In [1]:
```
# Import standard packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.set_theme(style="darkgrid")
from collections import Counter
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
import time
# from sklearn import metrics
from sklearn.metrics import precision_score, recall_score, accuracy_sco
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV, Ran
from sklearn.metrics import confusion_matrix, plot_confusion_matrix

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
```

Tanzania Machine Learning Water Pump Classification

Modeling Notebook

Author: Dylan Dey

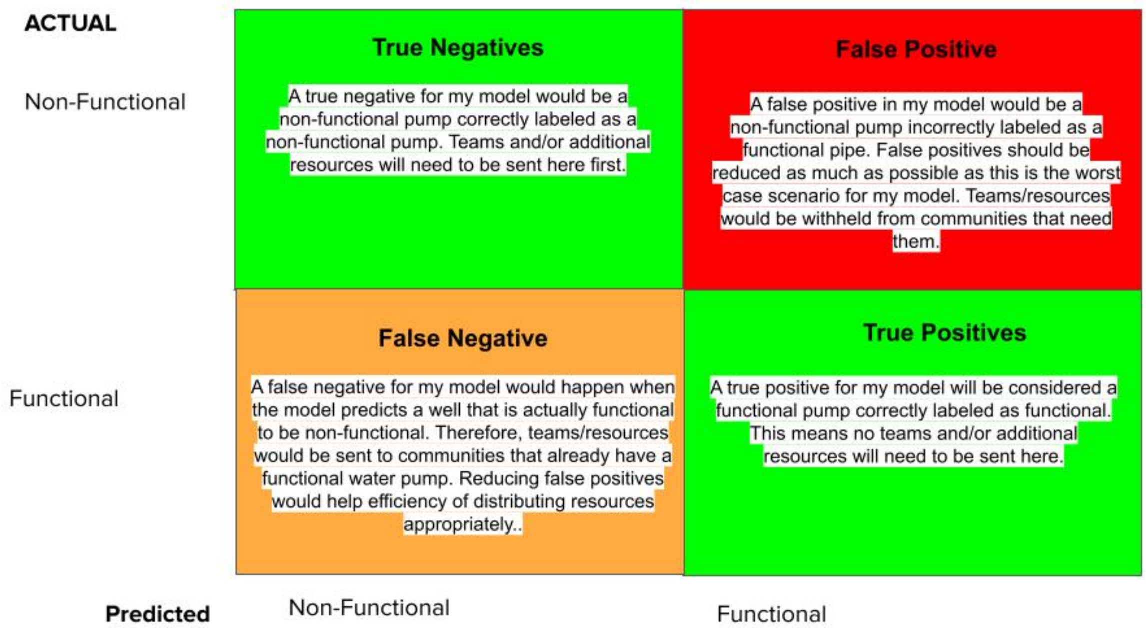
This project it available on github here:
https://github.com/ddey117/Tanzanian_Water_Pump_Classification
(https://github.com/ddey117/Tanzanian_Water_Pump_Classification)

The Author can be reached at the following email: ddey2985@gmail.com
(mailto:ddey2985@gmail.com)

Blog: *insert blog link*

Classification Metric Understanding

Below is a confusion matrix that would be produced from a model performing predictive maintenance on behalf of the Ministry of Water. There are four possible outcomes to be considered. The confusion matrix below is a visual aid to help in understanding what classification metrics to consider when building the model.



A true positive in the current context would be when the model correctly identifies a functional pump as functional. A true negative would be when the model correctly identifies a non-functional pump as non-functional. Both are important and both can be described by the overall **accuracy** of the model.

True negatives are really at the heart of the model, as this is the situation in which the Ministry of Water would have a call to action. An appropriately outfitted team would be set to *all* pumps that my model identifies as non-functional. Thus, this is the situation in which the correct resources are being derived to the correct water pumps as quickly as possible. High accuracy would mean that more resources are going to the correct locations from the get-go.

True positives are also important. This is where the model will really be saving time, resources, and money for the Ministry of Water. *Any* pumps identified as functional would no longer need to be physically checked and the Ministry of Water can withhold additional resources from going to pumps that do not actually need them.

Notice the emphasis on *any* and *all* pumps in my description of true negatives and true positives above. The true cost/resource analysis is really the consideration of this fact: no model I create will ever correctly identify every single pump appropriately. This is the cost of predictive maintenance and a proper understanding of false positives and false negatives is extremely important in production of classification models in the given context.

False positives in the current context are the worst case scenario for modeling. This is the scenario in which the model **incorrectly** identifies a non-functional model as functional. Thus, resources would be withheld and no team would be sent to physically check these pumps, as the Ministry of Water would have to assume they are indeed functional if they want to use the model appropriately. False positives therefore describe the number of non-functional pumps that will go unvisited and unfixed until they can be resolved by other means. Reducing false positives as much as possible is very important.

Well, why would I want to build the model if these false positives cannot be completely avoided? Cost/resource management, of course! Afterall, it is about making sure as many people get clean water as quickly as possible. The reality is there that resources are finite, and without the model the Ministry of Water likely would not have the resources to physically check all the pumps and then fix all of the pumps in any sort of reasonable timeline, and even less communities would have access to fresh water when compared to using the model for predictive maintenance.

False negatives are also important to consider. While false positives can be considered more harmful overall, false negatives are also important to reduce as much as possible. In the given context, false negatives describe the situation in which the model **incorrectly** identifies a functional pump as non-functional. Because the Ministry of water will deploy fully equipped teams to visit all pumps that my model predicts to be non-functional, these will be the pumps that will waste resources. Resources will be sent to locations that they aren't needed, and the metric that describes this would be false negatives. Thus, reduction of false negatives is essential in improving the efficiency of resource management through predictive maintenance.

In summary, overall accuracy of the model and a reduction of both false negatives and false positives are the most important metrics to consider when developing a model in this context. More specifically, models will be tuned to **maximize accuracy and f1-score.**

Accuracy: The number of correct predictions made by model divided by the total number of predictions. This measures how succesfull my model is at labeling functional pumps as functional and non functional pumps as non functional. It is a very important metric to consider.

f1-score: I stated above that both false negatives and false positives are important to avoid. This means that I really want precision and recall to go be high and therefore false negatives and false positives to be low. The f1-score is the harmonic mean of precision and recall. Maximizing for this metric is the best way to get a balance of both false positives and false negatives.

Function Definition

Below are all of the functions used for preprocessing data before modeling.

In [2]:

```
def drop_cols(water_pump_df):
    to_drop_final = ['id', 'recorded_by', 'num_private',
            'waterpoint_type_group', 'source',
            'source_class', 'extraction_type',
            'extraction_type_group', 'payment_type',
            'management_group', 'scheme_name',
            'water_quality', 'quantity_group',
            'scheme_management', 'longitude',
            'latitude', 'date_recorded',
            'amount_tsh', 'gps_height',
            'region_code', 'district_code']
            #'population'

    return water_pump_df.drop(columns=to_drop_final, axis=1)

#helper function to bin construction year
def construction_wrangler(row):
    if row['construction_year'] >= 1960 and row['construction_year'] <
        return '60s'
    elif row['construction_year'] >= 1970 and row['construction_year']
        return '70s'
    elif row['construction_year'] >= 1980 and row['construction_year']
        return '80s'
    elif row['construction_year'] >= 1990 and row['construction_year']
        return '90s'
    elif row['construction_year'] >= 2000 and row['construction_year']
        return '00s'
    elif row['construction_year'] >= 2010:
        return '10s'
    else:
        return 'unknown'

def bin_construction_year(water_pump_df):
    water_pump_df['construction_year'] = water_pump_df.apply(lambda ro
    return water_pump_df


#takes zero placeholders and NAN values and converts them into 'unknow
def fill_unknowns(water_pump_df):
    installer_index_0 = water_pump_df['installer'] =='0'
    funder_index_0 = water_pump_df['funder'] =='0'
    water_pump_df.loc[installer_index_0, 'installer'] = 'unknown'
    water_pump_df.loc[funder_index_0, 'funder'] = 'unknown'
    water_pump_df.fillna({'installer':'unknown',
                  'funder':'unknown',
                  'subvillage': 'unknown'}, inplace=True)
    return water_pump_df

#returns back boolean features without NANs while maintaining same rat
def fill_col_normal_data(water_pump_df):
    filt = water_pump_df['permit'].isna()
    probs = water_pump_df['permit'].value_counts(normalize=True)
    water_pump_df.loc[filt, 'permit'] = np.random.choice([True, False]
                        size=int(filt.sum()),
                        p = [probs[True], probs[False]])
    filt = water_pump_df['public_meeting'].isna()
```

```
    probs = water_pump_df['public_meeting'].value_counts(normalize=Tru
    water_pump_df.loc[filt, 'public_meeting'] = np.random.choice([True
                            size=int(filt.sum()),
                            p = [probs[True], probs[False]])
    return water_pump_df



def apply_cardinality_reduct(water_pump_df, reduct_dict):
    for col, categories_list in reduct_dict.items():
        water_pump_df[col] = water_pump_df[col].apply(lambda x: x if x
    return water_pump_df



#one_hot_incode categorical data
def one_hot(water_pump_df):
    final_cat = ['funder', 'installer', 'wpt_name', 'basin', 'subvilla
        'lga', 'ward', 'public_meeting', 'permit', 'construction_year',
        'extraction_type_class', 'management', 'payment', 'quality_grou
        'quantity', 'source_type', 'waterpoint_type']

    water_pump_df = pd.get_dummies(water_pump_df[final_cat], drop_firs

    return water_pump_df



#master function for cleaning dataFrame
def clean_dataFrame(water_pump_df, reduct_dict):
    water_pump_df = drop_cols(water_pump_df)
    water_pump_df = bin_construction_year(water_pump_df)
    water_pump_df = fill_unknowns(water_pump_df)
    water_pump_df = fill_col_normal_data(water_pump_df)
    water_pump_df = apply_cardinality_reduct(water_pump_df, reduct_dic
    water_pump_df = one_hot(water_pump_df)

    return water_pump_df

###############################################
# The rest of the functions in this section
#define functions that reduce cardinality
#by mapping infrequent values ot other
#the dictionary derived from these functions
#will be used by my_funk in my master
#clean_dataFrame function

#helper function for reducing cardinality
def cardinality_threshold(column,threshold=0.65):
    #calculate the threshold value using
    #the frequency of instances in column
    threshold_value=int(threshold*len(column))
    #initialize a new list for lower cardinality column
    categories_list=[]
    #initialize a variable to calculate sum of frequencies
    s=0
    #Create a dictionary (unique_category: frequency)
```

```
    counts=Counter(column)

    #Iterate through category names and corresponding frequencies afte
    #by descending order of frequency
    for i,j in counts.most_common():
        #Add the frequency to the total sum
        s += dict(counts)[i]
        #append the category name to the categories list
        categories_list.append(i)
        #Check if the global sum has reached the threshold value, if s
        if s >= threshold_value:
            break
        #append the new 'Other' category to list
        categories_list.append('Other')

    #Take all instances not in categories below threshold
    #that were kept and lump them into the
    #new 'Other' category.
    new_column = column.apply(lambda x: x if x in categories_list else
#     return new_column
    return categories_list

 #reduces the cardinality of appropriate categories
def get_col_val_mapping(water_pump_df):
    col_threshold_list = [
        ('funder',0.65),
        ('installer', 0.65),
        ('wpt_name', 0.15),
        ('subvillage', 0.07),
        ('lga', 0.6),
        ('ward', 0.05)
    ]

    reduct_dict = {}

    for col, thresh in col_threshold_list:
        reduct_dict[col] = cardinality_threshold(water_pump_df[col],
                                                   threshold= thresh)

    return reduct_dict

# reduct_dict is a key value mapper that will
# be used for both training and testing sets
# in order to reduce cardinality of the data
```

Import The Data From Multiple Sources

I used a number of sources for my data to use for modeling in this notebook. The cell below imports the original data from the DrivenData competition, data derived from DrivenData in QGIS and opensource hydrology data, and population data from Tanzania government census in 2012.

In [3]:
```
#import data from DrivenData
train_labels = pd.read_csv('data/0bf8bc6e-30d0-4c50-956a-603fc693d966.c
train_features = pd.read_csv('data/4910797b-ee55-40a7-8668-10efd5c1b960
df = train_features.merge(train_labels, on='id').copy()
#import QGIS derived data and prepare for model
river_df = pd.read_csv('data/river_dist2.csv')
#removing outliers
index_riv = river_df[river_df['HubDist'] >66].index
river_median = river_df['HubDist'].median()
river_df.loc[index_riv, 'HubDist'] = river_median

#create boolean for pump being within 8 km of river
river_s = river_df['HubDist'].copy()
river_s.rename('near_river', inplace=True)
near_river = river_s[river_s < 8].apply(lambda x: 1 if not pd.isnull(x)
df = df.join(near_river)
df.near_river.fillna(0, inplace=True)

#import population data from 2012 government census
df_pop = pd.read_excel('data/tza-pop-popn-nbs-baselinedata-xlsx-1.xlsx'

#create a dictionary of values with format {Ward : Total Population}

pop_index = df_pop.groupby('Ward_Name')['total_both'].sum().index
pop_values = df_pop.groupby('Ward_Name')['total_both'].sum().values
pop_dict = dict(zip(pop_index, pop_values))

#create pandas Dataframe for merging
pop_dataframe = pd.DataFrame.from_dict(pop_dict, orient='index')
#rename column for clarity
pop_dataframe.rename(columns={0: 'ward_pop'}, inplace=True)

#merge dataframes
df_pop_merge = df.merge(pop_dataframe,
                              how='left',
                              left_on='ward',
                              right_index=True)

#replace null values of ward population with
#median ward population

ward_pop_median = df_pop_merge['ward_pop'].median()
df_pop_merge.fillna(value=ward_pop_median, inplace=True)
# merge back into df and drop pop column
ward_pop_s = df_pop_merge['ward_pop'].copy()
df = df.join(ward_pop_s)
df.drop(columns=['population'], axis=1, inplace=True)

#bin functional needs repair and non function together
#simplify problem into binary classifcation
need_repair_index = df['status_group'] == 'functional needs repair'
df_binary = df.copy()
df_binary.loc[need_repair_index, 'status_group'] = 'non functional'

#workaround for Label Encoding mapping
#Label Encoding doesn't have a mapper method
```

```
#but it seems to encode alphabetically
#I want Non functional to be zero but encoded as 1
#unless make it come before 'functional' alphabetically
funct_index = df_binary['status_group'] == 'functional'
nonfunct_index = df_binary['status_group'] == 'non functional'
df_binary.loc[funct_index, 'status_group'] = 'Zfunctional'
df_binary.loc[nonfunct_index, 'status_group'] = 'Anon functional'
```

Transform Target to Numerical Data with Label Encoding

As shown at the beginning of the project, the functional state of the water supply point is described as: functional — It is working; non functional — it is not working; functional needs repair — Running, but needing maintenance.

These groups will be relabeled as: functional: 0 functional needs repair: 1 non functional: 2

This would be for future work in which I didn't bin funcitonal needs repair with non functional. I chose this simplified way to deal with the imbalanced dataset not only due to time-constraints but because it is extremely difficult to get meanigful predictive power for the 'functional needs repair' label due to the nature of the dataset.

For the modeling in this notebook the target data will be relabeled as the following after binning function needs repar with non-functional:

non_functional: 0

functional: 1

In my preprocessing steps I had to use a work around for LabelEncoder to play nice. There is no value mapper method and it labels strings alphabetically. Therefore, I temporarily changed the status group labels to get encoded in the order I want them to be encoded.

In [4]:
```
df_binary['status_group'].value_counts(normalize=True)
```

Out[4]:
```
Zfunctional        0.543081
Anon functional    0.456919
Name: status_group, dtype: float64
```

In [5]:
```
# le = LabelEncoder()

# le.fit(['functional', 'functional needs repair', 'non functional'])

# df['status_group'] = le.transform(df.status_group)

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

In [6]:
```
le = LabelEncoder()

le.fit(['Anon functional','Zfunctional'])

df_binary['status_group'] = le.transform(df_binary.status_group)

df_binary['status_group'].value_counts(normalize=True)
```

Out[6]:
```
1    0.543081
0    0.456919
Name: status_group, dtype: float64
```

Split Data into Test and Training Sets

In [7]:
```
from sklearn.model_selection import train_test_split

y = df_binary['status_group']
X = df_binary.drop(columns=['status_group'], axis=1)

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3,
```

Run Master Cleaning Function

In [8]:
```
value_maps = get_col_val_mapping(df_binary)
X_train = clean_dataFrame(X_train, value_maps)
X_test = clean_dataFrame(X_test, value_maps)
display(X_train.shape)
X_test.shape
```

```
(41580, 239)
```

Out[8]:
```
(17820, 239)
```

Data Modeling

Baseline model

I decided to use logistic regression as my baseline model. I chose to use an arbitrary large value for C and set the solver to 'liblinear.' I did not choose to fit an intercept for the baseline model.

In [9]:

```
# Instantiate the model
logreg = LogisticRegression(fit_intercept=False, C=1e12, solver='liblin

# Fit the model
logreg.fit(X_train, y_train)
y_hat_train = logreg.predict(X_train)
y_hat_test = logreg.predict(X_test)

print('Training Precision: ', precision_score(y_train, y_hat_train))
print('Testing Precision: ', precision_score(y_test, y_hat_test))
print('\n\n')

print('Training Recall: ', recall_score(y_train, y_hat_train))
print('Testing Recall: ', recall_score(y_test, y_hat_test))
print('\n\n')

print('Training Accuracy: ', accuracy_score(y_train, y_hat_train))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test))
print('\n\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test))
```

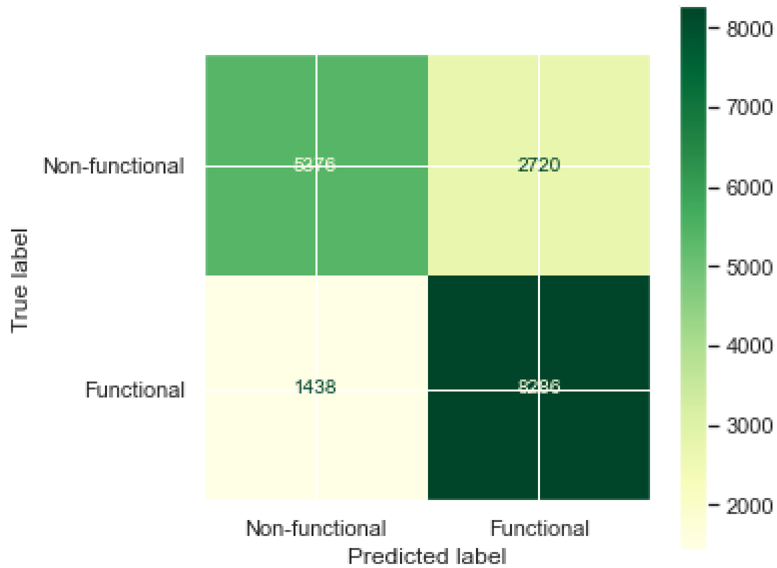
```
Training Precision:  0.7527743044701469
Testing Precision:  0.7528620752316918



Training Recall:  0.8548923896161527
Testing Recall:  0.8521184697655286



Training Accuracy:  0.7691919191919192
Testing Accuracy:  0.7666666666666667



Training F1-Score:  0.8005901053462713
Testing F1-Score:  0.7994211287988422
```

In [10]:

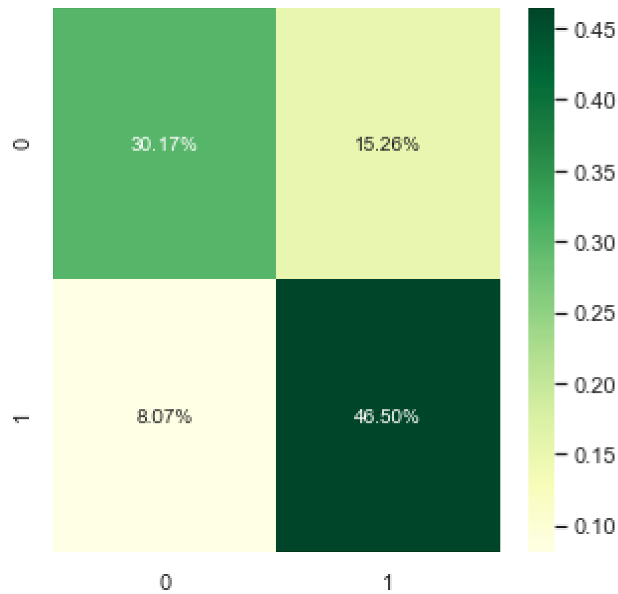
```
#Get the confusion matrix
lr_base_matrix = confusion_matrix(y_test, y_hat_test)
print(lr_base_matrix)
```

```
[[5376 2720]
 [1438 8286]]
```

In [11]:
```
fig_labels = ['Non-functional', 'Functional']
fig, ax = plt.subplots(figsize=(5, 5))

plot_confusion_matrix(logreg,
                      X_test,
                      y_test,
                      ax=ax,
                      cmap='YlGn',
                      display_labels=fig_labels)

plt.show();
```



In [14]:
```
#confusion matrix percentages
fig, ax = plt.subplots(figsize=(5, 5))


sns.heatmap(lr_base_matrix/np.sum(lr_base_matrix), annot=True,
            fmt='.2%', cmap='YlGn', ax=ax)

plt.show();
```



The baseline logisitc regression model had fairly decent scores all around but I believe they can be improved.

Below a random grid search was used to create 300 of 600 possible combinations to narrow down the paramters to do a full brute force gridCV search. The score used for determining the 'best" model was precision.

This is clearly not the way to build the model. Although I am greatly concerned with increasing my precision in order to reduce false positives, false negatives still have a fairly high cost as well.

The precision of the model is excellent. However, the accuracy and F1 scores are unacceptable.

I will describe why this is a poor model in more detail below under the display of the confusion matrix.

In [15]:

```
start = time.time()

classifier_penalties = ['l1', 'l2']
classifier_Cs = np.logspace(-5, 5, 100)
classifier__solver = ['liblinear']

random_grid = {'penalty' : classifier_penalties,
'C' : classifier_Cs,
'solver' : classifier__solver}

lr = LogisticRegression()
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
lr_random_precision = RandomizedSearchCV(scoring='precision',
                              estimator = lr,
                              param_distributions = random_grid,
                              n_iter = 100, cv = 3, verbose=2,
                              random_state=42,
                              n_jobs = -1)
# Fit the random search model
lr_random_precision.fit(X_train, y_train)



end = time.time()
print(end - start)

lr_best_rand_precision = lr_random_precision.best_estimator_
lr_best_rand_precision.fit(X_train,y_train)
y_hat_train_rp = lr_best_rand_precision.predict(X_train)
y_hat_test_rp = lr_best_rand_precision.predict(X_test)
```

```
Fitting 3 folds for each of 100 candidates, totalling 300 fits
245.77547788619995
```

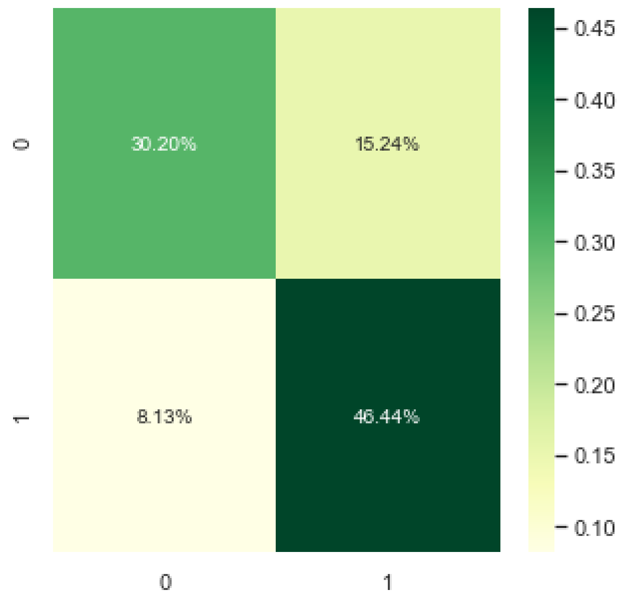
In [16]:
```
print('Training Precision: ', precision_score(y_train, y_hat_train_rp))
print('Testing Precision: ', precision_score(y_test, y_hat_test_rp))
print('\n\n')


print('Training Accuracy: ', accuracy_score(y_train, y_hat_train_rp))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test_rp))
print('\n\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train_rp))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test_rp))
```

```
Training Precision:  0.7525507212384192
Testing Precision:  0.7529572338489536



Training Accuracy:  0.7687830687830688
Testing Accuracy:  0.7663299663299663



Training F1-Score:  0.8001912045889101
Testing F1-Score:  0.7989765376074153
```

In [17]:
```
#Get the confusion matrix
lr_rand_prec_matrix = confusion_matrix(y_test, y_hat_test_rp)
print(lr_rand_prec_matrix)
```

```
[[5381 2715]
 [1449 8275]]
```

In [18]:
```
# Visualize your confusion matrix
fig, ax = plt.subplots(figsize=(5,5))


sns.heatmap(lr_rand_prec_matrix/np.sum(lr_rand_prec_matrix), annot=True
            fmt='.2%', cmap='YlGn', ax=ax)

plt.show();
```



The model did not perform much different than before the random search. Perhaps scoring on precision is not the way to go, as we also want to lower false negatives.

I will try scoring on f1_score as my target for iterative modeling as well as precision and compare.

Below I decided to run a full grid search with the same paramaters as my random grid, only this time with focus on improving f1_score.

In [19]:

```
start = time.time()

classifier_penalties = ['l1', 'l2']
classifier_Cs = np.logspace(-5, 5, 100)
classifier__solver = ['liblinear']

lr_grid = {'penalty' : classifier_penalties,
           'C' : classifier_Cs,
           'solver' : classifier__solver
          }

lr = LogisticRegression()
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
lr_grid_f1 = GridSearchCV(estimator=lr,
                          param_grid=lr_grid,
                          scoring='f1',
                          cv=3,
                          n_jobs = -1,
                          verbose=2
                         )

# RandomizedSearchCV(estimator = lr,
#                                param_distributions = random_grid,
#                                n_iter = 100, cv = 3, verbose=2,
#                                random_state=42,
#                                n_jobs = -1)
# Fit the random search model
lr_grid_f1.fit(X_train, y_train)


end = time.time()
print(end - start)


# lr_best_grid_f1 = lr_grid_f1.best_estimator_
# lr_best_grid_f1.fit(X_train,y_train)
# y_hat_train_gs = lr_best_grid_f1.predict(X_train)
# y_hat_test_gs = lr_best_grid_f1.predict(X_test)
```

```
Fitting 3 folds for each of 200 candidates, totalling 600 fits
447.88261699676514
```

In [20]:

```
lr_grid_f1.best_estimator_
```

Out[20]:

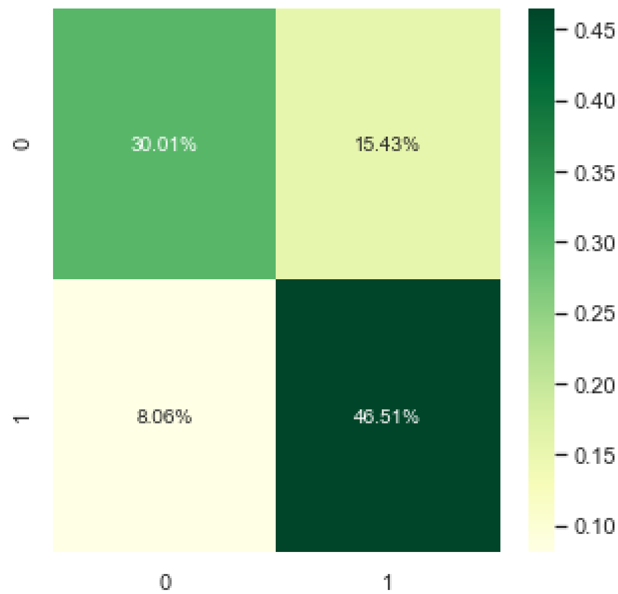
```
LogisticRegression(C=0.44306214575838776, penalty='l1', solver='liblinea
r')
```

In [21]:
```
lr_best_grid_f1 = lr_grid_f1.best_estimator_
lr_best_grid_f1.fit(X_train,y_train)
y_hat_train_gs = lr_best_grid_f1.predict(X_train)
y_hat_test_gs = lr_best_grid_f1.predict(X_test)


print('Training Precision: ', precision_score(y_train, y_hat_train_gs))
print('Testing Precision: ', precision_score(y_test, y_hat_test_gs))
print('\n\n')


print('Training Accuracy: ', accuracy_score(y_train, y_hat_train_gs))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test_gs))
print('\n\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train_gs))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test_gs))
```

```
Training Precision:  0.7507864383082838
Testing Precision:  0.7509286943915919



Training Accuracy:  0.7686387686387687
Testing Accuracy:  0.7651515151515151



Training F1-Score:  0.8007621572363517
Testing F1-Score:  0.7984201146380233
```

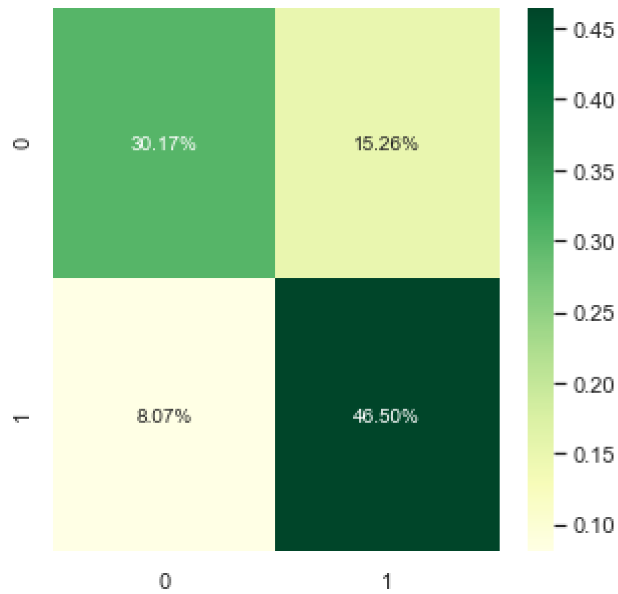
In [22]:
```
#best grid search confusion matrix
lr_grid_f1_matrix = confusion_matrix(y_test, y_hat_test_gs)
print(lr_grid_f1_matrix)

# Visualize your confusion matrix
fig, ax = plt.subplots(figsize=(5,5))


sns.heatmap(lr_grid_f1_matrix/np.sum(lr_grid_f1_matrix), annot=True,
            fmt='.2%', cmap='YlGn', ax=ax)

plt.show();
```

```
[[5347 2749]
 [1436 8288]]
```



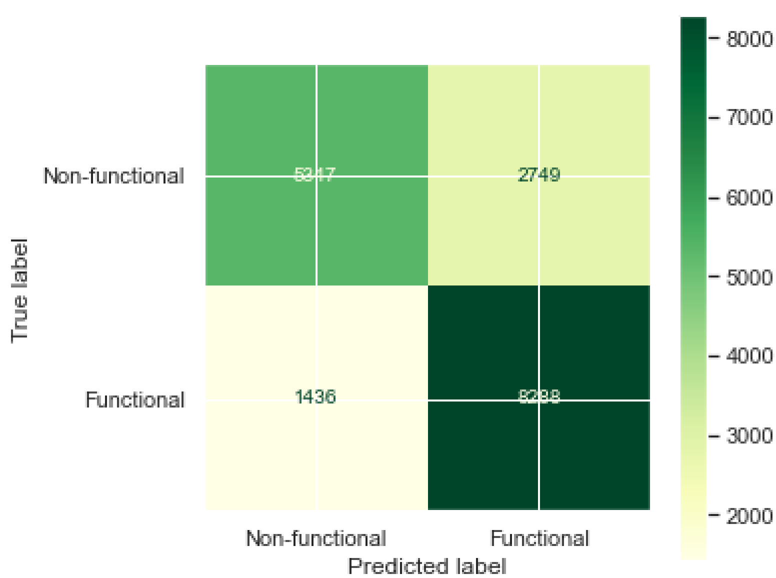
In [23]:

```
#baseline confusion matrix
fig, ax = plt.subplots(figsize=(5,5))


sns.heatmap(lr_base_matrix/np.sum(lr_base_matrix), annot=True,
            fmt='.2%', cmap='YlGn', ax=ax)

plt.show();
```



In [24]:

```
fig, ax = plt.subplots(figsize=(5,5))

plot_confusion_matrix(lr_best_grid_f1,
                      X_test,
                      y_test,
                      ax=ax,
                      cmap='YlGn',
                     display_labels=fig_labels)

plt.show();
```



The gridsearch did not do much to improve the model.

Random Forest Classifier

Below I have created a grid of paramters in order to tune a random forest classifier. This grid will be used to perform a random grid search in order to narrow down the paramters to run a more robust grid search. The run time for a Random Forest has potential to be much higher than a logistic regression model. Therefore, a random search to narrow down the paramters for a grid search was carried out in order to reduce total run time while still keeping a fairly broad search.

In [25]:

```
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, n
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf}
random_grid
```

Out[25]:

```
{'n_estimators': [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 200
0],
 'max_features': ['auto', 'sqrt'],
 'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, None],
 'min_samples_split': [2, 5, 10],
 'min_samples_leaf': [1, 2, 4]}
```

In [26]:

```
start = time.time()

# Use the random grid to search for best hyperparameters
# First create the base model to tune
rfc = RandomForestClassifier()
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
rfc_random = RandomizedSearchCV(scoring='f1',
                                estimator = rfc,
                                param_distributions = random_grid,
                                n_iter = 33,
                                cv = 3,
                                verbose=2,
                                random_state=42,
                                n_jobs = -1)
# Fit the random search model

rfc_random.fit(X_train,y_train)
rfc_best_random = rfc_random.best_estimator_
rfc_best_random.fit(X_train,y_train)
y_hat_train_rrf = rfc_best_random.predict(X_train)
y_hat_test_rrf = rfc_best_random.predict(X_test)


end = time.time()
print(end - start)

print('Training Precision: ', precision_score(y_train, y_hat_train_rrf)
print('Testing Precision: ', precision_score(y_test, y_hat_test_rrf))
print('\n\n')


print('Training Accuracy: ', accuracy_score(y_train, y_hat_train_rrf))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test_rrf))
print('\n\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train_rrf))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test_rrf))
```

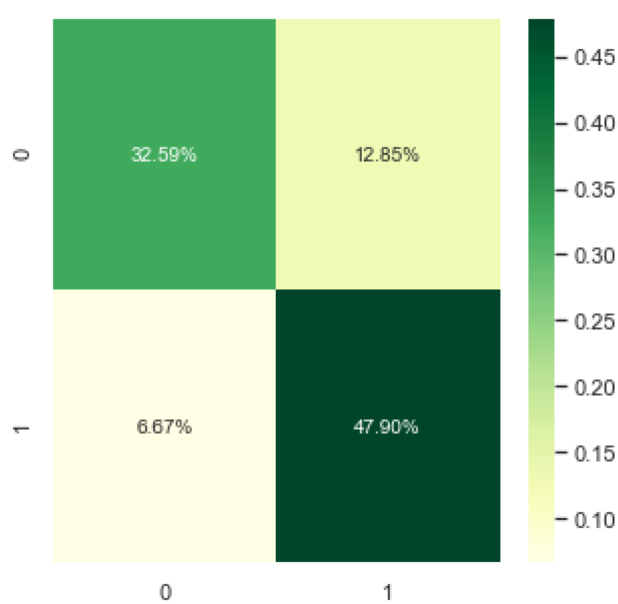
```
Fitting 3 folds for each of 33 candidates, totalling 99 fits
1702.4333066940308
Training Precision:  0.8238002619775334
Testing Precision:  0.7885450346420323



Training Accuracy:  0.8504088504088504
Testing Accuracy:  0.8048821548821549



Training F1-Score:  0.8696781763325511
Testing F1-Score:  0.8307946858727917
```

In [27]:
```
rfc_random.best_estimator_.get_params()
```

Out[27]:
```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': 40,
 'max_features': 'sqrt',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 2,
 'min_samples_split': 5,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 1000,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

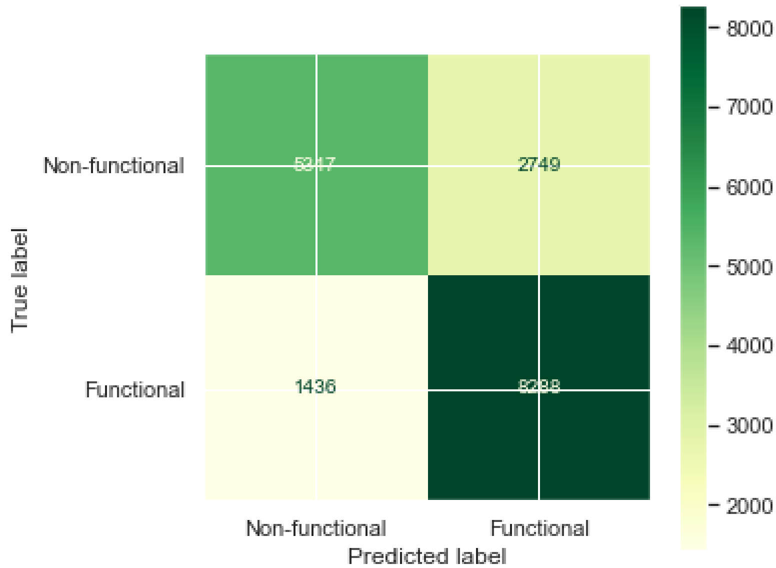
In [28]:
```
#best random search confusion matrix for Random Forest
lr_grid_f1_matrix_rrf = confusion_matrix(y_test, y_hat_test_rrf)
print(lr_grid_f1_matrix_rrf)

# Visualize your confusion matrix
fig, ax = plt.subplots(figsize=(5,5))


sns.heatmap(lr_grid_f1_matrix_rrf/np.sum(lr_grid_f1_matrix_rrf), annot=
            fmt='.2%', cmap='YlGn', ax=ax)

plt.show();
```

```
[[5807 2289]
 [1188 8536]]
```



In [29]:

```
fig, ax = plt.subplots(figsize=(5,5))

plot_confusion_matrix(lr_best_grid_f1,
                      X_test,
                      y_test,
                      ax=ax,
                      cmap='YlGn',
                     display_labels=fig_labels)

plt.show();
```



This random forest classifier outperforms the best logistic regression model in every relevant metric. It improves upon the model accuracy and precision while also lowering the occurance false negatives.

A grid search will be performed below to see if any improvements can be made to model performance.

In [30]:

```
# Number of trees in random forest
n_estimators = [500, 600, 700]
# Number of features to consider at every split
max_features = [14,15,16]
# Maximum number of levels in tree
# Minimum number of samples required to split a node
min_samples_split = [8,10,12]
# Minimum number of samples required at each leaf node
# Create the random grid
cv_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'min_samples_split': min_samples_split}

cv_grid
```

Out[30]:

```
{'n_estimators': [500, 600, 700],
 'max_features': [14, 15, 16],
 'min_samples_split': [8, 10, 12]}
```

In [31]:
```
start = time.time()

rfc = RandomForestClassifier()
# Instantiate the grid search model
rfc_gs = GridSearchCV(scoring='f1',
                      estimator = rfc,
                      param_grid = cv_grid,
                      cv = 3,
                      n_jobs = -1,
                      verbose = 2)


rfc_gs.fit(X_train,y_train)
rfc_best_gs = rfc_gs.best_estimator_
rfc_best_gs.fit(X_train,y_train)
y_hat_train_grf = rfc_best_gs.predict(X_train)
y_hat_test_grf = rfc_best_gs.predict(X_test)


end = time.time()
print(end - start)

print('Training Precision: ', precision_score(y_train, y_hat_train_grf)
print('Testing Precision: ', precision_score(y_test, y_hat_test_grf))
print('\n\n')


print('Training Accuracy: ', accuracy_score(y_train, y_hat_train_grf))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test_grf))
print('\n\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train_grf))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test_grf))
```

```
Fitting 3 folds for each of 27 candidates, totalling 81 fits
958.5107200145721
Training Precision:  0.8589955394019494
Testing Precision:  0.7988826815642458



Training Accuracy:  0.8761183261183261
Testing Accuracy:  0.8025813692480359



Training F1-Score:  0.889811110873425
Testing F1-Score:  0.8250273550184024
```

Random forest models using random and grid search CV with scoring set to 'f1'

Testing random search Precision: 0.7885450346420323

Testing random search Accuracy: 0.8048821548821549

Testing random search F1-Score: 0.8307946858727917

[5807 2289]
[1188 8536]

vs

Testing grid search Precision: 0.7988826815642458

Testing grid search Accuracy: 0.8025813692480359

Testing grid search F1-Score: 0.8250273550184024

[6008 2088]
[1430 8294]

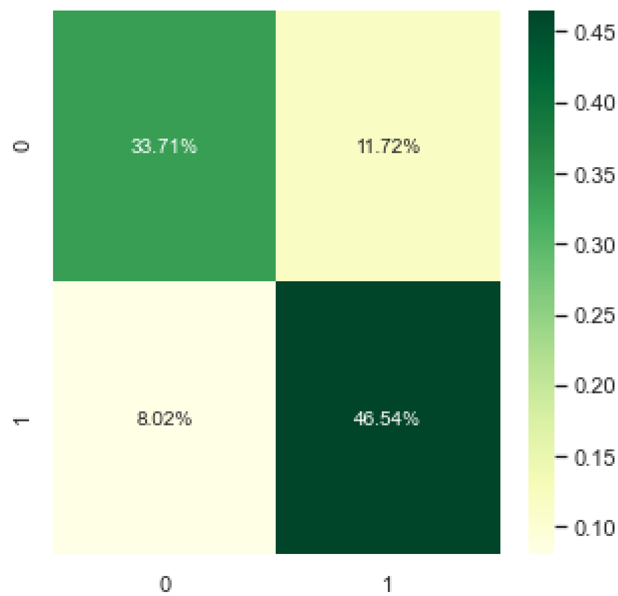
The grid-search-tuned random forest classifier correctly labeled about 200 more non-functional pumps and had about 200 less false positives than the random-search-tuned random forest classifier.

I believe the higher precision of the grid search random forest classifier is more important than the slight hit to the f1-score and accuracy.

In [66]:

```
#best grid search confusion matrix for Random Forest
lr_grid_f1_matrix_grf = confusion_matrix(y_test, y_hat_test_grf)
print(lr_grid_f1_matrix_grf)

# Visualize your confusion matrix
fig, ax = plt.subplots(figsize=(5, 5))


sns.heatmap(lr_grid_f1_matrix_grf/np.sum(lr_grid_f1_matrix_grf),
            annot=True,
            fmt='.2%',
            cmap='YlGn',
            ax=ax)

plt.show();

#current best model
```

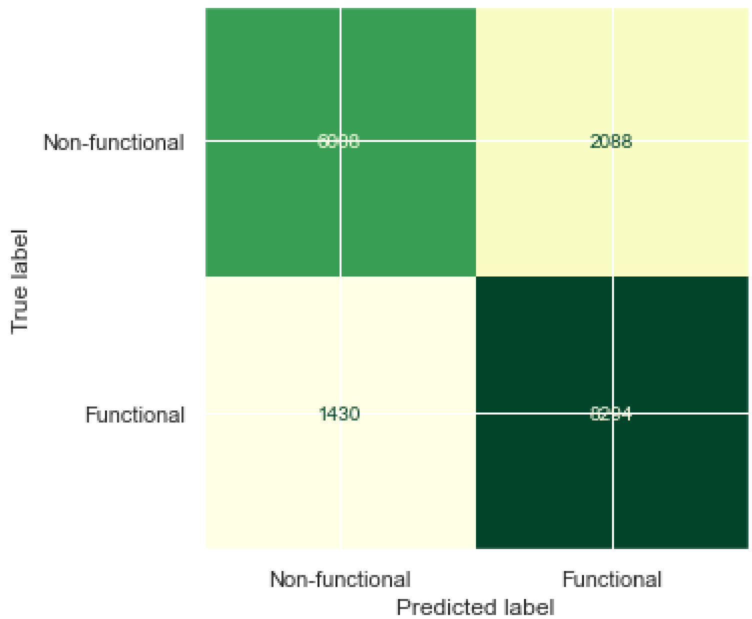
```
[[6008 2088]
 [1430 8294]]
```



The results of the grid search reduced the number of false positives, which is great, but it slightly increased the number of false negatives. It identified a similar number of functional pumps as the other random forest model, but it also identified slightly more non-functional pumps correctly. Performance was slightly improved overall.

In [68]:

```
#current best model

fig, ax = plt.subplots(figsize=(5, 5))

plot_confusion_matrix(rfc_best_gs,
                      X_test,
                      y_test,
                      ax=ax,
                      cmap='YlGn',
                      colorbar=False,
                     display_labels=fig_labels)

plt.show();
```



In [57]:
```
start = time.time()

# Use the random grid to search for best hyperparameters
# First create the base model to tune
rfc = RandomForestClassifier()
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
rfc_random2 = RandomizedSearchCV(scoring='precision',
                                 estimator = rfc,
                                 param_distributions = random_grid,
                                 n_iter = 33,
                                 cv = 3,
                                 verbose=2,
                                 random_state=42,
                                 n_jobs = -1)
# Fit the random search model

rfc_random2.fit(X_train,y_train)
rfc_best_random2 = rfc_random2.best_estimator_
rfc_best_random2.fit(X_train,y_train)
y_hat_train_rrf2 = rfc_best_random2.predict(X_train)
y_hat_test_rrf2 = rfc_best_random2.predict(X_test)


end = time.time()
print(end - start)

print('Training Precision: ', precision_score(y_train, y_hat_train_rrf2
print('Testing Precision: ', precision_score(y_test, y_hat_test_rrf2))
print('\n\n')


print('Training Accuracy: ', accuracy_score(y_train, y_hat_train_rrf2))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test_rrf2))
print('\n\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train_rrf2))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test_rrf2))
```

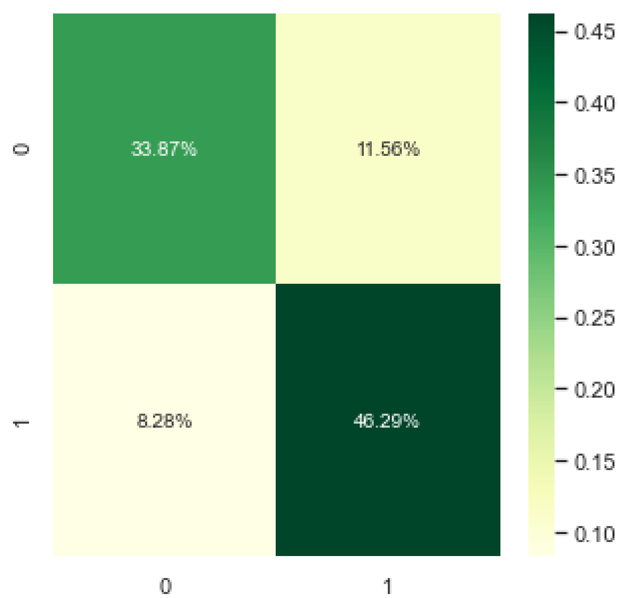
```
Fitting 3 folds for each of 33 candidates, totalling 99 fits
1520.5148952007294
Training Precision:  0.878849845588849
Testing Precision:  0.8001746047143273




Training Accuracy:  0.8946849446849446
Testing Accuracy:  0.8016273849607183




Training F1-Score:  0.9058218809815688
Testing F1-Score:  0.8235411570908002
```

In [54]:
```
rfc_best_gs2
```

Out[54]:
```
RandomForestClassifier(max_features=16, min_samples_split=8, n_estimators
=500)
```

In [70]:
```
#best grid search confusion matrix for Random Forest
lr_grid_p_matrix_grf = confusion_matrix(y_test, y_hat_test_rrf2)
print(lr_grid_p_matrix_grf)

# Visualize your confusion matrix
fig, ax = plt.subplots(figsize=(5, 5))


sns.heatmap(lr_grid_p_matrix_grf/np.sum(lr_grid_p_matrix_grf),
            annot=True,
            fmt='.2%',
            cmap='YlGn',
            ax=ax)

plt.show();
```

```
[[6036 2060]
 [1475 8249]]
```



current model (random search presicion scoring)

Testing Precision: 0.8001746047143273

Testing Accuracy: 0.8016273849607183

Testing F1-Score: 0.8235411570908002

[6036, 2060]
[1475, 8249]

vs

best model so far (grid search f1 scoring)

Testing grid search Precision: 0.7988826815642458

Testing grid search Accuracy: 0.8025813692480359

Testing grid search F1-Score: 0.8250273550184024

[6008, 2088]
[1430, 8294]

Both models perform very simlarly. The current model has slightly less false positives than the best model so far.

In [71]:
```
# Number of trees in random forest
n_estimators = [500, 600, 700]
# Number of features to consider at every split
max_features = [16,17,18]
# Maximum number of levels in tree
# Minimum number of samples required to split a node
min_samples_split = [8,10,12]
# Minimum number of samples required at each leaf node
# Create the random grid
cv_grid2 = {'n_estimators': n_estimators,
               'max_features': max_features,
               'min_samples_split': min_samples_split}

cv_grid2
```

Out[71]:
```
{'n_estimators': [500, 600, 700],
 'max_features': [16, 17, 18],
 'min_samples_split': [8, 10, 12]}
```

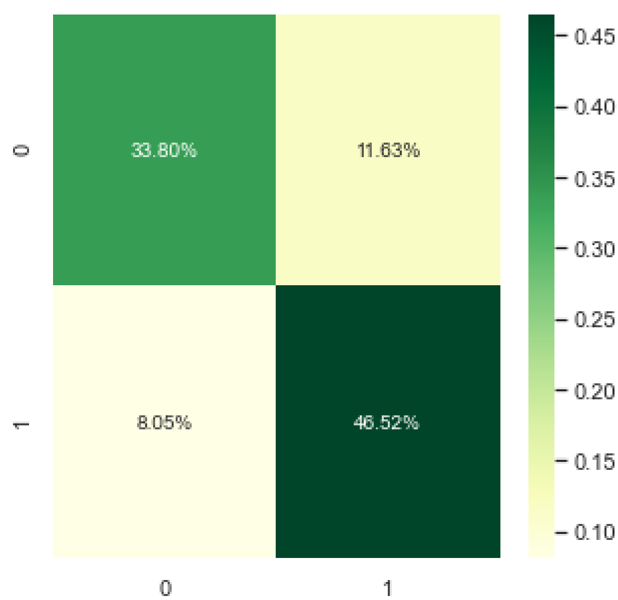
In [72]:

```
start = time.time()

rfc = RandomForestClassifier()
# Instantiate the grid search model
rfc_gs2 = GridSearchCV(scoring='precision',
                      estimator = rfc,
                      param_grid = cv_grid2,
                      cv = 3,
                      n_jobs = -1,
                      verbose = 2)


rfc_gs2.fit(X_train,y_train)
rfc_best_gs2 = rfc_gs2.best_estimator_
rfc_best_gs2.fit(X_train,y_train)
y_hat_train_grf2 = rfc_best_gs2.predict(X_train)
y_hat_test_grf2 = rfc_best_gs2.predict(X_test)


end = time.time()
print(end - start)

print('Training Precision: ', precision_score(y_train, y_hat_train_grf2
print('Testing Precision: ', precision_score(y_test, y_hat_test_grf2))
print('\n\n')


print('Training Accuracy: ', accuracy_score(y_train, y_hat_train_grf2))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test_grf2))
print('\n\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train_grf2))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test_grf2))
```

```
Fitting 3 folds for each of 27 candidates, totalling 81 fits
1050.9529011249542
Training Precision:  0.8690639127369251
Testing Precision:  0.8000386025863733



Training Accuracy:  0.8860509860509861
Testing Accuracy:  0.8032547699214366



Training F1-Score:  0.8984264459975132
Testing F1-Score:  0.825450562580902
```

In [74]:
```
#best grid search confusion matrix for Random Forest
lr_grid_p_matrix_grf2 = confusion_matrix(y_test, y_hat_test_grf2)
print(lr_grid_p_matrix_grf2)

# Visualize your confusion matrix
fig, ax = plt.subplots(figsize=(5, 5))


sns.heatmap(lr_grid_p_matrix_grf2/np.sum(lr_grid_p_matrix_grf2),
            annot=True,
            fmt='.2%',
            cmap='YlGn',
            ax=ax)

plt.show();
```

```
[[6024 2072]
 [1434 8290]]
```



I believe the random forest classifier using the hyperparamters determined through the grid search using precision as the tuning parameter produced the best model as a balance between accuracy and low false positives and false negatives.

grid search precision scoring (best model)

Testing Precision: 0.8000386025863733

Testing Accuracy: 0.8032547699214366

Testing F1-Score: 0.825450562580902 [6024, 2072]
[1434, 8290]

random search presicion scoring

Testing Precision: 0.8001746047143273

Testing Accuracy: 0.8016273849607183

Testing F1-Score: 0.8235411570908002

[6036, 2060]
[1475, 8249]

vs

grid search f1 scoring

Testing grid search Precision: 0.7988826815642458

Testing grid search Accuracy: 0.8025813692480359

Testing grid search F1-Score: 0.8250273550184024

[6008, 2088]
[1430, 8294]

XGBoost Classification Iterative Modeling

A similar method of iterative modeling was carried out using XGBoost as was used for tuning the random forest classifier: a random grid cv will be used to reduce the number of paramters to perform in a more robust grid search. A baseline XGBoost classifier was first created using default parameters before iterative modeling/hyper-paramter tuning was carried out.

In [35]:

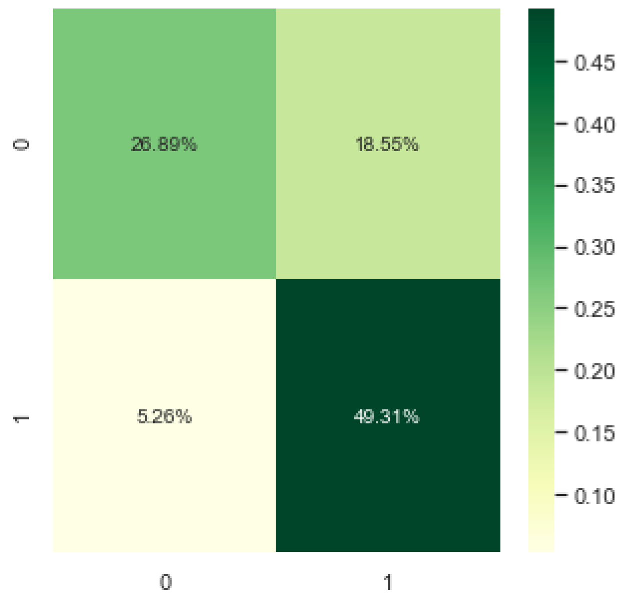
```
# Instantiate the model
xgc_base = XGBClassifier()

# Fit the model
xgc_base.fit(X_train, y_train)
y_hat_train = xgc_base.predict(X_train)
y_hat_test = xgc_base.predict(X_test)

print('Training Precision: ', precision_score(y_train, y_hat_train))
print('Testing Precision: ', precision_score(y_test, y_hat_test))
print('\n\n')

print('Training Recall: ', recall_score(y_train, y_hat_train))
print('Testing Recall: ', recall_score(y_test, y_hat_test))
print('\n\n')

print('Training Accuracy: ', accuracy_score(y_train, y_hat_train))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test))
print('\n\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test))
```

```
Training Precision:  0.7247322505142209
Testing Precision:  0.7266787958981145



Training Recall:  0.9068560017750167
Testing Recall:  0.9036404771698889



Training Accuracy:  0.7628427128427129
Testing Accuracy:  0.7619528619528619



Training F1-Score:  0.8056294719413399
Testing F1-Score:  0.8055555555555556
```

In [36]:

```
#baseline XGSBoost model
xgs_base_matrix = confusion_matrix(y_test, y_hat_test)
print(xgs_base_matrix)

# Visualize your confusion matrix
fig, ax = plt.subplots(figsize=(5, 5))


sns.heatmap(xgs_base_matrix/np.sum(xgs_base_matrix), annot=True,
            fmt='.2%', cmap='YlGn', ax=ax)

plt.show();
```

```
[[4791 3305]
 [ 937 8787]]
```



The baseline XGSBoost model has a low number of false negatives, but a relatively high number of false positives. It also has a lower accuracy than either random forest model. Some hyper-parameter tuning will be carried out below in an attempt to improve the model.

In [37]:
```
params = {
         'min_child_weight': [1, 5, 10],
         'gamma': [0.5, 1, 1.5, 2, 5],
         'subsample': [0.6, 0.8, 1.0],
         'colsample_bytree': [0.6, 0.8, 1.0],
         'max_depth': [3, 4, 5]
         }

xgc = XGBClassifier()
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
xgc_random = RandomizedSearchCV(scoring='f1',
                                estimator = xgc,
                                param_distributions = params,
                                n_iter = 33, cv = 3, verbose=2,
                                random_state=42,
                                n_jobs = -1)
# Fit the random search model
xgc_random.fit(X_train, y_train)


end = time.time()
print(end - start)


# xgc_best_random = xgc_random.best_estimator_
# xgc_best_random.fit(X_train,y_train)
# y_hat_train_brm = xgc_best_random.predict(X_train)
# y_hat_test_brm = xgc_best_random.predict(X_test)

# print('Training Precision: ', precision_score(y_train, y_hat_train_br
# print('Testing Precision: ', precision_score(y_test, y_hat_test_brm))
# print('\n\n')

# print('Training Recall: ', recall_score(y_train, y_hat_train_brm))
# print('Testing Recall: ', recall_score(y_test, y_hat_test_brm))
# print('\n\n')

# print('Training Accuracy: ', accuracy_score(y_train, y_hat_train_brm)
# print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test_brm))
# print('\n\n')

# print('Training F1-Score: ', f1_score(y_train, y_hat_train_brm))
# print('Testing F1-Score: ', f1_score(y_test, y_hat_test_brm))
```

```
Fitting 3 folds for each of 33 candidates, totalling 99 fits
1733.5508239269257
```

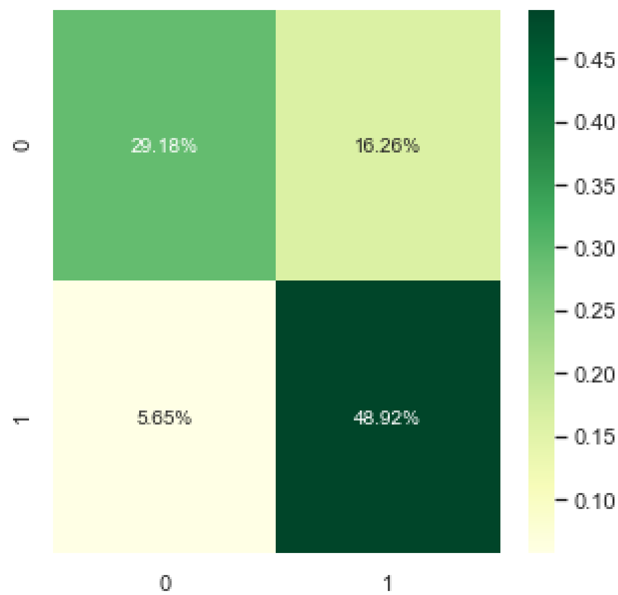
In [38]:

```
xgc_best_random = xgc_random.best_estimator_
xgc_best_random.fit(X_train,y_train)
y_hat_train_brm = xgc_best_random.predict(X_train)
y_hat_test_brm = xgc_best_random.predict(X_test)

print('Training Precision: ', precision_score(y_train, y_hat_train_brm)
print('Testing Precision: ', precision_score(y_test, y_hat_test_brm))
print('\n\n')

print('Training Recall: ', recall_score(y_train, y_hat_train_brm))
print('Testing Recall: ', recall_score(y_test, y_hat_test_brm))
print('\n\n')

print('Training Accuracy: ', accuracy_score(y_train, y_hat_train_brm))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test_brm))
print('\n\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train_brm))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test_brm))
```

```
Training Precision:  0.7548979817732354
Testing Precision:  0.75055966936456



Training Recall:  0.9079210117594853
Testing Recall:  0.896441793500617



Training Accuracy:  0.7903318903318903
Testing Accuracy:  0.7809203142536476



Training F1-Score:  0.8243684274144808
Testing F1-Score:  0.8170400224950791
```

In [39]:
```
#the model performed better than the base XGSboost model
#however it still underperforms compared to my best
#random forest classifier

xgs_rand_matrix = confusion_matrix(y_test, y_hat_test_brm)
print(xgs_rand_matrix)

# Visualize your confusion matrix
fig, ax = plt.subplots(figsize=(5, 5))


sns.heatmap(xgs_rand_matrix/np.sum(xgs_rand_matrix), annot=True,
            fmt='.2%', cmap='YlGn', ax=ax)

plt.show();
```

```
[[5199 2897]
 [1007 8717]]
```



Improvements to accuracy and f1-score. Still needs improvement to both to compete with the random forest models.

In [40]:
```
xgc_best_random
```

Out[40]:
```
XGBClassifier(colsample_bytree=0.8, gamma=0.5, max_depth=5, subsample=1.
0)
```

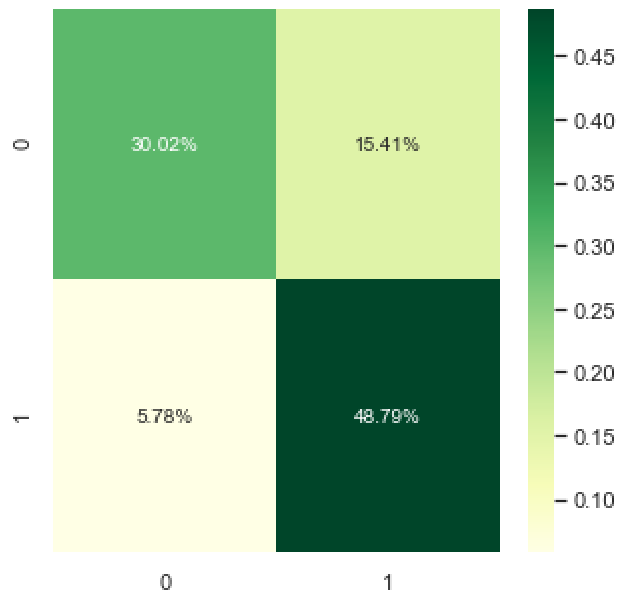
In [41]:

```python
params2 = {
        'gamma': [0.1, 0.3, 0.5],
        'colsample_bytree': [0.8, 0.9],
        'max_depth': [4, 5, 6]
        }

start = time.time()

xgc2 = XGBClassifier()
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
xgc_gs = GridSearchCV(scoring='f1',
                      estimator = xgc2,
                      param_grid = params2,
                      cv = 3,
                      verbose=2,
                      n_jobs = -1)


xgc_gs.fit(X_train, y_train)
xgc_best_gs = xgc_gs.best_estimator_
xgc_best_gs.fit(X_train,y_train)
y_hat_train_bgs = xgc_best_gs.predict(X_train)
y_hat_test_bgs = xgc_best_gs.predict(X_test)


end = time.time()
print(end - start)

print('Training Precision: ', precision_score(y_train, y_hat_train_bgs))
print('Testing Precision: ', precision_score(y_test, y_hat_test_bgs))
print('\n\n')


print('Training Accuracy: ', accuracy_score(y_train, y_hat_train_bgs))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test_bgs))
print('\n\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train_bgs))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test_bgs))
```

```
Fitting 3 folds for each of 18 candidates, totalling 54 fits
582.7580349445343
Training Precision:  0.7660139023843336
Testing Precision:  0.759965034965035



Training Accuracy:  0.8004088504088505
Testing Accuracy:  0.7881032547699215



Training F1-Score:  0.8316393808451504
Testing F1-Score:  0.8215838215838216
```

In [73]:
```
#slight improvement to f1-score as compared to before the grid-search
#still too many false positives to compete with random forest classifie
xgs_grid_matrix = confusion_matrix(y_test, y_hat_test_bgs)
print(xgs_grid_matrix)

# Visualize your confusion matrix
fig, ax = plt.subplots(figsize=(5, 5))


sns.heatmap(xgs_grid_matrix/np.sum(xgs_grid_matrix), annot=True,
            fmt='.2%', cmap='YlGn', ax=ax)

plt.show();
```

```
[[5350 2746]
 [1030 8694]]
```



In [52]:

```
#change scoring to 'precision' for the random search
#will do the same for the grid search afterwards


params = {
        'min_child_weight': [1, 5, 10],
        'gamma': [0.5, 1, 1.5, 2, 5],
        'subsample': [0.6, 0.8, 1.0],
        'colsample_bytree': [0.6, 0.8, 1.0],
        'max_depth': [3, 4, 5]
        }

xgc = XGBClassifier()
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
xgc_random2 = RandomizedSearchCV(scoring='precision',
                               estimator = xgc,
                               param_distributions = params,
                               n_iter = 33, cv = 3, verbose=2,
                               random_state=42,
                               n_jobs = -1)
# Fit the random search model
xgc_random2.fit(X_train, y_train)


end = time.time()
print(end - start)


xgc_best_random2 = xgc_random2.best_estimator_
xgc_best_random2.fit(X_train,y_train)
y_hat_train_brm2 = xgc_best_random2.predict(X_train)
y_hat_test_brm2 = xgc_best_random2.predict(X_test)

print('Training Precision: ', precision_score(y_train, y_hat_train_brm2
print('Testing Precision: ', precision_score(y_test, y_hat_test_brm2))
print('\n\n')

print('Training Recall: ', recall_score(y_train, y_hat_train_brm2))
print('Testing Recall: ', recall_score(y_test, y_hat_test_brm2))
print('\n\n')

print('Training Accuracy: ', accuracy_score(y_train, y_hat_train_brm2))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test_brm2))
print('\n\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train_brm2))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test_brm2))
```

```
Fitting 3 folds for each of 33 candidates, totalling 99 fits
1486.0256688594818
Training Precision:  0.7548979817732354
Testing Precision:  0.75055966936456



Training Recall:  0.9079210117594853
```

```
Testing Recall:  0.896441793500617



Training Accuracy:  0.7903318903318903
Testing Accuracy:  0.7809203142536476



Training F1-Score:  0.8243684274144808
Testing F1-Score:  0.8170400224950791
```

In [55]:

```python
xgc_best_random2
```

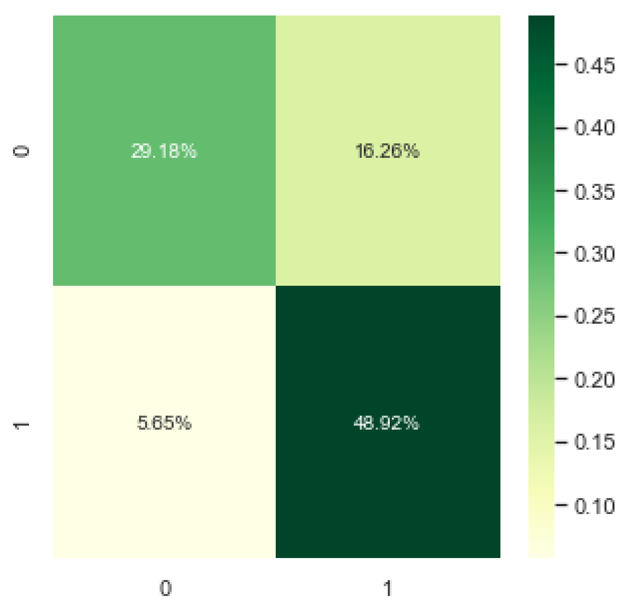
Out[55]:

```
XGBClassifier(colsample_bytree=0.8, gamma=0.5, max_depth=5, subsample=1.
0)
```

In [75]:

```python
#still not performing as well as random forest classifier


xgs_grid_matrix_p = confusion_matrix(y_test, y_hat_test_brm2)
print(xgs_grid_matrix_p)

# Visualize your confusion matrix
fig, ax = plt.subplots(figsize=(5, 5))


sns.heatmap(xgs_grid_matrix_p/np.sum(xgs_grid_matrix_p), annot=True,
            fmt='.2%', cmap='YlGn', ax=ax)

plt.show();
```

```
[[5199 2897]
 [1007 8717]]
```



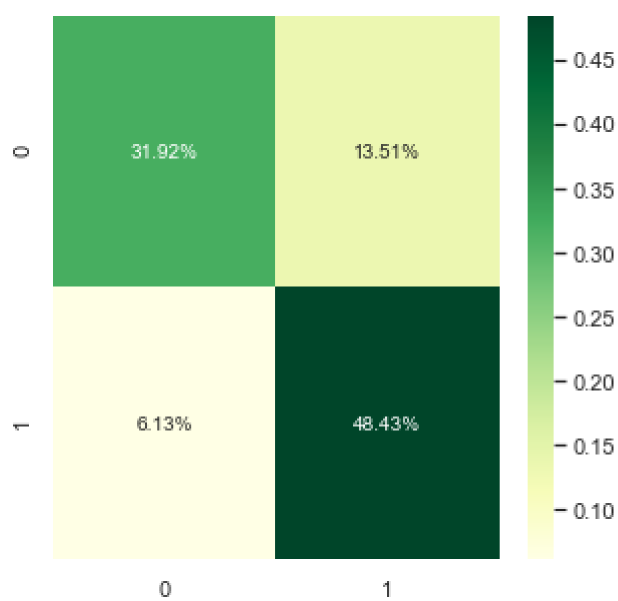
In [59]:

```
params_p = {
        'gamma': [0.3,0.4,0.5],
        'colsample_bytree': [0.8],
        'max_depth': [5,8,10]
        }

start = time.time()

xgc2 = XGBClassifier()
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
xgc_gs2 = GridSearchCV(scoring='precision',
                      estimator = xgc2,
                      param_grid = params_p,
                      cv = 3,
                      verbose=2,
                      n_jobs = -1)


xgc_gs2.fit(X_train, y_train)
xgc_best_gs2 = xgc_gs2.best_estimator_
xgc_best_gs2.fit(X_train,y_train)
y_hat_train_bgs2 = xgc_best_gs2.predict(X_train)
y_hat_test_bgs2 = xgc_best_gs2.predict(X_test)


end = time.time()
print(end - start)

print('Training Precision: ', precision_score(y_train, y_hat_train_bgs2
print('Testing Precision: ', precision_score(y_test, y_hat_test_bgs2))
print('\n\n')


print('Training Accuracy: ', accuracy_score(y_train, y_hat_train_bgs2))
print('Testing Accuracy: ', accuracy_score(y_test, y_hat_test_bgs2))
print('\n\n')

print('Training F1-Score: ', f1_score(y_train, y_hat_train_bgs2))
print('Testing F1-Score: ', f1_score(y_test, y_hat_test_bgs2))
```

```
Fitting 3 folds for each of 9 candidates, totalling 27 fits
503.37574887275696
Training Precision:  0.8044286156704862
Testing Precision:  0.781864299302473



Training Accuracy:  0.8362914862914863
Testing Accuracy:  0.8035353535353535



Training F1-Score:  0.8592635474600451
Testing F1-Score:  0.8313827481577806
```

In [76]:
```
#decent model but not the best model so far

xgs_grid_matrix_gp = confusion_matrix(y_test, y_hat_test_bgs2)
print(xgs_grid_matrix_gp)

# Visualize your confusion matrix
fig, ax = plt.subplots(figsize=(5, 5))


sns.heatmap(xgs_grid_matrix_gp/np.sum(xgs_grid_matrix_gp), annot=True,
            fmt='.2%', cmap='YlGn', ax=ax)

plt.show();
```

```
[[5688 2408]
 [1093 8631]]
```



The grid search produced a pretty good model, but my random forest classifier still performed better.

Evaluation

My random forest models outperformed my best logistic regression and XGSBoost models in regards to the metrics that are most important given the business problem at hand.

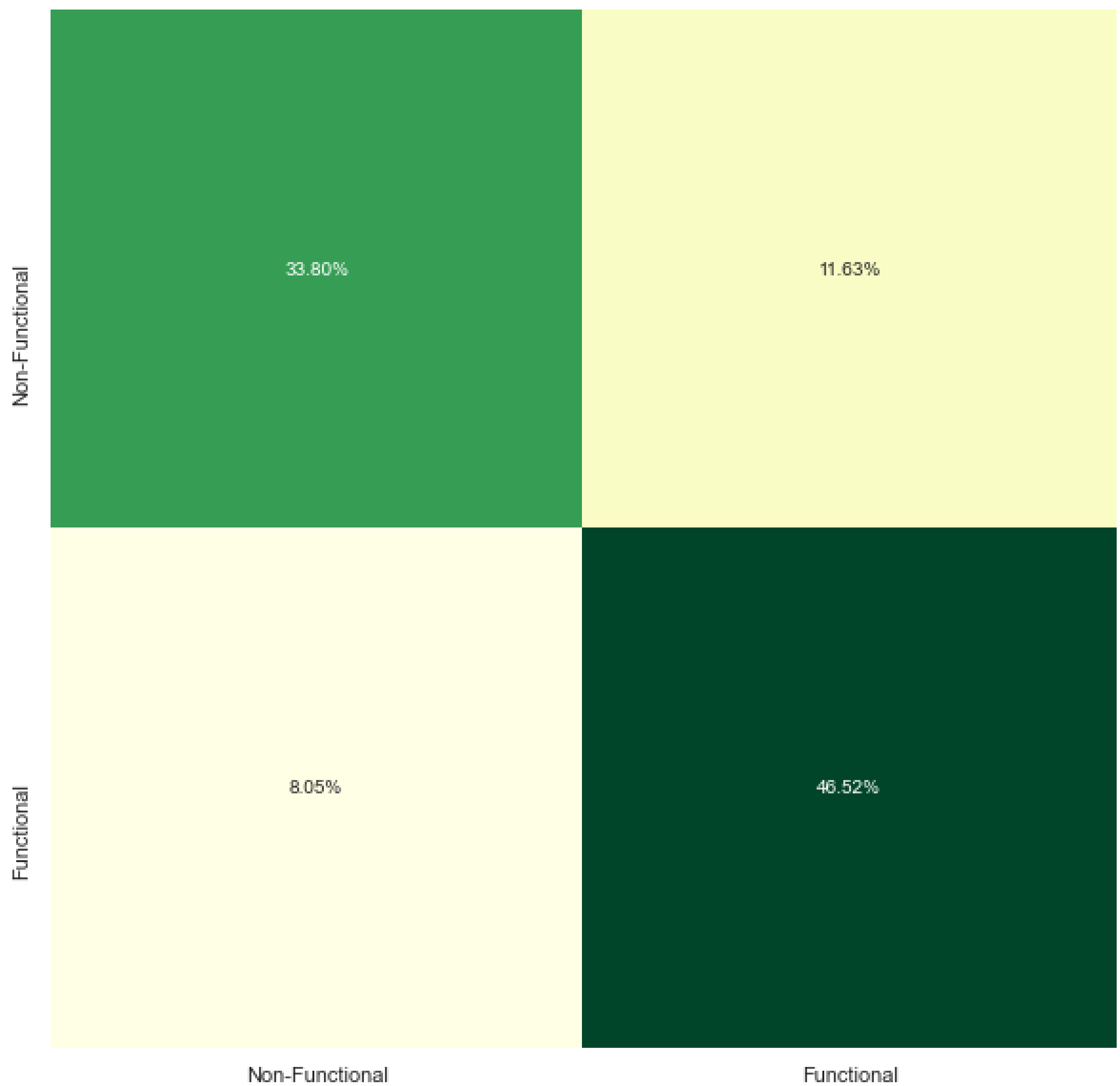
The best random forest model had a great balance between accuracy, precision, and f1-score.

11.63% of pumps would be misclassified as functional using my best model. This means that 11.63% of the pumps would go untreated if this classifier was deployed to conduct predictive maintenance. However, it correctly identifies a high number of functional pumps correctly, which would save a lot of valuable resources, time and money, and it also identifies a large number of non-functional pumps correctly. Only a8.05% of functional pumps would be incorrectly identified as non-functional. This is the resource/time/money sink of my model, so keeping it so low is great.

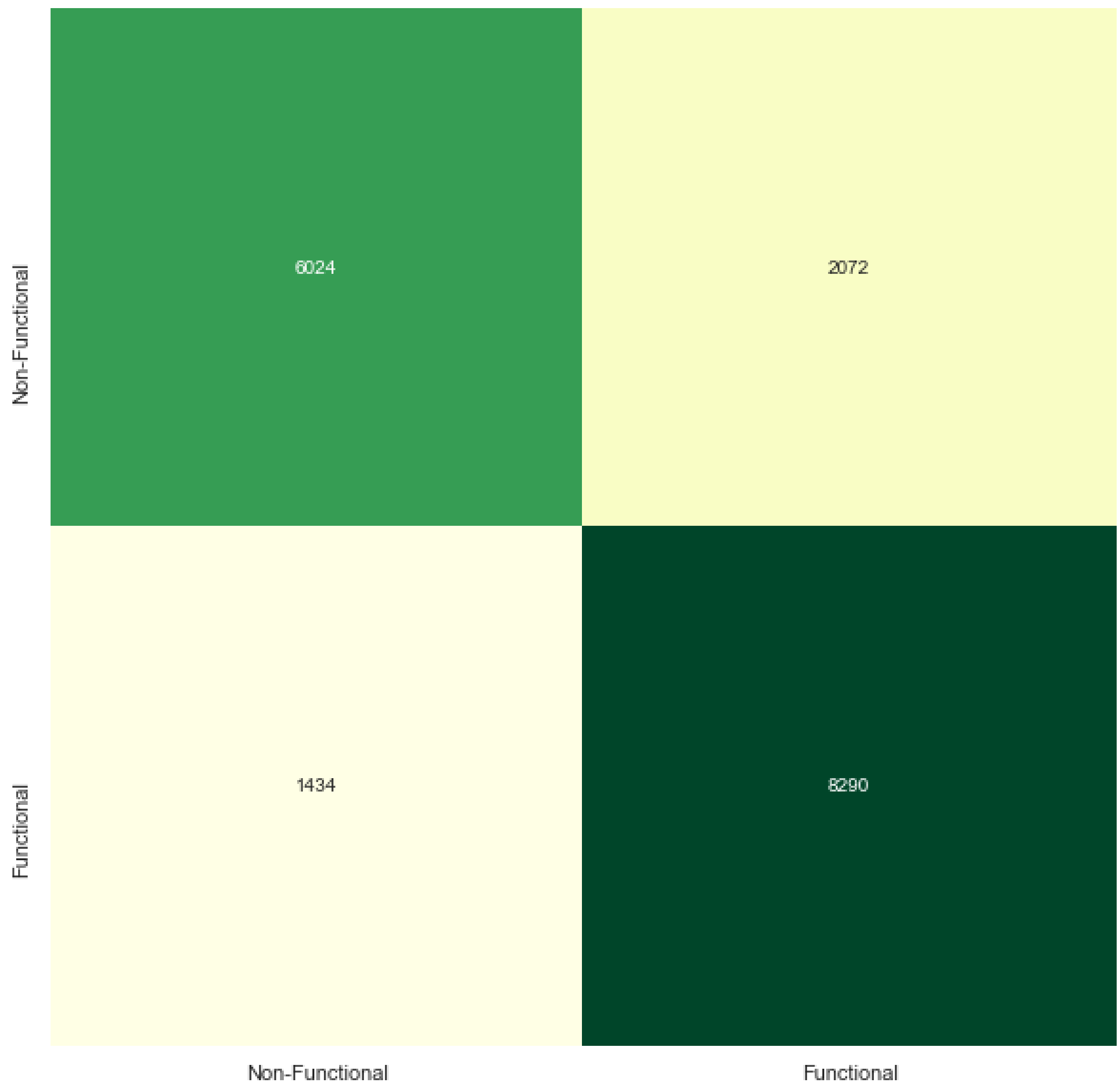
In [90]:

```
fig, ax = plt.subplots(figsize=(10, 10))
print(lr_grid_p_matrix_grf2)

sns.heatmap(lr_grid_p_matrix_grf2/np.sum(lr_grid_p_matrix_grf2),
            annot=True,
            fmt='.2%',
            cmap='YlGn',
            cbar=False,
            xticklabels=['Non-Functional', 'Functional'],
            yticklabels=['Non-Functional', 'Functional'],
            ax=ax)
plt.savefig('images/best_rfc_matrix')
plt.show();
```

```
[[6024 2072]
 [1434 8290]]
```



In [91]:
```
fig, ax = plt.subplots(figsize=(10, 10))

sns.heatmap(lr_grid_p_matrix_grf2,
            annot=True,
            fmt='n',
            cmap='YlGn',
            cbar=False,
            xticklabels=['Non-Functional', 'Functional'],
            yticklabels=['Non-Functional', 'Functional'],
            ax=ax)
plt.savefig('images/best_rfc_matrix2')
plt.show();
```

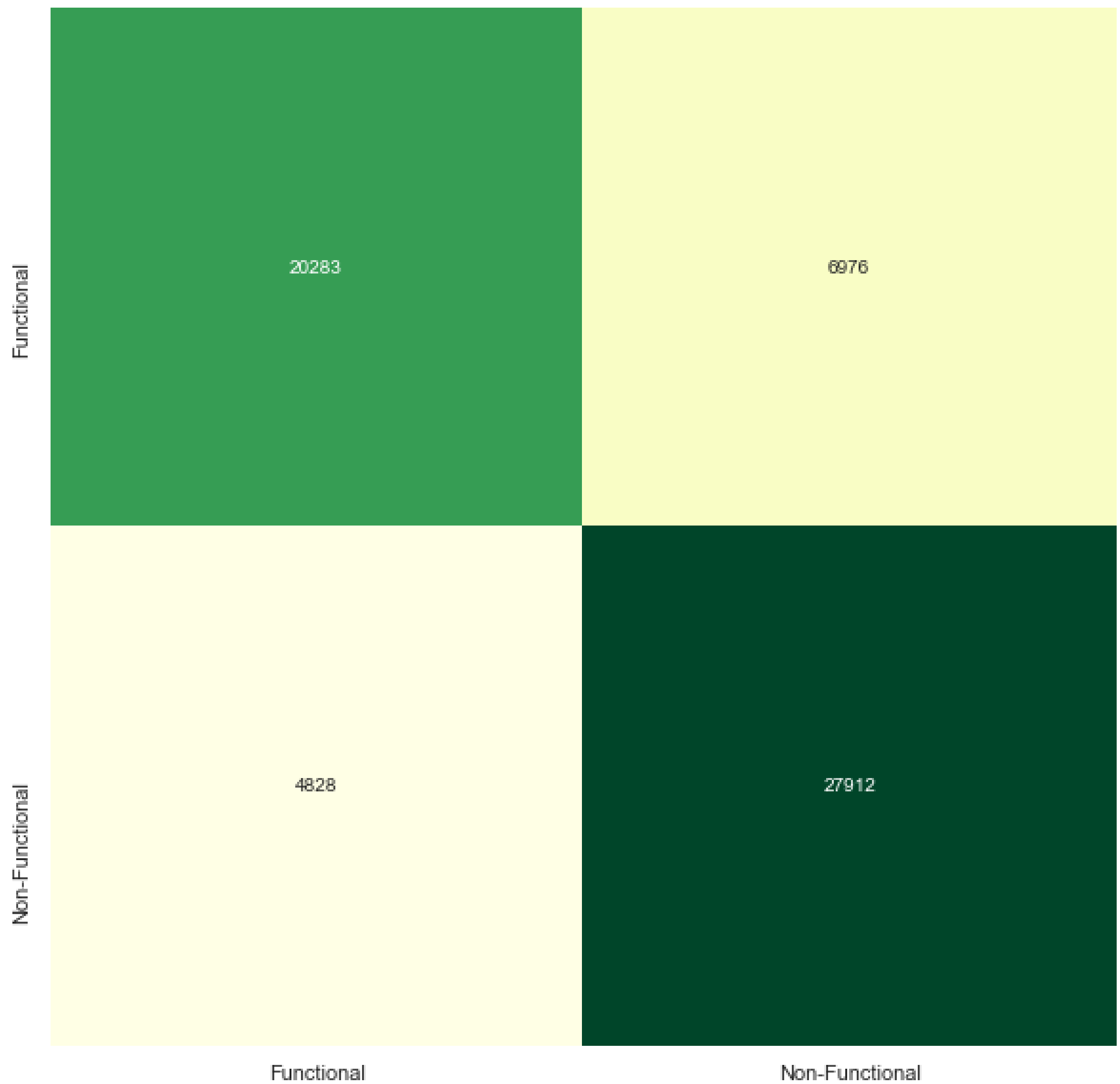


Conclusions

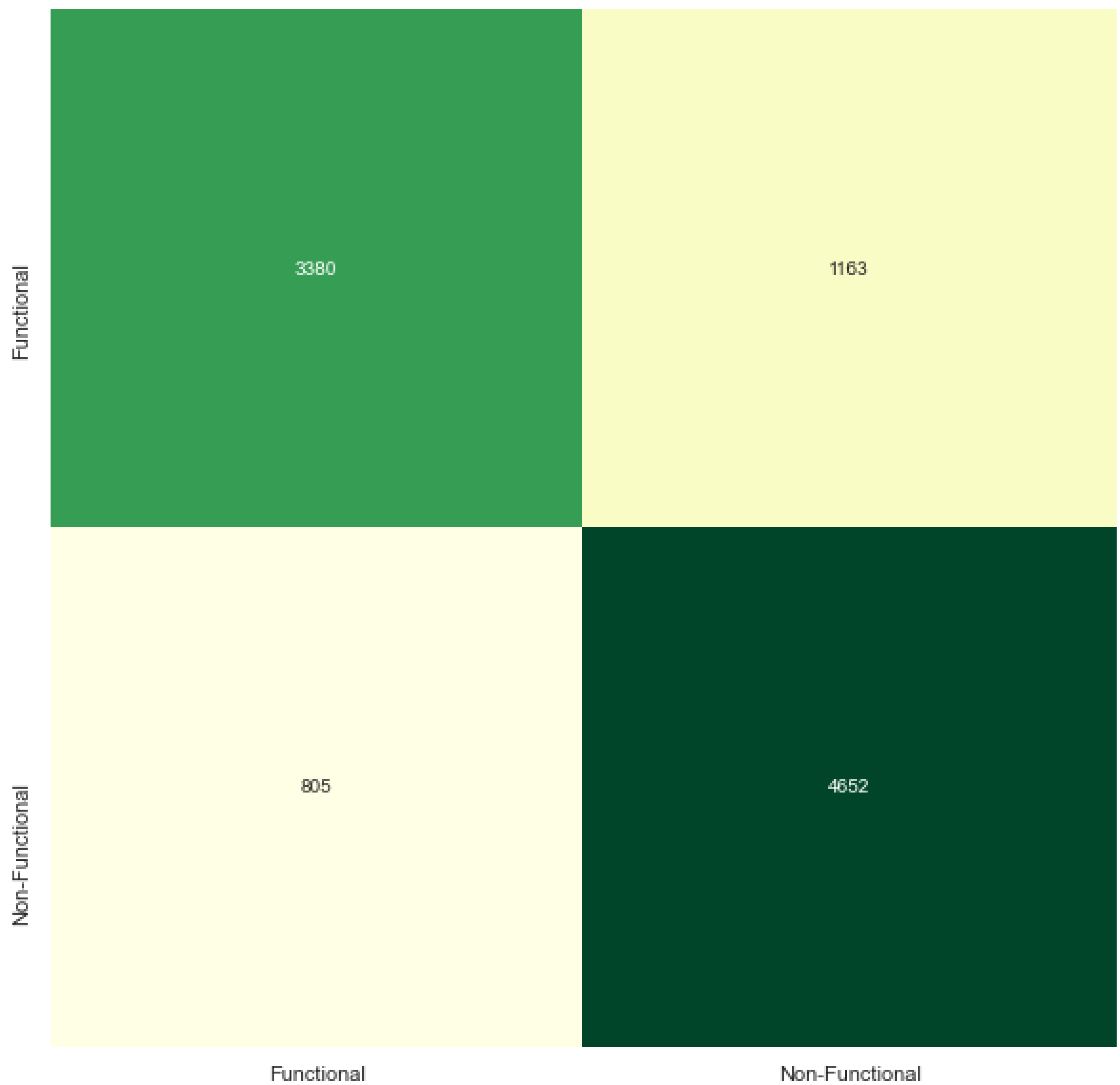
I believe that my best classification model provides a powerful enough predictive ability to prove very valuable to the Ministry of Water. The amount of resources saved, the relatively low number of misclassified functional pumps, and the elimination of the need to physically sweep the functionality of all pumps can bring access to potable drinking water to a larger number of communities than before without predictive maintenance.

In [93]:

```
fig_matrix_60k = 60000*(lr_grid_p_matrix_grf2/np.sum(lr_grid_p_matrix_g
fig_matrix_10k = 10000*(lr_grid_p_matrix_grf2/np.sum(lr_grid_p_matrix_g

fig, ax = plt.subplots(figsize=(10, 10))


sns.heatmap(fig_matrix_60k,
            annot=True,
            fmt='.0f',
            cmap='YlGn',
            cbar=False,
            ax=ax,
            xticklabels=['Functional', 'Non-Functional'],
            yticklabels=['Functional', 'Non-Functional']
           )
plt.savefig('images/best_rfc_matrix_60k')
plt.show();
```



In [94]:
```
fig, ax = plt.subplots(figsize=(10, 10))


sns.heatmap(fig_matrix_10k,
            annot=True,
            fmt='.0f',
            cmap='YlGn',
            cbar=False,
            ax=ax,
            xticklabels=['Functional', 'Non-Functional'],
            yticklabels=['Functional', 'Non-Functional']
           )
plt.savefig('images/best_rfc_matrix_10k')
plt.show();
```



Thank you! For questions or comments please feel free to reach me by email.

Author: Dylan Dey

Email: ddey2985@gmail.com (mailto:ddey2985@gmail.com)

github: https://github.com/ddey117/Tanzanian_Water_Pump_Classification (https://github.com/ddey117/Tanzanian_Water_Pump_Classification)