSR	36.164893	-6.0
...	...	...	...	...	...	...	...	

59400 data points, 39 dependent variables

In [4]:
```
df.columns
```

Out[4]:
```
Index(['status_group', 'amount_tsh', 'date_recorded', 'funder', 'gps_height',
       'installer', 'longitude', 'latitude', 'wpt_name', 'num_private',
       'basin', 'subvillage', 'region', 'region_code', 'district_code', 'lga',
       'ward', 'population', 'public_meeting', 'recorded_by',
       'scheme_management', 'scheme_name', 'permit', 'construction_year',
       'extraction_type', 'extraction_type_group', 'extraction_type_class',
       'management', 'management_group', 'payment', 'payment_type',
       'water_quality', 'quality_group', 'quantity', 'quantity_group',
       'source', 'source_type', 'source_class', 'waterpoint_type',
       'waterpoint_type_group'],
      dtype='object')
```

In [5]:
```
for col in df.columns:
    print(df[col].value_counts(normalize=True), '\n')
```

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

0.0         0.700993
500.0       0.052222
50.0        0.041616
1000.0      0.025051
20.0        0.024630
              ...
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

2011-03-15    0.009630
```

Notes about features:

- class imbalance in status_group (dependent variable): 54%-38%-7% split, may need to look into class imbalance options
- amount_tsh: over 70% 0 - an actual value of 0 would indicate no water, so this feature is likely very incorrect and mostly useless - also one extreme outlier of 350000
- date_recorded: over a range of dates from 2002 to 2013, not useful to prediciton
- gps_height: over a third are at exactly zero, with another 2.5% below zero, I'm not sure what to make of this, but the data will likely not be used
- longitude and latitude: the points with 0 longitude match exactly with -2e-8 latitude, neither of which are in Tanzania, but these only take up 3% of the data
- subvillage, region, region_code, district_code, lga, and ward: all seem to refer to location, so not all will be used, likely only one
- num_private: no description given for this data; with 99% 0 value, this is likely useless anyways
- population: nearly 50% of the data at 0 or 1 means this isn't likely factual information
- public_meeting: no information given about this feature
- recorded_by: not useful to prediction
- construction_year: the 35% of values with a 0 are obviously wrong
- payment and payment_type: these features seemt o be exact copies of each other
- water_quality and quality_group: near copies of each other, water_quality is a bit more precise
- quantity and quantity_group: exact copies of each other
- source, source_type, and source_class: near copies of each other, source is a bit more precise
- waterpoint_type and waterpoint_type_group: near copies of each other, waterpoint_type is a bit more precise

In [6]:
```
target = ['status_group']
to_drop = ['amount_tsh', 'date_recorded', 'gps_height', 'wpt_name', 'num_private'
           'payment_type', 'extraction_type_group', 'extraction_type_class', 'qua
           'management_group', 'source_type', 'source_class', 'waterpoint_type_gr
categorical = ['funder', 'installer', 'basin', 'subvillage', 'region', 'region_co
               'scheme_management', 'scheme_name', 'permit', 'extraction_type', '
               'water_quality', 'quantity', 'source', 'waterpoint_type']
continuous = ['longitude', 'latitude', 'population', 'construction_year']
```

Categorical Variables

Now I take a closer look at just the categorical variables. Many of these features have near copies that need to be removed, while others are less than useful without cleaning. I then need to make dummy columns of each of these features.

In [7]:
```
for col in categorical:
    print(col, df[col].value_counts().count())
```

```
funder 1897
installer 2145
basin 9
subvillage 19287
region 21
region_code 27
district_code 20
lga 125
ward 2092
scheme_management 12
scheme_name 2696
permit 2
extraction_type 18
management 12
payment 7
water_quality 8
quantity 5
source 10
waterpoint_type 7
```

Many of the categoricals variables have way too many categories, as many as 19287. In order to only bother with important features, I remove categories with representation of less than 1% of the sample, changing them to a new "other" category.

In [8]:
```
categories_to_remove = {}
for col in categorical:
    df_tmp = pd.DataFrame(df[col].value_counts(normalize=True))
    other_categories = list(df_tmp.loc[df_tmp[col]<0.01].index)
    df[col] = df[col].map(lambda x: 'other' if x in other_categories else x)
    categories_to_remove[col] = other_categories
```

In [9]:
```
categories_to_remove
```

Out[9]:
```
{'funder': ['Dwe',
  'Netherlands',
  'Hifab',
  'Adb',
  'Lga',
  'Amref',
  'Fini Water',
  'Oxfam',
  'Wateraid',
  'Rc Church',
  'Isf',
  'Rudep',
  'Mission',
  'Private',
  'Jaica',
  'Roman',
  'Rural Water Supply And Sanitat',
  'Adra',
  'Ces(gmbh)',
```

In [10]:
```
for col in categorical:
    print(df[col].value_counts(normalize=True), '\n', df[col].value_counts(normal
```

```
other                     0.483081
Government Of Tanzania    0.162898
Danida                    0.055841
Hesawa                    0.039487
Rwssp                     0.024639
World Bank                0.024191
Kkkt                      0.023079
World Vision              0.022344
Unicef                    0.018955
Tasaf                     0.015727
District Council          0.015117
Dhv                       0.014866
Private Individual        0.014812
Dwsp                      0.014543
0                         0.013933
Norad                     0.013718
Germany Republi           0.010939
Tcrs                      0.010795
Ministry Of Water         0.010580
```

With a smaller number of categories per feature, I can now take a closer look at them and see that some of them already have categories that don't correlate with an actual value. Those will be changed to this new "other" category.

Of the geographical locations, "region" seems most useful, so "subvillage", "region_code", "district_code", "lga", and "ward" will be dropped entirely. The "scheme_name" feature has nearly 90% of its values as "other", so it will also be dropped.

In [11]:
```
df['funder'] = df[col].map(lambda x: 'other' if x=='0' else x)
df['installer'] = df[col].map(lambda x: 'other' if x=='0' else x)
df['scheme_management'] = df[col].map(lambda x: 'other' if x=='Other' else x)
```

In [12]:
```
for item in ['subvillage', 'region_code', 'district_code', 'lga', 'ward', 'scheme_
    to_drop.append(item)
    categorical.remove(item)
```

In [13]:
```
corr = pd.get_dummies(df.drop(to_drop+target , axis=1)).corr()
df_corr=corr.abs().stack().reset_index().sort_values(0, ascending=False)
df_corr['pairs'] = list(zip(df_corr.level_0, df_corr.level_1))
df_corr.set_index(['pairs'], inplace = True)
df_corr.drop(columns=['level_1', 'level_0'], inplace = True)
df_corr.columns = ['cc']
df_corr.drop_duplicates(inplace=True)
df_corr[(df_corr.cc>.75) & (df_corr.cc <1)]
```

Out[13]:

	cc
pairs	
(permit_False, permit_True)	0.888369

Most models I'm going to run don't require uncorrelated data, but with so many features, I still want to trim the extra where I can, to save time in running the models and make the results easier to interpret.

In [14]:
```
df.drop(to_drop+target , axis=1).columns
```

Out[14]:
```
Index(['funder', 'installer', 'longitude', 'latitude', 'basin', 'region',
       'population', 'scheme_management', 'permit', 'construction_year',
       'extraction_type', 'management', 'payment', 'water_quality', 'quantity',
       'source', 'waterpoint_type'],
      dtype='object')
```

In [15]:
```
df_dummies = pd.get_dummies(df.drop(to_drop+target, axis=1))
df_dummies.sort_values(by='id')
```

Out[15]:

id	longitude	latitude	population	construction_year	funder_communal standpipe	funder_communal standpipe multiple
0	33.125828	-5.118154	0	0	0	0
1	34.770717	-9.395642	20	2008	0	0
2	36.115056	-6.279268	0	0	0	1
3	37.147432	-3.187555	25	1999	1	0
4	36.164893	-6.099289	0	0	0	0
...	...	...	...	...	...	...
74240	37.007726	-3.280868	350	2012	1	0
74242	33.724987	-8.940758	0	0	1	0
74243	33.963539	-1.429477	95	1983	0	1

Finally, I make dummy variables of the categorical features, ending with 100 variables.

Continuous Variables

Here I take a closer look at the continuous variables. The biggest problem is with missing values in important variables, which will be dealt with by KNN imputing.

In [16]:
```
import numpy as np
from sklearn.impute import KNNImputer
from sklearn.preprocessing import MinMaxScaler
```

In [17]:
```
df_dummies[continuous]
```

Out[17]:

id	longitude	latitude	population	construction_year
69572	34.938093	-9.856322	109	1999
8776	34.698766	-2.147466	280	2010
34310	37.460664	-3.821329	250	2009
67743	38.486161	-11.155298	58	1986
19728	31.130847	-1.825359	0	0
...	...	...	...	...
60739	37.169807	-3.253847	125	1999
27263	35.249991	-9.070629	56	1996
37057	34.017087	-8.750434	0	0
31282	35.861315	-6.378573	0	0

In [18]:
```
df_dummies[continuous].describe()
```

Out[18]:

	longitude	latitude	population	construction_year
count	59400.000000	5.940000e+04	59400.000000	59400.000000
mean	34.077427	-5.706033e+00	179.909983	1300.652475
std	6.567432	2.946019e+00	471.482176	951.620547
min	0.000000	-1.164944e+01	0.000000	0.000000
25%	33.090347	-8.540621e+00	0.000000	0.000000
50%	34.908743	-5.021597e+00	25.000000	1986.000000
75%	37.178387	-3.326156e+00	215.000000	2004.000000
max	40.345193	-2.000000e-08	30500.000000	2013.000000

In [19]:
```
for col in df_dummies[continuous].columns:
    print(df_dummies[col].value_counts(normalize=True), '\n')
```

```
0.000000     0.030505
37.540901    0.000034
33.010510    0.000034
39.093484    0.000034
32.972719    0.000034
               ...
37.579803    0.000017
33.196490    0.000017
34.017119    0.000017
33.788326    0.000017
30.163579    0.000017
Name: longitude, Length: 57516, dtype: float64

-2.000000e-08    0.030505
-6.985842e+00    0.000034
-3.797579e+00    0.000034
-6.981884e+00    0.000034
-7.104625e+00    0.000034
                   ...
```

All values need to be scaled in order to let KNN imputing work, but I don't want the scaling to be impacted by the missing variables, which are often 0. So, I need to replace all missing values with nan before applying a min-max scaler to every item in the dataframe.

In [20]:
```
df_dummies['longitude'].replace(0.0, np.NaN, inplace=True)
df_dummies['latitude'].replace(-2.000000e-08, np.NaN, inplace=True)
df_dummies['population'].replace(0, np.NaN, inplace=True)
df_dummies['construction_year'].replace(0, np.NaN, inplace=True)
```

In [21]:
```
df_dummies.isna().sum()
```

Out[21]:
```
longitude                                    1812
latitude                                     1812
population                                  21381
construction_year                           20709
funder_communal standpipe                       0
                                              ...
waterpoint_type_communal standpipe              0
waterpoint_type_communal standpipe multiple     0
waterpoint_type_hand pump                       0
waterpoint_type_improved spring                 0
waterpoint_type_other                           0
Length: 100, dtype: int64
```

In [22]:
```
scaler = MinMaxScaler()
df_scaled = pd.DataFrame(scaler.fit_transform(df_dummies), index=df_dummies.index
df_scaled.sort_values(by='id')
```

Out[22]:

id	longitude	latitude	population	construction_year	funder_communal standpipe	funder_communal standpipe multiple	f
0	0.327685	0.613210	NaN	NaN	0.0	0.0	
1	0.480868	0.211605	0.000623	0.905660	0.0	0.0	
2	0.606062	0.504195	NaN	NaN	0.0	1.0	
3	0.702203	0.794470	0.000787	0.735849	1.0	0.0	
4	0.610703	0.521093	NaN	NaN	0.0	0.0	
...	...	...	...	...	...	...	
74240	0.689193	0.785709	0.011443	0.981132	1.0	0.0	
74242	0.383483	0.254313	NaN	NaN	1.0	0.0	
74243	0.405698	0.959533	0.003082	0.433962	0.0	1.0	

Finally, I call the KNN imputer on the continuous data. I use the defaults of 5 neighbors and uniform weights.

In [23]:
```
imputer = KNNImputer()
df_imputed = pd.DataFrame(imputer.fit_transform(df_scaled), index=df_scaled.index
```

In [24]:
```
df_imputed[continuous].describe()
```

Out[24]:

	longitude	latitude	population	construction_year
count	59400.000000	59400.000000	59400.000000	59400.000000
mean	0.511598	0.549665	0.011306	0.703045
std	0.240618	0.264280	0.017296	0.212365
min	0.000000	0.000000	0.000000	0.000000
25%	0.340404	0.291881	0.002328	0.566038
50%	0.493859	0.622183	0.007508	0.754717
75%	0.705097	0.781079	0.014066	0.879245
max	1.000000	1.000000	1.000000	1.000000

In [25]:
```
df_imputed.isna().sum()
```

Out[25]:
```
longitude                                  0
latitude                                   0
population                                 0
construction_year                          0
funder_communal standpipe                  0
                                          ..
waterpoint_type_communal standpipe         0
waterpoint_type_communal standpipe multiple    0
waterpoint_type_hand pump                  0
waterpoint_type_improved spring            0
waterpoint_type_other                      0
Length: 100, dtype: int64
```

In [26]:
```
df_imputed.sort_values(by='id')
```

Out[26]:

id	longitude	latitude	population	construction_year	funder_communal standpipe	funder_communal standpipe multiple	f
0	0.327685	0.613210	0.022735	0.849057	0.0	0.0	
1	0.480868	0.211605	0.000623	0.905660	0.0	0.0	
2	0.606062	0.504195	0.008512	0.871698	0.0	1.0	
3	0.702203	0.794470	0.000787	0.735849	1.0	0.0	
4	0.610703	0.521093	0.006531	0.860377	0.0	0.0	
...	...	...	...	...	...	...	
74240	0.689193	0.785709	0.011443	0.981132	1.0	0.0	
74242	0.383483	0.254313	0.002525	0.377358	1.0	0.0	
74243	0.405698	0.959533	0.003082	0.433962	0.0	1.0	

I now have culled and cleaned the X-data into a useable format. I combine it with the y-data to complete the dataset, which I save as a csv so I can start the modelling process.

In [27]:
```
df_cleaned = pd.concat([df_y, df_imputed], axis=1)
df_cleaned.sort_values(by='id')
```

Out[27]:

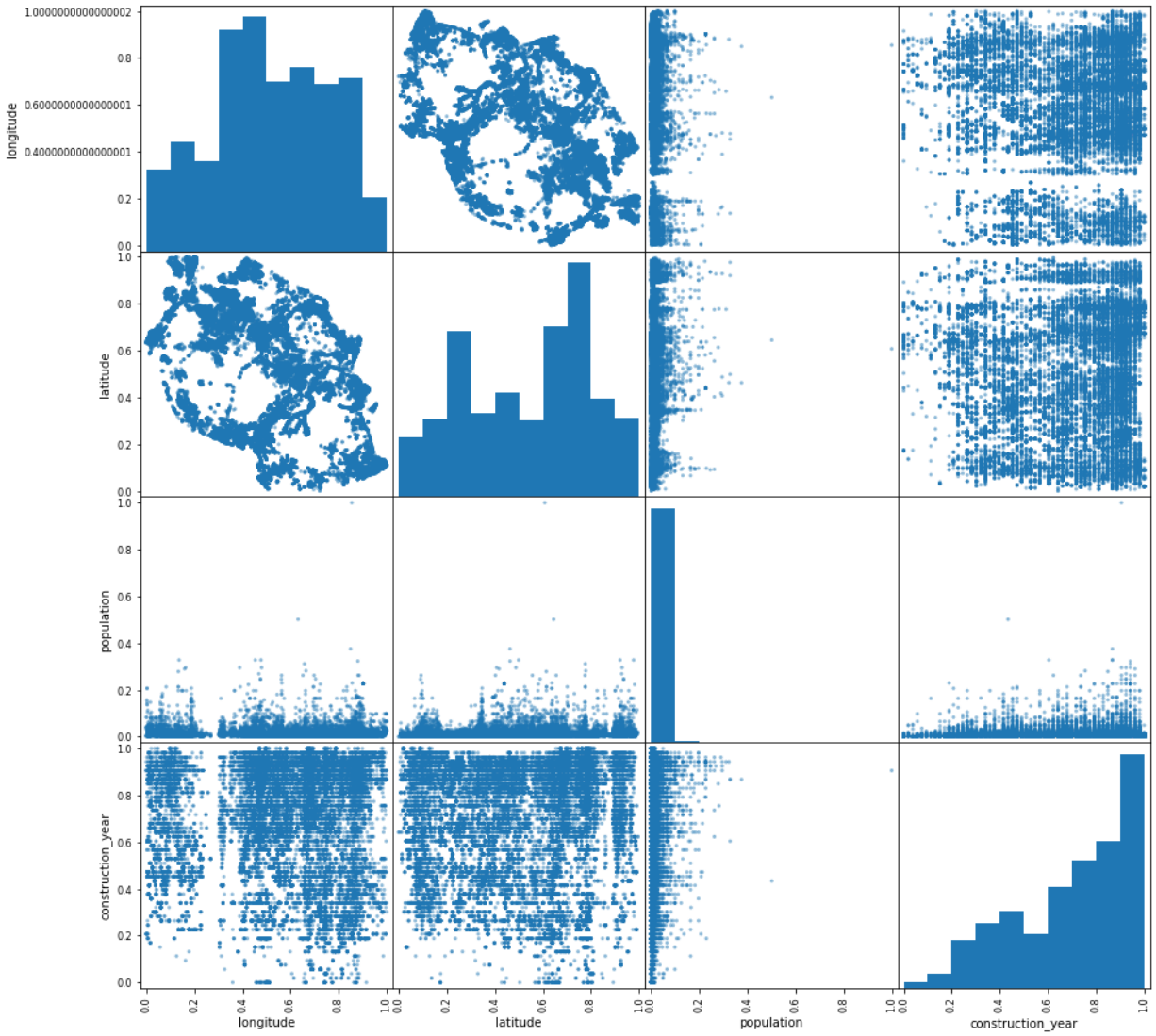
id	status_group	longitude	latitude	population	construction_year	funder_communal standpipe	funde
0	non functional	0.327685	0.613210	0.022735	0.849057	0.0	
1	functional	0.480868	0.211605	0.000623	0.905660	0.0	
2	functional	0.606062	0.504195	0.008512	0.871698	0.0	
3	functional	0.702203	0.794470	0.000787	0.735849	1.0	
4	non functional	0.610703	0.521093	0.006531	0.860377	0.0	
...	...	...	...	...	...	...	
74240	functional	0.689193	0.785709	0.011443	0.981132	1.0	
74242	functional	0.383483	0.254313	0.002525	0.377358	1.0	
74243	non functional	0.405698	0.959533	0.003082	0.433962	0.0	

In [28]:
```
df_cleaned.to_csv('data/cleaned_data.csv')
```

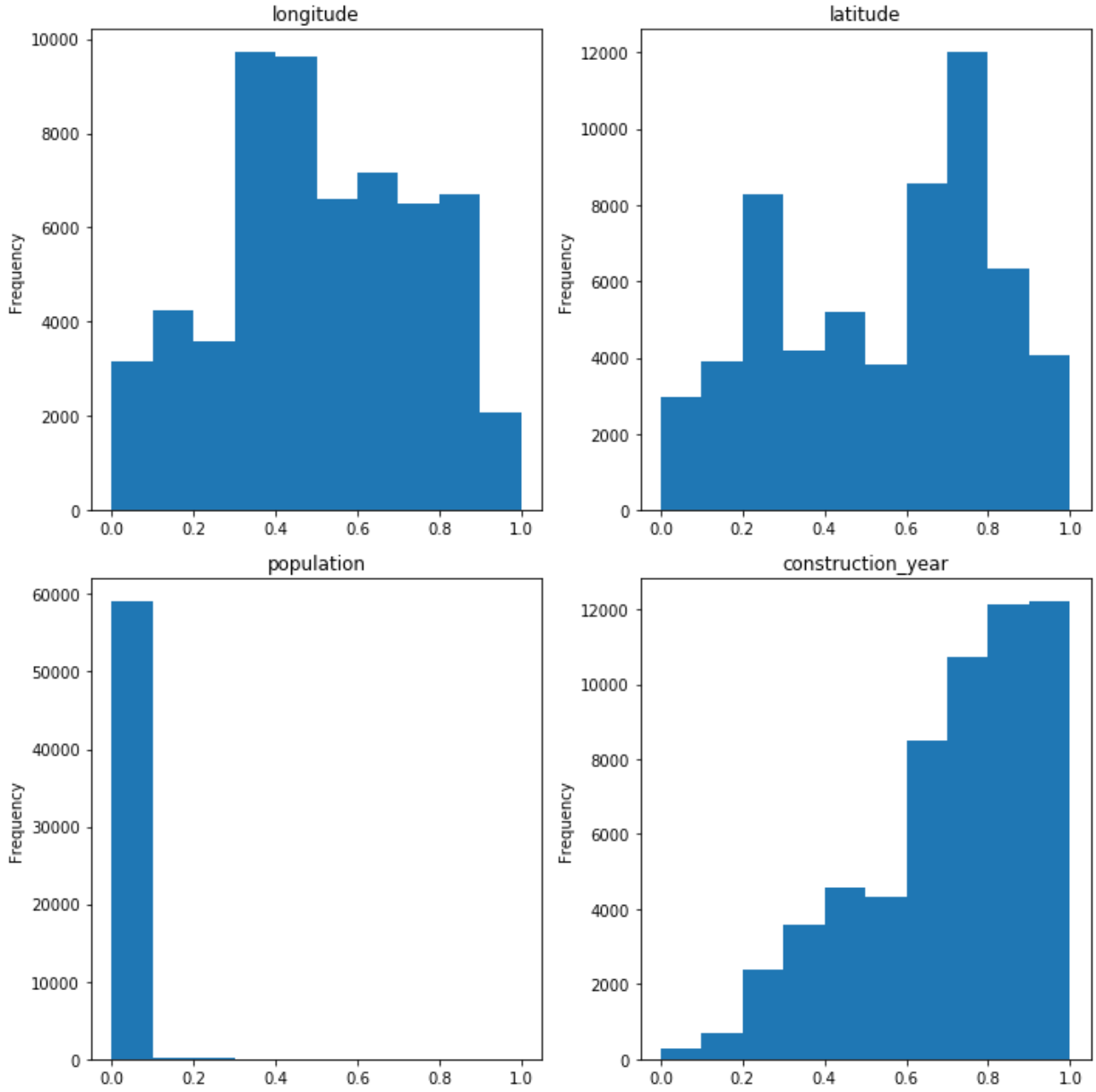
Visualizations

In [29]:
```
import matplotlib.pyplot as plt
```

In [30]:
```
pd.plotting.scatter_matrix(df_scaled[continuous], figsize=(15, 15))
plt.show()
```



In [31]:
```
fig = plt.figure(figsize=(10, 10))
for i, col in enumerate(continuous):
    ax = plt.subplot(2, 2, i+1)
    df_cleaned[col].plot(kind='hist', ax=ax, title=col)
plt.tight_layout()
plt.savefig(f'visualizations/histograms.png')
plt.show()
```



Submission Data

In order to make use of this data for the contest, I need to do the same thing to the submission X data as I did to my test data here.

In [32]:
```
X_submission = pd.read_csv('data/submission_x.csv', index_col='id')
X_submission.sort_values(by='id')
```

Out[32]:

id	amount_tsh	date_recorded	funder	gps_height	installer	longitude	latitude
10	0.0	2011-03-13	Roman Catholic	197	DWE	38.790694	-5.113207
13	0.0	2013-02-08	Tasaf	803	TASAF	36.905545	-10.765602
14	0.0	2013-03-27	Government Of Tanzania	1804	DWE	36.570357	-3.251609
29	0.0	2011-04-11	Water	0	Commu	35.882325	-6.067613
32	0.0	2011-07-27	Kyela Council	0	DWE	33.886334	-9.457260

In [33]:
```
for col, other_categories in categories_to_remove.items():
    X_submission[col] = X_submission[col].map(lambda x: 'other' if x in other_cat
X_submission['funder'] = X_submission[col].map(lambda x: 'other' if x =='0' else
X_submission['installer'] = X_submission[col].map(lambda x: 'other' if x =='0' el
X_submission['scheme_management'] = X_submission[col].map(lambda x: 'other' if x
```

In [34]:
```
X_submission_dummies = pd.get_dummies(X_submission.drop(to_drop, axis=1))
X_submission_dummies.sort_values(by='id')
```

Out[34]:

id	longitude	latitude	population	construction_year	funder_communal standpipe	funder_communal standpipe multiple
10	38.790694	-5.113207	250	1999	1	0
13	36.905545	-10.765602	1	2009	0	0
14	36.570357	-3.251609	200	1980	0	0
29	35.882325	-6.067613	0	0	0	1
32	33.886334	-9.457260	0	0	1	0
...	...	...	...	...	...	...
74241	36.563607	-6.688495	0	0	1	0
74244	37.559881	-3.498092	360	1993	1	0
74245	37.581890	-3.284912	1	1970	1	0

In [35]:
```
X_submission_dummies['longitude'].replace(0.0,np.NaN, inplace=True)
X_submission_dummies['latitude'].replace(-2.000000e-08,np.NaN, inplace=True)
X_submission_dummies['population'].replace(0,np.NaN, inplace=True)
X_submission_dummies['construction_year'].replace(0,np.NaN, inplace=True)
X_submission_scaled = pd.DataFrame(scaler.transform(X_submission_dummies), index=
X_submission_scaled.sort_values(by='id')
```

Out[35]:

id	longitude	latitude	population	construction_year	funder_communal standpipe	funder_communal standpipe multiple	f
10	0.855235	0.613675	0.008164	0.735849	1.0	0.0	
13	0.679677	0.082982	0.000000	0.924528	0.0	0.0	
14	0.648462	0.788457	0.006525	0.377358	0.0	0.0	
29	0.584388	0.524067	NaN	NaN	0.0	1.0	
32	0.398508	0.205820	NaN	NaN	1.0	0.0	
...	...	...	...	...	...	...	
74241	0.647834	0.465774	NaN	NaN	1.0	0.0	
74244	0.740613	0.765315	0.011771	0.622642	1.0	0.0	
74245	0.742663	0.785330	0.000000	0.188679	1.0	0.0	

In [36]:
```
X_submission_imputed = pd.DataFrame(imputer.transform(X_submission_scaled), index
X_submission_imputed.sort_values(by='id')
```

Out[36]:

id	longitude	latitude	population	construction_year	funder_communal standpipe	funder_communal standpipe multiple	f
10	0.855235	0.613675	0.008164	0.735849	1.0	0.0	
13	0.679677	0.082982	0.000000	0.924528	0.0	0.0	
14	0.648462	0.788457	0.006525	0.377358	0.0	0.0	
29	0.584388	0.524067	0.016689	0.494340	0.0	1.0	
32	0.398508	0.205820	0.001266	0.445283	1.0	0.0	
...	...	...	...	...	...	...	
74241	0.647834	0.465774	0.007508	0.656604	1.0	0.0	
74244	0.740613	0.765315	0.011771	0.622642	1.0	0.0	
74245	0.742663	0.785330	0.000000	0.188679	1.0	0.0	

In [37]:
```
X_submission_imputed.to_csv('data/X_submission_cleaned.csv')
```

Baseline Models

Here, I will run some baseline models on the data. After splitting the data into train and test sets, I will run it through various model types to see which ones perform the best. Those that work best will be fine-tuned later. I will use accuracy as my deciding metric, but precision and recall will let me know what values I'm having trouble classifying, and where I can improve.

In [1]:
```
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
import pandas as pd
from scripts import get_metrics
from sklearn.model_selection import train_test_split
```

In [3]:
```
df = pd.read_csv('data/cleaned_data.csv', index_col='id')
df
```

Out[3]:

id	status_group	longitude	latitude	population	construction_year	funder_communal standpipe	funde
69572	functional	0.496455	0.168353	0.003541	0.735849	1.0	
8776	functional	0.474167	0.892122	0.009148	0.943396	1.0	
34310	functional	0.731374	0.734967	0.008164	0.924528	0.0	
67743	non functional	0.826875	0.046394	0.001869	0.490566	0.0	
19728	functional	0.141899	0.922364	0.013692	0.852830	1.0	
...	...	...	...	...	...	...	
60739	functional	0.704287	0.788246	0.004066	0.735849	1.0	
27263	functional	0.525501	0.242120	0.001803	0.679245	1.0	
37057	functional	0.410685	0.272182	0.003836	0.924528	0.0	

In [4]:
```
y = df['status_group']
X = df.drop(['status_group'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=212)
```

In [5]:
```
models = []
```

Class Imbalance

As noted in the EDA, there is a class imbalance in the data; 54:39:7 funcitonal:non functional:needs repair. In order to address this, I will use the combined SMOTE and Tomek Links functions of the imbalanced learn library.

In [6]:
```
#from imblearn.combine import SMOTETomek
```

In [7]:
```
#resampler = SMOTETomek(random_state=42)
#X_train_resampled, y_train_resampled = resampler.fit_resample(X_train, y_train)
```

In [8]:
```
#pd.DataFrame(y_train_resampled)[0].value_counts(normalize=True)
```

After running tests with this, I have realized that under- and over-sampling methods reduce accuracy. They improved precision and recall - and therefore the f1 scores - but those aren't my metrics for success in this project. The competition is using accuracy as it's deciding metric, and so I won't use these methods in my final models.

Logistic Regression

In [9]:
```
from sklearn.linear_model import LogisticRegression
```

In [10]:
```
logreg = LogisticRegression(fit_intercept=False, C=1e12, solver='liblinear')
logreg.fit(X_train, y_train);
```

In [11]:
```
metrics = get_metrics(y_test, X_test, logreg)
metrics['name'] = 'Logistic Regression'
models.append(metrics)
```

K Nearest Neighbors

In [12]:
```
from sklearn.neighbors import KNeighborsClassifier
```

In [13]:
```
knn = KNeighborsClassifier()
knn.fit(X_train, y_train);
```

In [14]:
```
metrics = get_metrics(y_test, X_test, knn)
metrics['name'] = 'K Nearest Neighbors'
models.append(metrics)
```

Naive Bayes

In [15]:
```
from sklearn.naive_bayes import GaussianNB
```

In [16]:
```
bayes = GaussianNB()
bayes.fit(X_train, y_train);
```

In [17]:
```
metrics = get_metrics(y_test, X_test, bayes)
metrics['name'] = 'Naive Bayes'
models.append(metrics)
```

Decision Tree

In [18]:
```
from sklearn.tree import DecisionTreeClassifier
```

In [19]:
```
tree = DecisionTreeClassifier(random_state=12)
tree.fit(X_train, y_train);
```

In [20]:
```
metrics = get_metrics(y_test, X_test, tree)
metrics['name'] = 'Decision Tree'
models.append(metrics)
```

Bagged Trees

In [21]:
```
from sklearn.ensemble import BaggingClassifier
```

In [22]:
```
bag = BaggingClassifier(DecisionTreeClassifier(random_state=12), random_state=12)
bag.fit(X_train, y_train);
```

In [23]:
```
metrics = get_metrics(y_test, X_test, bag)
metrics['name'] = 'Bagged Trees'
models.append(metrics)
```

Random Forest

In [24]:
```
from sklearn.ensemble import RandomForestClassifier
```

In [25]:
```
forest = RandomForestClassifier(random_state=12)
forest.fit(X_train, y_train);
```

In [26]:
```
metrics = get_metrics(y_test, X_test, forest)
metrics['name'] = 'Random Forest'
models.append(metrics)
```

AdaBoost

In [27]:
```
from sklearn.ensemble import AdaBoostClassifier
```

In [28]:
```
adaboost = AdaBoostClassifier(random_state=12)
adaboost.fit(X_train, y_train);
```

In [29]:
```
metrics = get_metrics(y_test, X_test, adaboost)
metrics['name'] = 'AdaBoost'
models.append(metrics)
```

Gradient Boosting

In [30]:
```
from sklearn.ensemble import GradientBoostingClassifier
```

In [31]:
```
grad_boost = GradientBoostingClassifier(random_state=12)
grad_boost.fit(X_train, y_train);
```

In [32]:
```
metrics = get_metrics(y_test, X_test, grad_boost)
metrics['name'] = 'Gradient Boosting'
models.append(metrics)
```

XGBoost

In [33]:
```
from xgboost import XGBClassifier
```

In [34]:
```
xgb = XGBClassifier(random_state=12)
xgb.fit(X_train, y_train);
```

In [35]:
```
metrics = get_metrics(y_test, X_test, xgb)
metrics['name'] = 'XG Boost'
models.append(metrics)
```

Support Vector Machines

In [36]:
```
from sklearn.svm import SVC
```

In [37]:
```
svc = SVC(random_state=12)
svc.fit(X_train, y_train);
```

In [38]:
```
metrics = get_metrics(y_test, X_test, svc)
metrics['name'] = 'Support Vector Machine'
models.append(metrics)
```

Analysis

In [39]:
```
models_df = pd.DataFrame(models)
models_df.sort_values(by='accuracy', ascending=False)
```

Out[39]:

	accuracy	f1	precision	recall	name
5	0.792997	0.786769	0.784668	0.792997	Random Forest
4	0.783771	0.777718	0.776013	0.783771	Bagged Trees
1	0.779798	0.771395	0.770994	0.779798	K Nearest Neighbors
9	0.770370	0.749068	0.765879	0.770370	Support Vector Machine
7	0.749899	0.724101	0.749291	0.749899	Gradient Boosting
3	0.746397	0.745234	0.744187	0.746397	Decision Tree
8	0.744175	0.716140	0.746941	0.744175	XG Boost
0	0.734007	0.704561	0.725226	0.734007	Logistic Regression
6	0.727407	0.700264	0.719085	0.727407	AdaBoost
2	0.542290	0.589444	0.694734	0.542290	Naive Bayes

Conclusions

I will go more in depth on some of the best performing models. I intend to tune their hyperparameters with GridSearchCV and find the best performing model. I will look further into KNearestNeighbors, RandomForests, and SVM. I also will try XGBoost; It didn't perform well here, but it is more sensitive to hyperparameter tuning, so I expect its performance will improve more than the others. Bagging trees also saw a lot of improvement, so I will try bagging these already successful models as well.

K Nearest Neighbors

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. Still, it performed well in the baseline, so it may work well in the end.

In [1]:
```
import pandas as pd
from scripts import get_metrics
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
```

In [2]:
```
df = pd.read_csv('data/cleaned_data.csv', index_col='id')
df
```

Out[2]:

id	status_group	longitude	latitude	population	construction_year	funder_communal standpipe	funde
69572	functional	0.496455	0.168353	0.003541	0.735849	1.0	
8776	functional	0.474167	0.892122	0.009148	0.943396	1.0	
34310	functional	0.731374	0.734967	0.008164	0.924528	0.0	
67743	non functional	0.826875	0.046394	0.001869	0.490566	0.0	
19728	functional	0.141899	0.922364	0.013692	0.852830	1.0	
...	...	...	...	...	...	...	
60739	functional	0.704287	0.788246	0.004066	0.735849	1.0	
27263	functional	0.525501	0.242120	0.001803	0.679245	1.0	
37057	functional	0.410685	0.272182	0.003836	0.924528	0.0	

In [3]:
```
y = df['status_group']
X = df.drop(['status_group'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=212)
```

Baseline Model

In [4]:
```
baseline = KNeighborsClassifier()
baseline.fit(X_train, y_train)
get_metrics(y_test, X_test, baseline)
```

Out[4]:
```
{'accuracy': 0.7797979797979798,
 'f1': 0.7713946545972173,
 'precision': 0.7709936608722784,
 'recall': 0.7797979797979798}
```

GridSearch CV

I found four hyperparameters that seemed like they might significantly impact the model's performance, but in reality only the first two are likely of much help. In the second pass of the GridSearchCV, I tried to narrow the values around the successful ones I had already found, while reducing the number of models to run to a more manageable size.

In [4]:
```
from sklearn.model_selection import GridSearchCV
```

In [6]:
```
param_grid = {
    'n_neighbors': [1, 5, 10], # default 5
    'weights': ['uniform', 'distance'], # default 'uniform'
    'leaf_size': [10, 20, 30, 40], # default 30
    'p': [1, 2] # default 2
}
knn = KNeighborsClassifier()
grid_search = GridSearchCV(knn, param_grid, cv=3, scoring='accuracy')
grid_search.fit(X_train, y_train)
```

Out[6]:
```
GridSearchCV(cv=3, estimator=KNeighborsClassifier(),
             param_grid={'leaf_size': [10, 20, 30, 40],
                         'n_neighbors': [1, 5, 10], 'p': [1, 2],
                         'weights': ['uniform', 'distance']},
             scoring='accuracy')
```

In [7]:
```
grid_search.best_params_
```

Out[7]:
```
{'leaf_size': 10, 'n_neighbors': 10, 'p': 1, 'weights': 'distance'}
```

In [8]:
```
knn_tuned = KNeighborsClassifier(n_neighbors=10, weights='distance', leaf_size=10
knn_tuned.fit(X_train, y_train)
get_metrics(y_test, X_test, knn_tuned)
```

Out[8]:
```
{'accuracy': 0.7844444444444445,
 'f1': 0.7786650114084512,
 'precision': 0.7761680005340963,
 'recall': 0.7844444444444445}
```

In [5]:
```
param_grid = {
    'n_neighbors': [8, 10, 12], # default 5
    'weights': ['distance'], # default 'uniform'
    'leaf_size': [5, 10, 15], # default 30
    'p': [1] # default 2
}
knn = KNeighborsClassifier()
grid_search = GridSearchCV(knn, param_grid, cv=3, scoring='accuracy')
grid_search.fit(X_train, y_train)
```

Out[5]:
```
GridSearchCV(cv=3, estimator=KNeighborsClassifier(),
             param_grid={'leaf_size': [5, 10, 15], 'n_neighbors': [8, 10, 12],
                         'p': [1], 'weights': ['distance']},
             scoring='accuracy')
```

In [6]:
```
grid_search.best_params_
```

Out[6]:
```
{'leaf_size': 15, 'n_neighbors': 12, 'p': 1, 'weights': 'distance'}
```

In [4]:
```
knn_tuned = KNeighborsClassifier(n_neighbors=12, weights='distance', leaf_size=15
knn_tuned.fit(X_train, y_train)
get_metrics(y_test, X_test, knn_tuned)
```

Out[4]:
```
{'accuracy': 0.7857239057239057,
 'f1': 0.7794444894980167,
 'precision': 0.7769044977516286,
 'recall': 0.7857239057239057}
```

Bagging

In my baseline models, bagging decision trees saw significant improvements. I decided to attempt bagging more successful models. I'm not sure if this is actually good practice, but I was curious if this would improve the models or not.

In [4]:
```
from sklearn.ensemble import BaggingClassifier
```

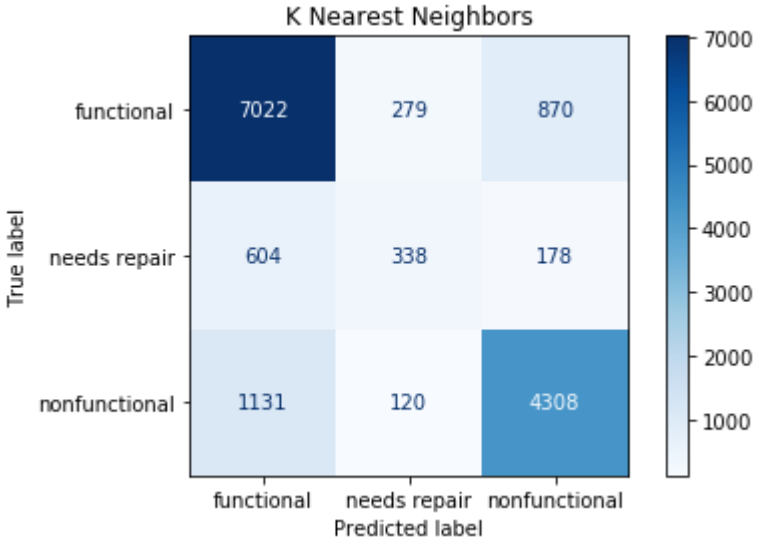
In [6]:
```
knn_bagged = BaggingClassifier(KNeighborsClassifier(n_neighbors=12, weights='dist
knn_bagged.fit(X_train, y_train)
get_metrics(y_test, X_test, knn_bagged)
```

Out[6]:
```
{'accuracy': 0.7852525252525252,
 'f1': 0.778485707736249,
 'precision': 0.7763541475578498,
 'recall': 0.7852525252525252}
```

Confusion Matrix

In [5]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix
```

In [7]:
```
plot_confusion_matrix(knn_tuned, X_test, y_test, display_labels=['functional', 'n
plt.title('K Nearest Neighbors')
plt.tight_layout()
plt.savefig(f'visualizations/knn-confusion-matrix.png')
plt.show()
```



Conclusions

The K Nearest Neighbors model performed better than most in the baseline models, but didn't improve significantly with hyperparemeter tuning. This is because KNN has relatively few hyperparameters: n_neighbors and weights are the two that have the most effect on its performance. It also takes in incredibly long time to run on this large dataset, making it impractical to tune further. Bagging the model saw no improvement, and so I won't bag the final model.

Random Forest

The random forest model performed the best among all the baseline models, and has significant tuning that can be done, so I'm hopeful that this one will perform well.

In [1]:
```
import pandas as pd
from scripts import get_metrics
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
```

In [2]:
```
df = pd.read_csv('data/cleaned_data.csv', index_col='id')
df
```

Out[2]:

id	status_group	longitude	latitude	population	construction_year	funder_communal standpipe	funde
69572	functional	0.496455	0.168353	0.003541	0.735849	1.0	
8776	functional	0.474167	0.892122	0.009148	0.943396	1.0	
34310	functional	0.731374	0.734967	0.008164	0.924528	0.0	
67743	non functional	0.826875	0.046394	0.001869	0.490566	0.0	
19728	functional	0.141899	0.922364	0.013692	0.852830	1.0	
...	...	...	...	...	...	...	
60739	functional	0.704287	0.788246	0.004066	0.735849	1.0	
27263	functional	0.525501	0.242120	0.001803	0.679245	1.0	
37057	functional	0.410685	0.272182	0.003836	0.924528	0.0	

In [3]:
```
y = df['status_group']
X = df.drop(['status_group'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=212)
```

Baseline Model

In [4]:
```
baseline = RandomForestClassifier()
baseline.fit(X_train, y_train)
get_metrics(y_test, X_test, baseline)
```

Out[4]:
```
{'accuracy': 0.7952861952861953,
 'f1': 0.7888460477820153,
 'precision': 0.7866511636737485,
 'recall': 0.7952861952861953}
```

GridSearch CV

The hyperparameters I found seem like they might have a significant impact on the results. Raising n_estimators too high increased the runtime significantly, so I didn't want to go any higher than 500.

In [2]:
```
from sklearn.model_selection import GridSearchCV
```

In [6]:
```
param_grid = {
    'n_estimators': [10, 100, 500], # default 100
    'max_depth': [None, 10], # default None
    'max_features': ['auto', 50, None] # default 'auto': auto=sqrt(# of features)=
}
forest = RandomForestClassifier()
grid_search = GridSearchCV(forest, param_grid, cv=3, scoring='accuracy')
grid_search.fit(X_train, y_train)
```

Out[6]:
```
GridSearchCV(cv=3, estimator=RandomForestClassifier(),
             param_grid={'max_depth': [None, 10],
                         'max_features': ['auto', 50, None],
                         'n_estimators': [10, 100, 500]},
             scoring='accuracy')
```

In [7]:
```
grid_search.best_params_
```

Out[7]:
```
{'max_depth': None, 'max_features': 'auto', 'n_estimators': 500}
```

In [8]:
```
forest_tuned = RandomForestClassifier(n_estimators=500, max_depth=None, max_featu
forest_tuned.fit(X_train, y_train)
get_metrics(y_test, X_test, forest_tuned)
```

Out[8]:
```
{'accuracy': 0.7934680134680134,
 'f1': 0.7870670131720358,
 'precision': 0.7848284765539788,
 'recall': 0.7934680134680134}
```

In [7]:
```
param_grid = {
    'n_estimators': [500], # default 100
    'max_depth': [None, 50], # default None
    'max_features': [5, 'auto', 15] # default 'auto': auto=sqrt(# of features)=10
}
forest = RandomForestClassifier()
grid_search = GridSearchCV(forest, param_grid, cv=3, scoring='accuracy')
grid_search.fit(X_train, y_train)
```

Out[7]:
```
GridSearchCV(cv=3, estimator=RandomForestClassifier(),
             param_grid={'max_depth': [None, 50],
                         'max_features': [5, 'auto', 15],
                         'n_estimators': [500]},
             scoring='accuracy')
```

In [8]:
```
grid_search.best_params_
```

Out[8]:
```
{'max_depth': None, 'max_features': 15, 'n_estimators': 500}
```

In [9]:
```
forest_tuned = RandomForestClassifier(n_estimators=500, max_depth=None, max_featu
forest_tuned.fit(X_train, y_train)
get_metrics(y_test, X_test, forest_tuned)
```

Out[9]:
```
{'accuracy': 0.7952861952861953,
 'f1': 0.7887222537260427,
 'precision': 0.7866326451468552,
 'recall': 0.7952861952861953}
```

Bagging

Bagging this model saw pretty good improvement: 0.5%. Random forests is already similar to bagged forests, so I didn't expect bagging to improve performance this much. But since it worked, I will use it in the final model.

In [4]:
```
from sklearn.ensemble import BaggingClassifier
```

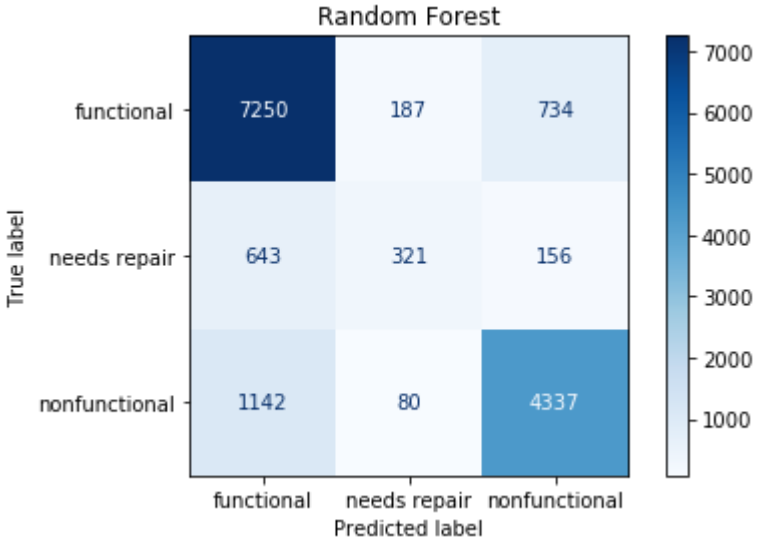
In [5]:
```
forest_bagged = BaggingClassifier(RandomForestClassifier(n_estimators=500, max_de
forest_bagged.fit(X_train, y_train)
get_metrics(y_test, X_test, forest_bagged)
```

Out[5]:
```
{'accuracy': 0.8018855218855219,
 'f1': 0.7930922693527286,
 'precision': 0.7933060737293601,
 'recall': 0.8018855218855219}
```

Confusion Matrix

In [6]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix
```

In [7]:
```
plot_confusion_matrix(forest_bagged, X_test, y_test, display_labels=['functional'
plt.title('Random Forest')
plt.tight_layout()
plt.savefig(f'visualizations/random-forest-confusion-matrix.png')
plt.show()
```



Conclusions

The Random Forests model saw some improvement here, but not as much as I had hoped. The helpful hyperparameters also raised the runtime of the model, forcing me to stop improving the hyperparameters. That said, it is outperforming my other models, and was responsive to bagging, making it the most useful to far.

XG Boost

Despite not performing well in the baselines, I know XGBoost has a large amount of hyperparameters that can be tuned, so I am hopeful that its performance will improve significantly.

In [1]:
```
import pandas as pd
from scripts import get_metrics
from sklearn.model_selection import train_test_split
from xgboost import XGBClassifier
```

In [2]:
```
df = pd.read_csv('data/cleaned_data.csv', index_col='id')
df
```

Out[2]:

id	status_group	longitude	latitude	population	construction_year	funder_communal standpipe	funde
69572	functional	0.496455	0.168353	0.003541	0.735849	1.0	
8776	functional	0.474167	0.892122	0.009148	0.943396	1.0	
34310	functional	0.731374	0.734967	0.008164	0.924528	0.0	
67743	non functional	0.826875	0.046394	0.001869	0.490566	0.0	
19728	functional	0.141899	0.922364	0.013692	0.852830	1.0	
...	...	...	...	...	...	...	
60739	functional	0.704287	0.788246	0.004066	0.735849	1.0	
27263	functional	0.525501	0.242120	0.001803	0.679245	1.0	
37057	functional	0.410685	0.272182	0.003836	0.924528	0.0	

In [3]:
```
y = df['status_group']
X = df.drop(['status_group'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=212)
```

Baseline Model

In [4]:
```
baseline = XGBClassifier()
baseline.fit(X_train, y_train)
get_metrics(y_test, X_test, baseline)
```

Out[4]:
```
{'accuracy': 0.7441750841750842,
 'f1': 0.7161396155265648,
 'precision': 0.7469413237149911,
 'recall': 0.7441750841750842}
```

GridSearch CV

Time and RAM constraints leave me unable to test every tunable hyperparameter this model offers, so I narrowed it down to a few that seemed the most important.

In [5]:
```
from sklearn.model_selection import GridSearchCV
```

In [6]:
```
param_grid = {
    'eta': [0.1, 0.3], # default 0.3, aka learning_rate
    'gamma': [0, 3], # default 0, aka min_loss_split
    'max_depth': [6, 8, 10], # default 6
    'min_child_weight': [0.5, 1], # default 1
    'subsample': [0.5, 1], # default 1
}
xgb = XGBClassifier()
grid_search = GridSearchCV(xgb, param_grid, cv=3, scoring='accuracy')
grid_search.fit(X_train, y_train)
```

Out[6]:
```
GridSearchCV(cv=3, estimator=XGBClassifier(),
             param_grid={'eta': [0.1, 0.3], 'gamma': [0, 3],
                         'max_depth': [6, 8, 10], 'min_child_weight': [0.5, 1],
                         'subsample': [0.5, 1]},
             scoring='accuracy')
```

In [7]:
```
grid_search.best_params_
```

Out[7]:
```
{'eta': 0.1,
 'gamma': 0,
 'max_depth': 10,
 'min_child_weight': 0.5,
 'subsample': 0.5}
```

In [8]:
```
xgb_tuned = XGBClassifier(eta=0.1, gamma=0, max_depth=10, min_child_weight=0.5, s
xgb_tuned.fit(X_train, y_train)
get_metrics(y_test, X_test, xgb_tuned)
```

Out[8]:
```
{'accuracy': 0.7997979797979798,
 'f1': 0.7865408402000733,
 'precision': 0.7941122092215788,
 'recall': 0.7997979797979798}
```

In [6]:
```
param_grid = {
    'eta': [0.05, 0.1], # default 0.3, aka learning_rate
    'gamma': [0], # default 0, aka min_loss_split
    'max_depth': [10, 12], # default 6
    'min_child_weight': [0.25, 0.5, 0.75], # default 1
    'subsample': [0.25, 0.5, 0.75], # default 1
}
xgb = XGBClassifier()
grid_search = GridSearchCV(xgb, param_grid, cv=3, scoring='accuracy')
grid_search.fit(X_train, y_train)
```

Out[6]:
```
GridSearchCV(cv=3, estimator=XGBClassifier(),
             param_grid={'eta': [0.05, 0.1], 'gamma': [0],
                         'max_depth': [10, 12],
                         'min_child_weight': [0.25, 0.5, 0.75],
                         'subsample': [0.25, 0.5, 0.75]},
             scoring='accuracy')
```

In [7]:
```
grid_search.best_params_
```

Out[7]:
```
{'eta': 0.05,
 'gamma': 0,
 'max_depth': 12,
 'min_child_weight': 0.75,
 'subsample': 0.75}
```

In [4]:
```
xgb_tuned = XGBClassifier(eta=0.05, gamma=0, max_depth=12, min_child_weight=0.75,
xgb_tuned.fit(X_train, y_train)
get_metrics(y_test, X_test, xgb_tuned)
```

Out[4]:
```
{'accuracy': 0.8024915824915825,
 'f1': 0.7911600451773849,
 'precision': 0.795282901250696,
 'recall': 0.8024915824915825}
```

Bagging

XGBoost already includes regularization that prevents over-fitting, so I don't expect bagging to improve its performance at all.

In [9]:
```
from sklearn.ensemble import BaggingClassifier
```

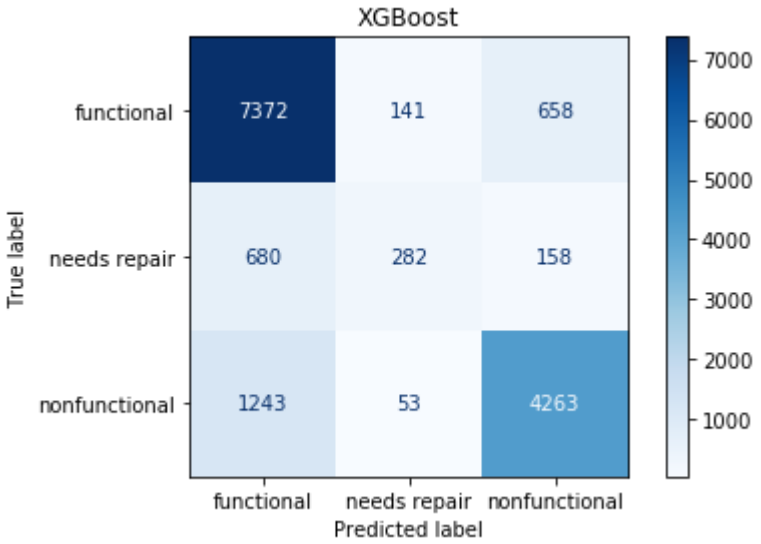
In [10]:
```
xgb_bagged = BaggingClassifier(XGBClassifier(eta=0.05, gamma=0, max_depth=12, min
xgb_bagged.fit(X_train, y_train)
get_metrics(y_test, X_test, xgb_bagged)
```

Out[10]:
```
{'accuracy': 0.8022895622895623,
 'f1': 0.7899042260108825,
 'precision': 0.7954930143029152,
 'recall': 0.8022895622895623}
```

Confusion Matrix

In [5]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix
```

In [7]:
```
plot_confusion_matrix(xgb_tuned, X_test, y_test, display_labels=['functional', 'n
plt.title('XGBoost')
plt.tight_layout()
plt.savefig(f'visualizations/xgboost-confusion-matrix.png')
plt.show()
```



Conclusions

As I thought, tuning the hyperparameters of XGBoost improved its performance significantly - it is not my best performing model. It also has a host of other parameters that can be tuned, so that would be one avenue to investigate if returning to improve this upon the model at a later time. Despite not focusing on other metrics, This XGBoost model also has better precision and recall than my other models.

Support Vector Machines

The SVM model is a distance-based algorithm, similar to KNN, but has more tunable parameters, so I wanted to try it out. This one also takes the longest to run, making tuning those parameters a real chore.

In [1]:
```
import pandas as pd
from scripts import get_metrics
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
```

In [2]:
```
df = pd.read_csv('data/cleaned_data.csv', index_col='id')
df
```

Out[2]:

id	status_group	longitude	latitude	population	construction_year	funder_communal standpipe	funde
69572	functional	0.496455	0.168353	0.003541	0.735849	1.0	
8776	functional	0.474167	0.892122	0.009148	0.943396	1.0	
34310	functional	0.731374	0.734967	0.008164	0.924528	0.0	
67743	non functional	0.826875	0.046394	0.001869	0.490566	0.0	
19728	functional	0.141899	0.922364	0.013692	0.852830	1.0	
...	...	...	...	...	...	...	
60739	functional	0.704287	0.788246	0.004066	0.735849	1.0	
27263	functional	0.525501	0.242120	0.001803	0.679245	1.0	
37057	functional	0.410685	0.272182	0.003836	0.924528	0.0	

In [3]:
```
y = df['status_group']
X = df.drop(['status_group'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=212)
```

Baseline Model

In [4]:
```
baseline = SVC()
baseline.fit(X_train, y_train)
get_metrics(y_test, X_test, baseline)
```

Out[4]:
```
{'accuracy': 0.7703703703703704,
 'f1': 0.7490681311609566,
 'precision': 0.7658791378307882,
 'recall': 0.7703703703703704}
```

GridSearch CV

Which parameters are tunable depend on the kernel chosen. rbf seems to be better for not-easily-seperable data, so I chose it. I believe the balanced value for class_weight would be better for most models, but I am using solely accuracy as my scoring metric, and so it is less useful in this case.

In [4]:
```
from sklearn.model_selection import GridSearchCV
```

In [ ]:
```
param_grid = {
    'C': [0.1, 1, 10, 100], # default 1.0
    'kernel': ['rbf'], # default 'rbf'
    'gamma': [0.001, 0.01, 0.1, 1], # default 'scale'
    'class_weight': [None, 'balanced'], # default None
}
svc = SVC()
grid_search = GridSearchCV(svc, param_grid, cv=3, scoring='accuracy')
grid_search.fit(X_train, y_train)
```

In [7]:
```
grid_search.best_params_
```

Out[7]:
```
{'C': 10, 'class_weight': None, 'gamma': 0.1, 'kernel': 'rbf'}
```

In [8]:
```
svc_tuned = SVC(C=10, kernel='rbf', gamma=0.1, class_weight=None)
svc_tuned.fit(X_train, y_train)
get_metrics(y_test, X_test, svc_tuned)
```

Out[8]:
```
{'accuracy': 0.775084175084175,
 'f1': 0.7610356464920425,
 'precision': 0.7664295252943402,
 'recall': 0.775084175084175}
```

In [5]:
```
param_grid = {
    'C': [5, 10, 50], # default 1.0
    'kernel': ['rbf'], # default 'rbf'
    'gamma': [0.05, 0.1, 0.5], # default 'scale'
    'class_weight': [None], # default None
}
svc = SVC()
grid_search = GridSearchCV(svc, param_grid, cv=3, scoring='accuracy')
grid_search.fit(X_train, y_train)
```

Out[5]:
```
GridSearchCV(cv=3, estimator=SVC(),
             param_grid={'C': [5, 10, 50], 'class_weight': [None],
                         'gamma': [0.05, 0.1, 0.5], 'kernel': ['rbf']},
             scoring='accuracy')
```

In [6]:
```
grid_search.best_params_
```

Out[6]:
```
{'C': 5, 'class_weight': None, 'gamma': 0.1, 'kernel': 'rbf'}
```

In [7]:
```
svc_tuned = SVC(C=5, kernel='rbf', gamma=0.1, class_weight=None)
svc_tuned.fit(X_train, y_train)
get_metrics(y_test, X_test, svc_tuned)
```

Out[7]:
```
{'accuracy': 0.7757575757575758,
 'f1': 0.7599310142186181,
 'precision': 0.7680019416187246,
 'recall': 0.7757575757575758}
```

Bagging

Bagging seems to have some small improvement for SVM, so I will use it in the final model.

In [4]:
```
from sklearn.ensemble import BaggingClassifier
```

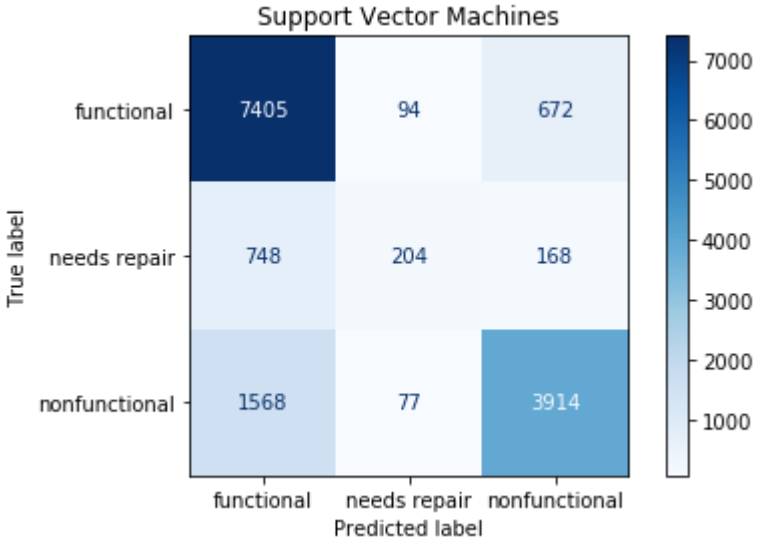
In [5]:
```
svc_bagged = BaggingClassifier(SVC(C=5, kernel='rbf', gamma=0.1, class_weight=Non
svc_bagged.fit(X_train, y_train)
get_metrics(y_test, X_test, svc_bagged)
```

Out[5]:
```
{'accuracy': 0.775959595959596,
 'f1': 0.7601798972167861,
 'precision': 0.7683720458118004,
 'recall': 0.775959595959596}
```

Confusion Matrix

In [6]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import plot_confusion_matrix
```

In [7]:
```
plot_confusion_matrix(svc_bagged, X_test, y_test, display_labels=['functional', '
plt.title('Support Vector Machines')
plt.tight_layout()
plt.savefig(f'visualizations/svc-confusion-matrix.png')
plt.show()
```



Conclusions

SVM performs worse than any of my other tuned models. This is likely because it performs best on more easily seperable data. I don't expect it to have good performance as the final model.

Creating Contest Submissions

This project is part of an online competition, and so there is unlabeled data provided to test against. I will use each of the tuned models to produce results from this data, and submit it to the competition.

Although I split the data provided to me into test and train data before, I do not do so here. The submission data is 1/3 of the training data, indicating a 75:25 train-test split was already performed. I will instead train the models on the entirety of the data I was given, using the hyperparameters tuned using my own train-test splits.

In [1]:
```
import pandas as pd
from scripts import get_metrics
from sklearn.ensemble import BaggingClassifier
```

In [2]:
```
df = pd.read_csv('data/cleaned_data.csv', index_col='id')
y = df['status_group']
X = df.drop(['status_group'], axis=1)
```

In [3]:
```
X_submission = pd.read_csv('data/X_submission_cleaned.csv', index_col='id')
```

K Nearest Neighbors

In [4]:
```
from sklearn.neighbors import KNeighborsClassifier
```

In [5]:
```
knn = KNeighborsClassifier(n_neighbors=12, weights='distance', leaf_size=15, p=1)
knn.fit(X, y)
```

Out[5]:
```
KNeighborsClassifier(leaf_size=15, n_neighbors=12, p=1, weights='distance')
```

In [6]:
```
y_submission = pd.DataFrame(knn.predict(X_submission), index=X_submission.index,
y_submission.to_csv('data/submissions/k_nearest_neighbors.csv')
```

79.99% accuracy upon submission.

Random Forest

In [4]:
```
from sklearn.ensemble import RandomForestClassifier
```

In [5]:
```
forest = BaggingClassifier(RandomForestClassifier(n_estimators=500, max_depth=Non
forest.fit(X, y)
```

Out[5]:
```
BaggingClassifier(base_estimator=RandomForestClassifier(max_features=15,
                                                        n_estimators=500))
```

In [12]:
```
y_submission = pd.DataFrame(forest.predict(X_submission), index=X_submission.inde
y_submission.to_csv('data/submissions/bagged_random_forests.csv')
```

81.49% accuracy upon submission.

XGBoost

In [4]:
```
from xgboost import XGBClassifier
```

In [5]:
```
xgb = XGBClassifier(eta=0.05, gamma=0, max_depth=12, min_child_weight=0.75, subsa
xgb.fit(X.drop('population', axis=1), y)
```

Out[5]:
```
XGBClassifier(eta=0.05, max_depth=12, min_child_weight=0.75,
              objective='multi:softprob', subsample=0.75)
```

In [6]:
```
y_submission = pd.DataFrame(xgb.predict(X_submission.drop('population', axis=1)),
y_submission.to_csv('data/submissions/xg_boost.csv')
```

81.50% accuracy upon submission.

Support Vector Machines

In [4]:
```
from sklearn.svm import SVC
```

In [5]:
```
svc = BaggingClassifier(SVC(C=5, kernel='rbf', gamma=0.1, class_weight=None))
svc.fit(X, y)
```

Out[5]:
```
BaggingClassifier(base_estimator=SVC(C=5, gamma=0.1))
```

In [6]:
```
y_submission = pd.DataFrame(svc.predict(X_submission), index=X_submission.index,
y_submission.to_csv('data/submissions/support_vector_machines.csv')
```

78.04% accuracy upon submission.

Conclusions

As I thought when I first started this project, XGBoost performed the best, scoring an 81.50% accuracy, and landing me a rank of 1303 of 10458 competitors. Interestingly, the Bagged Random Forests model only performed worse by one or two data points. I can't see the answers for the test data, but there are nearly 1000 of the nearly 1500 data points predicted differently between the two models.

The KNN and SVM models did not perform nearly as well as these two. The data is likely not seperable enough for these distance-based models to be useful.

```python
from sklearn.metrics import precision_score, recall_score, accuracy_score, f1_score

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='weighted')
    metrics['precision'] = precision_score(labels, preds, average='weighted')
    metrics['recall'] = recall_score(labels, preds, average='weighted')

    return metrics
```