Brian-Njau / dsc-phase-3-project Public

0 stars 0 forks

Star Watch

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master

Brian-Njau Tanzanian water wells ... 1 minute ago 8

View code

README.md

Introduction

Tanzania, as a developing country, struggles with providing clean water to its population of over 57,000,000. There are many water points already established in the country, but some are in need of repair while others have failed altogether.

The purpose of this project is to build a classifier that can predict the condition of a water well, given information about the sort of pump, when it was installed, etc. Our audience is the Government of Tanzania which is looking to find patterns in non-functional wells in order to influence how new wells are built.

Note that this is a ternary classification problem by default, but can be engineered to be binary.

Lets get started

Importing Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OrdinalEncoder
```

Tuning Pandas

```
pd.options.mode.chained_assignment = None
pd.set_option('display.max_columns',None)
```

Data Undertsanding

The data for this project comes from the Taarifa waterpoints dashboard, which aggregates data from the Tanzanian Ministry of Water.

Learn More here https://taarifa.org/

```
#Reading files
X_train=pd.read_csv('data/trainingsetvalues.csv')
y_train=pd.read_csv('data/trainingsetlabels.csv')
#Merging files
df=pd.merge(X_train,y_train,on='id')
```

```
#Displaying first five rows
df.head()
```

We are provided the following set of information about the waterpoints:

```
amount_tsh - Total static head (amount water available to waterpoint)

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 waterpoint if there is one

num_private -

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 waterpoint

scheme_name - Who operates the waterpoint

permit - If the waterpoint is permitted

construction_year - Year the waterpoint was constructed

extraction_type - The kind of extraction the waterpoint uses

extraction_type_group - The kind of extraction the waterpoint uses

extraction_type_class - The kind of extraction the waterpoint uses

management - How the waterpoint is managed

management_group - How the waterpoint 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 waterpoint

waterpoint_type_group - The kind of waterpoint
```

Exploratory Data Analysis

```
#(Number of rows,number of columns)
df.shape
```

```
#Overall Data Information
df.info()
```

```
#Basic Data Statistics
df.describe()
```

```
#Unique Target Variables
df['status_group'].value_counts(normalize=True)
```

```
#Null Values
df.isnull().sum()
```

Preprocessing

Encoding and Imputing All Categorical Features

```
#Initializing Encoder
encoder = OrdinalEncoder()

#Extracting Categorical Data
categorical_data=df.select_dtypes('object')
categorical_columns=categorical_data.columns

#Defining Encoder Function
def encode_data(col_data):
    #function to encode non-null data and replace it in the original data
    non_nulls=np.array(col_data.dropna())
    #reshaping the data for encoding
    reshaped_data=non_nulls.reshape(-1,1)
    #encoding
    encoded_data=encoder.fit_transform(reshaped_data)
    #replace data
    col_data.loc[col_data.notnull()]=np.squeeze(encoded_data)
    return col_data

#Defining Imputer Function
def impute_data(col_data):
    col_data.fillna(col_data.value_counts().index[0], inplace=True)
    return col_data

#Transforming Data
for column in categorical_columns:
    encode_data(categorical_data[column])
    impute_data(categorical_data[column])
categorical_data
```

```
#Null values
categorical_data.isnull().sum()
```

Checking Numeric Features and Updating DataFrame

```
#Extracting numeric data
numeric_data=df.select_dtypes(['float64','int64'])
#Checking null values
numeric_data.isnull().sum()
```

```
#Updating DataFrame
df[categorical_columns]=categorical_data
df.head()
```

Standardizing Features

```
#Importing Function
from sklearn.preprocessing import StandardScaler
#Scaling data
scaler=StandardScaler()
df=pd.DataFrame(scaler.fit_transform(df),columns=df.columns)
df.head()
```

Dropping `id` Column in preparation for modeling

```
df.drop('id',axis=1,inplace=True)
```

```
df[:1]
```

Building, Tuning and Evaluating Models

```
#Performing a train-test split
from sklearn.model_selection import train_test_split
X=df.drop('status_group',axis=1)
y=df['status_group'].astype(int)
X_train,X_test,y_train,y_test=train_test_split(X,y,random_state=42,test_size=0.4)
```

> We will be using GridSearchCV as our hyperparameter tuning method and accuracy score as our evaluation metric

```
# Importing Tools
from sklearn.metrics import accuracy_score
from sklearn.model_selection import GridSearchCV
rs=11
```

```
# Function for fitting and testing
def fit_and_test(model):
    model.fit(X_train,y_train)
    #Making Predictions
    train_predictions=model.predict(X_train)
    test_predictions=model.predict(X_test)
    #Computing Accuracy
    train_accuracy=accuracy_score(y_train,train_predictions)
    test_accuracy=accuracy_score(y_test,test_predictions)

    print(f'{str(model)} Results: \
    \n Train Accuracy: {train_accuracy}\
    \n Test Accuracy: {test_accuracy}')
```

```
# Function for hyperparameter tuning
def find_params(model,param_grid):
    model_cv=GridSearchCV(model,param_grid,cv=5)
    model_cv.fit(X_train,y_train)
    params=model_cv.best_params_
    values=list(params.values())
    return (values[i] for i in range(len(values)))
```

Note: Tuning cells have been commented after computation to reduce run

MultiClass Logistic Regressor

```
Vanilla
```

```
#Importing modules
from sklearn.linear_model import LogisticRegression
#Initializing Model
logistic_model=LogisticRegression(multi_class='multinomial',solver='lbfgs',random_state=rs)
#Fitting and Testing Model
fit_and_test(logistic_model)
```

```
Tuned
```

```
# lg_param_grid = {'C': [1,5],
#                'max_iter': [100, 500],
#                'tol': [0.0001]}
# #Extracting Optimal Parameters
# c,m,t=find_params(logistic_model,lg_param_grid)
# #Building Tuned Model
# tuned_logistic_model=LogisticRegression(C=c,max_iter=m,tol=t)
# #Fitting and Testing Model
# fit_and_test(tuned_logistic_model)
```

Decision Tree Classifier

Vanilla

```
#Importing functions
from sklearn.tree import DecisionTreeClassifier
#Initializing Model
decision_tree_model=DecisionTreeClassifier(max_depth=14)
#Fitting and Testing Model
fit_and_test(decision_tree_model)
```

Tuned

```
# dt_param_grid = {'criterion': ['entropy','gini'],
#                  'min_samples_split': [20,26]}
# c,ms=find_params(decision_tree_model,dt_param_grid)
# #Building Tuned Model
# tuned_decision_tree_model=DecisionTreeClassifier(criterion=c,min_samples_split=ms)
# #Fitting and Testing Model
# fit_and_test(tuned_decision_tree_model)
```

K-Nearest Neighbors

Vanilla

```
#Importing Functions
from sklearn.neighbors import KNeighborsClassifier
# import warnings filter
from warnings import simplefilter
# ignore all future warnings
simplefilter(action='ignore', category=FutureWarning)
```

```
#Initializing model
knn_model=KNeighborsClassifier(n_neighbors=5)
#Fitting and Testing Model
fit_and_test(knn_model)
```

Tuned

```
# kn_param_grid = {'n_neighbors': [3, 5, 7],
#                  'weights': ['uniform', 'distance']}
# n,w=find_params(knn_model,kn_param_grid)
# #Building tuned model
# tuned_knn_model=KNeighborsClassifier(n_neighbors=n,weights=w)
# #Fitting and Testing Model
# fit_and_test(tuned_knn_model)
```

Random Forest

Vanilla

```
#Importing Function
from sklearn.ensemble import RandomForestClassifier
#Initializing Model
forest_model=RandomForestClassifier()
#Fitting and Testing Model
fit_and_test(forest_model)
```

Tuned

```
# rf_param_grid = {
#     'n_estimators': [50, 100],
#     'max_depth' : [40,80]
# }
```

```
# n,mf,md,c=find_params(forest_model,rf_param_grid)
# #Building Tuned Model
# tuned_forest_model = RandomForestClassifier(n_estimators=n,max_features=mf,max_depth=md,criterion=c)
# #Fitting and Testing Model
# fit_and_test(tuned_forest_model)
```

Bayesian Classifier

```
Vanilla
```

```
#Importing module
from sklearn.naive_bayes import GaussianNB
#Initializing model
bayes_model=GaussianNB()
#Fitting and Testing Model
fit_and_test(bayes_model)
```

```
Tuned
```

```
# bayes_param_grid = {
#     'var_smoothing': np.logspace(0,-9, num=100)
# }
# var=find_params(bayes_model,bayes_param_grid)
# #Building Tuned Model
# tuned_bayes_model=GaussianNB(var_smoothing=var)
# #Fitting and Testing Model
# fit_and_test(tuned_bayes_model)
```

Adaptive Boosting Classifier

```
Vanilla
```

```
#Importing module
from sklearn.ensemble import AdaBoostClassifier
#Initializing model
adaboost_model=AdaBoostClassifier()
#Fitting and Testing Model
fit_and_test(adaboost_model)
```

Tuned

```
# ab_param_grid = {
#     'base_estimator': [DecisionTreeClassifier(max_depth=1), DecisionTreeClassifier(max_depth=2)],
#     'n_estimators': [50, 100, 200],
#     'learning_rate': [0.01, 0.1, 1],
# }
# b,n,l=find_params(adaboost_model,ab_param_grid)
# #Building tuned model
# tuned_adaboost_model=AdaBoostClassifier(base_estimator=b,n_estimators=n,learning_rate=l)
# #Fitting and testing Model
# fit_and_test(tuned_adaboost_model)
```

Extra Trees Classifier

Vanilla

```
#Importing Module
from sklearn.ensemble import ExtraTreesClassifier
#Initializing model
extra_trees_model=ExtraTreesClassifier()
#Fitting and Testing Model
fit_and_test(extra_trees_model)
```

Tuned

```
# et_param_grid = {
#     'n_estimators': [50, 100, 200],
#     'max_depth': [1, 2, 3],
#     'min_samples_split': [2, 4],
#     'min_samples_leaf': [1, 2],
# }
# n,md,ms,ml=find_params(extra_trees_model,et_param_grid)
# #Building Tuned Model
# tuned_extra_trees_model=ExtraTreesClassifier(n_estimators=n,max_depth=md,min_samples_split=ms,min_samples_leaf=ml)
# #Fitting and Testing Model
# fit_and_test(tuned_extra_trees_model)
```

Gradient Boosting Classifier

Vanilla

```
#Importing module
from sklearn.ensemble import GradientBoostingClassifier
#Initializing model
gradient_boost_model=GradientBoostingClassifier()
#Fitting Data
fit_and_test(gradient_boost_model)
```

Tuned

```
# gb_param_grid = {
#     'n_estimators': [50, 100, 200],
#     'max_depth': [1, 2, 3],
#     'min_samples_split': [2, 4],
```

```
#     'min_samples_leaf': [1, 2],
# }
# n,md,ms,ml=find_params(gradient_boost_model,gb_param_grid)
# #Building Tuned Model
# tuned_gradient_boost_model=GradientBoostClassifier(n_estimators=n,max_depth=md,min_samples_split=ms,min_samples_le
# #Fitting and Testing model
# fit_and_test(tuned_gradient_boost_model)
```

Model Selection

ROC and AUC

(Receiver Operating Characteristics)

```
#Importing Module
import sklearn.metrics as metrics
#Listing models
models = [
    { 'label': 'Logistic Regression', 'model': logistic_model},
    { 'label': 'Decision Tree', 'model': decision_tree_model},
    { 'label': 'K-Neighbors', 'model': decision_tree_model},
    { 'label': 'Random Forest', 'model': forest_model},
    { 'label': 'Gaussian Bayes', 'model': bayes_model},
    { 'label': 'Adaptive Boosting', 'model': adaboost_model},
    { 'label': 'Extra Trees', 'model': extra_trees_model},
    { 'label': 'Gradient Boosting', 'model': gradient_boost_model}
]
#Plotting ROC
plt.figure()
for m in models:
    model = m['model'] # select the model
```

```
    y_pred=model.predict(X_test) # predict the test data
# Compute False postive rate, and True positive rate
    fpr, tpr, thresholds = metrics.roc_curve(y_test, model.predict_proba(X_test)[:,1])
# Calculate Area under the curve to display on the plot
    auc = metrics.roc_auc_score(y_test,y_pred)
# Now, plotting the computed values
    plt.plot(fpr, tpr, label='%s ROC (area = %0.2f)' % (m['label'], auc))
# Custom settings for the plot
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('1-Specificity(False Positive Rate)')
plt.ylabel('Sensitivity(True Positive Rate)')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()   # Display
```

> The **Random Forests** model has the highest `AUC` (0.84) as well as the highest `accuracy score` (86%). We shall therefore be selecting it as our final and best model. **Our model of choice**

Pickling the model

```
import pickle
import joblib

# Save the model as a pickle in a file
joblib.dump(forest_model, 'random_forest_model.pkl')
```

Testing Pickled Model

```
f = open('random_forest_model.pkl','rb')
loaded_model = joblib.load(f)
f.close()
load_prediction = loaded_model.predict(X_test)
load_prediction_accuracy = accuracy_score(y_test,load_prediction)
print(f'Loaded model accuracy: {np.round(load_prediction_accuracy*100,2)}%')
```

Recommendation

With such predictive power, the Government of Tanzania can now forecast and infer the condition of a water well. We therefore recommend the utilization of this algorithm to improve operational efficiency

Conclusion

More and more businesses are leveraging on the use of data and machine learning to drive decision making. It has been an honor working with the Tanzanian Government. When we work together we grow together

No packages published
Publish your first package

Languages

- **Jupyter Notebook** 100.0%