Table of Contents

Phase3 Project

Business Objective

To help the Goverment of Tanzania monitor the condition of installed waterpumps across the country.Given a set of parameters, the model should be able to predict the status of a waterpump. Status can be as classified as:

1. Functional
2. Functional needs repair
3. non functional

Dataset

Dataset sourced from: https://www.drivendata.org/competitions/7/pump-it-up-data-mining-the-water-table/page/23/

Import libraries

In [1]:
```
import pandas as pd
import numpy as np
import geopandas as gpd
from shapely.geometry import Point, Polygon
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.metrics import f1_score,accuracy_score,precision_score,recall_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import plot_confusion_matrix
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
import warnings
warnings.filterwarnings('ignore')
```

EDA

In [2]:
```
# importing dataset
df = pd.read_csv('waterwell.csv')
df.head()
```

Out[2]:

id	amount_tsh	date_recorded	funder	gps_height	installer	longitude	latitude	wpt_name

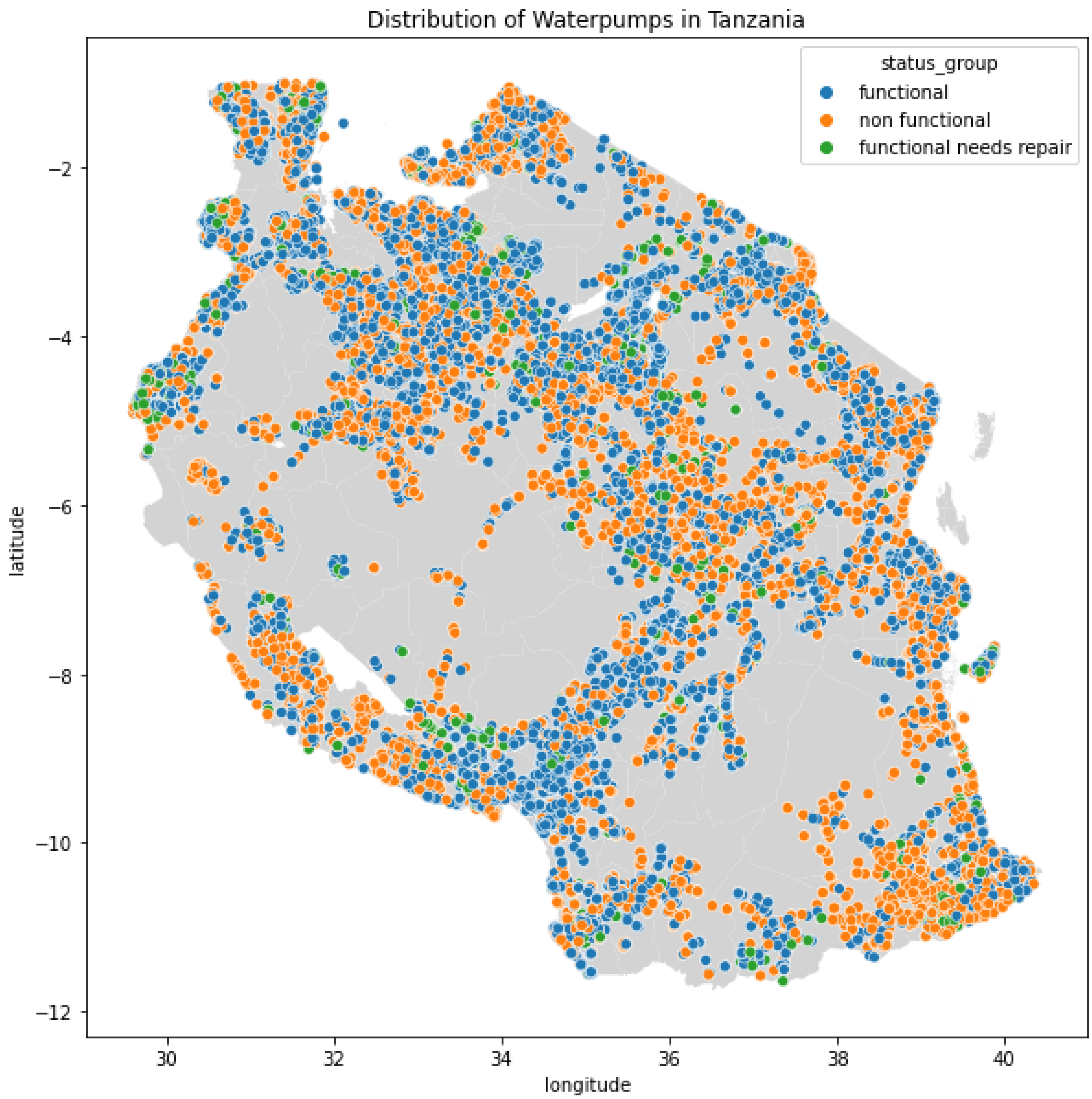
	id	amount_tsh	date_recorded	funder	gps_height	installer	longitude	latitude	wpt_name
0	69572	6000.0	3/14/2011	Roman	1390	Roman	34.938093	-9.856322	none
1	8776	0.0	3/6/2013	Grumeti	1399	GRUMETI	34.698766	-2.147466	Zahanati
2	34310	25.0	2/25/2013	Lottery Club	686	World vision	37.460664	-3.821329	Kwa Mahundi
3	67743	0.0	1/28/2013	Unicef	263	UNICEF	38.486161	-11.155298	Zahanati Ya Nanyumbu
4	19728	0.0	7/13/2011	Action In A	0	Artisan	31.130847	-1.825359	Shuleni

5 rows × 41 columns

Mapping waterpump distribution

In [3]:

```
#create a new df
mapdf = df.copy()

#filter out the longitude values
mapdf = mapdf[mapdf['longitude'] > 0]

#read the shape file with geopandas
tanzania_map = gpd.read_file('Districts and TC as 2020.shp')
# tanzania_map.plot(color='lightgrey',figsize=(8,8)); just look at the map of tanzania

crs = {'init':'EPSG:4326'} #define CRS
geometry = [Point(xy) for xy in zip(mapdf['longitude'], df['latitude'])] #create Points
geo_df = gpd.GeoDataFrame(mapdf,
                          crs = crs,
                          geometry = geometry) #define the geometry df

#plot the data
fig, ax = plt.subplots(figsize = (10,10))
tanzania_map.to_crs(epsg=4326).plot(ax=ax, color='lightgrey')
sns.scatterplot(x="longitude", y="latitude",data=mapdf,hue='status_group',ax=ax);
ax.set_title('Distribution of Waterpumps in Tanzania');
```



Data Understanding

In [4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 59400 entries, 0 to 59399
Data columns (total 41 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   id                     59400 non-null  int64  
 1   amount_tsh             59400 non-null  float64
 2   date_recorded          59400 non-null  object 
 3   funder                 55765 non-null  object 
 4   gps_height             59400 non-null  int64  
 5   installer              55745 non-null  object 
 6   longitude              59400 non-null  float64
 7   latitude               59400 non-null  float64
 8   wpt_name               59400 non-null  object 
 9   num_private            59400 non-null  int64  
 10  basin                  59400 non-null  object 
 11  subvillage             59029 non-null  object 
 12  region                 59400 non-null  object 
```

```
 13  region_code            59400 non-null  int64
 14  district_code          59400 non-null  int64
 15  lga                    59400 non-null  object
 16  ward                   59400 non-null  object
 17  population             59400 non-null  int64
 18  public_meeting         56066 non-null  object
 19  recorded_by            59400 non-null  object
 20  scheme_management      55523 non-null  object
 21  scheme_name            31234 non-null  object
 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
 40  status_group           59400 non-null  object
dtypes: float64(3), int64(7), object(31)
memory usage: 18.6+ MB
```

Just a glance at the features, we can see that some features like `latitude,longitude,date_recorded,water_quality,construction_year` and so on could be important for modelling. On the other hand, features like `wpt_name,scheme_name,lga,ward` can be considered as superflous and can be omitted during modelling. We will make informed decisions based on EDA and `feature_importances_` after we build models.

Checking the datatypes

In [5]:

```
# examining the data types of the df
df.dtypes.value_counts()
```

Out[5]:

```
object     31
int64       7
float64     3
dtype: int64
```

We can see that most of the features are categorical

Checking for Null values

In [6]:

```
# checking for null values and returning it as a pandas series
empty=df.isna().sum()
empty
```

Out[6]:

```
id                          0
amount_tsh                  0
date_recorded               0
funder                   3635
gps_height                  0
```

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

In [7]:

```
#converting the empty series into a dictionary
empty_dict = dict(empty)

#looping thru dictionary to isolate the columns that have null values
empty_list =[]
for key,value in empty_dict.items():
    if value != 0:
        empty_list.append(key)

empty_list
# we now have the list of columns that have null values
```

Out[7]:

```
['funder',
 'installer',
 'subvillage',
 'public_meeting',
 'scheme_management',
 'scheme_name',
 'permit']
```

In [8]:

```
# examining those columns
df_empty = df[empty_list]
df_empty
```

Out[8]:

	funder	installer	subvillage	public_meeting	scheme_management	scheme_name	permit

	funder	installer	subvillage	public_meeting	scheme_management	scheme_name	permit
0	Roman	Roman	Mnyusi B	True	VWC	Roman	False
1	Grumeti	GRUMETI	Nyamara	NaN	Other	NaN	True
2	Lottery Club	World vision	Majengo	True	VWC	Nyumba ya mungu pipe scheme	True
3	Unicef	UNICEF	Mahakamani	True	VWC	NaN	True
4	Action In A	Artisan	Kyanyamisa	True	NaN	NaN	True
...	...	...	...	...	...	...	...
59395	Germany Republi	CES	Kiduruni	True	Water Board	Losaa Kia water supply	True
59396	Cefa-njombe	Cefa	Igumbilo	True	VWC	Ikondo electrical water sch	True
59397	NaN	NaN	Madungulu	True	VWC	NaN	False
59398	Malec	Musa	Mwinyi	True	VWC	NaN	True
59399	World Bank	World	Kikatanyemba	True	VWC	NaN	True

59400 rows × 7 columns

In [9]:
```
df_empty.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 59400 entries, 0 to 59399
Data columns (total 7 columns):
 #   Column             Non-Null Count  Dtype 
---  ------             --------------  ----- 
 0   funder             55765 non-null  object
 1   installer          55745 non-null  object
 2   subvillage         59029 non-null  object
 3   public_meeting     56066 non-null  object
 4   scheme_management  55523 non-null  object
 5   scheme_name        31234 non-null  object
 6   permit             56344 non-null  object
dtypes: object(7)
memory usage: 3.2+ MB
```

We can see that all the columns that have null values are categorical. Also,recall from the original df that total number of rows is `59400`

In [10]:
```
# Looking at scheme_name first since it has the highest number of null values
df['scheme_name'].value_counts()
```

Out[10]:
```
K                       682
None                    644
Borehole                546
Chalinze wate           405
M                       400
                       ... 
MANGISA                   1
```

```
Mradi wa Maji Kitaraka      1
Mradi wa maji Vijini        1
Mongwe r                    1
Mkabenga spring source      1
Name: scheme_name, Length: 2696, dtype: int64
```

Since `scheme_name` has approx. 47% of the data missing, even classifying this as `missing` might skew the analysis. Hence it's best to remove it from the analysis.

In [11]:
```python
# removing 'scheme_name' from the df
df.drop('scheme_name',axis=1,inplace=True)
```

Since, the rest of the columns have approx. only 6% of the data missing, we can either choose to drop it or classify it as `MISSING` for the analysis. Let's classify it as `MISSING` .

In [12]:
```python
#replacing the null values as 'MISSING'
df.fillna('MISSING',inplace=True)
```

In [13]:
```python
# checking the df to see if we still have missing values
df.isna().any()
```

Out[13]:
```
id                       False
amount_tsh               False
date_recorded            False
funder                   False
gps_height               False
installer                False
longitude                False
latitude                 False
wpt_name                 False
num_private              False
basin                    False
subvillage               False
region                   False
region_code              False
district_code            False
lga                      False
ward                     False
population               False
public_meeting           False
recorded_by              False
scheme_management        False
permit                   False
construction_year        False
extraction_type          False
extraction_type_group    False
extraction_type_class    False
management               False
management_group         False
payment                  False
payment_type             False
water_quality            False
quality_group            False
quantity                 False
quantity_group           False
source                   False
source_type              False
source_class             False
waterpoint_type          False
waterpoint_type_group    False
status_group             False
dtype: bool
```

We can see that there no more missing values

Checking for duplicate data

In [14]:
```
#checking for duplicate data based on all the columns
df[df.duplicated()]
```

Out[14]:

id	amount_tsh	date_recorded	funder	gps_height	installer	longitude	latitude	wpt_name	num_pri

0 rows × 40 columns

We can that there are no duplicate data.

Feature Exploration

Let's explore some of the features and see if we can glean some information:

In [15]:
```
df[['payment','payment_type']]
```

Out[15]:

	payment	payment_type
0	pay annually	annually
1	never pay	never pay
2	pay per bucket	per bucket
3	never pay	never pay
4	never pay	never pay
...	...	...
59395	pay per bucket	per bucket
59396	pay annually	annually
59397	pay monthly	monthly
59398	never pay	never pay
59399	pay when scheme fails	on failure

59400 rows × 2 columns

Since they are the same, we can delete either `payment` or `payment_type`

In [16]:
```
#dropping 'payment_type' from the df
df.drop('payment_type',axis=1,inplace=True)
```

In [17]:
```
df[['extraction_type', 'extraction_type_group', 'extraction_type_class']]
```

Out[17]:

	extraction_type	extraction_type_group	extraction_type_class
0	gravity	gravity	gravity
1	gravity	gravity	gravity

	extraction_type	extraction_type_group	extraction_type_class
2	gravity	gravity	gravity
3	submersible	submersible	submersible
4	gravity	gravity	gravity
...	...	...	...
59395	gravity	gravity	gravity
59396	gravity	gravity	gravity
59397	swn 80	swn 80	handpump
59398	nira/tanira	nira/tanira	handpump
59399	nira/tanira	nira/tanira	handpump

59400 rows × 3 columns

Again, these are similar and we can chose to eliminate 2 of them from our analysis

In [18]:
```
df.drop(['extraction_type_group','extraction_type_class'],axis=1,inplace=True)
```

In [19]:
```
df[['management','management_group']]
```

Out[19]:

	management	management_group
0	vwc	user-group
1	wug	user-group
2	vwc	user-group
3	vwc	user-group
4	other	other
...	...	...
59395	water board	user-group
59396	vwc	user-group
59397	vwc	user-group
59398	vwc	user-group
59399	vwc	user-group

59400 rows × 2 columns

We can remove `management_group` since `management` provides more detail.

In [20]:
```
#drop
df.drop('management_group',axis=1,inplace=True)
```

In [21]:
```
df[['source','source_type','source_class']]
```

Out[21]:

	source	source_type	source_class
0	spring	spring	groundwater
1	rainwater harvesting	rainwater harvesting	surface
2	dam	dam	surface
3	machine dbh	borehole	groundwater
4	rainwater harvesting	rainwater harvesting	surface
...	...	...	...
59395	spring	spring	groundwater
59396	river	river/lake	surface
59397	machine dbh	borehole	groundwater
59398	shallow well	shallow well	groundwater
59399	shallow well	shallow well	groundwater

59400 rows × 3 columns

We can choose `source` over the other two features

In [22]:

```
#drop
df.drop(['source_type','source_class'],axis=1,inplace=True)
```

In [23]:

```
df[['water_quality','quality_group','quantity','quantity_group','waterpoint_type','wate
```

Out[23]:

	water_quality	quality_group	quantity	quantity_group	waterpoint_type	waterpoint_type_grou
0	soft	good	enough	enough	communal standpipe	communal standpipe
1	soft	good	insufficient	insufficient	communal standpipe	communal standpipe
2	soft	good	enough	enough	communal standpipe multiple	communal standpipe
3	soft	good	dry	dry	communal standpipe multiple	communal standpipe
4	soft	good	seasonal	seasonal	communal standpipe	communal standpipe
...	...	...	...	...	...	..
59395	soft	good	enough	enough	communal standpipe	communal standpipe
59396	soft	good	enough	enough	communal standpipe	communal standpipe
59397	fluoride	fluoride	enough	enough	hand pump	hand pump
59398	soft	good	insufficient	insufficient	hand pump	hand pump

	water_quality	quality_group	quantity	quantity_group	waterpoint_type	waterpoint_type_group
59399	salty	salty	enough	enough	hand pump	hand pump

59400 rows × 6 columns

In [24]:
```
#dropping quality_group,quantity_group and waterpoint_type_group
df.drop(['quality_group','waterpoint_type_group','quantity_group'],axis=1,inplace=True)
# df.info()
```

In [25]:
```
df[['region','region_code','district_code']]
```

Out[25]:

	region	region_code	district_code
0	Iringa	11	5
1	Mara	20	2
2	Manyara	21	4
3	Mtwara	90	63
4	Kagera	18	1
...	...	...	...
59395	Kilimanjaro	3	5
59396	Iringa	11	4
59397	Mbeya	12	7
59398	Dodoma	1	4
59399	Morogoro	5	2

59400 rows × 3 columns

In [26]:
```
df['public_meeting'].unique()
```

Out[26]:
```
array([True, 'MISSING', False], dtype=object)
```

In [27]:
```
df['recorded_by'].unique()
```

Out[27]:
```
array(['GeoData Consultants Ltd'], dtype=object)
```

In [28]:
```
df['num_private'].unique()
```

Out[28]:
```
array([   0,   39,    5,   45,    6,    3,  698,   32,   15,    7,   25,
        102,    1,   93,   14,   34,  120,   17,  213,   47,    8,   41,
         80,  141,   20,   35,  131,    4,   22,   11,   87,   61,   65,
        136,    2,  180,   38,   62,    9,   16,   23,   42,   24,   12,
        668,  672,   58,  150,  280,  160,   50, 1776,   30,   27,   10,
         94,   26,  450,  240,  755,   60,  111,  300,   55, 1402],
      dtype=int64)
```

In [29]:
```
df[['scheme_management','permit']]
```

Out[29]:

	scheme_management	permit
0	VWC	False
1	Other	True
2	VWC	True
3	VWC	True
4	MISSING	True
...	...	...
59395	Water Board	True
59396	VWC	True
59397	VWC	False
59398	VWC	True
59399	VWC	True

59400 rows × 2 columns

Looks like `scheme_management` has the same info as `management` and hence can be removed.

In [30]:
```
#dropping scheme_management
df.drop('scheme_management',axis=1,inplace=True)
```

In [31]:
```
#dropping id,wpt_name since they will not have a bearing on the analysis
df.drop(['id','wpt_name'],axis=1,inplace=True)
```

In [32]:
```
#check the shape of the df after removing redundant features
df.shape
```

Out[32]: (59400, 28)

Preprocessing

In [33]:
```
#splitting train and test sets to avoid data leakage
X = df.drop('status_group',axis=1)
y = df['status_group']

X_train,X_test,y_train,y_test = train_test_split(X,y,random_state=42)
```

In [34]:
```
#getting the list of column names with numeric data
num_cols = [col for col in X_train.columns if X_train[col].dtypes!='O']
cat_cols = [col for col in X_train.columns if X_train[col].dtypes=='O']
```

Column Transformer

In [35]:
```
# instantiate the transformer to scale the numeric data and encode categorical data
ct = ColumnTransformer([
    ('scale', MinMaxScaler(), num_cols),
    ('encode', OneHotEncoder(sparse=False,handle_unknown='ignore'), cat_cols)
])
```

Since we have a large amount of categorical data, doing OHE would result in an explosion in the size of the dataset. Let's check to see what we're dealing with after OHE

In [36]:
```
df_cat = df[cat_cols]
df_cat_ohe = pd.get_dummies(df_cat)
df_cat_ohe.shape
```

Out[36]:
```
(59400, 26009)
```

Obviously this would not be feasible to work with. We will have to reduce the number of features to make the dataset tractable.

In [37]:
```
#getting the numer of unique values
d ={}
for col in X_train[cat_cols]:
    d[col] = X_train[col].nunique()
#sorting the values in descending order
sort_d = dict( sorted(d.items(),
                           key=lambda item: item[1],
                           reverse=True))
sort_d
```

Out[37]:
```
{'subvillage': 16672,
 'ward': 2071,
 'installer': 1861,
 'funder': 1645,
 'date_recorded': 349,
 'lga': 125,
 'region': 21,
 'extraction_type': 18,
 'management': 12,
 'source': 10,
 'basin': 9,
 'water_quality': 8,
 'payment': 7,
 'waterpoint_type': 7,
 'quantity': 5,
 'public_meeting': 3,
 'permit': 3,
 'recorded_by': 1}
```

We can that some features have very high unique values that would make the dataset unmanageable after OHE. Let's eliminate all features that have 'unique values > 100`

In [38]:
```
cat_cols.remove('subvillage')
cat_cols.remove('ward')
cat_cols.remove('installer')
cat_cols.remove('funder')
cat_cols.remove('date_recorded')
cat_cols.remove('lga')
```

In [39]:
```
#dropping the columns from the train and test sets
X_train.drop(columns=['subvillage','ward','installer','funder','date_recorded','lga'],
             axis=1,inplace=True)
X_test.drop(columns=['subvillage','ward','installer','funder','date_recorded','lga'],
            axis=1,inplace=True)
```

In [40]:
```
#transforming the categrical data to str for the encoder to work
```

```
X_train[cat_cols] = X_train[cat_cols].astype('str')

#fitting the transformer on the train set
ct.fit(X_train)
```

Out[40]:
```
ColumnTransformer(transformers=[('scale', MinMaxScaler(),
                                 ['amount_tsh', 'gps_height', 'longitude',
                                  'latitude', 'num_private', 'region_code',
                                  'district_code', 'population',
                                  'construction_year']),
                                ('encode',
                                 OneHotEncoder(handle_unknown='ignore',
                                               sparse=False),
                                 ['basin', 'region', 'public_meeting',
                                  'recorded_by', 'permit', 'extraction_type',
                                  'management', 'payment', 'water_quality',
                                  'quantity', 'source', 'waterpoint_type'])])
```

In [41]:
```
#creating a df of the transformed train set
X_train = pd.DataFrame(data=ct.transform(X_train))
X_train.head()
```

Out[41]:

	0	1	2	3	4	5	6	7	8	9	...	103	104
0	0.000057	0.138722	0.944941	0.477474	0.0	0.051020	0.0125	0.002623	0.979632	0.0	...	0.0	0.0
1	0.000000	0.022238	0.000000	1.000000	0.0	0.163265	0.0125	0.000000	0.000000	0.0	...	1.0	0.0
2	0.000000	0.022238	0.825683	0.758435	0.0	0.183673	0.0500	0.000000	0.000000	0.0	...	0.0	0.0
3	0.000000	0.566537	0.862136	0.584350	0.0	0.122449	0.0500	0.000754	0.998510	1.0	...	0.0	0.0
4	0.000000	0.206848	0.859110	0.080872	0.0	0.091837	0.0375	0.000033	1.000000	0.0	...	0.0	1.0

5 rows × 113 columns

In [42]:
```
#applying the transformer to the test set
X_test = pd.DataFrame(data=ct.transform(X_test))
X_test.head()
```

Out[42]:

	0	1	2	3	4	5	6	7	8	9	...	103	104
0	0.000000	0.022238	0.792800	0.691285	0.0	0.163265	0.0625	0.000000	0.000000	0.0	...	1.0	0.0
1	0.000000	0.022238	0.813797	0.575521	0.0	0.132653	0.0750	0.000000	0.000000	0.0	...	1.0	0.0
2	0.000029	0.613484	0.879616	0.635858	0.0	0.204082	0.0125	0.004852	0.997516	1.0	...	0.0	1.0
3	0.000000	0.022238	0.821432	0.222333	0.0	0.112245	0.0750	0.000000	0.000000	0.0	...	1.0	0.0
4	0.000143	0.413696	0.848108	0.619679	0.0	0.122449	0.0125	0.007705	0.999006	1.0	...	0.0	0.0

5 rows × 113 columns

Label Encoding

We will employ the `label encoding` method for the target variable.

In [43]:
```
#instantiate
le = LabelEncoder()

#fit on the train set
le.fit(y_train)

#transfomring the train set
y_train = le.transform(y_train)
```

In [44]:
```
#transforming the test set
y_test = le.transform(y_test)
```

We have now compelted the preprocessing steps for the train and sets independently of each other to prevent data leakage

Building baseline models

Logistic Regression

In [45]:
```
#instantiate logistic regression model
logreg = LogisticRegression(random_state=123)

#fit the model onto the train sets
logreg.fit(X_train,y_train)
logreg

#predict values of the model
y_hat_train = logreg.predict(X_train)
y_hat_test = logreg.predict(X_test)

#evaluate model
names = ['functional','non functional','functional needs repair']
f1_logreg = round(f1_score(y_test,y_hat_test,average='macro'),3)
print(classification_report(y_test,y_hat_test,target_names=names))
```

```
                         precision    recall  f1-score   support

             functional       0.74      0.81      0.77      8098
         non functional       0.28      0.29      0.28      1074
functional needs repair       0.75      0.65      0.70      5678

               accuracy                           0.71     14850
              macro avg       0.59      0.58      0.59     14850
           weighted avg       0.71      0.71      0.71     14850
```

Decision Tree

In [46]:
```
#instantiate
clf = DecisionTreeClassifier(criterion='entropy',random_state=123)

#fit the model onto the train sets
clf.fit(X_train,y_train)

#predict
y_hat_train = clf.predict(X_train)
```

```
y_hat_test = clf.predict(X_test)

#evaluate model
f1_tree = round(f1_score(y_test,y_hat_test,average='macro'),3)
print(classification_report(y_test,y_hat_test,target_names=names))
```

```
                         precision    recall  f1-score   support

             functional       0.79      0.78      0.78      8098
         non functional       0.35      0.37      0.36      1074
functional needs repair       0.74      0.75      0.74      5678

               accuracy                           0.74     14850
              macro avg       0.63      0.63      0.63     14850
           weighted avg       0.74      0.74      0.74     14850
```

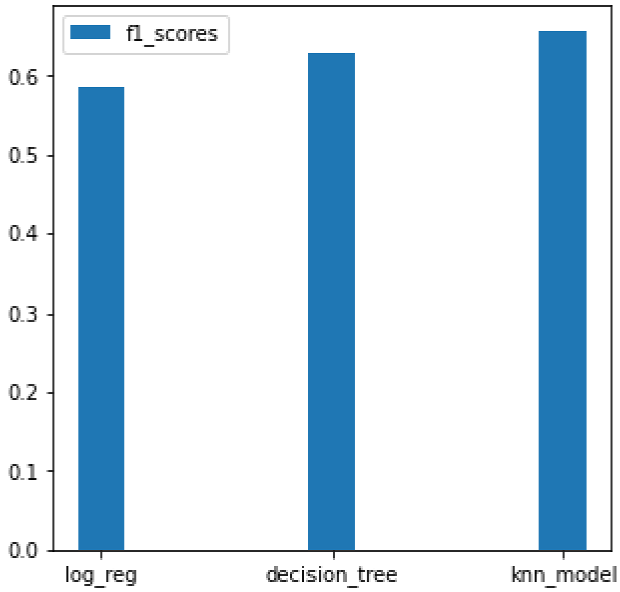
KNN model

In [47]:
```
#instantiate
knn_baseline_model = KNeighborsClassifier()

#fit onto the data
knn_baseline_model.fit(X_train,y_train)

#predict
y_hat_train = knn_baseline_model.predict(X_train)
y_hat_test = knn_baseline_model.predict(X_test)

#evaluate model
f1_knn = round(f1_score(y_test,y_hat_test,average='macro'),3)
print(classification_report(y_test,y_hat_test,target_names=names))
```

```
                         precision    recall  f1-score   support

             functional       0.78      0.86      0.82      8098
         non functional       0.48      0.32      0.38      1074
functional needs repair       0.80      0.73      0.76      5678

               accuracy                           0.77     14850
              macro avg       0.69      0.64      0.66     14850
           weighted avg       0.77      0.77      0.77     14850
```

Selecting a model

In [48]:
```
#visualizing baseline model performance
fig,ax=plt.subplots(figsize=(5,5))
plt.bar(x=['log_reg','decision_tree','knn_model'],height=[f1_logreg,f1_tree,f1_knn],lab
       width=0.2);
plt.legend();
```



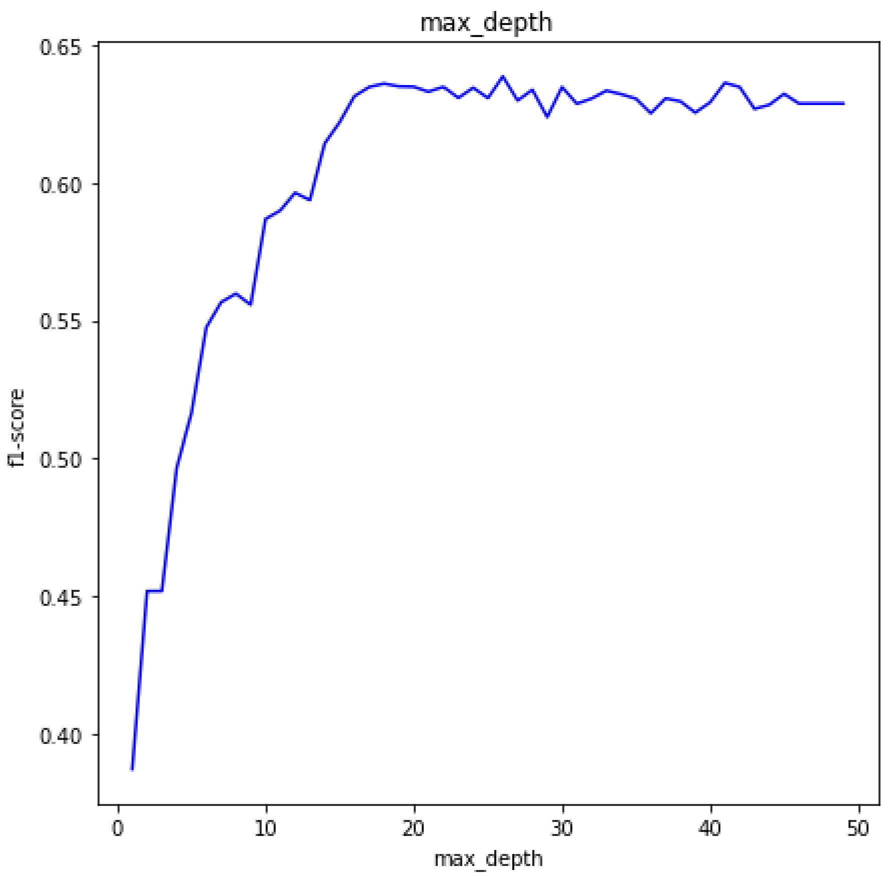
We can see that the `KNN` baseline model has the highest performance score among the three baseline models. But since,computation time is large for `KNN` and the performance gain in only marginal between `KNN` and `DecisionTree` , let's go with `DecisionTree` .

Hyperparameter Tuning

Max_depth

In [49]:
```
#creating a list of depth values
max_depth = np.arange(1,50)

#initiate dict to store the score and the depth value
f1_scores=[]
scores_depth = {}

# create a loop for the classifier to run with the different depth values
for depth in max_depth:

    #instantiate
    classifier = DecisionTreeClassifier(criterion='entropy',max_depth=depth,random_stat

    #fit the model
    classifier.fit(X_train,y_train)

    #predict values
    y_hat_test = classifier.predict(X_test)

    #calculate
    f1 = f1_score(y_test,y_hat_test,average='macro')

    #append test_scores list
    f1_scores.append(f1)

    #adding to the dict
    scores_depth[f1] = depth

#sort the dict
scores_depth = dict(sorted(scores_depth.items(),
```

```
                          key=lambda item:item[0],
                          reverse=True))

#visualize the data
import matplotlib.pyplot as plt
fig,ax = plt.subplots(figsize=(7,7))
ax.plot(max_depth,f1_scores,c='b')
ax.set_xlabel('max_depth')
ax.set_ylabel('f1-score')
ax.set_title('max_depth')
plt.show()
```



In [50]:

```
max_depth= list(scores_depth.items())[0][1]
print(f'f1-score is the highest at a depth value of {max_depth}')
```

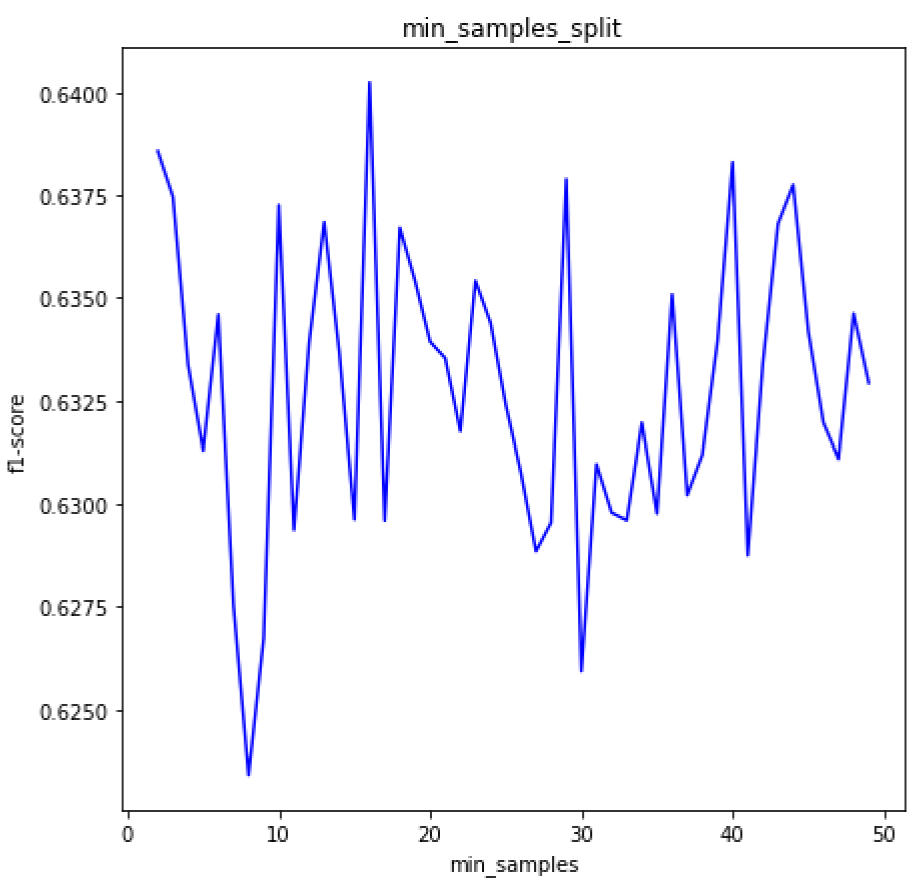
```
f1-score is the highest at a depth value of 26
```

Min_samples_split

In [51]:

```
#use the optimum value of depth
max_depth=max_depth

#define a range of min_samples_for each split
min_samples_range = np.arange(2,50)

#create a loop with the optimum depth and different min_samples
#creating an empty list to store scores for each split and sample
f1_scores = []
scores_split = {}

# create a loop for the classifier to run with the different depth values
for sample in min_samples_range:
```

```
    #instantiate
    classifier = DecisionTreeClassifier(criterion='entropy',max_depth=max_depth,min_sam

    #fit the model
    classifier.fit(X_train,y_train)

    #predict values
    y_hat_test = classifier.predict(X_test)

    #calculate
    f1 = f1_score(y_test,y_hat_test,average='macro')

    #add the scores to the list
    f1_scores.append(f1)

    #adding to the dict
    scores_split[f1] = sample

#sort the dict
scores_split = dict(sorted(scores_split.items(),
                          key=lambda item:item[0],
                          reverse=True))

#visualize
fig,ax = plt.subplots(figsize=(7,7))
ax.plot(min_samples_range,f1_scores,c='b')

ax.set_xlabel('min_samples')
ax.set_ylabel('f1-score')
ax.set_title('min_samples_split')
plt.show();
```

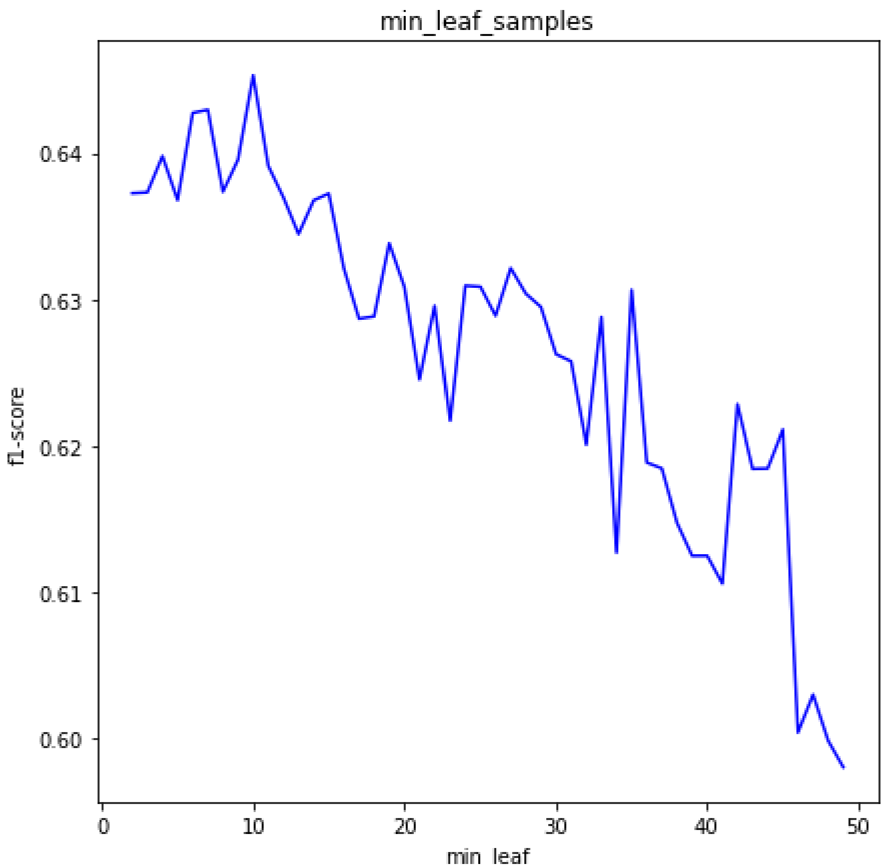


```
In [52]: min_samples_split= list(scores_split.items())[0][1]
         print(f'f1-score is the highest at a depth value of {min_samples_split}')
```

```
f1-score is the highest at a depth value of 16
```

Min_samples_leaf

```
In [53]: #use the optimum value of depth,min_samples_split
         depth = max_depth
         min_samples_split = min_samples_split

         #define a range of min_samples_for each split
         min_leaf_range = np.arange(2,50)

         #create a loop with the optimum depth and different min_samples
         #creating an empty list to store scores for each depth
         f1_scores = []
         scores_leaf = {}

         # create a loop for the classifier to run with the different depth values
         for sample in min_leaf_range:

             #instantiate
             classifier = DecisionTreeClassifier(criterion='entropy',max_depth=depth,min_samples_
                                                  min_samples_leaf=sample, random_state=123)

             #fit the model
             classifier.fit(X_train,y_train)

             #predict values
             y_hat_test = classifier.predict(X_test)

             #calculate
             f1 = f1_score(y_test,y_hat_test,average='macro')

             #add the scores to the list
             f1_scores.append(f1)

              #adding to the dict
             scores_leaf[f1] = sample


         #sort the dict
         scores_leaf = dict(sorted(scores_leaf.items(),
                                   key=lambda item:item[0],
                                   reverse=True))

         #visual
         fig,ax = plt.subplots(figsize=(7,7))
         ax.plot(min_leaf_range,f1_scores,c='b')

         ax.set_xlabel('min_leaf')
         ax.set_ylabel('f1-score')
         ax.set_title('min_leaf_samples')

         plt.show();
```



In [54]:
```
min_samples_leaf= list(scores_leaf.items())[0][1]
print(f'f1-score is the highest at a depth value of {min_samples_leaf}')
```

```
f1-score is the highest at a depth value of 10
```

Model with optimized parameters

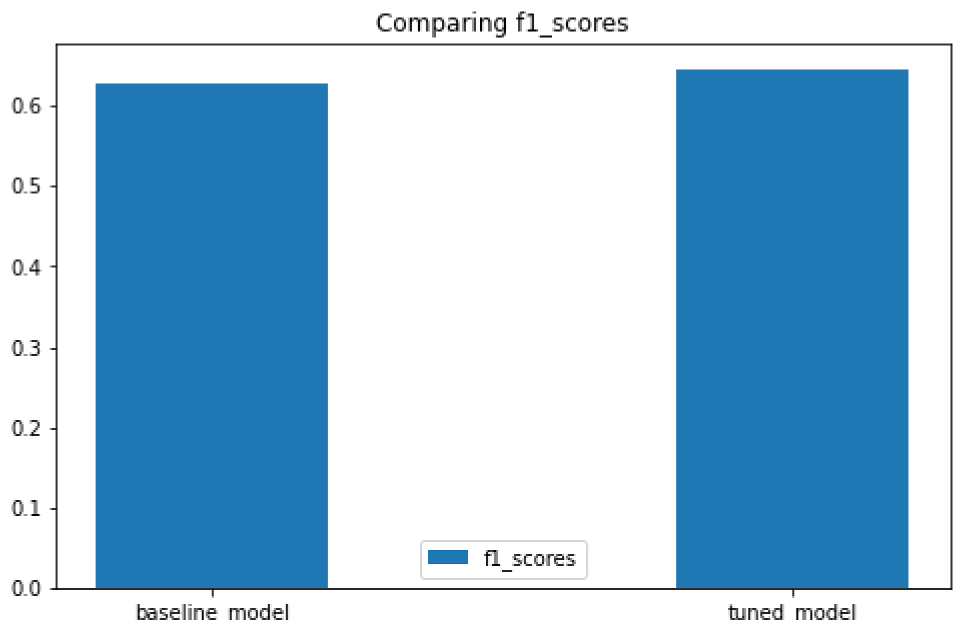
In [55]:
```
#run the model with the optimized parameters
max_depth=max_depth
min_samples_split=min_samples_split
min_samples_leaf=min_samples_leaf

#instantiate
classifier = DecisionTreeClassifier(criterion='entropy',max_depth=max_depth,min_samples_
                                    min_samples_leaf=min_samples_leaf,random_state=123)

#fit the model
classifier.fit(X_train,y_train)

#predict values
y_hat_train = classifier.predict(X_train)
y_hat_test = classifier.predict(X_test)


print('TEST SCORES')
print('-----------------------------------------------------------')
print(classification_report(y_test,y_hat_test,target_names=names))

f1_score_optimized_train = f1_score(y_train,y_hat_train,average='macro')
f1_score_optimized_test = f1_score(y_test,y_hat_test,average='macro')
```

```
TEST SCORES
```

```
----------------------------------------------------------
                         precision    recall  f1-score   support

            functional       0.78      0.85      0.81      8098
        non functional       0.48      0.30      0.37      1074
functional needs repair       0.78      0.73      0.76      5678

              accuracy                           0.77     14850
             macro avg       0.68      0.63      0.65     14850
          weighted avg       0.76      0.77      0.76     14850
```

In [56]:

```python
#comparing baseline model with the model with tuned hyperparameters
fig,ax=plt.subplots(figsize=(8,5))
ax.bar(x=['baseline_model','tuned_model'],height=[f1_tree,f1_score_optimized_test],labe
      width=0.4);
ax.set_title('Comparing f1_scores');
ax.legend();
```



We can see that there is only marginal improvement

Feature_importance

In [57]:

```python
#creating a df with just feature_importance data
df_importance = pd.DataFrame({'features':X_train.columns,
                              'importance':classifier.feature_importances_})

df_importance
```

Out[57]:

	features	importance
0	0	0.032682
1	1	0.050008
2	2	0.139233
3	3	0.126083
4	4	0.000029

	features	importance
...	...	...
108	108	0.021166
109	109	0.000000
110	110	0.002658
111	111	0.002251
112	112	0.081298

113 rows × 2 columns

In [58]:

```
#sorting the importance in ascending order
df_importance_sorted = df_importance.sort_values(by=['importance'],ascending=True)
df_importance_sorted.head()
```

Out[58]:

	features	importance
56	56	0.0
63	63	0.0
19	19	0.0
30	30	0.0
33	33	0.0

Let's build a model based on the top_10 features to see if we can get better results

Extracting top_10 features

In [59]:

```
#create new train and test sets with the top_10 features alone
#get top10 features as a df
cols = df_importance_sorted['features'].tail(10)
cols=pd.DataFrame(data=cols)
cols
```

Out[59]:

	features
108	108
6	6
0	0
7	7
1	1
8	8
112	112
3	3
2	2

	features
91	91

In [60]:
```
#df with top10 features for train and test sets
X_train_top10 =X_train[list(cols['features'])]
X_test_top10 =X_test[list(cols['features'])]
```

Decision Tree with the top10 features

GridSearch CV

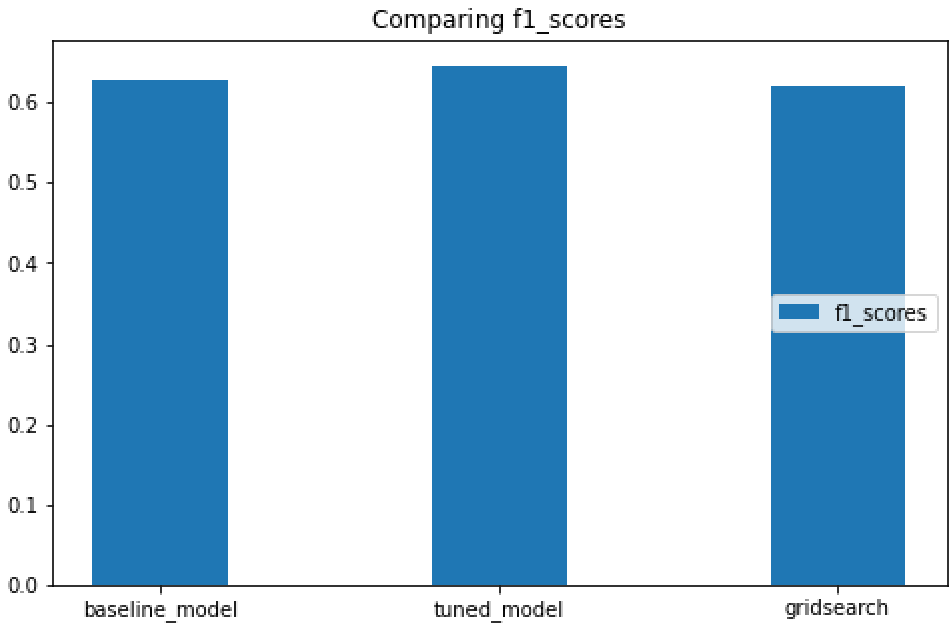
Rather than use the earlier approach, where we built a baseline model and then tuned each hyper parameter seperately, we can combine all these steps into one using `GridSearchCV`

In [61]:
```
#instantiate
clf_top10 = DecisionTreeClassifier(random_state=123)

#define the parameter grid
#constraining size based on tuned model params to reduce computation time
param_grid = {'max_depth': np.arange(max_depth, max_depth+5),
              'min_samples_split': np.arange(min_samples_split,min_samples_split+5),
              'min_samples_leaf': np.arange(min_samples_leaf,min_samples_leaf+5)
             }

#instantiate
gs_tree = GridSearchCV(estimator=clf_top10,param_grid=param_grid,cv=5)

#fit
gs_tree.fit(X_train_top10,y_train)

#predict
gs_train = gs_tree.predict(X_train_top10)
gs_test = gs_tree.predict(X_test_top10)

print('TEST SCORES')
print('--------------------------------------------------')
print(classification_report(y_test,gs_test))

f1_score_gs_tree_train = f1_score(y_train,gs_train,average='macro')
f1_score_gs_tree_test = f1_score(y_test,gs_test,average='macro')
```

```
TEST SCORES
--------------------------------------------------
              precision    recall  f1-score   support

           0       0.76      0.85      0.81      8098
           1       0.48      0.23      0.31      1074
           2       0.77      0.71      0.74      5678

    accuracy                           0.76     14850
   macro avg       0.67      0.60      0.62     14850
weighted avg       0.75      0.76      0.75     14850
```

In [62]:
```
#visualizing scores
#comparing baseline model with the model with tuned hyperparameters
fig,ax=plt.subplots(figsize=(8,5))
```

```
ax.bar(x=['baseline_model','tuned_model','gridsearch'],height=[f1_tree,f1_score_optimiz
       label='f1_scores',width=0.4);
ax.set_title('Comparing f1_scores');
ax.legend(loc=7);
```



We can see that the model performance has regressed using just the top10 features. Hence, we can nix that approach.

Random Forest

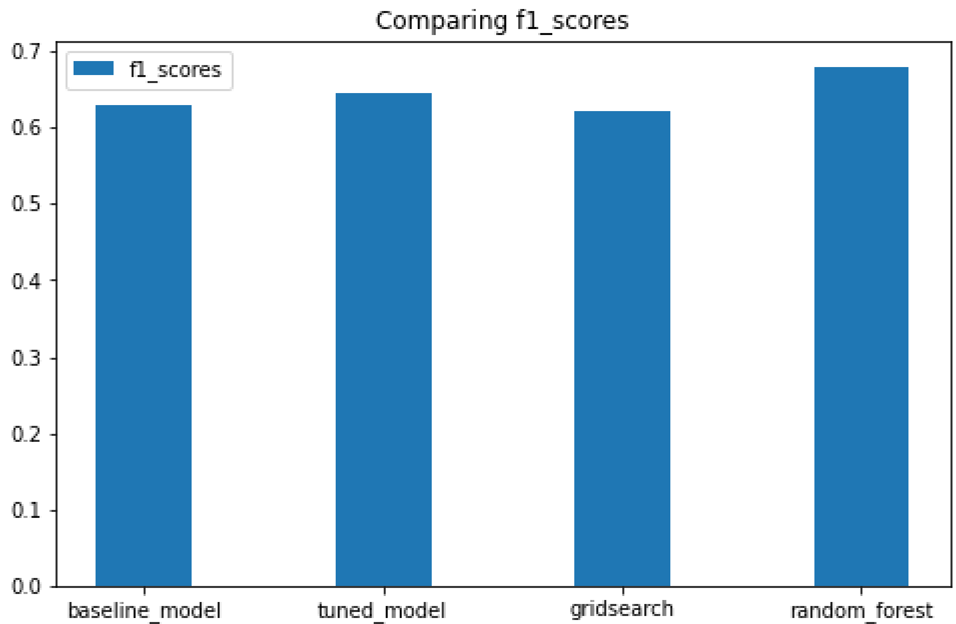
Let's use a popular ensemble method called Random Forest to see if we can make improvements. Random Forest combines Bootstrapping and Sub-Space Sampling methods to build models that are robust and immune to noise in the data.

In [63]:
```
#instantiate the classifier
forest =RandomForestClassifier(n_estimators=100,criterion='entropy')

#fit the data
forest.fit(X_train,y_train)

#predict
y_hat_train = forest.predict(X_train)
y_hat_test = forest.predict(X_test)

print('TEST SCORES')
print('------------------------------------------------------------')
print(classification_report(y_test,y_hat_test,target_names=names))

f1_score_forest_test = f1_score(y_test,y_hat_test,average='macro')
```

```
TEST SCORES
----------------------------------------------------------
                         precision    recall  f1-score   support

             functional       0.80      0.89      0.84      8098
         non functional       0.54      0.31      0.40      1074
functional needs repair       0.84      0.76      0.80      5678
```

```
    accuracy                           0.80     14850
   macro avg       0.73      0.66      0.68     14850
weighted avg       0.79      0.80      0.79     14850
```

In [64]:
```
#visualize scores
fig,ax=plt.subplots(figsize=(8,5))
ax.bar(x=['baseline_model','tuned_model','gridsearch','random_forest'],
       height=[f1_tree,f1_score_optimized_test,f1_score_gs_tree_test,f1_score_forest_te
       label='f1_scores',width=0.4);
ax.set_title('Comparing f1_scores');
ax.legend();
```

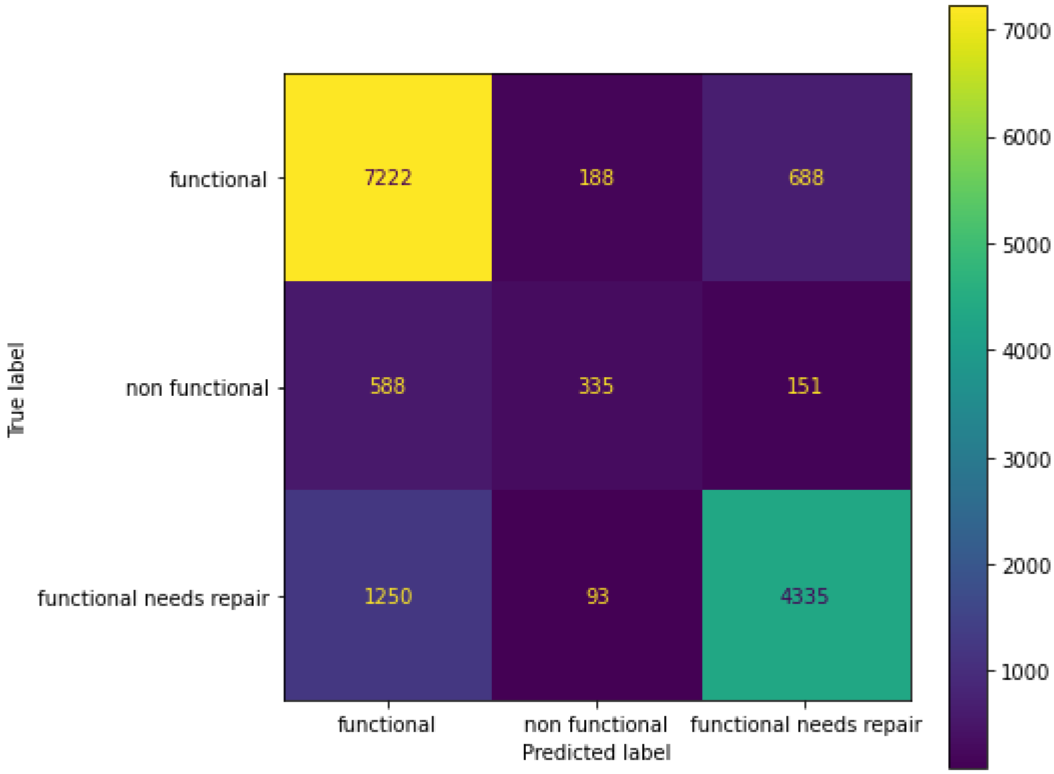


Again, the gains are only marginal

Confusion matrix

The Confusion Matrix is a graphical representation of `[TP,FP,TN,FN]` values. While the f1-score helps us evaluate model performance, we cannot ascertain how to move forward. By using the confusion matrix, we can the see the performance of the model across the different classes and get a clearer picture of model performance

In [65]:
```
#plotting the confusion matrix for the tuned model since that has the highest f1-score
fig,(ax1)=plt.subplots(figsize=(7,7))
plot_confusion_matrix(forest,X_test,y_test,ax=ax1,display_labels=names);
```



We can clearly see that the `functional needs repair` is a problem for the model. Pumps that `need repair` are being classified as `functional`. While the model does reasonably well in comparision to the other two classes, the `functional needs repair` class clearly outweighs the other two classes in poor performance.

Examining the target feature

In [66]:
```
#examining the target variable
df['status_group'].value_counts()
```

Out[66]:
```
functional                 32259
non functional             22824
functional needs repair     4317
Name: status_group, dtype: int64
```

It is very evident that there is a class imbalance. We can try the following approaches to address this imbalance:

1. We can reclassify the `functional needs repair` class as `non_functional`. This way we have more data as `non functional` and this could mitigate some of the imbalance.
2. We can eliminate the `functional needs repair` class completely from the dataset.

Let's move forward using approach 1

Creating a new dataset

In [67]:
```
#reclassifying the values in the  dataset
df1 = df.copy()
df1['status_group'] = df1['status_group'].replace('functional needs repair','non functi
df1['status_group'].value_counts()
```

```
Out[67]: functional        32259
         non functional    27141
         Name: status_group, dtype: int64
```

Preprocessing steps as before

```
In [68]: #splitting the train and test sets
         X=df1.drop('status_group',axis=1)
         y=df1['status_group']

         X_train,X_test,y_train,y_test = train_test_split(X,y,random_state=123)
```

```
In [69]: #getting the list of column names with numeric and categorical data
         #getting the list of column names with numeric data
         num_cols = [col for col in X_train.columns if X_train[col].dtypes!='O']
         cat_cols = [col for col in X_train.columns if X_train[col].dtypes=='O']
```

```
In [70]: #dropping cat columns for a easier computation like before
         cat_cols.remove('subvillage')
         cat_cols.remove('ward')
         cat_cols.remove('installer')
         cat_cols.remove('funder')
         cat_cols.remove('date_recorded')
         cat_cols.remove('lga')
```

```
In [71]: X_train.drop(columns=['subvillage','ward','installer','funder','date_recorded','lga'],
                      axis=1,inplace=True)
         X_test.drop(columns=['subvillage','ward','installer','funder','date_recorded','lga'],
                     axis=1,inplace=True)
```

```
In [72]: #transforming the categrical data to str for the encoder to work
         X_train[cat_cols] = X_train[cat_cols].astype('str')
```

```
In [73]: # instantiate the transformer to scale the numeric data and encode categorical data
         ct = ColumnTransformer([
             ('scale', MinMaxScaler(), num_cols),
             ('encode', OneHotEncoder(sparse=False,handle_unknown='ignore'), cat_cols),
         ])
```

```
In [74]: #transforming the train and test sets
         ct.fit(X_train)
         X_train = pd.DataFrame(data=ct.transform(X_train))
         X_test = pd.DataFrame(data=ct.transform(X_test))
```

```
In [75]: #label encode the target variable
         #instantiate
         le = LabelEncoder()

         #fit on the train set
         le.fit(y_train)

         #transfomring the train set
         y_train = le.transform(y_train)
         y_test = le.transform(y_test)
```

Decision Tree on the new dataset

In [76]:
```
names = ['functional','non functional']

#instantiate
clf = DecisionTreeClassifier(criterion='entropy',random_state=123)

#fit the model onto the train sets
clf.fit(X_train,y_train)

#predict
y_hat_train = clf.predict(X_train)
y_hat_test = clf.predict(X_test)

#evaluate model
f1_balanced = round(f1_score(y_test,y_hat_test,average='macro'),3)
print(classification_report(y_test,y_hat_test,target_names=names))
```

```
                precision    recall  f1-score   support

    functional       0.79      0.78      0.79      7963
non functional       0.75      0.76      0.75      6887

      accuracy                           0.77     14850
     macro avg       0.77      0.77      0.77     14850
  weighted avg       0.77      0.77      0.77     14850
```

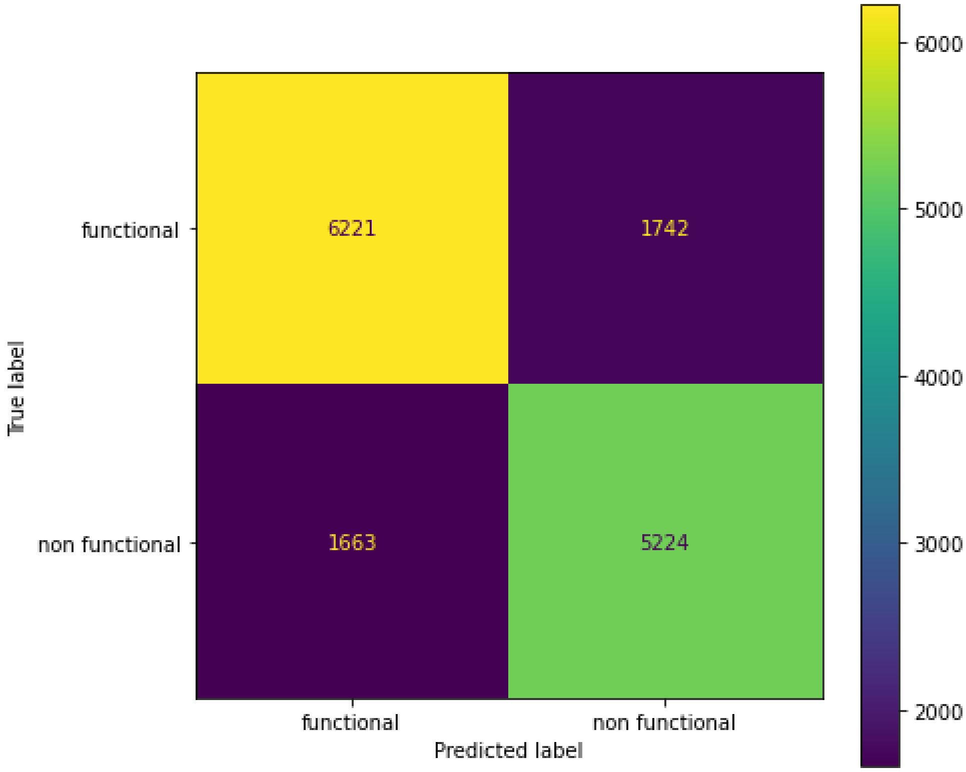
In [77]:
```
#comparing the different models
fig,ax=plt.subplots(figsize=(8,5))
ax.bar(x=['baseline_model','tuned_model','gridsearch','random_forest','reclassified_set
       height=[f1_tree,f1_score_optimized_test,f1_score_gs_tree_test,f1_score_forest_te
       label='f1_scores',width=0.4);
ax.set_title('Comparing f1_scores');
ax.legend();
```

We can see that the model has improved significantly by using the reclassified data.

Confusion Matrix

In [78]:

```
#confusion matrix
fig,(ax1)=plt.subplots(figsize=(7,7))
plot_confusion_matrix(clf,X_test,y_test,ax=ax1,display_labels=names);
```



From the confusion matrix, we can see that there is still room for improvement

GridSearchCV

Let's do a GridSearch on the model to optimize. We'll take a conservative approach while defining the parameters to reduce computation time and base them from our previous model

In [79]:

```
#instantiate
clf = DecisionTreeClassifier(criterion='entropy',random_state=123)

#define the parameter grid
#constraining size based on tuned model params to reduce computation time
param_grid = {'max_depth': np.arange(25,30),
              'min_samples_split': np.arange(15,20),
              'min_samples_leaf': np.arange(10,15)
             }

#instantiate
gs_tree = GridSearchCV(estimator=clf,param_grid=param_grid,cv=5)

#fit
gs_tree.fit(X_train,y_train)

#predict
gs_test = gs_tree.predict(X_test)

print('TEST SCORES')
print('--------------------------------------------------')
```

```python
print(classification_report(y_test,gs_test))

f1_score_gs = f1_score(y_test,gs_test,average='macro')
```

```
TEST SCORES
--------------------------------------------------
              precision    recall  f1-score   support

           0       0.77      0.84      0.81      7963
           1       0.79      0.72      0.75      6887

    accuracy                           0.78     14850
   macro avg       0.78      0.78      0.78     14850
weighted avg       0.78      0.78      0.78     14850
```

By runnning the model with parameters obtained earlier, we can see that gain is only marginal there by suggesting that more tuning is required

Combining GridSearch and RandomForest

In [80]:

```python
# #define the parameter grid
# #constraining size based on tuned model params to reduce computation time
# param_grid = {'max_depth': np.arange(25,30),
#               'min_samples_split': np.arange(15,20),
#               'min_samples_leaf': np.arange(10,15)
#              }

# #instantiate the classifier
# forest =RandomForestClassifier(n_estimators=100,criterion='entropy')
# gs = GridSearchCV(estimator=forest,param_grid=param_grid,cv=5)

# #fit the data
# gs.fit(X_train,y_train)

# #predict
# y_hat_test = gs.predict(X_test)

# print('TEST SCORES')
# print('-----------------------------------------------------------')
# print(classification_report(y_test,y_hat_test,target_names=names))

# f1_score_grid = f1_score(y_test,y_hat_test,average='macro')
```

Not running due to extended running time

Next Steps

1. Hyperparameters should be tuned for the new dataset optimization.
2. We can combine GrdSearchCV and RandomForest and evaluate performance
3. Since we only used some the categorical features for our model, we can selectively add more features and check for performance.