PREDICTING THE CONDITION OF WATER PUMPS IN TANZANIA

Business Understanding

The Problem

Tanzania is a developing country in eastern Africa. It is a geographically diverse country with mountainous terrain and flat plains. The country borders the Indian Ocean in the east, and the Great Rift Valley on its western border. A good portion of the country is below sea level, and much of it is 900 ft above sea level. Access to natural resources is spotty, and water is one of the most, if not the most, scarce resource in the country. Many organizations have installed water pumps in villages around the country in an effort to provide clean drinking water to the poeple. These pumps vary in how they extract water, the water quality, and what basin the water comes from, how they are managed, the population around the well, and whether the users have to pay for water or not. Pumps break down regularly and require maintenance which can be quite a task considering there are over 59,000 pumps in the country.

The Task

We have been asked by the Tanzanian Government to create a predicition model that will predict the condition of a water pump; functional, or non-functional. This will allow them to strategically mobilize repair teams in areas with a high concentration of non-functional pumps, and to efficiently

react when resources are needed in a particular area, such as supplying potable water to a village whose pump is non-functional and awaiting repair.

Hypothesis

Our null hypothesis is that we CANNOT predict whether a well is functional or not.

Our alternative hypothesis is that we CAN predict the condition of a well.

> - A false positive would be to predict a well is non-functional when it is functional.
> - A false negative would be to predict a well is functional when it is not functional.

Data Understanding

The dataset is provided by Taarifa which aggregates data from the Tanzania Ministry of Water on the over 59,000 water pumps in he country. The data contains information about the location, operation, management, installation, water quality, and population of users of a particular well.

Method:

- Determine relevant features for prediction
- Create several models and look for best predictor metrics
- Use validation methods to ensure performance
- Tune final model for optimal predictions

Model Selection

NOTE: a positive prediction is a well that is non-functional.

Maximize Recall

We are seeking a model that will primarily maximize Recall. Recall will measure how well we are predicting wells that are actually non-functional. It is calculated by dividing the number of non-funtioning wells that were correctly predicted by the total number of non-funtional wells in our test set. Recall does not account for false positives.

> Recall:
>
> - Minimize false negatives
> - Resources are directed to the people who need them
> - More human lives are saved

With human lives at stake and counting on our model to get it right we will place primary importance on the recall score.

Maximize Precision

Secondarily we will be seeking a model that will maximize precision, but not at the expense of recall. Precision is a measure of how accurate the positive predictions are. It is calculated by dividing the number of correctly predicted positives by the total number of positive predictions. Precision does not account for false negatives.

> Precision:
>
> - Minimize false positives
> - Resources are not directed where they are not needed
> - Less logistic strain on the system, resources and man power

Repair materials and man power are also a scarce resource in this problem ad we cannot afford to be sending repair crews out to wells that have been predicted as non-functional but are functional. However, human life is tantamount and we will not sacrifice recall for more precision.

The two metrics have an inverse relationship and it will be tricky to design a model that will maximize both.

Explore the data and process it for modeling

In [1]:

```
# IMPORT DEPENDENCIES

import numpy as np
import pandas as pd
import missingno as msno
import matplotlib.pyplot as plt
import seaborn as sns
import yellowbrick as yb
import folium

from scipy import stats as stats
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, plot_confusion_matrix,\
    precision_score, recall_score, accuracy_score, f1_score,\
    classification_report
from sklearn.model_selection import train_test_split, GridSearchCV, cro
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.naive_bayes import GaussianNB

sns.set(style="whitegrid")
pd.set_option('display.max_columns',None)
pd.set_option("mode.chained_assignment", None)
```

In [2]:
```
# IMPORT THE DATA

V = pd.read_csv('Data/values.csv')
y = pd.read_csv('Data/labels.csv')
```

In [3]:
```
# TRANSFORM OUR LABELS INTO A BINARY CLASS

y = y.replace({'status_group': {'functional' : 0, 'non functional' : 1,
```

In [4]:
```
# LOOK FOR IMBALANCE IN THE DATA
# This looks good and should not cause an issue, hwever we will use bal

y.status_group.value_counts()
```

Out[4]:
```
0    32259
1    27141
Name: status_group, dtype: int64
```

In [5]:
```
# USE THE LATITUDE AND LONGITUDE AND COMBINE IT WITH OUR LABELS TO PLOT

df_lat_long = V['latitude'].to_frame().join(V['longitude']).join(y)
map_center = [df_lat_long['latitude'].mean(), df_lat_long['longitude'].
```

In [45]:
```
# PLOT THE MAP
# Notice the opacity denotes concentration of wells

map1 = folium.Map(location = map_center, tiles='Openstreetmap', zoom_st

for index, loc in df_lat_long.iterrows():
    if loc['status_group']==0:
        color = 'green'
    elif loc['status_group']==1:
        color = "red"
    else:
        color = 'white'

    folium.CircleMarker([loc['latitude'], loc['longitude']], radius=2,

folium.LayerControl().add_to(map1)

map1
```

Out[45]: Make this Notebook Trusted to load map: File -> Trust Notebook

In [7]:
```
# Create a dataframe with our of our data

df_all = V.join(y.set_index('id'), on='id')
```

In [8]:
```
# Explore all of our data in one place

df_all.head()
```

Out[8]:

	id	amount_tsh	date_recorded	funder	gps_height	installer	longitude	latitude	wpt_na
0	69572	6000.0	2011-03-14	Roman	1390	Roman	34.938093	-9.856322	r
1	8776	0.0	2013-03-06	Grumeti	1399	GRUMETI	34.698766	-2.147466	Zaha
2	34310	25.0	2013-02-25	Lottery Club	686	World vision	37.460664	-3.821329	Mah
3	67743	0.0	2013-01-28	Unicef	263	UNICEF	38.486161	-11.155298	Zaha Nanyu
4	19728	0.0	2011-07-13	Action In A	0	Artisan	31.130847	-1.825359	Shu

In [9]:
```
# Drop an outlier we found on the map (zoom out)

df_all.drop(labels=df_all.id[49651], inplace=True)
```

In [10]:
```
# No duplicates

df_all.duplicated().value_counts()
```

Out[10]:
```
False    59399
dtype: int64
```

In [11]:
```
# Drop features that have no influence, and are repetitive

df = df_all.drop(columns=['id', 'amount_tsh', 'date_recorded', 'funder'
       'longitude', 'latitude', 'wpt_name', 'num_private',
       'subvillage', 'region', 'region_code', 'district_code',
       'ward', 'public_meeting', 'recorded_by',
       'scheme_management', 'scheme_name', 'permit',
       'extraction_type_group', 'extraction_type_class',
       'management_group', 'payment_type',
       'quality_group', 'quantity', 'quantity_group',
       'source_type', 'source_class',
       'waterpoint_type_group'])
```

In [12]:
```
df.columns
```

Out[12]:
```
Index(['gps_height', 'installer', 'basin', 'lga', 'population',
       'construction_year', 'extraction_type', 'management', 'payment',
       'water_quality', 'source', 'waterpoint_type', 'status_group'],
      dtype='object')
```

In [13]:
```
# Check again for duplicates

df.duplicated().value_counts()
```

Out[13]:
```
False    44034
True     15365
dtype: int64
```

We have duplicates now that we've dropped the 'id' column, and need to make a decision whether or not to drop them. We will not because we know they are independent wells with different 'id' numbers and they need to be represented when we train our models. It seems like we are overfitting models with repetitive information, to drop them would effectively be under-representing certaiin features and our models will not be accurate.

In [14]:
```
# df.drop_duplicates(inplace=True)
```

In [15]:
```
# Look for Null values

df.isna().sum().sum()
```

Out[15]:
```
3655
```

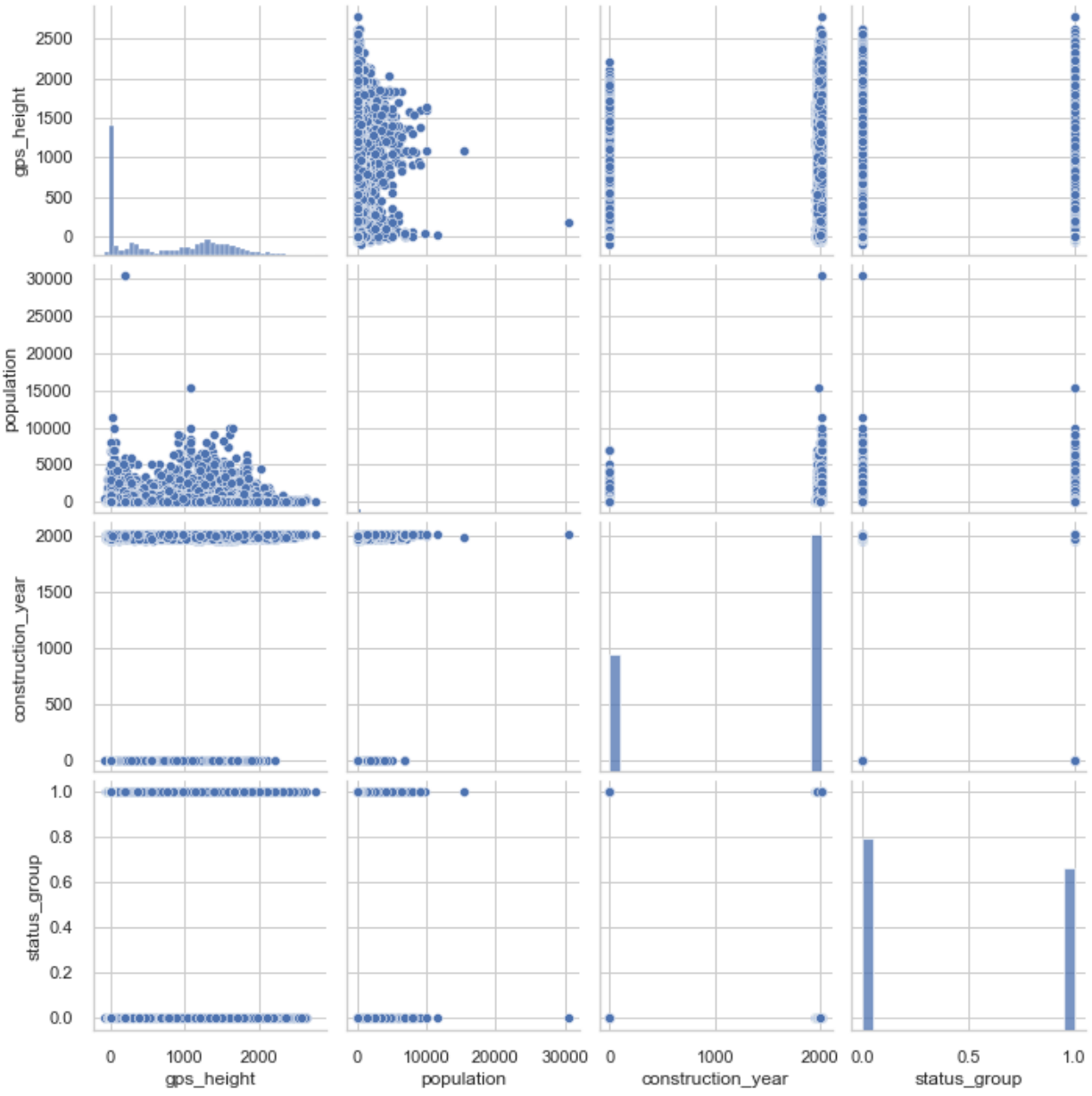
In [16]:
```
# Drop the Null values

df.dropna(inplace=True)
```

In [17]:

```
# Plot our numerical values against each other to look for any patterns

sns.pairplot(df)
plt.show()
```



Not much can be drawn about patterns, but we find some interesting information, however there are many construction_year' and 'population' entries with 0 values. 'gps_height' also shows many 0 values and even negative values, however this in line with what we know about Tanzania. some areas of the country are in valleys below sea level, and much of the counrty is above 900 ft. the median of our data set is approx. 990.

We will fix the construction year entries by imputing the median year, but the population 0 values may be accurate and reflect the population around a well and tell a story about the data, like perhaps an area has been abandoned due to lack of water, etc...

In [18]:

```
# Create our feature and label datasets for modeling

X = df.drop('status_group', axis=1)
y = df.status_group
```

In [19]:

```
#  Create a function that will process our data and split it into train

# A function that will process and scale numerical data
def process_scale (X, y):
    # Encode categorical data
    X_cat = X.select_dtypes('object')
    ohe = OneHotEncoder(sparse=False, handle_unknown='ignore')
    dums = ohe.fit_transform(X_cat)
    dums_df = pd.DataFrame(dums, columns=ohe.get_feature_names(), index

    # combine encoded columns back with numerical columns
    X_nums = X.select_dtypes('int64')
    X = pd.concat([X_nums, dums_df], axis=1)

    # Split the data into training and test sets, we will test on only
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size

    # We do the following processing on train and test sets separately

    # Replace 0's in 'construction_year' with the column median
    X_train.construction_year.where(X_train.construction_year != 0, X_t

    # Scale numerical data and combine back with encoded categorical co
    X_train_nums = X_train.select_dtypes('int64')
    ss = StandardScaler()
    ss.fit(X_train_nums)
    nums_df = pd.DataFrame(ss.transform(X_train_nums),
                           index=X_train_nums.index)
    X_cats = X_train.select_dtypes('float64')
    X_train_clean = pd.concat([nums_df, X_cats], axis=1)

    # Replace 0's in 'construction_year' with the column median
    X_test.construction_year.where(X_test.construction_year != 0, X_tes

    # Scale numerical data and combine back with encoded categorical co
    X_test_nums = X_test.select_dtypes('int64')
    ss = StandardScaler()
    ss.fit(X_test_nums)
    test_nums_df = pd.DataFrame(ss.transform(X_test_nums),
                           index=X_test_nums.index)
    X_test_cats = X_test.select_dtypes('float64')
    X_test_clean = pd.concat([test_nums_df, X_test_cats], axis=1)

    return X_train_clean, X_test_clean, y_train, y_test


# The same process as above but with no scaling for tree-based models t
def process_no_scale (X, y):
    X_cat = X.select_dtypes('object')
    ohe = OneHotEncoder(sparse=False, handle_unknown='ignore')
    dums = ohe.fit_transform(X_cat)
    dums_df = pd.DataFrame(dums, columns=ohe.get_feature_names(), index

    X_nums = X.select_dtypes('int64')
    X = pd.concat([X_nums, dums_df], axis=1)

```

```
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size

    X_train.construction_year.where(X_train.construction_year != 0, X_t

    X_test.construction_year.where(X_test.construction_year != 0, X_tes

    return X_train, X_test, y_train, y_test
```

In [20]:
```
# Load our training and testing sets into memory

X_train, X_test, y_train, y_test = process_scale(X, y)

X_train_ns, X_test_ns, y_train_ns, y_test_ns = process_no_scale(X, y)
```

CREATE SOME MODELS AND SCORE THEM

> **We are looking to maximize Recall primarily.**
>
> **We are also looking to maximize Precision, but not at the expense of Recall**

Logistic Regression

In [21]:
```
# Instantiate and fit the model

logreg = LogisticRegression(fit_intercept=False, class_weight='balanced
model_log = logreg.fit(X_train, y_train)
```

In [22]:
```
print(f"training accuracy: {model_log.score(X_train, y_train)}")
```

```
training accuracy: 0.7462177838904502
```

In [23]:
```
# Create predictions and score them

y_hat_test = model_log.predict(X_test)

print(confusion_matrix(y_test, y_hat_test))
```

```
[[2310  675]
 [ 749 1841]]
```

In [24]:
```
precision = precision_score(y_test, y_hat_test)
recall = recall_score(y_test, y_hat_test)
accuracy = accuracy_score(y_test, y_hat_test)
F1 = f1_score(y_test, y_hat_test)

print(precision)
print(accuracy)
print(recall)
print(F1)
```

```
0.7317170111287759
0.7445739910313901
0.7108108108108108
0.7211124167645906
```

K Nearest Neighbors

In [25]:
```
# Instantiate and fit the model

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

Out[25]:
```
KNeighborsClassifier()
```

In [26]:
```
print(f"training accuracy: {knn.score(X_train, y_train)}")
```

```
training accuracy: 0.8259283621359804
```

In [27]:
```
# Create predictions and score them

knn_preds = knn.predict(X_test)

print(confusion_matrix(y_test, knn_preds))
```

```
[[2449  536]
 [ 675 1915]]
```

In [28]:
```
precision = precision_score(y_test, knn_preds)
recall = recall_score(y_test, knn_preds)
accuracy = accuracy_score(y_test, knn_preds)
F1 = f1_score(y_test, knn_preds)

print(precision)
print(accuracy)
print(recall)
print(F1)
```

```
0.7813137494900041
0.782780269058296
0.7393822393822393
0.7597698869271969
```

Those are some good scores

Recall: 74%
Precision: 78%

Naive Bayes

In [29]:
```
# Instantiate and fit the model

gnb = GaussianNB()

gnb.fit(X_train_ns, y_train)

print(f"training accuracy: {gnb.score(X_train, y_train)}")
# print(f"testing accuracy: {gnb.score(X_test, y_test)}")
```

```
training accuracy: 0.5396360302178637
```

In [30]:
```
# Create predictions and score them

gnb_preds = gnb.predict(X_test)
print(confusion_matrix(y_test, gnb_preds))
```

```
[[ 476 2509]
 [  71 2519]]
```

In [31]:
```
precision = precision_score(y_test, gnb_preds)
recall = recall_score(y_test, gnb_preds)
accuracy = accuracy_score(y_test, gnb_preds)
F1 = f1_score(y_test, gnb_preds)

print(precision)
print(accuracy)
print(recall)
print(F1)
```

```
0.500994431185362
0.537219730941704
0.9725868725868726
0.6613284326594907
```

Decision Tree

In [32]:
```
# Instantiate and fit the model

tree = DecisionTreeClassifier(max_depth=10, criterion='gini')

tree.fit(X_train_ns, y_train_ns)

print(f"training accuracy: {tree.score(X_train_ns, y_train_ns)}")
# print(f"testing accuracy: {tree.score(X_test_ns, y_test_ns)}")
```

```
training accuracy: 0.7315473698897725
```

In [33]:
```
# Create predictions and score them

tree_preds = tree.predict(X_test_ns)
print(confusion_matrix(y_test_ns, tree_preds))
```

```
[[2346  639]
 [ 938 1652]]
```

In [34]:
```
precision = precision_score(y_test_ns, tree_preds)
recall = recall_score(y_test_ns, tree_preds)
accuracy = accuracy_score(y_test_ns, tree_preds)
F1 = f1_score(y_test_ns, tree_preds)

print(precision)
print(accuracy)
print(recall)
print(F1)
```

```
0.7210824967263204
0.7171300448430493
0.6378378378378379
0.6769104691661545
```

Random Forest

In [35]:
```
# Instantiate and fit the model

forest = RandomForestClassifier(max_features=None, class_weight="balanc
                                max_depth = 10)

forest.fit(X_train_ns, y_train_ns)

print(f"training accuracy: {forest.score(X_train_ns, y_train_ns)}")
# print(f"testing accuracy: {forest.score(X_train_ns, y_train_ns)}")
```

```
training accuracy: 0.7529151468038031
```

In [36]:
```
# Create predictions and score them

forest_preds = forest.predict(X_test_ns)
print(confusion_matrix(y_test_ns, forest_preds))
```

```
[[2429  556]
 [ 888 1702]]
```

In [37]:
```
precision = precision_score(y_test_ns, forest_preds)
recall = recall_score(y_test_ns, forest_preds)
accuracy = accuracy_score(y_test_ns, forest_preds)
F1 = f1_score(y_test_ns, forest_preds)

print(precision)
print(accuracy)
print(recall)
print(F1)
```

```
0.7537643932683791
0.7409865470852018
0.6571428571428571
0.7021452145214522
```

Based on Recall and Precision scores, our best performing model was the *K Nearest Neighbor model*

Tuning the KNN Model

We will iterate through a couple grid searches to be sure we are finding the best parameters

In [38]:
```
knn_tuned = KNeighborsClassifier(algorithm='auto')

parameters_KNN = {
    'n_neighbors': (5, 23, 33, 43, 53),
    'weights': ('uniform', 'distance')
}

grid_search_KNN = GridSearchCV(
    estimator=knn_tuned,
    param_grid=parameters_KNN,
    scoring = 'recall',
    n_jobs = -1,
    cv = 5
)

KNN = grid_search_KNN.fit(X_train, y_train)

print('Best parameters:', grid_search_KNN.best_params_ )
print('--------------------------')
print('Best Recall Score:', grid_search_KNN.best_score_ )
```

```
Best parameters: {'n_neighbors': 5, 'weights': 'distance'}
--------------------------
Best Recall Score - KNN: 0.7173736947961599
```

Lets fine tune this with n-neighbors closer to the above result

In [39]:
```
knn_tuned = KNeighborsClassifier(algorithm='auto')

parameters_KNN = {
    'n_neighbors': (1, 3, 5, 7, 9, 11, 13),
    'weights': ('uniform', 'distance')
}

grid_search_KNN = GridSearchCV(
    estimator=knn_tuned,
    param_grid=parameters_KNN,
    scoring = 'recall',
    n_jobs = -1,
    cv = 5
)

KNN = grid_search_KNN.fit(X_train, y_train)

print('Best parameters:', grid_search_KNN.best_params_ )
print('--------------------------')
print('Best Recall Score:', grid_search_KNN.best_score_ )
```

```
Best parameters: {'n_neighbors': 5, 'weights': 'distance'}
--------------------------
Best Recall Score - KNN: 0.7173736947961599
```

It looks like our default number of neighbors (5) was the top scoring number parameter, along with being weighted by distance. Let's see

how these parameters score on the test set.

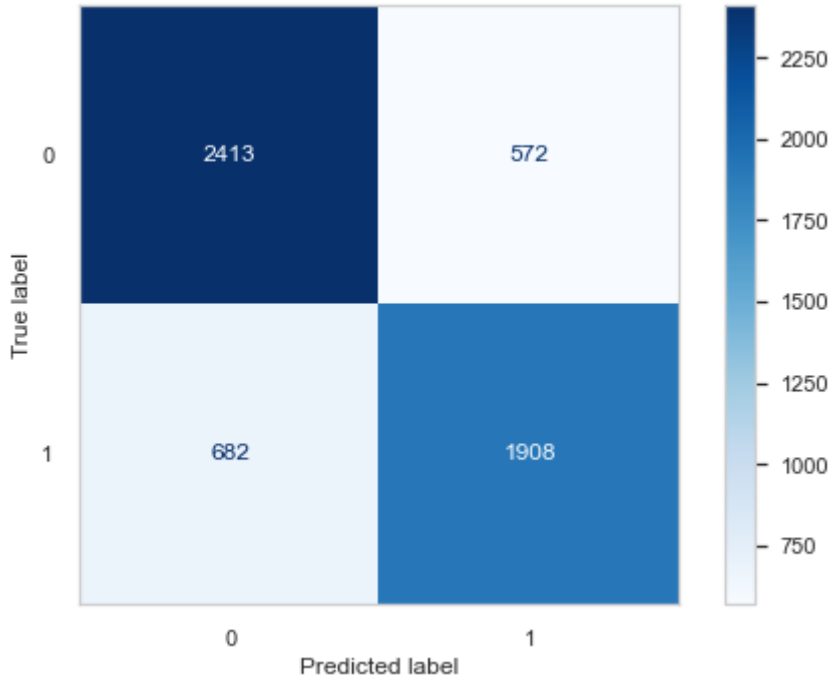
> distance weighting means that within "n" number of neighbors, closer neighbors are weighted heavier than neighbors that are further away

In [40]:
```
knn_tuned = KNeighborsClassifier(n_neighbors=5, weights='distance')

knn_tuned.fit(X_train, y_train)

knn_tuned_preds = knn_tuned.predict(X_test)

plot_confusion_matrix(knn_tuned, X_test, y_test, cmap='Blues')
plt.grid(False)
```



In [41]:
```
precision = precision_score(y_test, knn_tuned_preds)
recall = recall_score(y_test, knn_tuned_preds)

print('Precision:', precision)
print('Recall:', recall)
```

```
Precision: 0.7693548387096775
Recall: 0.7366795366795367
```

Let's compare these scores with our original default test scores.

Only difference is that uniform weighting is the default

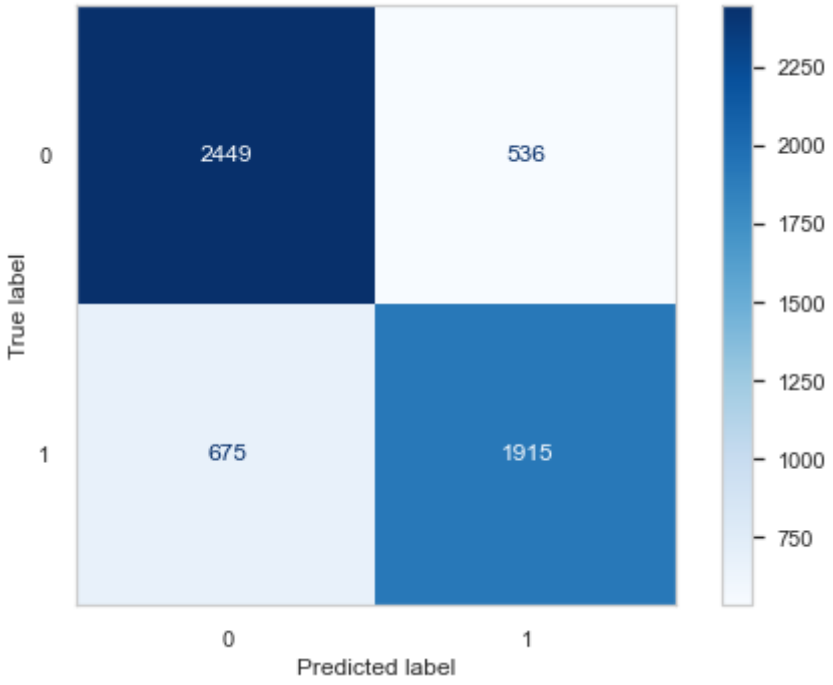
> Uniform means all neighbors within "n" are weighted equally

In [42]:
```
knn_tuned = KNeighborsClassifier(n_neighbors=5, weights='uniform')

knn_tuned.fit(X_train, y_train)

knn_tuned_preds = knn_tuned.predict(X_test)

plot_confusion_matrix(knn_tuned, X_test, y_test, cmap='Blues')
plt.grid(False)
```



In [43]:
```
precision = precision_score(y_test, knn_tuned_preds)
recall = recall_score(y_test, knn_tuned_preds)

print('Precision:', precision)
print('Recall:', recall)
```

```
Precision: 0.7813137494900041
Recall: 0.7393822393822393
```

The uniform weighting model performed better on test data.

Reccomendations

- Track seasonal droughts, water conditions and basin levels so we will know what non-functional pumps are un-repairable to better target efforts
- Install more pumps
- Gather more data

Next Steps

- Gather more data
- Re-assess features for significance
- Further tune the model
- Assess more advanced and resource intensive models

In [ ]:
```
1
```