MMA 869 FINAL PROJECT
PUMP IT UP

- Team Kipling

01
Background
&
Purpose

Problems and Background

4 Million

Tanzanian people lack **safe water**

30 Million

Tanzanian people don't have the access to **improved sanitation**

50% of

Tanzanian people don't have **clean drinking water**

Photo Source: https://www.thenorthernecho.co.uk/news/14205315.event-provide-clean-water-africa/

Purpose

Goal?

Support Tanzanian communities to maintain safe water access

How?

Develop predictive models to determine the overall functionality of a waterpoint

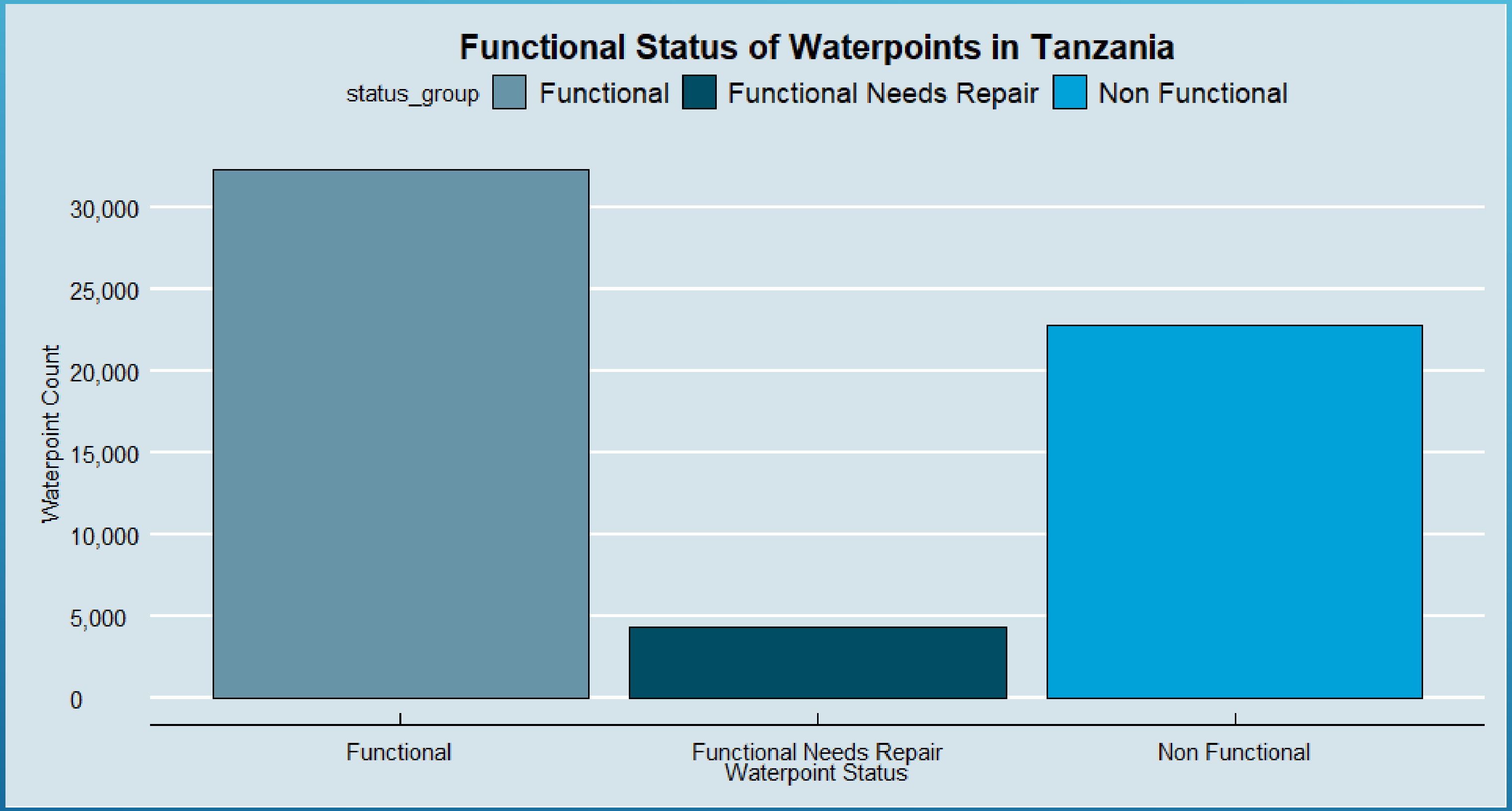
Benefits?

Improve maintenance operations & ensure that potable water is available to communities in Tanzania

02
Exploratory Data Analysis

EDA – Target Class Split

Current state of Waterpoints in Tanzania

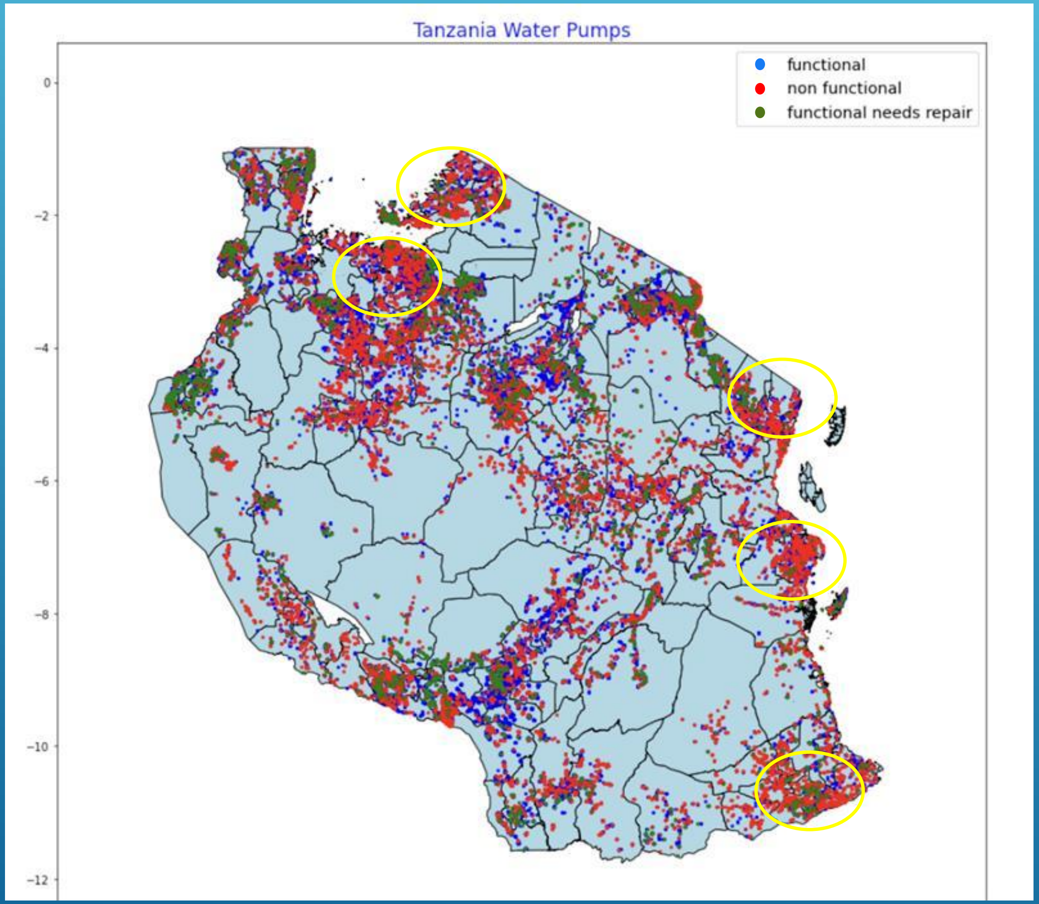


Insights:

- 59,400 total waterpoints in Tanzania
- **45%** of total waterpoints (27,141) either non functioning or in need of repair

EDA – Population Map & Summary Statistics

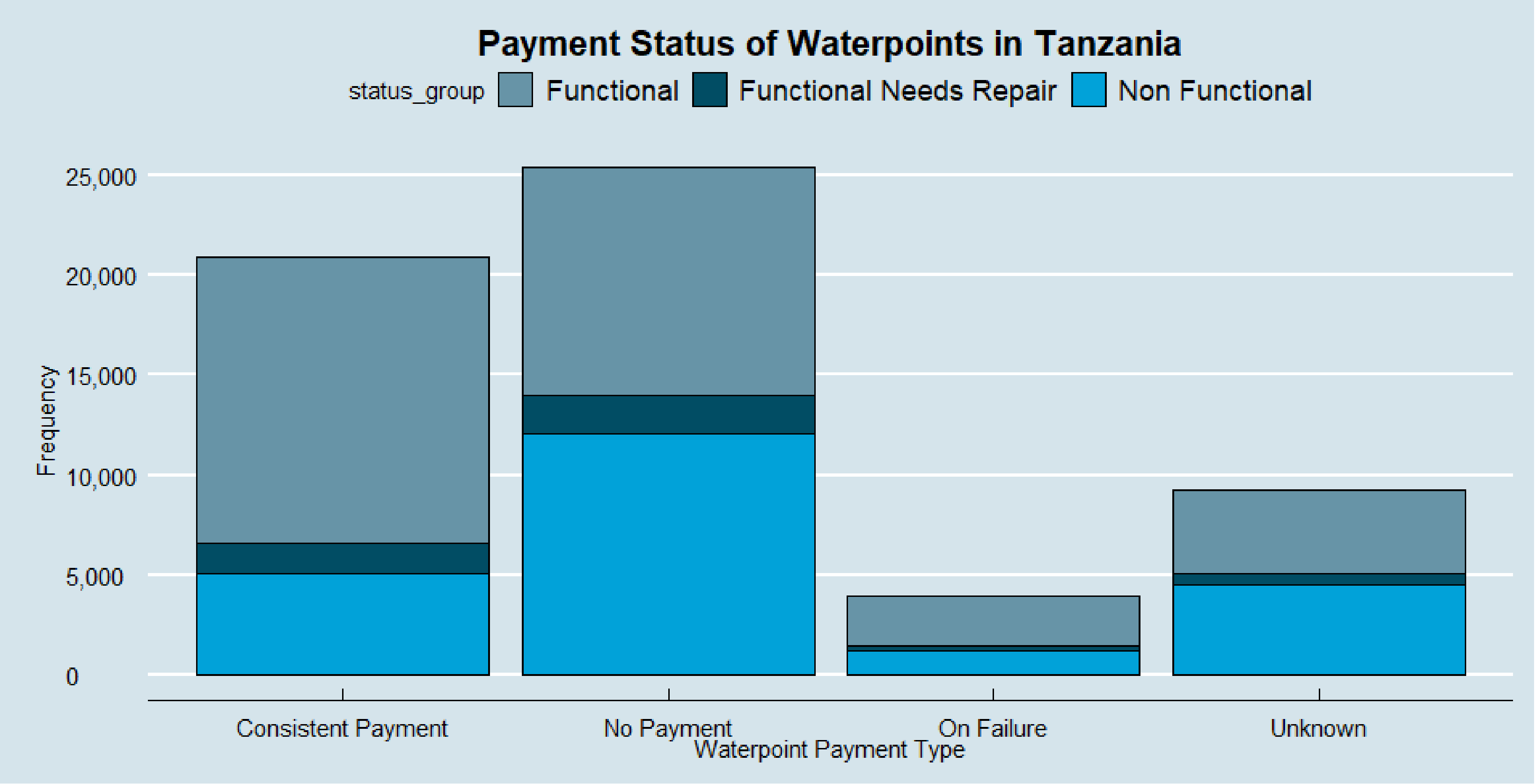
Distribution of Pumps Across Tanzania



Regions with relatively more non-functional pumps:

- Mara, Mwanza, Tanga, Pwani, and Mtwara

EDA – Payments



Insights:

- Majority of Waterpoints are not paid for
- It is also evident that the unpaid Waterpoints have a higher % non functional than any other type
- Waterpoints consistently paid for are the most functional

- **24% Non Functional for Consistent Payment vs 47.6% for No Payment**

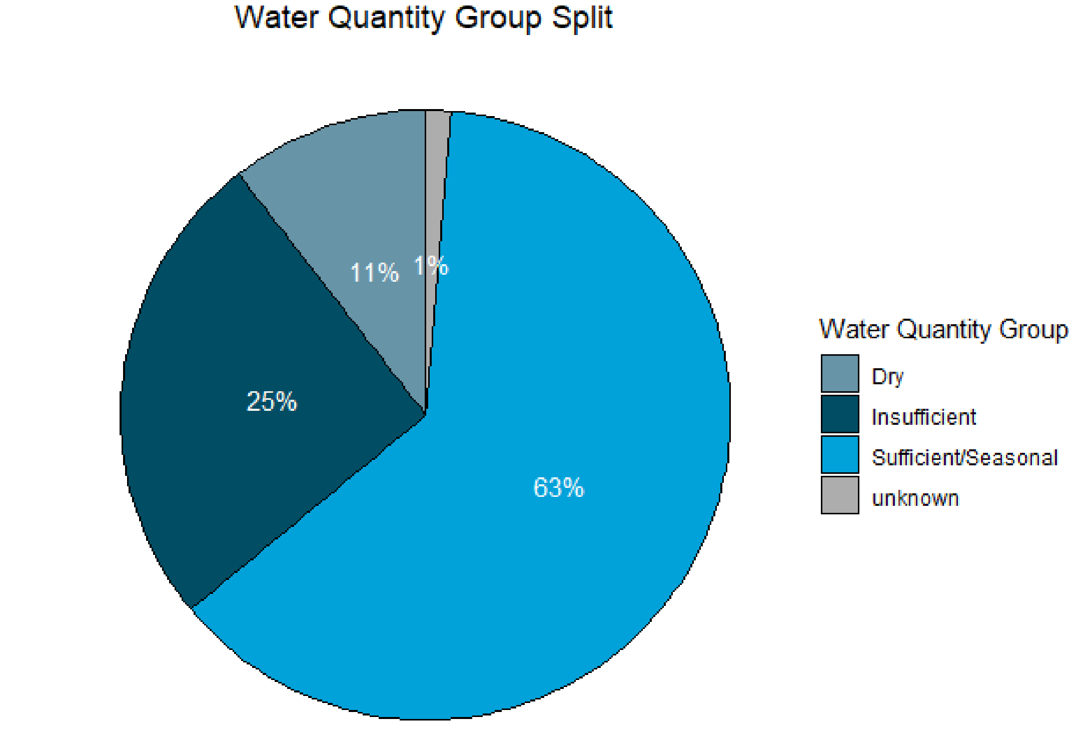
EDA – Water Quantity & Quality

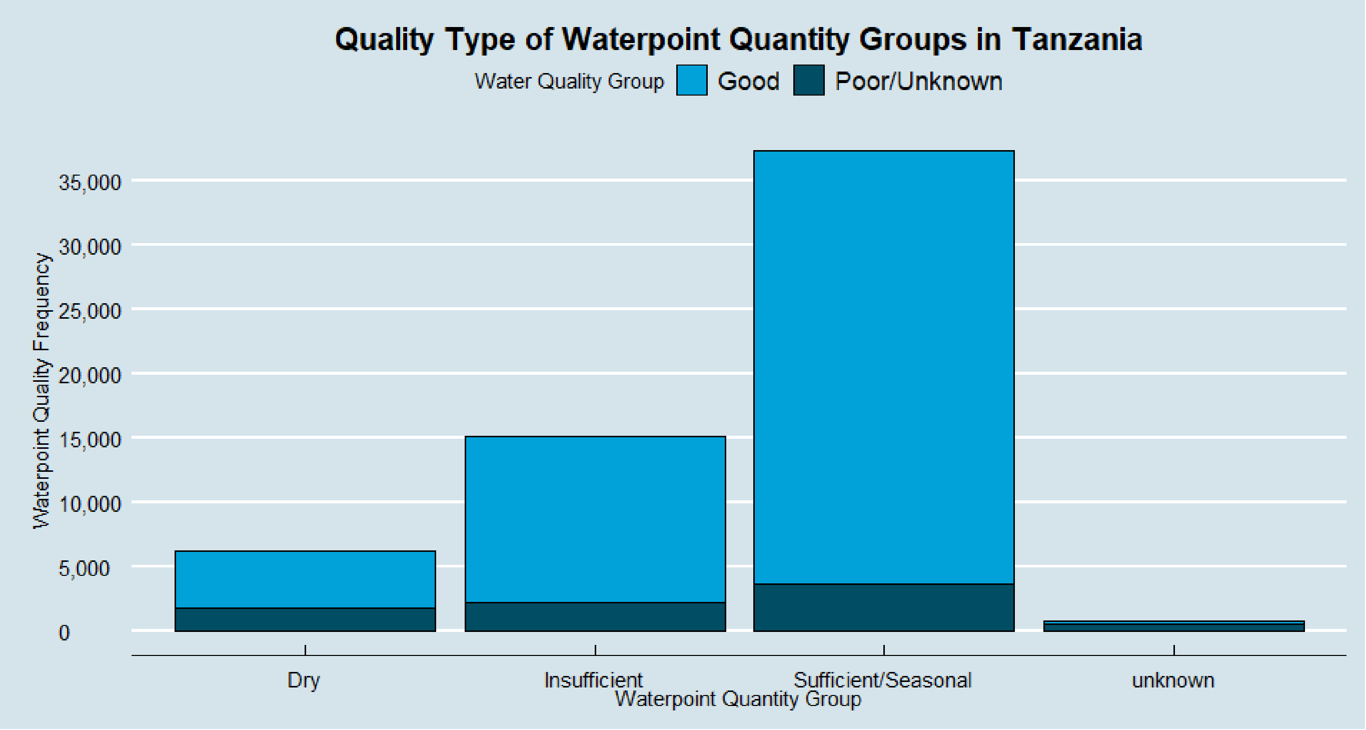
Quantity:

- More than 63% of pumps are sufficient of water
- More than 36% of pumps are either dry or insufficient of water

Quality:

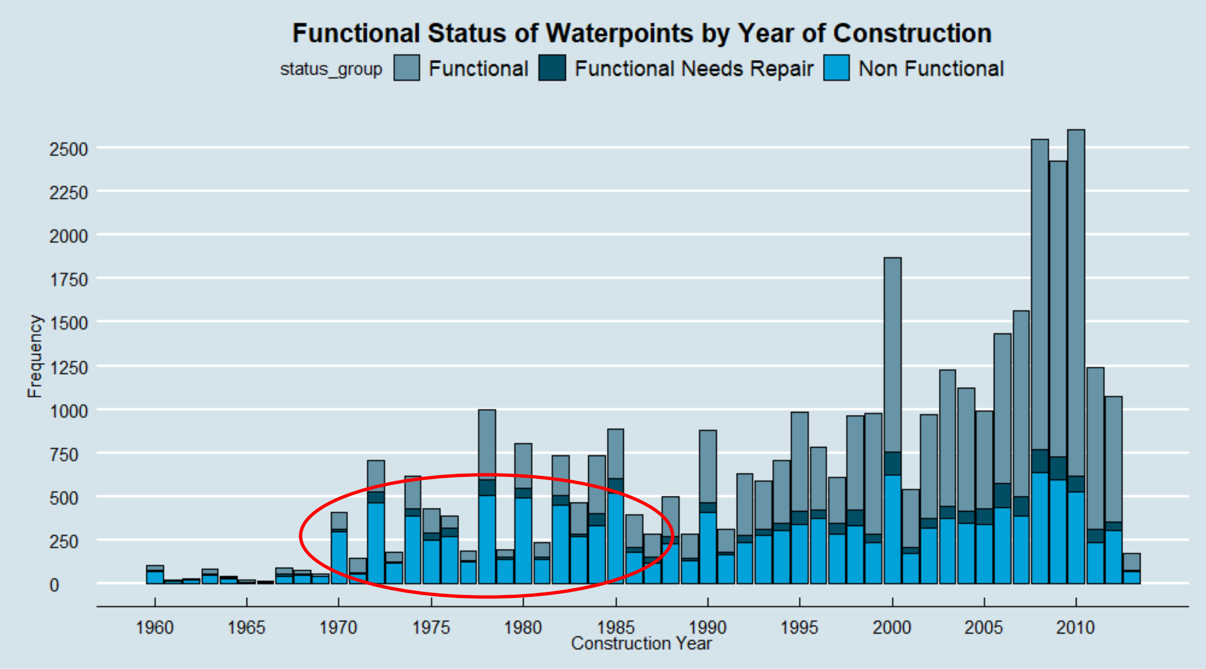
- 90% of Water Quality is Good where Water Quantity is Sufficient
- Poor Water Quality where Quantity is not up to par





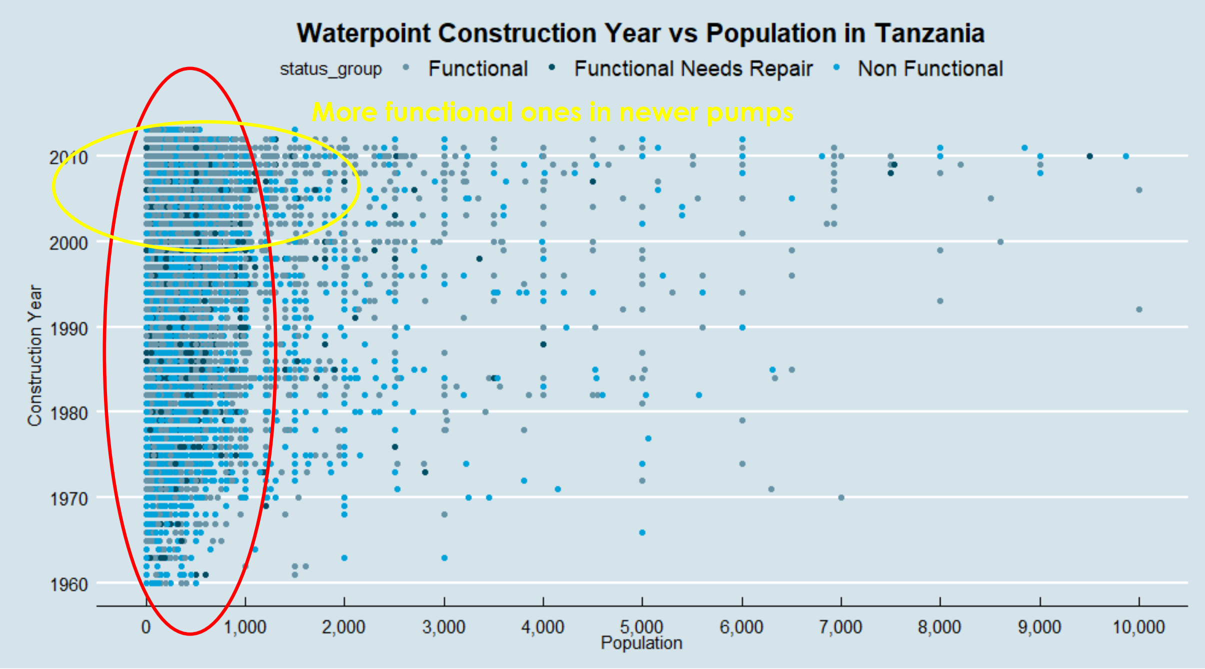
EDA – Construction Year

Does Age affect Functional Status?



- Large amount of pumps built from the 2000's onwards with a big spike from 2008-2010
- Newer pumps (built in 2000's) are far more likely to be functional than those built pre 2000

How does Population Factor in?



- Most pumps serve for less populated places
- Similar to prior graph, grey highlighting in top left shows that newer pumps are more functional than older ones

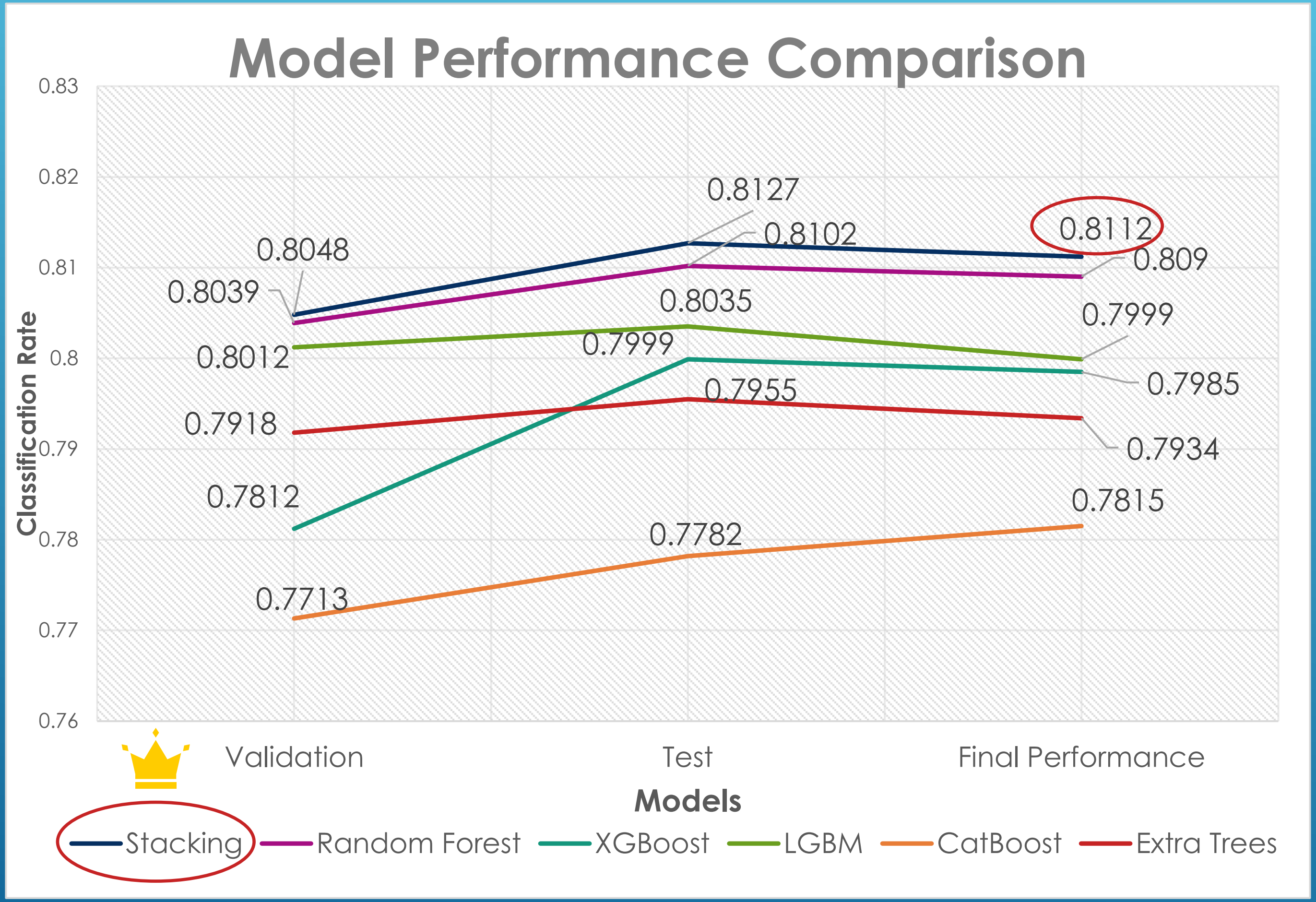
03
Modelling Process
&
Results

Hot Shot Kipling’s Guide to Data Preprocessing

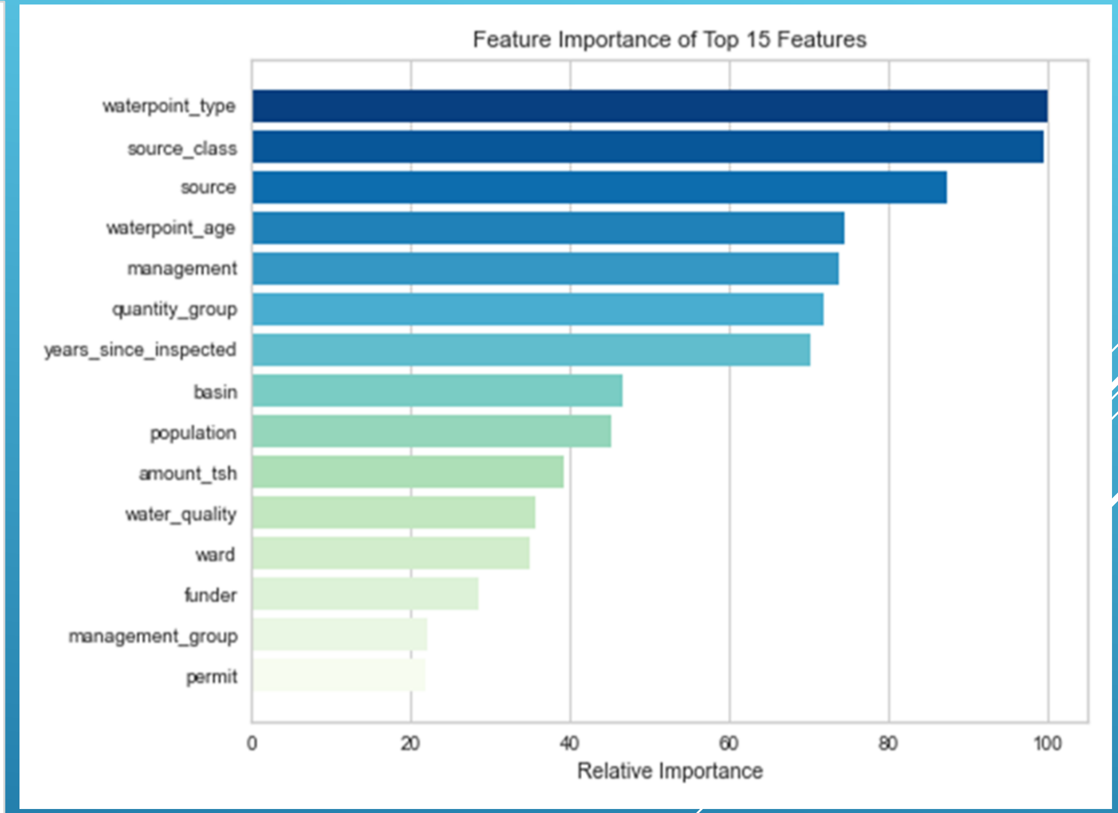
Feature Engineering	
Remove redundant features	😎
Reduce cardinality of features	😎
Feature Creation	
Waterpoint Age	😎
Payment Indicator	😎
Sufficient & Good Water Quality	😎
Above Average Population Indicator	😎
Transformation	
Iterative Imputation	😎
Normalization	😎
M-Estimate Encoding	😎
Failures	
Dimensionality Reduction	😟
Feature Selection	😟
Clustering	😟

Modelling Process – Model Comparison & Learnings

Model Comparison



Feature Importance



Modelling Process – Best Model Performance

Classification Report

Class	Precision	Recall	F1-Score	Support
Functional (0)	0.80	0.90	0.85	3,199
Functional Needs Repair (1)	0.85	0.78	0.81	2,297
Non Functional (2)	0.64	0.32	0.43	444
---------------	--------------	-----------	-------------	-------------
Accuracy			0.81	5,940
Macro Avg	0.76	0.67	0.70	5,940
Weighted Avg (Micro)	0.81	0.81	0.80	5,940

Confusion Matrix

	Predicted Class		
Truth Class	**2,892**	246	61
	485	**1,793**	19
	236	65	**143**

04
Future Efforts
&
Conclusion

Next Steps – What can improve this effort in the future?

Improve Quality of Data

- Reach out to Tanzanian Ministry of Water to improve data collection efforts (ie reduce missing values)
- Enables more focus on features we know are important

Advanced Modeling Procedure

- Lack of computational power has been an obstacle for our team
- Use of higher-powered machines

Consult Industry Experts

- This would broaden our understanding of the data and enable us to build more insightful features

Conclusion – Total Impact

Economic Impact

- Assumption: Each functioning Waterpoint can help generate $2,000 USD in economic activity for a village:
  - $500 USD to fix a Waterpoint

Our model can generate an additional $1,795,000 USD for the 5,940 communities in our test group

Social Impact

- Based on our final model, it's ability to detect Waterpoints in need of repair would enable 535,000+ Tanzanians access to clean, sustainable water


Thank you

Appendix – Cleaning Steps

Binning

```
Name: subvillage, dtype: int64

[1653]: #Lets bin values from above under 50

        subvillage_under_50 = combined_df['subvillage'].value_counts().loc[lambda x: x<=100].index.tolist()
        len(subvillage_under_50)

[1653]: 21389

[1654]: #Function to bin values <50
        combined_df['subvillage'] = combined_df['subvillage'].apply(lambda x: 'other' if x in subvillage_under_50 else x)
        combined_df['subvillage'].value_counts()

[1654]: other        66663
        Shuleni        646
        Majengo        631
        Madukani       629
        Kati           467
        Mtakuja        322
        Sokoni         294
```

- Goal: Reduce cardinality and have no more than ~50 levels per feature
- This same process was repeated for the following variables:
    - Construction_year, region_code, district_code, wpt_name. lga, ward, scheme_management, extraction_type, source, num_private,

Appendix – Cleaning Steps

Replacing “0” values with NaN

```
[1657]: #Converting 0 vals into NA for future imputation
combined_df['construction_year'] = combined_df['construction_year'].replace({0:np.nan})
combined_df['waterpoint_age'] = combined_df['waterpoint_age'].replace({0:np.nan})

print('Total NA vals', combined_df['construction_year'].isnull().sum())
print(combined_df['construction_year'].value_counts(dropna=False))
```

```
Total NA vals 25969
NaN       25969
2010.0     3314
2008.0     3243
2009.0     3196
2000.0     2578
2007.0     1960
```

- We believe a construction year of 0 does not make logical sense
- Therefore, we changed 0 to NaN and imputed with Pipeline during modeling process
- The same logic was also applied to the below features:
  - Amount_tsh, gps_height, longitude

Appendix – Feature Creation

Creating “waterpoint_age” Feature

```
[1656]: #Adding in feature to represent age of Waterpoint Type

combined_df['waterpoint_age'] = combined_df['construction_year'].apply(lambda x: 2015 - x if x > 0 else x)

combined_df[['waterpoint_age', 'construction_year']]
```

[1656]:

	waterpoint_age	construction_year
0	16	1999
1	5	2010
2	6	2009
3	29	1986
4	0	0
...	...	...
74245	27	1988
74246	21	1994
74247	5	2010
74248	6	2009
74249	7	2008

74250 rows × 2 columns

- Represents the age in years of the waterpoint
- Assuming data collection was completed in 2015

Appendix – Feature Creation

Creating "good_qual_sufficient" Feature

```
[1704]: #Creating another feature indicating if an instance has enough water and good water quality

        def good_qual_sufficient(df):
            if df['quantity_group'] == 'enough' and df['quality_group'] == 'good':
                return 1
            else:
                return 0
```

```
[1705]: combined_df['good_qual_sufficient'] = combined_df.apply(good_qual_sufficient, axis = 1)

        print(combined_df[['good_qual_sufficient', 'quantity_group', 'quality_group']])

       good_qual_sufficient quantity_group quality_group
0                         1         enough          good
1                         0   insufficient          good
2                         1         enough          good
3                         0            dry          good
4                         0       seasonal          good
...                     ...            ...           ...
74245                     1         enough          good
74246                     0   insufficient         salty
74247                     0   insufficient          good
74248                     0   insufficient          good
74249                     0            dry          good

[74250 rows x 3 columns]
```

- Binary (1/0) indicator representing if an instance has both sufficient water quantity and good water quality

Appendix – Feature Creation

Creating “consistent_payment” Feature

```
[1696]: #Creating a payment_type option to indicate a consistent income stream

        def consistent_income(df):
            if df['payment_type'] == 'monthly' or df['payment_type'] == 'annually' or df['payment_type'] == 'per bucket':
                return 1
            else:
                return 0

        combined_df['consistent_payment'] = combined_df.apply(consistent_income, axis =1)

[1697]: combined_df['consistent_payment'] = combined_df['consistent_payment'].astype('object')

        print(combined_df[['consistent_payment', 'payment_type']])

              consistent_payment payment_type
        0                      1     annually
        1                      0    never pay
        2                      1   per bucket
        3                      0    never pay
        4                      0    never pay
        ...                  ...          ...
        74245                  0    never pay
        74246                  1     annually
        74247                  0    never pay
        74248                  0    never pay
        74249                  0    never pay

        [74250 rows x 2 columns]
```

- Binary (1/0) indicator representing if an instance pays “consistently” for their water
- This includes payment types “monthly”, “annually” and “per bucket”

Appendix – Feature Creation

Creating "used_pump" Feature

```
[1687]: #Creating feature that indicates whether or not a pump was used

        def used_pump(df):
            if df['extraction_type_class'] == 'handpump' or df['extraction_type_class'] == 'motorpump' or df['extraction_type_class'] == 'rope pump':
                return 1
            else:
                return 0

        combined_df['used_pump'] = combined_df.apply(used_pump, axis = 1)
```

```
[1688]: #Changing to type object
        combined_df['used_pump'] = combined_df['used_pump'].astype('object')
        print(combined_df[['used_pump', 'extraction_type_class']])
```

```
      used_pump extraction_type_class
0             0               gravity
1             0               gravity
2             0               gravity
3             0           submersible
4             0               gravity
...         ...                   ...
74245         1             motorpump
74246         1              handpump
74247         0               gravity
74248         0               gravity
74249         0               gravity
```

- Binary (1/0) indicator representing if an instance uses a pump to extract their water

Appendix – Feature Creation

Creating “above_avg_pop” Feature

```
[1735]: #Lets create an indicator if the population size is above average
        def above_avg_pop(df):
            mean = 179.909983
            if df['population'] >= mean:
                return 1
            else:
                return 0

        combined_df['above_avg_pop'] = combined_df.apply(above_avg_pop, axis =1)

[1736]: combined_df['above_avg_pop'] = combined_df['above_avg_pop'].astype('object')

        combined_df[['above_avg_pop', 'population']]
```

[1736]:

	above_avg_pop	population
0	0	109.0
1	1	280.0
2	1	250.0
3	0	58.0

- Binary (1/0) indicator representing if an instance is above/below the national average sub village population

Appendix – Data type Conversion

```
#Converting all numeric categories to numeric dtype
combined_df['amount_tsh'] = pd.to_numeric(combined_df['amount_tsh'])
combined_df['gps_height'] = pd.to_numeric(combined_df['gps_height'])
combined_df['longitude'] = pd.to_numeric(combined_df['longitude'])
combined_df['latitude'] = pd.to_numeric(combined_df['latitude'])
combined_df['population'] = pd.to_numeric(combined_df['population'])
combined_df['years_since_inspected'] = pd.to_numeric(combined_df['years_since_inspected'])
```

- Converted above features to type “numeric” for use in upcoming predictive models

Appendix – Data type Conversion

```
#Doing the same for categorical variables
for col in combined_df.columns:
    if combined_df[col].dtype == 'object':
        combined_df[col] = combined_df[col].astype('category')
```

- Converting remaining feature dtypes to “category” to make it easier to decipher between levels of categorical variables
- Same thought process as converting to factor in R

Appendix – Removing Redundant & Unimportant Features

```
[1742]: #Attempting to drop some columns w issues that might improve performance


#As per 07/28
def remove(df):
        cols = ['id', 'num_private_binned', 'wpt_name',
          'recorded_by', 'subvillage', 'scheme_name_2', 'extraction_type', 'extraction_type_class']
        for x in cols:
            del df[x]
        return df


#(original as per 07/27 working condition)
#def remove(df):
#        cols = ['id', 'amount_tsh', 'funder', 'installer', 'num_private_binned', 'wpt_name',
#          'recorded_by', 'subvillage', 'scheme_name_2', 'region', 'extraction_type_group']
#        for x in cols:
#            del df[x]
#        return df
```

- Above features were removed due to redundancy to other features or lack of feature importance
- Other features dropped: source_type, waterpoint_type_group, num_private

Appendix – Removing Redundant Features

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 59400 entries, 0 to 59399
Data columns (total 35 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   amount_tsh             17761 non-null  float64
 1   funder                 55765 non-null  object 
 2   gps_height             38962 non-null  float64
 3   installer              55745 non-null  object 
 4   longitude              57588 non-null  float64
 5   latitude               59400 non-null  float64
 6   basin                  59400 non-null  object 
 7   region                 59400 non-null  object 
 8   region_code            59400 non-null  object 
 9   district_code          59400 non-null  object 
 10  lga                    59400 non-null  object 
 11  ward                   59400 non-null  object 
 12  population             38019 non-null  float64
 13  public_meeting         56066 non-null  object 
 14  scheme_management      55523 non-null  object 
 15  permit                 56344 non-null  object 
 16  construction_year      38691 non-null  object 
 17  extraction_type_group  59400 non-null  object 
 18  management             59400 non-null  object 
 19  management_group       59400 non-null  object 
 20  payment_type           59400 non-null  object 
 21  water_quality          59400 non-null  object 
 22  quality_group          59400 non-null  object 
 23  quantity_group         59400 non-null  object 
 24  source                 59400 non-null  object 
 25  source_class           59400 non-null  object 
 26  waterpoint_type        59400 non-null  object 
 27  status_group           59400 non-null  float64
 28  waterpoint_age         38691 non-null  float64
 29  years_since_inspected  59400 non-null  int64  
 30  used_pump              59400 non-null  int64  
 31  consistent_payment     59400 non-null  int64  
 32  water_quant_class      59400 non-null  object 
 33  good_qual_sufficient   59400 non-null  int64  
 34  above_avg_pop          59400 non-null  int64  
dtypes: float64(7), int64(5), object(23)
memory usage: 15.9+ MB
```

- All remaining/created features that are used in the predictive models

Appendix – Train/Validation/Test Split

```
[1757]: #Lets view the splits of our data in a df
        #This should give us appropriate data to train, validate and test on

        splits = pd.DataFrame({'Train': [X_train.shape[0]], 'Validation': [X_val.shape[0]],
                               'Test': [X_test.shape[0]]})

        splits
```

[1757]:

	Train	Validation	Test
0	47520	5940	5940

```
[1758]: #Now lets check y vals to ensure they have been stratified
        #Train Y
        #Good
        round(y_train.value_counts().apply(lambda x: x/47520),3)

[1758]: 0.0    0.543
        1.0    0.384
        2.0    0.073
        Name: status_group, dtype: float64

[1759]: #Validation Y
        #Good
        round(y_val.value_counts().apply(lambda x: x/5940),3)

[1759]: 0.0    0.548
        1.0    0.382
        2.0    0.071
        Name: status_group, dtype: float64

[1760]: #Test Y
        #Good
        round(y_test.value_counts().apply(lambda x: x/5940),3)

[1760]: 0.0    0.539
        1.0    0.387
        2.0    0.075
        Name: status_group, dtype: float64
```

- Split into train/validation/test sets
- Split ratio: 80%/10%/10%
  - This was because we felt dataset was robust enough to limit size of validation/test sets and retain more for training
- Ensured y dataset maintained imbalance
  - We dealt with imbalance in the models themselves by enabling class_weights = 'balanced'

Appendix – Package Details

```
#Lets start by creating a pipeline and our first classifier

#from imblearn.pipeline import Pipeline
from imblearn.pipeline import make_pipeline, Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder, OrdinalEncoder
from sklearn.model_selection import GridSearchCV, cross_validate
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.compose import make_column_transformer
from sklearn.compose import make_column_selector as selector
from sklearn.ensemble import RandomForestClassifier
from imblearn.over_sampling import SMOTE
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score, f1_score
import category_encoders as ce
from sklearn.decomposition import PCA
from sklearn.feature_selection import VarianceThreshold
from sklearn.feature_selection import RFE
```

- These were some of the original packages that we used for our models
  - In addition to Pandas, Numpy, Matplotlib and Seaborn as used in the EDA section
- Additional packages used:
  - **Models**
  - XGBClassifier, ExtraTrees Classifier, Catboost Classifier, FLAML (AutoML) scipy.cluster, sklearn.cluster Kmeans, DBSCan, Agglomerative Clustering
  - **Metrics**
  - Sklearn silhouette_score flaml.sklearn_metric_loss-score
  - **Other**
  - Scikitplot, yellowbrick ROCAUC, yellowbrick Feature Importance, ce.CatBoostEncoder, ce.MEstimateEncoder

Appendix – Splitting Features into Lists

```
[1766]: #Pulling a fresh list of numeric features as we made some changes during EDA
        numeric_feats = X.select_dtypes(include='number').columns.tolist()
        print(numeric_feats)
        print(len(numeric_feats))

['amount_tsh', 'gps_height', 'longitude', 'latitude', 'population', 'waterpoint_age', 'years_since_inspected', 'used_
pump', 'consistent_payment', 'good_qual_sufficient', 'above_avg_pop']
11
```

```
[1767]: categorical_feats = X.select_dtypes(exclude='number').columns.tolist()
        print(categorical_feats)
        print(len(categorical_feats))

['funder', 'installer', 'basin', 'region', 'region_code', 'district_code', 'lga', 'ward', 'public_meeting', 'scheme_m
anagement', 'permit', 'construction_year', 'extraction_type_group', 'management', 'management_group', 'payment_type',
'water_quality', 'quality_group', 'quantity_group', 'source', 'source_class', 'waterpoint_type', 'water_quant_class']
23
```

- Above are the final lists created to store both numeric and categorical features
- This makes it easier to transform features in upcoming Pipeline

Appendix – Creating a Pipeline

```
#Create numeric and categorical pipelines
numeric_pipeline = Pipeline(steps=[
    ('impute', IterativeImputer()),
    ('scale', StandardScaler())])

categorical_pipeline_2 = Pipeline(steps=[
    ('mestimate_encode', ce.MEstimateEncoder()),
    ('impute', SimpleImputer(strategy='most_frequent'))])

#Creating Full Pipeline
full_processor_2 = ColumnTransformer(transformers=[
    ('categorical', categorical_pipeline_2, categorical_feats),
    ('numeric', numeric_pipeline, numeric_feats)
])
```

- Above is the best performing pipeline we put together
- Normalization and iterative imputation applied to numeric features
- MEstimate encoding and mode imputation applied to categorical features
- Most notably, this iteration includes MEstimate encoding
    - Significant performance upgrade when used compared to Catboost Encoding and One Hot encoding

Appendix – Modeling Details: Best Model

- Top overall learner:
  - FLAML Stacked Classifier consistent of LGBM, XGBoost, Catboost, Random Forest and Extra Trees
- Below are the parameters of the FLAML automl settings:

```
[1862]: #Now trying FLAML
        from flaml import AutoML

        automl = AutoML()
```

```
[1863]: automl_settings = {
            'time_budget':300,
            'metric': 'macro_f1',
            'task': 'classification',
            'estimator_list' : ['lgbm', 'xgboost', 'catboost', 'rf', 'extra_tree'],
            'eval_method': ['cv'],
            'n_splits': 5,
        }

        #ensemble = True
        #Can also try running each model individually ie estimator_list just lgbm and then put each of them into a stacking
        automl.fit(X_train_transformed, y_train, **automl_settings, ensemble=True)
```

Appendix – Modeling Details: Best Model

- Within the stacking classifier, below are the hyperparameters of the best individual performing model:

```
[1864]: #Viewing Performance
print('Best ML leaner:', automl.best_estimator)
print('Best hyperparmeter config:', automl.best_config)
print('Best accuracy on validation data: {0:.4g}'.format(1-automl.best_loss))
print('Training duration of best run: {0:.4g} s'.format(automl.best_config_train_time))
```

```
Best ML leaner: xgboost
Best hyperparmeter config: {'n_estimators': 197.0, 'max_leaves': 340.0, 'min_child_weight': 1.2481290195206753, 'lear
ning_rate': 0.18085024148873766, 'subsample': 0.8413048297641477, 'colsample_bylevel': 0.44908511189495054, 'colsampl
e_bytree': 0.6204654035998071, 'reg_alpha': 0.11100569389257602, 'reg_lambda': 1.7669866785090698, 'FLAML_sample_siz
e': 47520}
Best accuracy on validation data: 0.6852
Training duration of best run: 82.11 s
```

Appendix – Modeling Details: Best Model

```
[1883]: #Computing Metrics
from flaml.ml import sklearn_metric_loss_score
from flaml.ml import multi_class_curves, norm_confusion_matrix
print('accuracy','=', 1- sklearn_metric_loss_score('accuracy', automl_preds_1, y_val))
print('Macro F1 Score','=', 1- sklearn_metric_loss_score('macro_f1', automl_preds_1, y_val))
print(classification_report(y_val, automl_preds_1))
print(confusion_matrix(y_val, automl_preds_1))
```

```
accuracy = 0.8048821548821549
Macro F1 Score = 0.6719394898808764
              precision    recall  f1-score   support

         0.0       0.80      0.90      0.85      3253
         1.0       0.85      0.77      0.80      2268
         2.0       0.54      0.28      0.37       419

    accuracy                           0.80      5940
   macro avg       0.73      0.65      0.67      5940
weighted avg       0.80      0.80      0.80      5940

[[2929  253   71]
 [ 504 1736   28]
 [ 246   57  116]]
```

```
[1885]: #Now Predict on test set to see if generalizes well
#Very similar to results in validation set; generalized well
automl_preds_test_1 = automl.predict(X_test_transformed)
automl_preds_test_probab = automl.predict_proba(X_test_transformed)

print('accuracy','=', 1- sklearn_metric_loss_score('accuracy', automl_preds_test_1, y_test))
print('Macro F1 Score','=', 1- sklearn_metric_loss_score('macro_f1', automl_preds_test_1, y_test))
print(classification_report(y_test, automl_preds_test_1))
print(confusion_matrix(y_test, automl_preds_test_1))
```

```
accuracy = 0.8127946127946128
Macro F1 Score = 0.6975634211558234
              precision    recall  f1-score   support

         0.0       0.80      0.90      0.85      3199
         1.0       0.85      0.78      0.81      2297
         2.0       0.64      0.32      0.43       444

    accuracy                           0.81      5940
   macro avg       0.76      0.67      0.70      5940
weighted avg       0.81      0.81      0.80      5940

[[2892  246   61]
 [ 485 1793   19]
 [ 236   65  143]]
```

- Stacking Classifier Predictions on Validation/Test data sets:
- Generalized well from Validation to Test:
  - In fact, performance was slightly higher than expected on Test, and even stronger than expected when submitted to the competition

Appendix – Modeling Details: Other Notable Models

```
rf_grid = {
    'clf__n_estimators':[500,750,1000],
    'clf__max_depth':[None,5,10,20],
    'clf__criterion': ['gini']
}

rf_model_2 = GridSearchCV(rf_pipeline_2, param_grid=rf_grid,
                          cv=10, n_jobs=-1,
                          scoring = 'f1_macro',
                          return_train_score=True,
                          verbose=2)

%time rf_grid_model_2 = rf_model_2.fit(X_train, y_train)
```

```
Fitting 10 folds for each of 12 candidates, totalling 120 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:  9.6min
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed: 33.0min finished
Wall time: 33min 50s
```

```
#Viewing results of grid search
#Better with base params; find out what they are
print(rf_grid_model_2.best_score_)
print(rf_grid_model_2.best_params_)
```

```
0.69849172889903942
{'clf__criterion': 'gini', 'clf__max_depth': 20, 'clf__n_estimators': 500}
```

- Our next best performing model was a Random Forest with the below hyperparameters:
    - Criterion: Gini, Max_depth: 20, n_estimators: 500

Appendix – Modeling Details: Other Notable Models

```
[1790]: #Running CV

        rf_final_pipe_cv = cross_validate(rf_pipeline_final, X_train, y_train,
                                          cv=5, scoring='f1_macro', n_jobs=-1, verbose=1,
                                            return_train_score=True, return_estimator=True)

        [Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
        [Parallel(n_jobs=-1)]: Done   2 out of   5 | elapsed:  1.2min remaining:  1.9min
        [Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:  1.3min finished

[1791]: #Viewing CV results
        print(rf_final_pipe_cv['test_score'].mean())

        0.6962508521738812
```

- Cross Validation results with our final random forest pipeline (using our best performing hyperparameters to retain on entire training set)

Appendix – Modeling Details: Other Notable Models

```
[1792]: #Predicting on validation set
        rf_final_pipe_val_preds = rf_pipeline_final.predict(X_val)

        print(f1_score(y_val, rf_final_pipe_val_preds, average='macro'))
        print(classification_report(y_val, rf_final_pipe_val_preds))
        print(confusion_matrix(y_val, rf_final_pipe_val_preds))
```

```
0.6955537521898251
              precision    recall  f1-score   support

         0.0       0.81      0.87      0.84      3253
         1.0       0.85      0.77      0.81      2268
         2.0       0.43      0.45      0.44       419

    accuracy                           0.80      5940
   macro avg       0.70      0.69      0.70      5940
weighted avg       0.80      0.80      0.80      5940

[[2815  251  187]
 [ 468 1736   64]
 [ 177   52  190]]
```

```
[1793]: #Predicting on test set

        rf_final_pipe_test_preds = rf_pipeline_final.predict(X_test)

        print(f1_score(y_test, rf_final_pipe_test_preds, average='macro'))
        print(classification_report(y_test, rf_final_pipe_test_preds))
        print(confusion_matrix(y_test, rf_final_pipe_test_preds))
```

```
0.7099132634420702
              precision    recall  f1-score   support

         0.0       0.82      0.88      0.85      3199
         1.0       0.86      0.78      0.82      2297
         2.0       0.47      0.46      0.47       444

    accuracy                           0.81      5940
   macro avg       0.72      0.71      0.71      5940
weighted avg       0.81      0.81      0.81      5940

[[2805  225  169]
 [ 447 1790   60]
 [ 174   66  204]]
```

- Predicting on both Validation and Test data sets
- Generalized well from Validation to Test

Appendix – Modeling Details: Final Predictions

```
[176]: #Creating submission file

       submission_1 = pd.DataFrame({'id': test_features.iloc[:,0], 'status_group': final_preds_1})

       def convert_status(df):
           if df['status_group'] == 0.0:
               return 'functional'
           elif df['status_group'] == 1.0:
               return 'non functional'
           elif df['status_group'] == 2.0:
               return 'functional needs repair'
```

```
[5470]: submission_1.head()
```

[5470]:

	id	status_group
0	50785	0.0
1	51630	2.0
2	17168	0.0
3	45559	1.0
4	49871	0.0

```
[5471]: submission_1['status_group'] = submission_1.apply(convert_status, axis=1)
```

```
[5473]: #Viewing Converted results
        print(len(submission_1))
        submission_1.head()
```

14850

[5473]:

	id	status_group
0	50785	functional
1	51630	functional needs repair
2	17168	functional
3	45559	non functional
4	49871	functional

```
[5474]: #Pushing to CSV
        submission_1.to_csv('rf_preds_pipe2.csv', index=False)
```

- Beside is the code we used to group our final predictions into a dataframe, and then convert our 0,1,2 classes into "Functional", "Non Functional" and "Functional Needs Repair"

Appendix – Final Competition Submissions

Submissions

BEST	CURRENT RANK	# COMPETITORS
0.8112	2114	12137

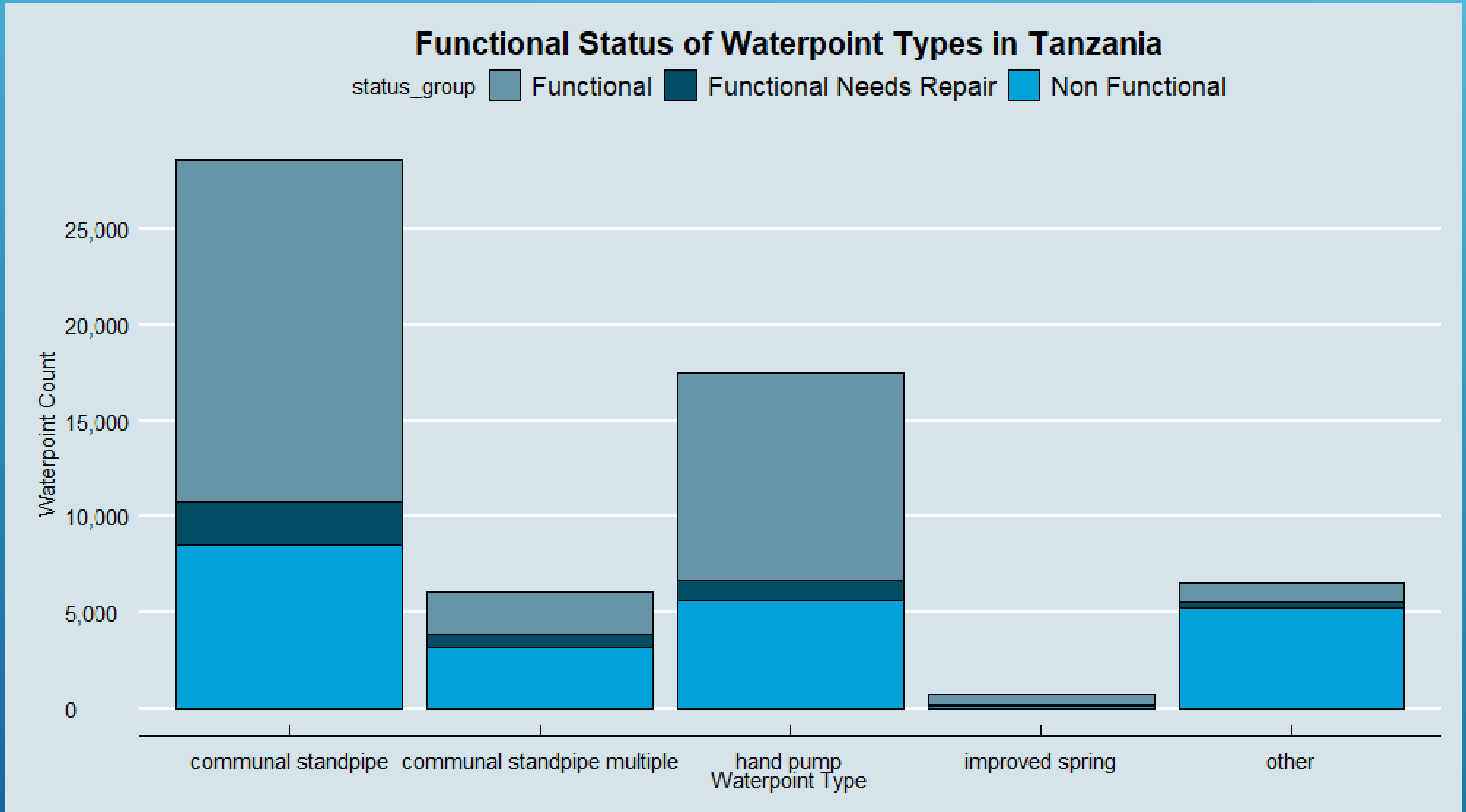
SUBMISSIONS

Score	Submitted by	Timestamp
0.7815	teamkipling	2021-07-21 21:10:31 UTC
0.8055	teamkipling	2021-07-26 21:27:09 UTC
0.8090	teamkipling	2021-07-28 01:59:15 UTC
0.8112	teamkipling	2021-08-01 21:10:20 UTC

- Beside are the results of our official submissions to the competition
- We were pleased to see a steady increase in each iteration
- Please note however that we had dozens of submissions on our individual accounts; only the strongest ones were submitted through our team's main account

Appendix – Additional Visuals; Waterpoint Types

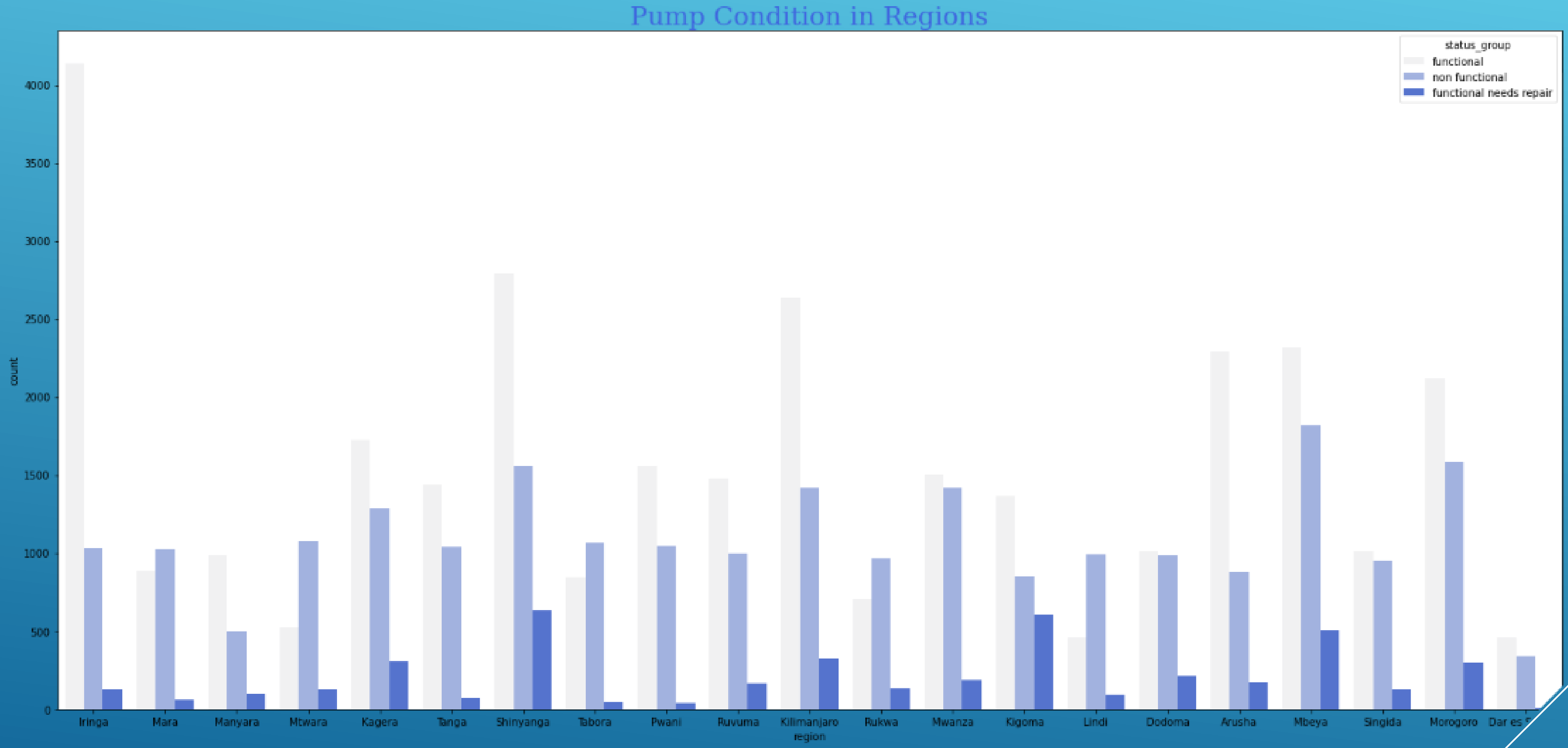
What sources do communities rely on for their water?



Insights:

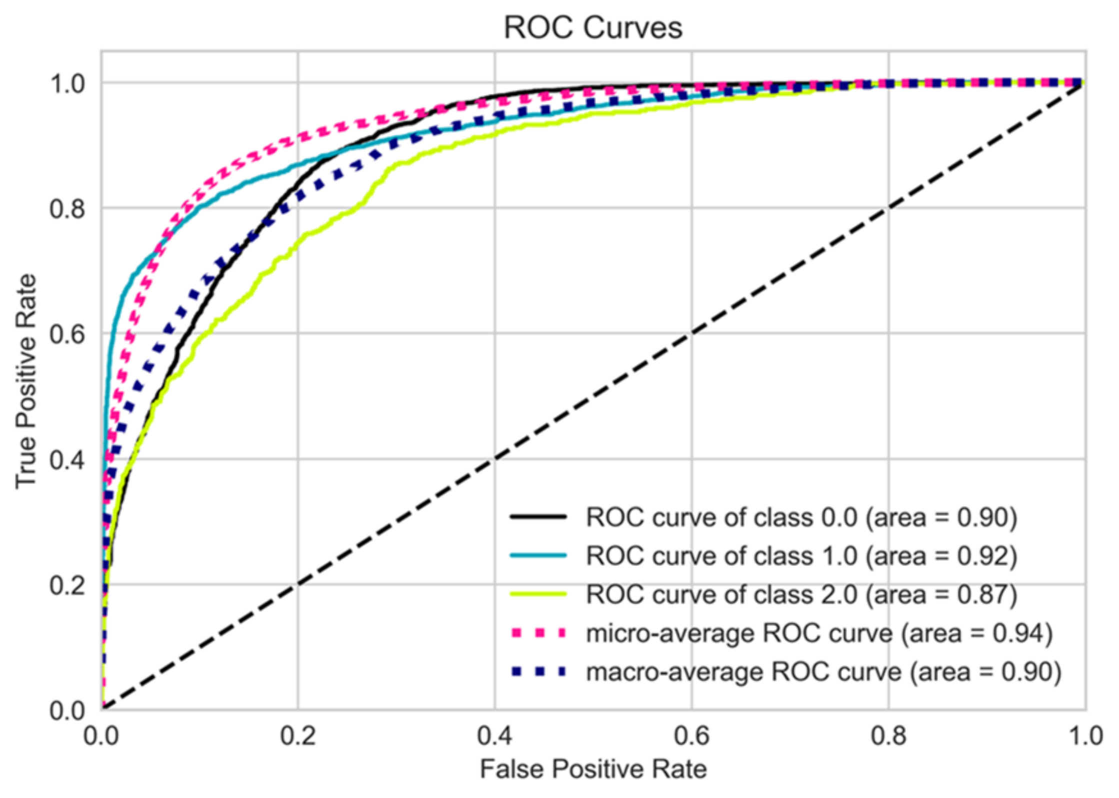
- Majority of the pumps are communal standpipe and hand pump
- Communal standpipe multiple has a higher chance being non-functional

Appendix – Additional Visuals



Appendix – Additional Visuals

ROC Curves from Best Model



Appendix – Total Impact Calculations

Estimated Economic Benefit/Cost

Economic Benefit of 1 Functioning Well	$2,000
Cost to Fix Functional Needs Repair	$500
Cost to fix Needs Repair (Avg)	$500

Confusion & Cost Matrix

	TN	FP				
	Predicted				Predicted	
Truth	2892	307		Truth	$ -	-$ 153,500
	721	2020			-$1,081,500	$3,030,000
	FN	TP				
				Total	$1,795,000	