Detection of Non-functional Water Wells Using Machine Learning Algorithms

Overview

Tanzania, as a developing country, struggles with providing clean water to its population of over 57,000,000. There are many wells (water points) already established in the country, but some are in need of repair while others have failed altogether. The Government of Tanzania is looking to find patterns in non-functional wells to influence how new wells are built.

Business Problem

Tanzania faces challenges in ensuring access to clean water for its population due to non-functional water wells. We will be looking to predict patterns in non-functional wells to inform more robust construction methods for new wells.

Data Understanding

Dataset

Driven Data - Tanzanian Water Wells

- Labels (https://www.drivendata.org/competitions/7/pump-it-up-data-mining-the-water-table/data/)
- Values (https://www.drivendata.org/competitions/7/pump-it-up-data-mining-the-water-table/data/)

Target

- functional : the well is operational and there are no repairs needed
- functional needs repair : the well is operational, but needs repairs
- non functional : the well is not operational

Features

- amount_tsh : Total static head (amount water available to well)
- date_recorded : The date the row was entered
- funder : Who funded the well
- gps_height : Altitude of the well
- installer : Organization that installed the well
- longitude : GPS coordinate
- latitude : GPS coordinate
- wpt_name : Name of the well if there is one
- num_private :Private use or not
- basin : Geographic water basin
- subvillage : Geographic location
- region : Geographic location
- region_code : Geographic location (coded)
- district_code : Geographic location (coded)
- lga : Geographic location
- ward : Geographic location
- population : Population around the well
- public_meeting : True/False
- recorded_by : Group entering this row of data
- scheme_management : Who operates the well
- scheme_name : Who operates the well
- permit : If the well is permitted
- construction_year : Year the well was constructed
- extraction_type : The kind of extraction the well uses
- extraction_type_group : The kind of extraction the well uses
- extraction_type_class : The kind of extraction the well uses
- management : How the well is managed
- management_group : How the well is managed
- payment : What the water costs
- payment_type : What the water costs
- water_quality : The quality of the water

- quality_group : The quality of the water
- quantity : The quantity of water
- quantity_group : The quantity of water
- source : The source of the water
- source_type : The source of the water
- source_class : The source of the water
- waterpoint_type : The kind of well
- waterpoint_type_group : The kind of well

Limitations

- Do not know what types of repairs are needed
- Not able to use the well age, due to missing construction years
- Without full population information, we do not know the supply needs

Data Preparation

Import and Read Datasets

In [51]:
```
# Import standard packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
import datetime

# Model Selection
from sklearn.model_selection import train_test_split, GridSearchCV, cross

# Classification Models
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.dummy import DummyClassifier
from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import plot_confusion_matrix, accuracy_score, f1_sc
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import roc_curve, auc, roc_auc_score

# Scalers
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

# Categorical Create Dummies
from sklearn.preprocessing import OneHotEncoder

# Column Transformer
from sklearn.compose import ColumnTransformer

# Pipeline
from sklearn.pipeline import Pipeline

# Base
from sklearn.base import BaseEstimator, TransformerMixin

import warnings
warnings.filterwarnings('ignore')
```

In [93]:
```
# Load the datasets
features_data = 'data/WellWaterData.csv'
target_data = 'data/TargetData.csv'

features = pd.read_csv(features_data)
target = pd.read_csv(target_data)

# Display the contents of the datasets
features.head(), target.head()
```

. . .

Merging Datasets

In [53]:
```
# Checking for unique IDs in both datasets to ensure they match
unique_ids_features = features['id'].nunique()
unique_ids_target = target['id'].nunique()

unique_ids_features, unique_ids_target
```

Out[53]:
```
(59400, 59400)
```

In [54]:
```
# Merging the datasets on the 'id' column
merged_data = pd.merge(features, target, on='id')

# Displaying the first few rows of the merged dataset
merged_data.head()
```

Out[54]:

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

5 rows × 41 columns

In [55]:
```
merged_data.info()
```

. . .

In [56]:
```
merged_data.value_counts()
```

...

Clean

Preparing the merged dataset for feature exploration and how they relate to the 'status_group' target variable.

In [57]:
```
# Converting 'date_recorded' to datetime
merged_data['date_recorded'] = pd.to_datetime(merged_data['date_recorded
```

Addressing Categorical Features with Parent and Subgroup Columns

In [58]:
```
grouped = merged_data.groupby(['extraction_type_class', 'extraction_type_
print(grouped)
```

...

In [59]:
```
grouped = merged_data.groupby(['management_group', 'management']).size()
print(grouped)
```

...

In [60]:
```
grouped = merged_data.groupby(['waterpoint_type_group', 'waterpoint_type
print(grouped)
```

...

Handling Missing Values

In [61]:
```
# Calculating the percentage of zero values for each column
zero_value_percentages = {}
for column in merged_data.columns:
    zero_count = (merged_data[column] == 0).sum()
    zero_value_percentages[column] = (zero_count / len(merged_data)) * 1


zero_value_percentages
```

...

In [62]:
```
# Calculating the percentage of missing values in each column
missing_values = merged_data.isnull().mean() * 100
missing_values = missing_values[missing_values > 0].sort_values(ascendin

missing_values
```

...

Since 'scheme_management', 'installer', 'funder', and 'permit' have less than 7% missing values and are potentially relevant, replacing them is a good option.

In [63]:
```
# Replacing the missing values
for column in ['scheme_management', 'installer', 'funder', 'permit']:
    merged_data[column].fillna('Unknown', inplace=True)

# Sanity check on missing values
remaining_missing_values = merged_data.isnull().sum()
remaining_missing_values[remaining_missing_values > 0]
```

...

Dropping Columns

In [64]:
```
# Dropping additional group columns
# Dropping irrelvant columns
# Dropping features with missing values over 45%
features_dropped = merged_data.drop(columns=['quantity_group', 'extracti
                                             'waterpoint_type_group', 'ma
                                             'payment', 'water_quality',
                                             'region_code', 'public_meet
                                             'wpt_name', 'district_code'
                                             'ward', 'subvillage', 'extra


features_dropped.info()
```

...

Removing Duplicates

In [65]:
```
features_dropped.duplicated().sum()
```

Out[65]: 685

In [66]:
```
data_dedup = features_dropped.drop_duplicates()

# Rechecking for duplicates
new_duplicate_count = data_dedup.duplicated().sum()
new_duplicate_count
```

Out[66]: 0

Feature Engineering

In [67]:
```
# Extracting year and month from 'date_recorded'
data_dedup['year_recorded'] = data_dedup['date_recorded'].dt.year
data_dedup['month_recorded'] = data_dedup['date_recorded'].dt.month

data_dedup[['longitude', 'gps_height', 'construction_year', 'year_record
```

...

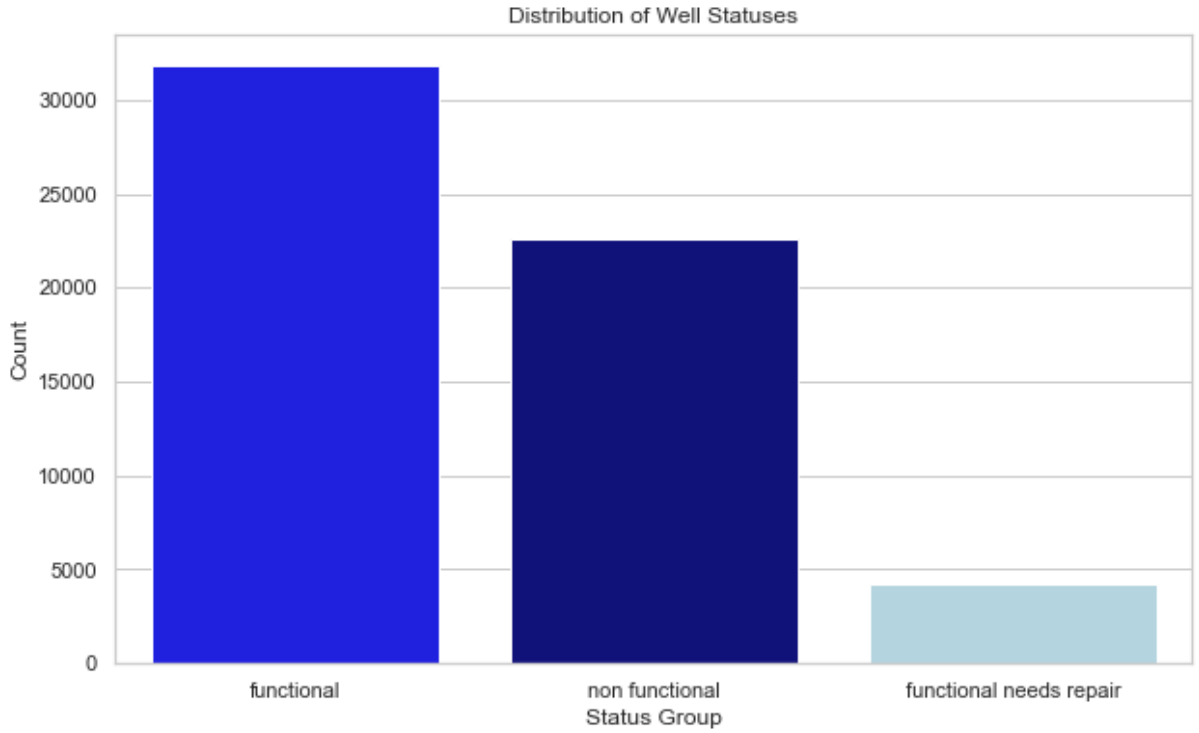
In [68]:
```
# Adding 'well_age' feature
data_dedup['well_age'] = data_dedup.apply(lambda row: 0 if row['construc
                                          else row['year_recorded'] - ro

# Displaying the first 50 rows to check the 'construction_year' and 'wel
data_dedup[['year_recorded', 'construction_year', 'well_age']].value_cou
```

...

Creating Binary Target Column

First, we'll need to understand the distribution of the target variable and visualize the proportion of functional vs. non-functional wells.

In [91]:
```
sns.set(style="whitegrid")

# Defining palette colors
palette_colors = {'functional': 'blue', 'non functional': 'darkblue', 'fu

# Plotting the distribution of well statuses
plt.figure(figsize=(10, 6))
sns.countplot(x='status_group', palette=palette_colors, data=data_dedup)
plt.title('Distribution of Well Statuses')
plt.ylabel('Count')
plt.xlabel('Status Group')
plt.show()
```



To address differing opinions on how to condense our target into a binary column, we will create two separte binary target columns and assess the better performer on our baseline model.

status_binary:

- Class 0 = non-functional & functional needs repair
- Class 1 = functional

status_binary_reversed:

- Class 0 = non-functional
- Class 1 = functional & functional needs repair

In [70]:
```
# Binary encoding of the 'status_group' column

# 'functional' is assigned 1 and 'non functional' or 'functional needs re
data_dedup['status_binary'] = data_dedup['status_group'].apply(lambda x:

# 'non-functional' is assigned 0 and 'functional' or 'functional needs re
data_dedup['status_binary_reversed'] = data_dedup['status_group'].apply(

data_dedup[['status_group', 'status_binary', 'status_binary_reversed']].
```

...

Creating Master Dataset

In [71]:
```
# Creating master dataset with all values of 'well_age' greater than or e
# to eliminate negative values where 'recorded_year' was likely listed in
master_data = data_dedup[data_dedup['well_age'] >= 0]
master_data.info()
```

...

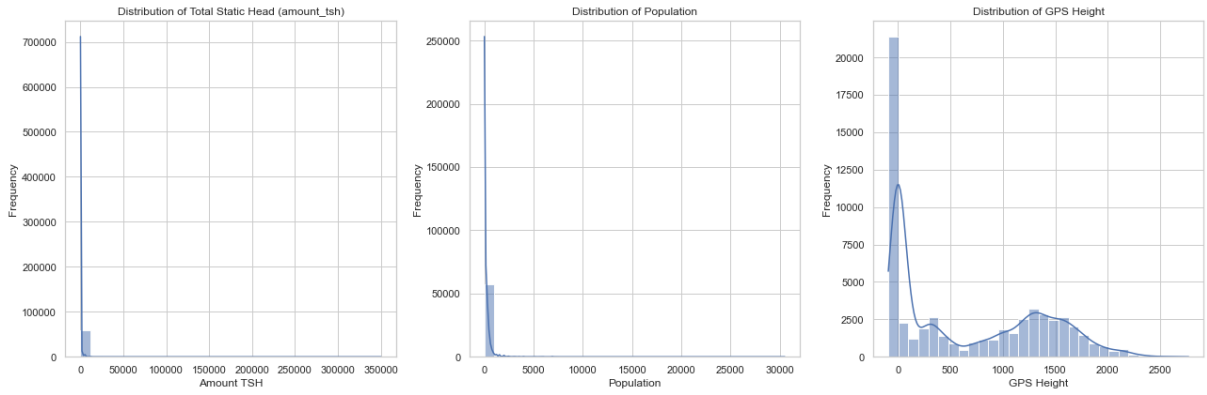
In [72]:
```
# Saving master dataset to csv
master_data.to_csv('data/master_data.csv', index=False)
```

EDA

Data Distribution Visualizations

Using histograms, we'll plot the distribution of key numeric variables like amount_tsh, population, and gps_height.

In [73]:
```
sns.set_style("whitegrid")

# Creating histograms for 'amount_tsh', 'population', and 'gps_height'
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# Plot for 'amount_tsh'
sns.histplot(master_data['amount_tsh'], bins=30, ax=axes[0], kde=True)
axes[0].set_title('Distribution of Total Static Head (amount_tsh)')
axes[0].set_xlabel('Amount TSH')
axes[0].set_ylabel('Frequency')

# Plot for 'population'
sns.histplot(master_data['population'], bins=30, ax=axes[1], kde=True)
axes[1].set_title('Distribution of Population')
axes[1].set_xlabel('Population')
axes[1].set_ylabel('Frequency')

# Plot for 'gps_height'
sns.histplot(master_data['gps_height'], bins=30, ax=axes[2], kde=True)
axes[2].set_title('Distribution of GPS Height')
axes[2].set_xlabel('GPS Height')
axes[2].set_ylabel('Frequency')

plt.tight_layout()
plt.show()
```



Observations

These distributions reflect the high number of zero values in our dataset for these features.

Total Static Head (amount_tsh): The distribution appears to be highly skewed to the right, indicating that most wells have a low static head value.

Population: This distribution is also right-skewed, showing that most wells serve a relatively small population, with fewer points serving larger populations.

GPS Height: The distribution is more varied, indicating a range of elevations at which wells are located.

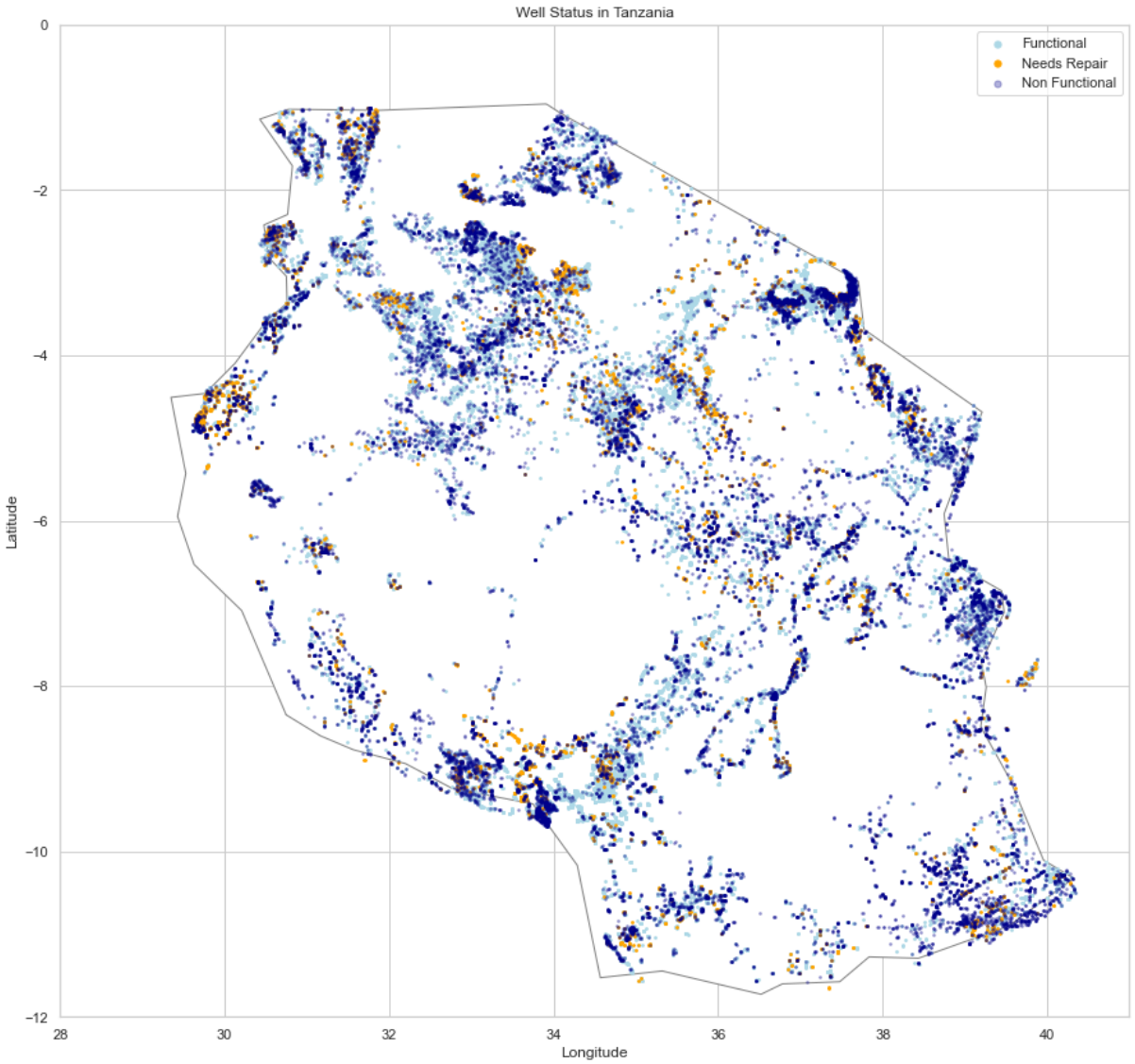
Geographical Analysis

We'll create a geographical plot using latitude and longitude to see if there is any geographical pattern in the status of wells.

In [74]:
```
!pip install geopandas
```

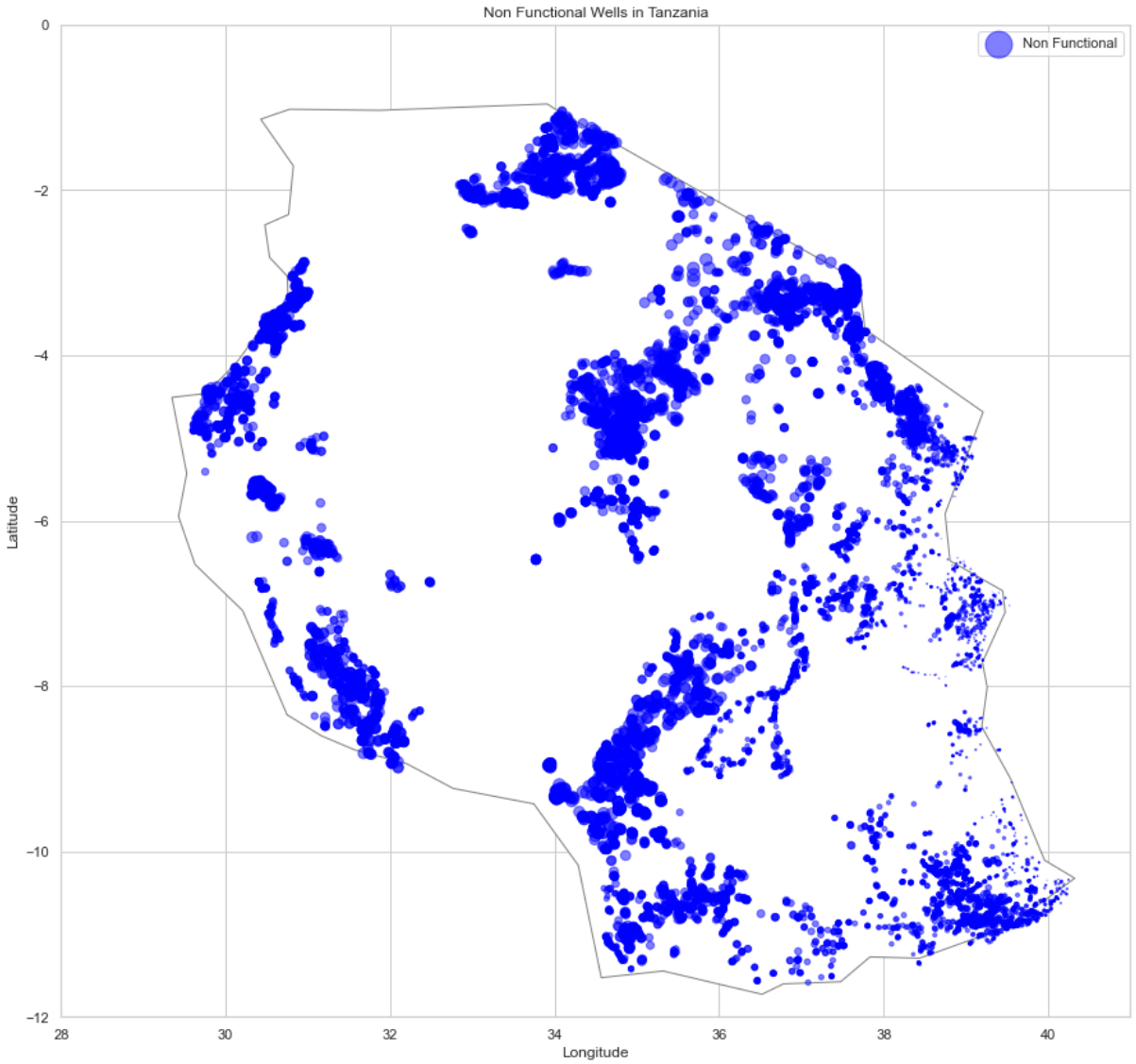
...

In [75]:
```
import geopandas
import matplotlib.pyplot as plt

# Create GeoDataFrame
gdf = geopandas.GeoDataFrame(
    master_data, geometry=geopandas.points_from_xy(master_data.longitude

# Assigning 'status_group'
functional = gdf[gdf['status_group'] == 'functional']
repair = gdf[gdf['status_group'] == 'functional needs repair']
non_functional = gdf[gdf['status_group'] == 'non functional']

# Load world shapefile
world_shapefile_path = 'data/ne_110m_admin_0_countries/ne_110m_admin_0_c
world = geopandas.read_file(world_shapefile_path)

# Filter for Tanzania
fig, ax = plt.subplots(figsize=(15, 15))
base = world[world.ADMIN == 'United Republic of Tanzania'].plot(color='wh

# Scatter plots for each category
ax.scatter(functional['longitude'], functional['latitude'], c='lightblue
ax.scatter(repair['longitude'], repair['latitude'], c='orange', alpha=1,
ax.scatter(non_functional['longitude'], non_functional['latitude'], c='da

# Limiting the display area
ax.set_ylim(-12, 0)
ax.set_xlim(28, 41)

# Adding labels and title
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
ax.set_title('Well Status in Tanzania')

# Adding legend
ax.legend(markerscale=3, loc='upper right')

plt.show()
```



To further identify geographic feature relationships, we'll plot only the non-functional wells and adjust the marker size by 'gps_height'.

In [76]:
```
# Filter for 'non functional' wells
non_functional = gdf[gdf['status_group'] == 'non functional']

# Filter for Tanzania and plot
fig, ax = plt.subplots(figsize=(15, 15))
base = world[world.ADMIN == 'United Republic of Tanzania'].plot(color='wh

# Plot for 'non functional' wells with marker size based on 'gps_height'
# Normalize 'gps_height' for visualization
max_height = non_functional['gps_height'].max()
marker_size = (non_functional['gps_height'] / max_height) * 100

ax.scatter(non_functional['longitude'], non_functional['latitude'], c='b

# Limiting the display area
ax.set_ylim(-12, 0)
ax.set_xlim(28, 41)

# Adding labels and title
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
ax.set_title('Non Functional Wells in Tanzania')

# Adding legend
ax.legend(markerscale=3, loc='upper right')

plt.show()
```

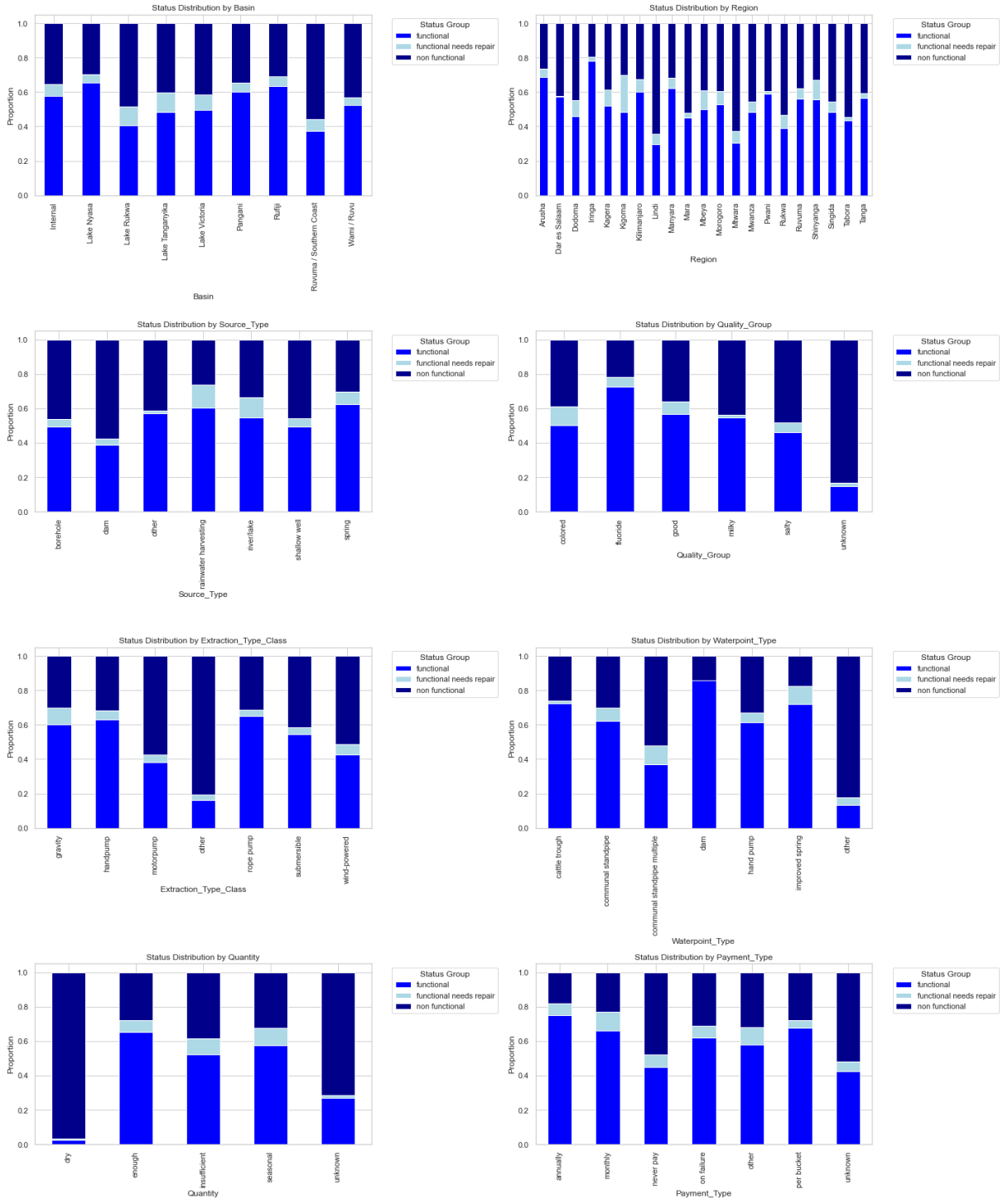


The majority of 'non functional' wells seem to be located at higher altitudes.

Categorical to Target Relationships

We'll explore the relationships between categorical variables and the target variable 'status_group'.

In [77]:
```
import math

categorical_vars = ['basin', 'region', 'source_type', 'quality_group', '

# Define custom colors for each status
color_map = {'functional': 'blue', 'non functional': 'darkblue', 'functi

# Determine the number of rows needed for the subplot (2 plots per row)
num_vars = len(categorical_vars)
num_rows = math.ceil(num_vars / 2)

# Creating subplots for each categorical variable
fig, axes = plt.subplots(nrows=num_rows, ncols=2, figsize=(20, num_rows *
axes = axes.flatten()  # Flatten the axes array for easy iteration

# Looping through the variables and creating a stacked bar plot for each
for i, var in enumerate(categorical_vars):
    # Creating a crosstab for the variable and status_group
    crosstab = pd.crosstab(master_data[var], master_data['status_group']

    # Creating a stacked bar plot with custom colors
    crosstab.plot(kind='bar', stacked=True, color=[color_map[status] for
    axes[i].set_title(f'Status Distribution by {var.title()}')
    axes[i].set_xlabel(var.title())
    axes[i].set_ylabel('Proportion')
    axes[i].legend(title='Status Group', bbox_to_anchor=(1.05, 1), loc='u

# Adjust the layout
plt.tight_layout()
plt.show()
```



Observations

Region: Similar to basins, each region has a unique distribution of well statuses. This seems to be an indicator for well status.

Payment Type: Whether a well is paid seems to be a crucial factor. Wells that are not paid have a high number of non-functional wells.

Waterpoint Type: The method used for the population to access the water from the wells is another crucial factor. Similarly to extraction methods, waterpoint types might be more robust and less prone to failure, while others could be more complex and require frequent

repairs

Machine Learning

Baseline Model #1: Binary Target Column

Class 0 = non-functional/needs repair
Class 1 = functional

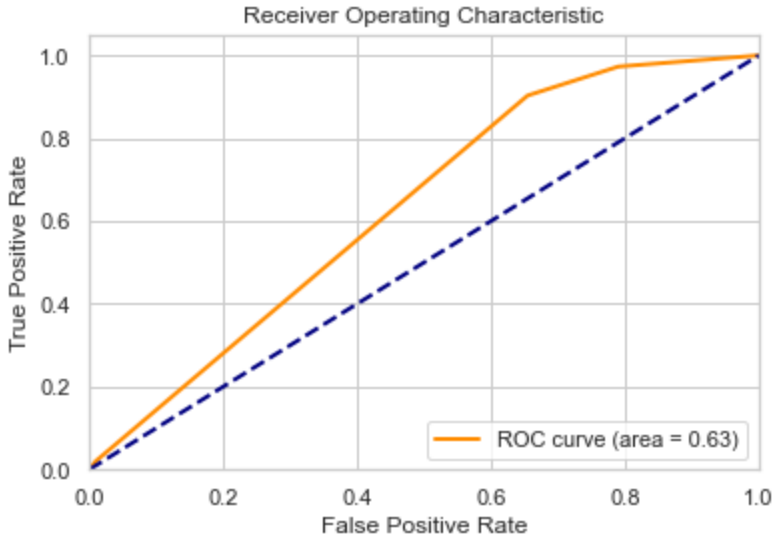
In the context of non-functional wells, focusing on recall (false negative) may be more important to ensure that most of the non-functional wells are correctly identified. We will test two separate baseline models, each with a different binary target column to see which performs best on recall.

In [78]:
```
# Intitating train_test_split
X = master_data[['waterpoint_type']]
y = master_data['status_binary']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33

# Encoding categorical variable
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder()
ohe.fit(X_train)
X_train_encoded = ohe.transform(X_train)
X_test_encoded = ohe.transform(X_test)

# Plotting Log Reg transform
logreg = LogisticRegression(random_state=42)
logreg.fit(X_train_encoded, y_train)

# Checking if the target is balanced
y_test.value_counts(normalize=True)

y_pred = logreg.predict(X_test_encoded)

print("Classification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
Classification Report:
               precision    recall  f1-score   support

           0       0.75      0.35      0.47      8894
           1       0.62      0.90      0.73     10479

    accuracy                           0.65     19373
   macro avg       0.69      0.62      0.60     19373
weighted avg       0.68      0.65      0.61     19373

Confusion Matrix:
 [[3070 5824]
 [1015 9464]]
```

Evaluating with ROC Curve

In [79]:
```
# Predict probas for the positive class
y_pred_proba = logreg.predict_proba(X_test_encoded)[:, 1]

# Compute AUC-ROC
roc_auc = roc_auc_score(y_test, y_pred_proba)

# Compute ROC curve
fpr, tpr, _ = roc_curve(y_test, y_pred_proba)

# Plotting ROC Curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```



Baseline Model #2: Reverse Binary Target Column

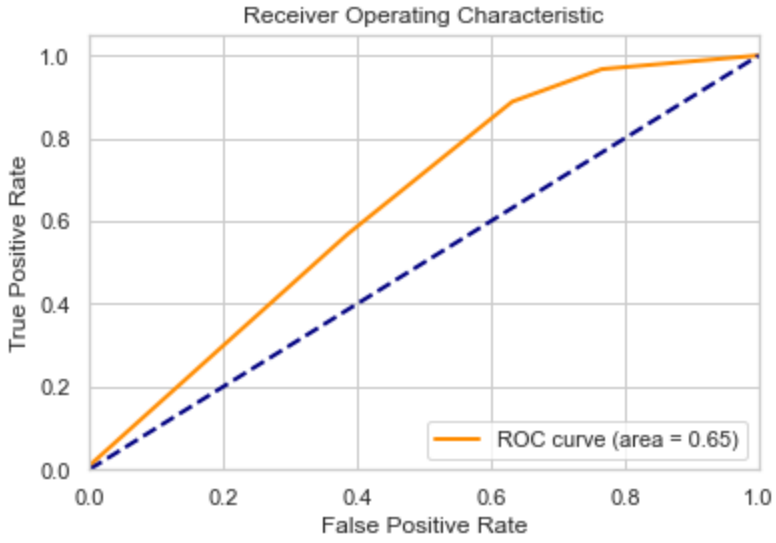
Class 0 = non-functional
Class 1 = functional/needs repair

In [80]:
```
# Intitating train_test_split
X = master_data[['waterpoint_type']]
y = master_data['status_binary_reversed']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33

# Encoding categorical variable
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder()
ohe.fit(X_train)
X_train_encoded = ohe.transform(X_train)
X_test_encoded = ohe.transform(X_test)

# Plotting Log Reg transform
logreg = LogisticRegression(random_state=42)
logreg.fit(X_train_encoded, y_train)

# Checking if the target is balanced
y_test.value_counts(normalize=True)

# Predicting and evaluating the model
y_pred = logreg.predict(X_test_encoded)
print("Classification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
Classification Report:
               precision    recall  f1-score   support

           0       0.68      0.37      0.48      7490
           1       0.69      0.89      0.78     11883

    accuracy                           0.69     19373
   macro avg       0.68      0.63      0.63     19373
weighted avg       0.68      0.69      0.66     19373

Confusion Matrix:
 [[ 2758  4732]
 [ 1327 10556]]
```

Evaluating with ROC Curve

In [81]:
```
# Predict probas for the positive class
y_pred_proba = logreg.predict_proba(X_test_encoded)[:, 1]

# Compute AUC-ROC
roc_auc = roc_auc_score(y_test, y_pred_proba)

# Compute ROC curve
fpr, tpr, _ = roc_curve(y_test, y_pred_proba)

# Plotting ROC Curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```



Baseline Comparison

Baseline 1

Precision:

- Class 0: 0.75 (Higher precision for class 0)
- Class 1: 0.62

Recall:

- Class 0: 0.35 (Lower recall for class 0)

- Class 1: 0.90 (Higher recall for class 1)

F1-Score:

- Class 0: 0.47 (Lower F1-score for class 0)
- Class 1: 0.73

Accuracy: 65%

Baseline 2

Precision:

- Class 0: 0.68 (Lower precision for class 0)
- Class 1: 0.69

Recall:

- Class 0: 0.37 (Slightly higher recall for class 0)
- Class 1: 0.89 (Slightly lower recall for class 1)

F1-Score:

- Class 0: 0.48 (Slightly higher F1-score for class 0)
- Class 1: 0.78

Accuracy: 69% (Higher)

Analysis

Baseline 2 shows improved overall performance, with better accuracy and a better balance in precision and recall for both classes. However, it is more prone to falsely identifying class 0 (non-functional) instances as class 1 (functional).

Baseline 1 while having higher precision for class 0 (non-functional), falls short in accurately identifying class 0 (non-functional) instances (lower recall).

Since we are more concerned with better recall, we will continue our modeling with **Baseline 2**.

Random Forest Classifier

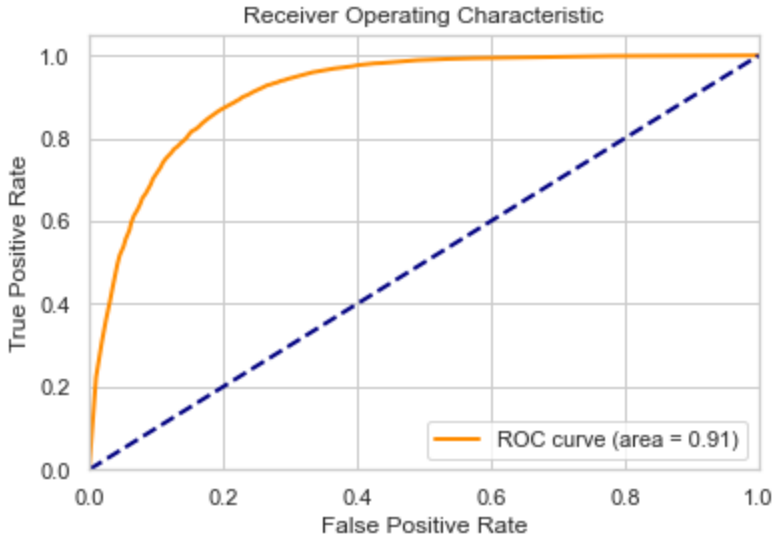
This model can provide insights into the importance of various features in predicting well functionality. It's less likely to overfit than individual decision trees and doesn't require feature scaling.

In [82]:
```
X = master_data.drop(['status_binary', 'status_binary_reversed', 'status_
                      'installer', 'permit', 'date_recorded', 'construct
y = master_data['status_binary_reversed']

# Initializing train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,

# Identifying categoricals
categorical_cols = X_train.select_dtypes(include=['object', 'category'])

# Creating a column transformer with OneHotEncoder for categoricals
column_transformer = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_cols
    ],
    remainder='passthrough'
)

# Applying the column transformer
X_train_encoded = column_transformer.fit_transform(X_train)
X_test_encoded = column_transformer.transform(X_test)

# Creating and training the Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train_encoded, y_train)

# Extracting feature names manually for older versions of scikit-learn
onehot_features = column_transformer.named_transformers_['cat'].get_featu
other_features = [col for col in X_train.columns if col not in categoric
feature_names = np.concatenate([onehot_features, other_features])

# Predicting and evaluating the model
y_pred = rf_model.predict(X_test_encoded)
print("Classification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
Classification Report:
               precision    recall  f1-score   support

           0       0.83      0.77      0.80      6856
           1       0.86      0.90      0.88     10756

    accuracy                           0.85     17612
   macro avg       0.85      0.84      0.84     17612
weighted avg       0.85      0.85      0.85     17612

Confusion Matrix:
 [[5290 1566]
 [1087 9669]]
```

Evaluating with ROC Curve

In [83]:
```
# Predict probas for the positive class
y_pred_proba = rf_model.predict_proba(X_test_encoded)[:, 1]

# Compute AUC-ROC
roc_auc = roc_auc_score(y_test, y_pred_proba)

# Compute ROC curve
fpr, tpr, _ = roc_curve(y_test, y_pred_proba)

# Plotting ROC Curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```



Evaluating Feature Importance

In [84]:
```
# Extracting and displaying feature importances
importances = rf_model.feature_importances_
importance_df = pd.DataFrame({'Feature': feature_names, 'Importance': imp
top_features = importance_df.sort_values(by='Importance', ascending=Fals
print(top_features)
```

```
                                Feature  Importance
84                             latitude    0.153992
83                            longitude    0.152580
62                        quantity_dry    0.101336
82                          gps_height    0.077973
85                          population    0.054906
80                waterpoint_type_other    0.040619
81                              amount_tsh    0.030302
33          extraction_type_class_other    0.030182
63                       quantity_enough    0.028917
87                        month_recorded    0.026894
64                quantity_insufficient    0.015844
51                payment_type_never pay    0.013426
75  waterpoint_type_communal standpipe    0.011889
30        extraction_type_class_gravity    0.010953
44                        management_vwc    0.010815
```

Tuning Random Forest Classifier

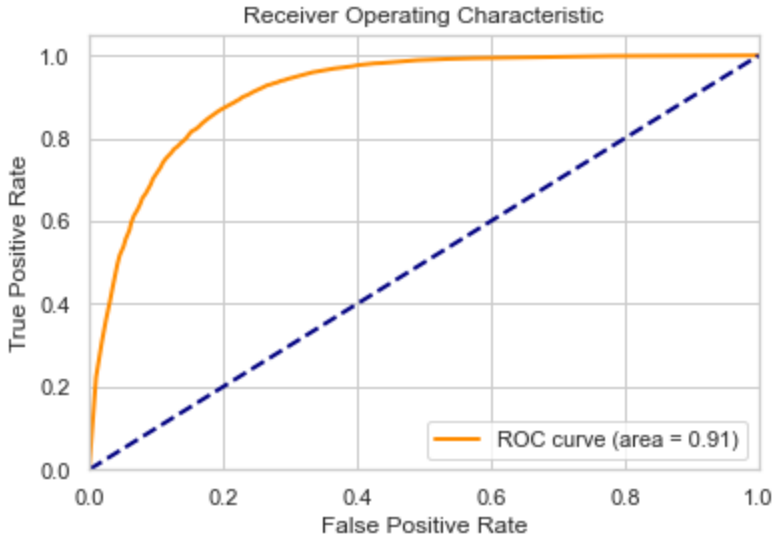
- SMOTE for oversampling the minority class or adjusting class weights in the model.
- Hyperparameter tuning of the Random Forest model
- RandomizedSearchCV will randomly sample 10 combos of parameters and use 3-fold cross-validation. This will reduce run time compared to GridSearchCV

In [85]:
```
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import RandomizedSearchCV

# Handling class imbalance with SMOTE
smote = SMOTE()
X_train_resampled, y_train_resampled = smote.fit_resample(X_train_encode

# Define the hyperparameter grid
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2']
}

# Hyperparameter tuning with Randomized Search
random_search = RandomizedSearchCV(
    RandomForestClassifier(random_state=42),
    param_grid,
    n_iter=10,
    cv=3,
    scoring='recall',
    n_jobs=-1
)

random_search.fit(X_train_resampled, y_train_resampled)

# Get the best model
best_model = random_search.best_estimator_

# Re-train and evaluate the model with the best params
best_model.fit(X_train_resampled, y_train_resampled)
y_pred = best_model.predict(X_test_encoded)

# Evaluate the model
print("Classification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
Classification Report:
               precision    recall  f1-score   support

           0       0.83      0.62      0.71      6856
           1       0.79      0.92      0.85     10756

    accuracy                           0.80     17612
   macro avg       0.81      0.77      0.78     17612
weighted avg       0.81      0.80      0.80     17612

Confusion Matrix:
 [[4258 2598]
 [ 862 9894]]
```

Evaluating with ROC Curve

In [86]:
```
# Predict probas for the positive class
y_pred_proba = rf_model.predict_proba(X_test_encoded)[:, 1]

# Compute AUC-ROC
roc_auc = roc_auc_score(y_test, y_pred_proba)

# Compute ROC curve
fpr, tpr, _ = roc_curve(y_test, y_pred_proba)

# Plotting ROC Curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```



Comparison Interpretations of Random Forest Models

Class 0 = non-functional/needs repair
Class 1 = functional

Precision:

- **Original Model:** Precision is 0.83 for both classes.
- **Tuned Model:** Precision increased to 0.85 for class 0 but decreased to 0.79 for class 1.

Recall:

- **Original Model:** Recall is 0.77 for class 0 and 0.90 for class 1.

- **Tuned Model:** Recall decreased to 0.62 for class 0 but increased to 0.93 for class 1.

F1-Score:

- **Original Model:** F1-scores are 0.80 (class 0) and 0.88 (class 1).
- **Tuned Model:** F1-scores are 0.72 (class 0) and 0.86 (class 1).

Accuracy:

- **Original Model:** Overall accuracy is 0.85.
- **Tuned Model:** Overall accuracy decreased to 0.81.

Macro and Weighted Averages:

- **Original Model:** Both macro and weighted averages are around 0.85.
- **Tuned Model:** Both macro and weighted averages are around 0.77 - 0.81.

AUC-ROC Score:

- **Original Model:** The area under the curve is 0.91, which is high. This means the model can effectively distinguish between the positive class (class 1) and the negative class (class 0).
- **Tuned Model:** The area under the curve is also 0.91.

Analysis of Comparison:

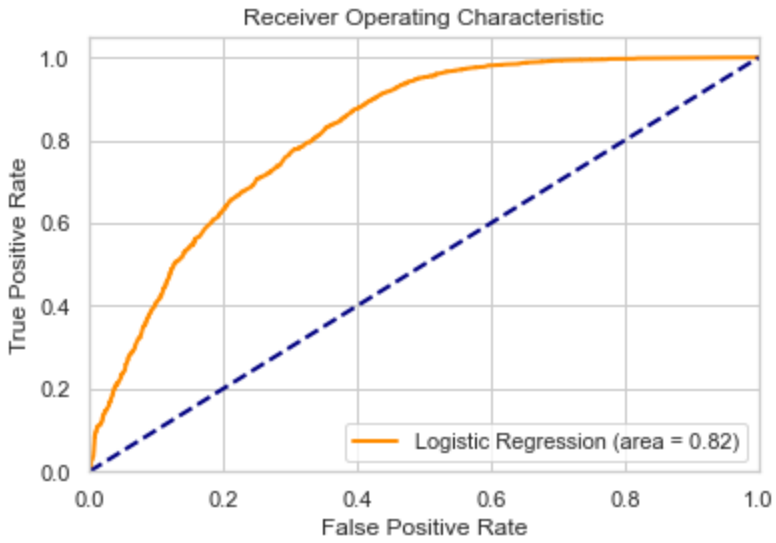
- Tuning the model appears to have made it more biased towards class 1, improving its ability to detect class 1 instances but worsening its performance for class 0 (higher false positives).
- The original model is more balanced in terms of precision and recall across both classes.
- The tuned model has a lower overall accuracy compared to the original model.

Logistic Regression Model: Most Important Features

In [87]:
```
# Selecting most important featues
features = ['region', 'quantity', 'gps_height', 'waterpoint_type', 'extra
X = master_data[features]
y = master_data['status_binary_reversed']

# Initializing train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,

# Identifying categorical and numerical
categorical_cols = ['quantity', 'region', 'waterpoint_type', 'extraction_
numerical_cols = ['gps_height']

# Creating a column transformer with OneHotEncoder and StandardScaler
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_cols),
        ('cat', OneHotEncoder(), categorical_cols)
    ]
)

# Creating a pipeline with preprocessing and log reg model
model_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', LogisticRegression(solver='saga', max_iter=1000, rando
])

# Training the model
model_pipeline.fit(X_train, y_train)

# Predicting and evaluating the model
y_pred = model_pipeline.predict(X_test)
print("Classification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.83      0.54      0.65      6856
           1       0.76      0.93      0.84     10756

    accuracy                           0.78     17612
   macro avg       0.79      0.73      0.74     17612
weighted avg       0.79      0.78      0.76     17612

Confusion Matrix:
 [[3694 3162]
 [ 768 9988]]
```

In [88]:
```
# Calculate probas, ROC curve, and AUC for log reg
logreg_probs = model_pipeline.predict_proba(X_test)[:, 1]
fpr_logreg, tpr_logreg, _ = roc_curve(y_test, logreg_probs)
roc_auc_logreg = auc(fpr_logreg, tpr_logreg)


# Plotting ROC Curve
plt.figure()
plt.plot(fpr_logreg, tpr_logreg, color='darkorange', lw=2, label='Logist
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```



Interpretation

Precision:

- For class 0: 83% precision means that out of all instances predicted as class 0, 83% were actually class 0.
- For class 1: 76% precision indicates that out of all instances predicted as class 1, 76% were actually class 1.

Recall:

- For class 0: The recall of 54% is moderate, meaning the model correctly identifies 54% of the actual class 0 instances.
- For class 1: A high recall of 93% indicates the model is very effective at identifying class 1 instances.

F1-Score:

- For class 0: The F1-score of 0.65 suggests a balance between precision and recall for class 0, but more weighted towards precision.
- For class 1: The F1-score of 0.84 indicates a strong balance between precision and recall for class 1, favoring recall.

Accuracy:

- The overall accuracy of 78% indicates that the model correctly predicts the class for 78% of all instances.

Macro and Weighted Averages:

- Macro average treats both classes equally, showing an average precision of 79%, recall of 73%, and F1-score of 74%.
- Weighted average considers class imbalance, showing slightly higher precision and recall, indicating better performance on the more prevalent class 1.

ROC Score:

- An ROC score of 0.82 suggests a good ability of the model to distinguish between the two classes. It indicates a favorable balance between the true positive rate and false positive rate across different thresholds.

Insights:

- The model performs well overall, especially in predicting class 1, which is indicated by the high recall and F1-score for class 1.
- Model is less effective in correctly identifying class 0 instances, as evidenced by the lower recall for class 0.
- The relatively high number of false positives for class 0 (3162) indicates that the model

Key Findings

- **Geographic Indicators:** Including region and altitude, geographic features are 21% MORE influential in identifying non-functional wells than other features.
- **Region:** Mbeya, Morogoro, and Kilimanjaro have the highest rates of non-functional wells. Altitude may play an important role in water source access.
- **Type of Wells:** Communal Standpipe wells are most likely to be a functional well. Other well types have the highest percentage of non-functional wells at 81.38%. Multi Communal Handpipe wells have the second highest percentage of non-functional wells at 53.85%.
- **Payment Type:** Whether a well is paid seems to be a crucial factor. Wells that are not paid have a high number of non-functional wells.
- **Random Forest Classifier:** Our best performing model gave us actionable insights into feature importance and effectively minimized false-negatives.

Conclusion

In this project, we have unearthed critical insights to steer the strategic decision-making of the Tanzanian Government. This includes pinpointing the precise types of wells that warrant prioritized construction efforts, as well as identifying the specific regions that should receive initial focus and substantial investment in well infrastructure.

Next Steps

- **Investigate Additional Features:** Concentrating on geographical indicators like climate, population, and amount of water available in the area.
- **Time-Series Analysis:** Further consideration of the well ages should be analyzed to predict the average lifetime of more robust well structures.
- **Repairs:** Local governments should look at what type of water wells are needing repairs, and the severity of those repairs, to fine-tune non-functional indicators.

Sources

- Driven Data - Tanzanian Water Wells (https://www.drivendata.org/competitions/7/pump-it-up-data-mining-the-water-table/page/23/)
  - Labels (https://www.drivendata.org/competitions/7/pump-it-up-data-mining-the-water-table/data/)
  - Values (https://www.drivendata.org/competitions/7/pump-it-up-data-mining-the-water-table/data/)

- The World Bank (https://www.worldbank.org/en/news/feature/2022/06/14/the-road-to-better-services-is-paved-with-strong-delivery-institutions-a-rural-water-story)
- Groundwater Wells (https://www.usgs.gov/special-topics/water-science-school/science/groundwater-wells)
- Detection of Non-Function Bore Wells Using Maching Learning Algorithms (https://www.researchgate.net/publication/349446319_Detection_of_Non-functional_Bore_wells_Using_Machine_Learning_Algorithms)

About Us

Goknur Kaya

- Github Lead
- Email: goknurkaya@gmail.com (mailto:goknurkaya@gmail.com)
- github.com/goknurk

Kari Primiano

- Tech Lead
- Email: kkprim@gmail.com (mailto:kkprim@gmail.com)

- github.com/kkprim

Mytreyi Abburu

- Presentation Lead
- Email: abburumk@gmail.com (mailto:abburumk@gmail.com)
- github.com/myt-hue