KImondorose / phase_3_project-Pump-it-Up-Data-Mining-the-Water-Table

Code Issues Pull requests Actions Projects Security Insights Settings

0 stars 0 forks 1 watching 1 Branch 0 Tags Activity

Private repository

1 Branch 0 Tags Go to file t Go to file + Add file Code ...

KImondorose Added repository pdf 6e0708c · now

data	loaded the data	last week
images	loaded the data	last week
.gitignore	loaded the data	last week
README.md	Added repository pdf	now
clean_data.csv	Finished EDA and Feature Exploration	3 days ago
notebook.ipynb	Final notebook run	6 minutes ago
notebook.pdf	Added Notebook PDF	4 minutes ago
presentation (2).pdf	Added presentation	17 minutes ago
repository.pdf	Added repository pdf	now

README

- Slideshow Presentation
- Jupyter Notebook with Cleaning & Questions
- Repository pdf
- Notebook PDF
- Data Sources
-

Introduction and Problem Statement

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 aim of this project is to build a classifier that predicts the condition of a water well (functional, non-function, or functional but needs repair), using information such as the extraction type, how it is managed, payment type, waterpoint type, the water source, whether it has a permit, and whether a public meeting was held.

With this model, the aim is to help the Government of Tanzania find patterns in non-functional wells to influence how new wells are built.

Objectives

1. Analyze the relationship between the following variables and the `status_group` (functional, non-functional, functional but needs repair) to identify patterns in non-functional wells:

- `Payment` : What the water costs
- `source` : The source of the water
- `management_group` : How the waterpoint is managed
- `extraction_type` : The kind of extraction the waterpoint uses
- `permit` : If the waterpoint is permitted
- `public _meeting` : Whether a public meeting was held for the well

2. Develop a classification model to predict the condition of a well (functional, non-functional, or non-functional but needs repair)

Data Understanding

The original data can be obtained on the [DrivenData 'Pump it Up: Data Mining the Water Table'] competition. Basically, there are 4 different data sets; submission format, training set, test set and train labels set which contains status of wells. With given training set and labels set, competitors are expected to build predictive model and apply it to test set to determine status of the wells and submit.

In this project, we will use train set and train label set. Train set has 59400 water points data with 40 features. Train labels data has the same 59400 water points as train set, but just has information about id of these points and status of them.

Limitations of the data

1. The target labels are ternary and highly imbalanced:

- functional: 32259
- non functional: 22824
- functional needs repair: 4317

  This might impact the performance of the model

2. The data contains 40 features, which are very many, which means that a lot of work will go into EDA and feature elimination

Methods/Data Analysis

After opening the raw training datasets with pandas, we cleaned and prepared the data by imputing missing values, eliminating some redundant columns that contained similar information, identifying the columns with the most variable information for modeling, removing outliers from some numerical columns, encoding categorial variables for modeling.
The 15 final selected features for the model were:

- `basin`
- `month_recorded`
- `region`
- `extraction_type_class`
- `management_group`
- `payment`
- `quality_group`
- `quantity`
- `source`
- `waterpoint_type`
- `gps_height`
- `population`
- `construction_age`
- `permit`
- `public_meeting`

EDA and Feature Exploration

training_data.shape We now have one dataframe with 59,400 rows and 41 columns, with the 41st column being the status_group column training_data.info() training_data.columns These are all the columns and their descriptions:

- **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
- **status_group** - The labels in this dataset with three possible values: functional, non-functional, and functional needs repair training_data.isna().sum() # to see the null values

Visualizations

Water Quality by Number of Wells

plt.figure(figsize=(10,8)) ax = sns.countplot(x='quality_group', hue="status_group", data=training_data, palette = 'icefire') ax.set_xlabel('Payment') ax.set_ylabel('Number of Wells') ax.set_title('Wells By Quality Group');

Wells by Payment Type

plt.figure(figsize=(10,8)) ax = sns.countplot(x='payment', hue="status_group", data=training_data, palette = 'icefire') ax.set_xlabel('Payment') ax.set_ylabel('Number of Wells') ax.set_title('Wells By Payment Type');

Wells by Waterpoint Type

plt.figure(figsize=(17,10)) ax = sns.countplot(x='waterpoint_type', hue="status_group", data=training_data, palette = 'icefire') ax.set_xlabel('Waterpoint Type') ax.set_ylabel('Number of Wells') ax.set_title('Wells By Waterpoint Type');

Wells by Quantity

plt.figure(figsize=(10,8)) ax = sns.countplot(x='quantity', hue="status_group", data=training_data, palette = 'icefire') ax.set_xlabel('Quantity') ax.set_ylabel('Number of Wells') ax.set_title('Wells By Quantity');

Wells By Basin

plt.figure(figsize=(16,8)) ax = sns.countplot(x='basin', hue="status_group", data=training_data, palette = 'icefire') ax.set_xlabel('Basin') ax.set_ylabel('Number of Wells') ax.set_title('Wells By Basin');

Wells by Source

plt.figure(figsize=(16,8)) ax = sns.countplot(x='source', hue="status_group", data=training_data, palette = 'icefire') ax.set_xlabel('Source') ax.set_ylabel('Number of Wells') ax.set_title('Wells By Source');

Wells By Management Group

plt.figure(figsize=(10,8)) ax = sns.countplot(x='management_group', hue="status_group", data=training_data, palette = 'icefire') ax.set_xlabel('Management Group') ax.set_ylabel('Number of Wells') ax.set_title('Wells By Management Group');

Wells By Extraction Type Class

plt.figure(figsize=(10,8)) ax = sns.countplot(x='extraction_type_class', hue="status_group", data=training_data, palette = 'icefire') ax.set_xlabel('Exatraction Type Class') ax.set_ylabel('Number of Wells') ax.set_title('Wells By Extraction Type Class');

Wells By Permit

plt.figure(figsize=(10,8)) ax = sns.countplot(x='permit', hue="status_group", data=training_data, palette = 'icefire') ax.set_xlabel('Permit') ax.set_ylabel('Number of Wells') ax.set_title('Wells By Permit');

Wells By Public Meeting

plt.figure(figsize=(10,8)) ax = sns.countplot(x='public_meeting', hue="status_group", data=training_data, palette = 'icefire') ax.set_xlabel('Public Meeting') ax.set_ylabel('Number of Wells') ax.set_title('Wells By Public Meeting');

Modeling

Model Training

Model 1: Logistic Regression Iteration 1

instantiate the model

logreg = LogisticRegression(solver='sag', max_iter=400, multi_class='auto', random_state=42)

fit the model

logreg.fit(X_train, y_train) y_pred_test = logreg.predict(X_test)

y_pred_test

Check accuracy score

from sklearn.metrics import accuracy_score

print('Model accuracy score: {0:0.4f}'. format(accuracy_score(y_test, y_pred_test))) Here, **y_test** are the true class labels and **y_pred_test** are the predicted class labels in the test-set.

Compare the train-set and test-set accuracy

Now, I will compare the train-set and test-set accuracy to check for overfitting. y_pred_train = logreg.predict(X_train)

y_pred_train print('Training-set accuracy score: {0:0.4f}'. format(accuracy_score(y_train, y_pred_train))) Check for overfitting or underfitting

print the scores on training and test set

print('Training set score: {:.4f}'.format(logreg.score(X_train, y_train)))

print('Test set score: {:.4f}'.format(logreg.score(X_test, y_test))) The training-set accuracy score is 0.7311 while the test-set accuracy is 0.7297. These two values are quite comparable. So, there is no question of overfitting.

In Logistic Regression, we use default value of C = 1. It provides good performance with approximately 73% accuracy on both the training and the test set. But the model performance on both the training and test set are very comparable. It is likely the case of underfitting.

I will increase C and fit a more flexible model.

fit the Logsitic Regression model with C=100

instantiate the model

logreg100 = LogisticRegression(C=100, solver='sag', random_state=42)

fit the model

logreg100.fit(X_train, y_train)

print the scores on training and test set

print('Training set score: {:.4f}'.format(logreg100.score(X_train, y_train))) print('Test set score: {:.4f}'.format(logreg100.score(X_test, y_test))) We can see that, C=100 results in slightly test set accuracy and also a slightly reduced training set accuracy. So, we can conclude that a more complex model does not perform better. Now, I will investigate, what happens if we use more regularized model than the default value of C=1, by setting C=0.01.

fit the Logsitic Regression model with C=001

instantiate the model

logreg001 = LogisticRegression(C=0.01, solver='sag', random_state=42)

fit the model

logreg001.fit(X_train, y_train)

print the scores on training and test set

print('Training set score: {:.4f}'.format(logreg001.score(X_train, y_train)))

print('Test set score: {:.4f}'.format(logreg001.score(X_test, y_test))) The training and test set accuracy scores are also lower than the original, so we will continue from the one that used default c= 1 This will be our baseline model. Now let's try to change up some things and see what that does to our model

from sklearn.metrics import confusion_matrix

cm_logreg1 = confusion_matrix(y_test, y_pred_test) print('Confusion matrix\n\n', cm_logreg1)

sns.heatmap(cm_logreg1, annot=True,fmt='d') plt.ylabel('Prediction',fontsize=12)
plt.xlabel('Actual',fontsize=12) plt.title('Confusion Matrix',fontsize=16)

plt.show();

Remember that:
0: Functional:
1: functional needs repair
2: non-functional from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_test)) `weighted`: Calculate precision and recalls metrics for each label, and find their average weighted by support (the number of true instances for each label).
This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall. from sklearn.metrics import precision_score, recall_score, balanced_accuracy_score, f1_score

precision = precision_score(y_test, y_pred_test, average='weighted') recall = recall_score(y_test, y_pred_test, average = "weighted") f1_score = f1_score(y_test, y_pred_test, average="weighted")

accuracy_score = accuracy_score(y_test, y_pred_test)

print("The weighted precision is: ",precision) print("The weighted recall is: ",recall) print("The accuracy score is: ", accuracy_score) print("The weighted f1score is: ", f1_score) print(" ")

print("Weighted precision, f1 score, and recall are ideal measures because the target labels are imbalanced")

Precision: 72%
Recall: 73% The weighted average precision and recall are preferred as evaluation metrics.
It is appropriate to use weighted averaging. This approach takes into account the balance of classes. You weigh each class based on its representation in the dataset. Then, you compute precision and recall as a weighted average of the precision and recall in individual classes.

Simply put, it would work like macro-averaging, but instead of dividing precision and recall by the number of classes, you give each class a fair representation based on the proportion it takes in the dataset.

This approach is useful if you have an imbalanced dataset, like we have here but want to assign larger importance to classes with more examples.

Model 2- Logistic Regression Iteration Two

Our target variable is hughly imbalanced. We will create a copy and use SMOTE to balance it then create a second iteration Xy_init.status_group.value_counts() Xy_init_smote = Xy_init.copy() # assign to protect original one

X_smote = Xy_init_smote.drop(['status_group'], axis=1) # assign X variables

y_smote = Xy_init_smote['status_group'] # Assign y variable

from sklearn.model_selection import train_test_split

X_train2, X_test2, y_train2, y_test2 = train_test_split(X_smote, y_smote, test_size = 0.2, random_state = 42) # Train test split

Remove outliers for numerical data

upper_thresholds = { 'population': 560.0, 'construction_age': 52.0, }

for df3 in [X_train2, X_test2]: for column, top in upper_thresholds.items(): df3[column] = df3[column].clip(upper=top)

Obe Hot encode categorical data

import category_encoders as ce

ohe = ce.one_hot.OneHotEncoder(cols=['basin','month_recorded','region', 'extraction_type_class', 'management_group', 'payment', 'quality_group', 'quantity', 'source', 'waterpoint_type'], handle_missing="value", handle_unknown="ignore")

ohe.fit(X_train2)

X_train2 = ohe.transform(X_train2) X_test2 = ohe.transform(X_test2)

Scale the data

from sklearn.preprocessing import MinMaxScaler

scaler2 = MinMaxScaler()

scaler2.fit(X_train2)

X_train2 = pd.DataFrame(scaler.transform(X_train2), columns=X_train2.columns)

X_test2 = pd.DataFrame(scaler.transform(X_test2), columns=X_test2.columns)

SMOTE

from imblearn.over_sampling import SMOTE

smt = SMOTE(sampling_strategy = 'auto', n_jobs = -1)

X_smote_sampled, y_smote_sampled = smt.fit_resample(X_train2, y_train2)

print(y_smote.value_counts())

y_smote_sampled = pd.Series(y_smote_sampled) # converting from array to np.series to see value_counts

print(y_smote_sampled.value_counts())

Instantiate and Fit Logistic Regression Model

instantiate the model

logreg2 = LogisticRegression(solver='sag', multi_class='auto', random_state=42)

fit the model

logreg2.fit(X_smote_sampled, y_smote_sampled) y_pred_test2 = logreg2.predict(X_test2)

y_pred_test2 from sklearn.metrics import accuracy_score

print('Model accuracy score: {0:0.4f}'. format(accuracy_score(y_test2, y_pred_test2)))

Compare the train-set and test-set accuracy

Now, I will compare the train-set and test-set accuracy to check for overfitting. y_pred_train2 = logreg2.predict(X_train2)

y_pred_train2 print('Training-set accuracy score: {0:0.4f}'. format(accuracy_score(y_train2, y_pred_train2))) Check for overfitting or underfitting

print the scores on training and test set

print('Training set score: {:.4f}'.format(logreg2.score(X_train2, y_train2)))

print('Test set score: {:.4f}'.format(logreg2.score(X_test2, y_test2))) The training and test accuracy score are much lower than the first model (62%), meaning that it is less accurate. Using SMOTE does not improve our model from sklearn.metrics import confusion_matrix

cm_logreg3 = confusion_matrix(y_test2, y_pred_test2) print('Confusion matrix\n\n', cm_logreg3) sns.heatmap(cm_logreg3, annot=True,fmt='d') plt.ylabel('Prediction',fontsize=12) plt.xlabel('Actual',fontsize=12) plt.title('Confusion Matrix',fontsize=16) plt.show(); from sklearn.metrics import classification_report

print(classification_report(y_test2, y_pred_test2)) `weighted`: Calculate precision and recalls metrics for each label, and find their average weighted by support (the number of true instances for each label).
This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall. from sklearn.metrics import precision_score, recall_score, balanced_accuracy_score, f1_score

precision2 = precision_score(y_test, y_pred_test2, average='weighted') recall2 = recall_score(y_test, y_pred_test2, average = "weighted") f1_score2 = f1_score(y_test, y_pred_test2, average="weighted")

accuracy_score2 = accuracy_score(y_test, y_pred_test2)

print("The weighted precision is: ",precision2) print("The weighted recall is: ",recall2) print("The accuracy score is: ", accuracy_score2) print("The weighted f1score is: ", f1_score2) print(" ")

print("Weighted precision, f1 score, and recall are ideal measures because the target labels are imbalanced")

Precision: 74%
Recall: 62% Let's now try using the standard scaler instead on the MinMax Scaler on the original data to see if this creates a better model

Model 3: Linear Regression Iteration 3

Xy_init_std_scaler = Xy_init.copy() # assign to protect original one

X_std_scaler = Xy_init_std_scaler.drop(['status_group'], axis=1) # assign X variables

y_std_scaler = Xy_init_std_scaler['status_group'] # Assign y variable

from sklearn.model_selection import train_test_split

X_train3, X_test3, y_train3, y_test3 = train_test_split(X_std_scaler, y_std_scaler, test_size = 0.2, random_state = 42) # Train test split

Remove outliers for numerical data

upper_thresholds = { 'population': 560.0, 'construction_age': 52.0, }

for df3 in [X_train3, X_test3]: for column, top in upper_thresholds.items(): df3[column] = df3[column].clip(upper=top)

Obe Hot encode categorical data

import category_encoders as ce

ohe = ce.one_hot.OneHotEncoder(cols=['basin','month_recorded','region', 'extraction_type_class', 'management_group', 'payment', 'quality_group', 'quantity', 'source', 'waterpoint_type'], handle_missing="value", handle_unknown="ignore")

ohe.fit(X_train3)

X_train3 = ohe.transform(X_train3) X_test3 = ohe.transform(X_test3)

Scale the data

from sklearn.preprocessing import StandardScaler

scaler3 = StandardScaler()

scaler3.fit(X_train3)

X_train3 = pd.DataFrame(scaler3.transform(X_train3), columns=X_train3.columns)

X_test3 = pd.DataFrame(scaler3.transform(X_test3), columns=X_test3.columns)

Instantiate and Fit Logistic Regression Model

instantiate the model

logreg3 = LogisticRegression(solver='sag', multi_class='auto', random_state=42)

fit the model

logreg3.fit(X_train3, y_train3) y_pred_test3 = logreg3.predict(X_test3)

y_pred_test3 from sklearn.metrics import accuracy_score

print('Model accuracy score: {0:0.4f}'. format(accuracy_score(y_test3, y_pred_test3)))

Compare the train-set and test-set accuracy

Now, I will compare the train-set and test-set accuracy to check for overfitting.

Compare the train-set and test-set accuracy

Now, I will compare the train-set and test-set accuracy to check for overfitting.

y_pred_train3 = logreg3.predict(X_train3)

y_pred_train3

print('Training-set accuracy score: {0:0.4f}'. format(accuracy_score(y_train3, y_pred_train3)))

Check for overfitting or underfitting

print the scores on training and test set

print('Training set score: {:.4f}'.format(logreg3.score(X_train3, y_train3)))

print('Test set score: {:.4f}'.format(logreg3.score(X_test3, y_test3)))

The training and test accuracy score are similar to the first model at 0.73 and 0.729 from sklearn.metrics import confusion_matrix

cm_logreg4 = confusion_matrix(y_test3, y_pred_test3) print('Confusion matrix\n\n', cm_logreg4)

sns.heatmap(cm_logreg4, annot=True,fmt='d') plt.ylabel('Prediction',fontsize=12) plt.xlabel('Actual',fontsize=12) plt.title('Confusion Matrix',fontsize=16) plt.show(); from sklearn.metrics import classification_report

print(classification_report(y_test3, y_pred_test3)) `weighted`: Calculate precision and recalls metrics for each label, and find their average weighted by support (the number of true instances for each label).
This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall. from sklearn.metrics import precision_score, recall_score, balanced_accuracy_score, f1_score

precision3 = precision_score(y_test, y_pred_test3, average='weighted') recall3 = recall_score(y_test, y_pred_test3, average = "weighted") f1_score3 = f1_score(y_test, y_pred_test3, average="weighted")

accuracy_score3 = accuracy_score(y_test, y_pred_test3)

print("The weighted precision is: ",precision3) print("The weighted recall is: ",recall3) print("The accuracy score is: ", accuracy_score3) print("The weighted f1score is: ", f1_score3) print(" ")

print("Weighted precision, f1 score, and recall are ideal measures because the target labels are imbalanced")

Precision: 72%
Recall: 73%

Decision Tree

Model 4: Decision Tree- Model Iteration 1

from sklearn import tree from sklearn.tree import DecisionTreeClassifier dt= DecisionTreeClassifier(criterion='gini', splitter='best', random_state=42)

dt.fit(X_train, y_train)

Compare the train-set and test-set accuracy

Now, I will compare the train-set and test-set accuracy to check for overfitting. print('Accuracy score of train data :{}'.format(dt.score(X_train,y_train))) print('Accuracy score of test data:{}'.format(dt.score(X_test,y_test))) This high variance in accuracy scores indicates that we are overfitting. Let's change the depth and use entropy and see the difference it brings y_pred_test4 = dt.predict(X_test)

y_pred_test4 from sklearn.metrics import confusion_matrix

cm_logreg2 = confusion_matrix(y_test, y_pred_test4) print('Confusion matrix\n\n', cm_logreg2)

sns.heatmap(cm_logreg2, annot=True,fmt='d')

plt.ylabel('Prediction',fontsize=12) plt.xlabel('Actual',fontsize=12) plt.title('Confusion Matrix',fontsize=16) plt.show();

from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_test4)) `weighted`: Calculate precision and recalls metrics for each label, and find their average weighted by support (the number of true instances for each label).
This alters ‘macro’ to account for label imbalance; it can result in an F-score that is not between precision and recall. from sklearn.metrics import precision_score, recall_score, balanced_accuracy_score, f1_score

precision4 = precision_score(y_test, y_pred_test4, average='weighted') recall4 = recall_score(y_test, y_pred_test4, average = "weighted") f1_score4 = f1_score(y_test, y_pred_test4, average="weighted")

accuracy_score4 = accuracy_score(y_test, y_pred_test4)

print("The weighted precision is: ",precision4) print("The weighted recall is: ",recall4) print("The accuracy score is: ", accuracy_score4) print("The weighted f1score is: ", f1_score4) print(" ")

print("Weighted precision, f1 score, and recall are ideal measures because the target labels are imbalanced")

Model 5: Decision Tree- Model Iteration Two: Hyperparameter Tuning for the Decision Tree

dt2= DecisionTreeClassifier(criterion='entropy', splitter='best', max_depth= 14, random_state=42)
dt2.fit(X_train, y_train)

Compare the train-set and test-set accuracy

Now, I will compare the train-set and test-set accuracy to check for overfitting.

print('Accuracy score of train data :{}'.format(dt2.score(X_train,y_train))) print('Accuracy score of test data: {}'.format(dt2.score(X_test,y_test))) y_pred_test5 = dt2.predict(X_test)

y_pred_test5 Accuracy = 75.8% .
This has improved both accuracy scores and we are not overfitting either. This could be perfect for the final model Let's create a confusion matrix for this and obtain the other classification metrics

from sklearn.metrics import confusion_matrix

cm_logreg5 = confusion_matrix(y_test, y_pred_test5) print('Confusion matrix\n\n', cm_logreg5)

sns.heatmap(cm_logreg5, annot=True,fmt='d')

plt.ylabel('Prediction',fontsize=12) plt.xlabel('Actual',fontsize=12) plt.title('Confusion Matrix',fontsize=16) plt.show();

Remember that:
0: Functional:
1: functional needs repair
2: non-functional from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_test5)) `weighted`: Calculate precision and recalls metrics for each label, and find their average weighted by support (the number of true instances for each label).
This alters ‘macro’ to account for label imbalance; it can result in an F-score that is not between precision and recall. from sklearn.metrics import precision_score, recall_score, balanced_accuracy_score, f1_score

precision5 = precision_score(y_test, y_pred_test5, average='weighted') recall5 = recall_score(y_test, y_pred_test5, average = "weighted") f1_score5 = f1_score(y_test, y_pred_test5, average="weighted") accuracy_score5 = accuracy_score(y_test, y_pred_test5)

print("The weighted precision is: ",precision5) print("The weighted recall is: ",recall5) print("The accuracy score is: ", accuracy_score5) print("The weighted f1score is: ", f1_score5) print(" ")

print("Weighted precision, f1 score, and recall are ideal measures because the target labels are imbalanced")

Precision = 75%
Recall = 76%

Conclusion

1. The relationship between the following variables and the `status_group` (functional, non-functional, functional but needs repair):
Wells likely to be non-functional or needing repair:

- `payment` : Wells where no payments are made
- `source` : Wells with a shallow well as the water source
- `management_group` : Wells managed by the user group
- `extraction_type` : Wells with gravity as the extraction type
- `permit` : Wells that are permitted