Building Energy Star Score Prediction Report

Suhas Londhe

17th Mar, 2021

Contents

1 Introduction

In this report I present findings from an exploration of the NYC benchmarking dataset which measures 60 variables related to energy use for over 11,000 buildings. Of primary interest is the Energy Star Score, which is often used as an aggregate measure of the overall efficiency of a building. The Energy Star Score is a percentile measure of a building's energy performance calculated from self-reported energy usage.

1.1 Objectives

- Identify predictors within the dataset for the Energy Star Score
- Build Regression/Classification models that can predict the Energy Star Score of a building given the building's energy data
- Interpret the results of the model and use the trained model to infer Energy Star Scores of new buildings

Preliminary data cleaning and exploration was completed in another Jupyer Notebook.

```
# Pandas and numpy for data manipulation
import pandas as pd
import numpy as np

# No warnings about setting value on copy of slice
pd.options.mode.chained_assignment = None

# Display up to 60 columns of a dataframe
pd.set_option('display.max_columns', 60)

# Matplotlib visualization
import matplotlib.pyplot as plt
%matplotlib inline

# Set default font size
plt.rcParams['font.size'] = 24
```

```
# Internal ipython tool for setting figure size
from IPython.core.pylabtools import figsize


# Seaborn for visualization
import seaborn as sns
sns.set(font_scale = 2)


# Splitting data into training and testing
from sklearn.model_selection import train_test_split


# Read in data into a dataframe


data = pd.read_csv('E:/Data Science class/Projects/input_data.csv')


# Display top of dataframe
data.head()
```

There are over 11,000 buildings in the dataset with 60 energy-related features each. Many of the columns contain a significant portion of missing values which were initially encoded as ’Not Available’. In a previous notebook, I filled in the missing values with nan and converted the appropriate columns to numerical values.

2 Data Exploration

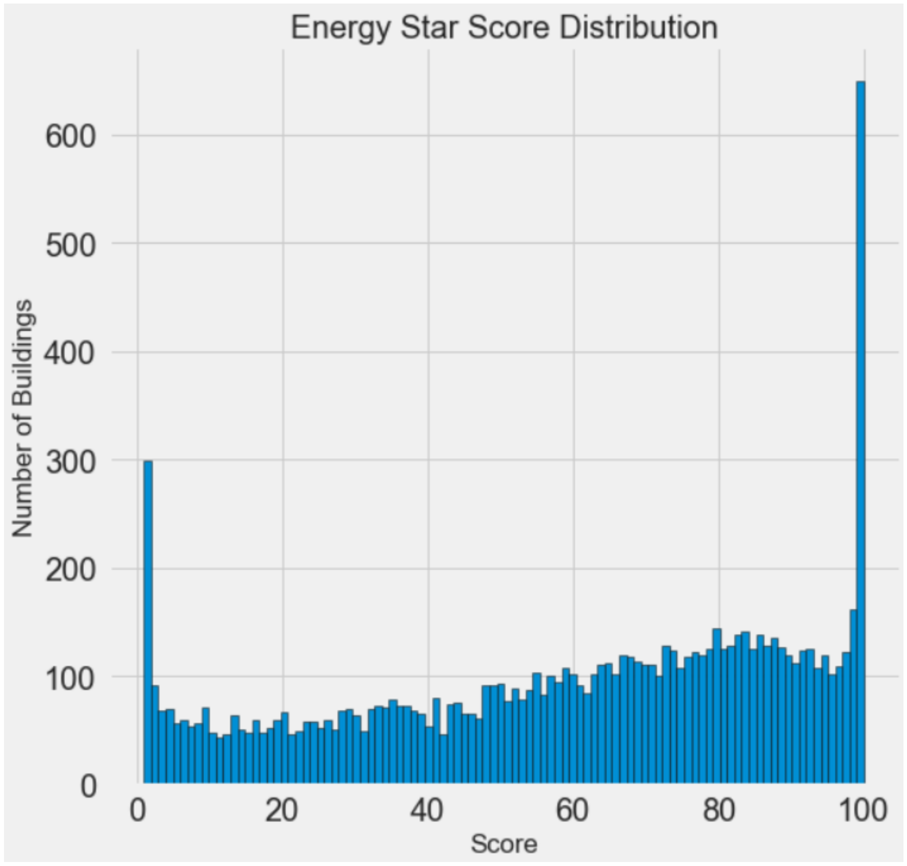
As we are concerned mainly with the energy star score, the first chart to make shows the distribution of this measure across all the buildings in the dataset that have a score (9642).

2.1 Univariate analysis – Single variable plots

A single variable (called univariate) plot shows the distribution of a single variable such as in a histogram.

```
figsize(8, 8)
```

```
# Rename the score
data = data.rename(columns = {'ENERGY STAR Score': 'score'})

# Histogram of the Energy Star Score
plt.style.use('fivethirtyeight')
plt.hist(data['score'].dropna(), bins = 100, edgecolor = 'k');
plt.xlabel('Score'); plt.ylabel('Number of Buildings');
plt.title('Energy Star Score Distribution');
```



2.2 Removing outliers

When we remove outliers, we want to be careful that we are not throwing away measurements just because they look strange. They may be the result of actual phenomenon that we should further investigate. When removing outliers, I try to be as conservative as possible, using the definition of an extreme outlier:

- On the low end, an extreme outlier is below $\text{First Quartile} - 3 * \text{Interquartile Range}$First Quartile−3∗Interquartile Range
- On the high end, an extreme outlier is above $\text{Third Quartile} + 3 * \text{Interquartile Range}$Third Quartile+3∗Interquartile Range

In this case, I will only remove the single outlying point and see how the distribution looks.

```
# Calculate first and third quartile

first_quartile = data['Site EUI (kBtu/ft²)'].describe()['25%']

third_quartile = data['Site EUI (kBtu/ft²)'].describe()['75%']


# Interquartile range

iqr = third_quartile - first_quartile


# Remove outliers

data = data[(data['Site EUI (kBtu/ft²)'] > (first_quartile - 3 * iqr)) &

            (data['Site EUI (kBtu/ft²)'] < (third_quartile + 3 * iqr))]
```

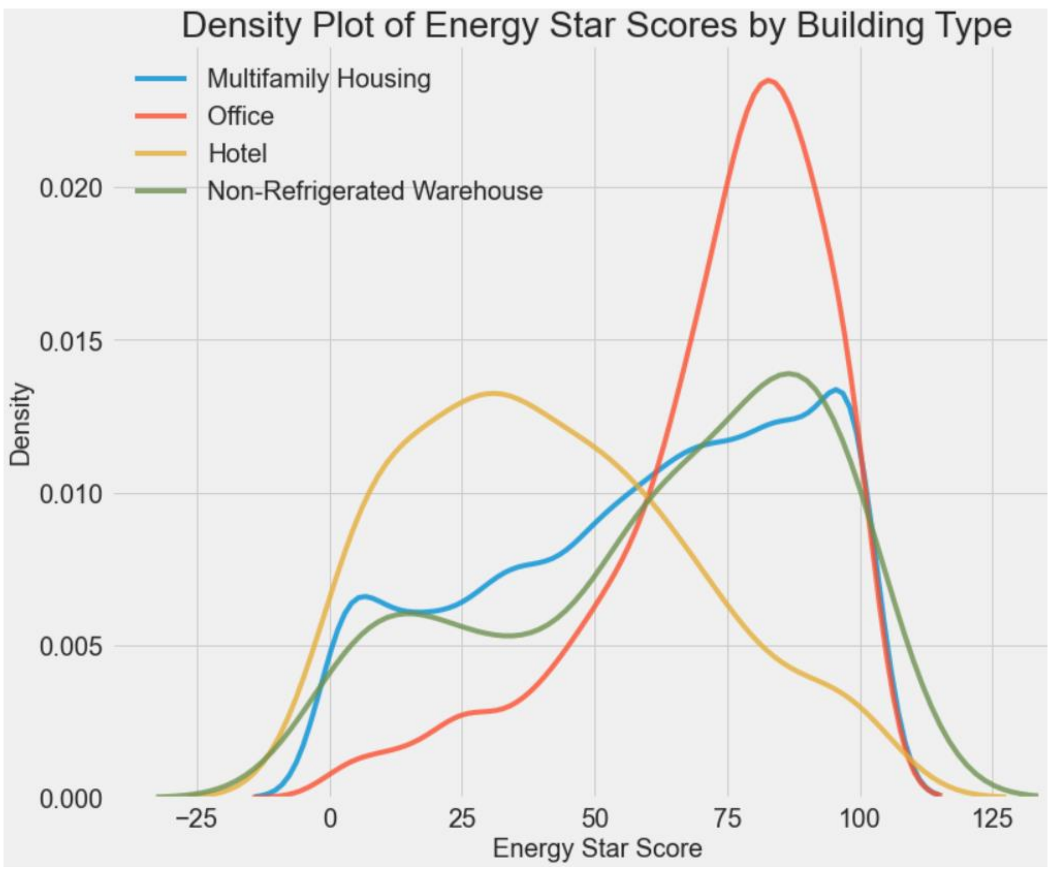
2.3 Looking for Relationships

In order to look at the effect of categorical variables on the score, we can make a density plot colored by the value of the categorical variable. Density plots also show the distribution of a single variable and can be thought of as a smoothed histogram. If we color the density curves by a categorical variable, this will shows us how the distribution changes based on the class.

The first plot we will make shows the distribution of scores by the property type. In order to not clutter the plot, we will limit the graph to building types that have more than 100 observations in the dataset.

```
# Create a list of buildings with more than 100 measurements

types = data.dropna(subset=['score'])

types = types['Largest Property Use Type'].value_counts()

types = list(types[types.values > 100].index)
```

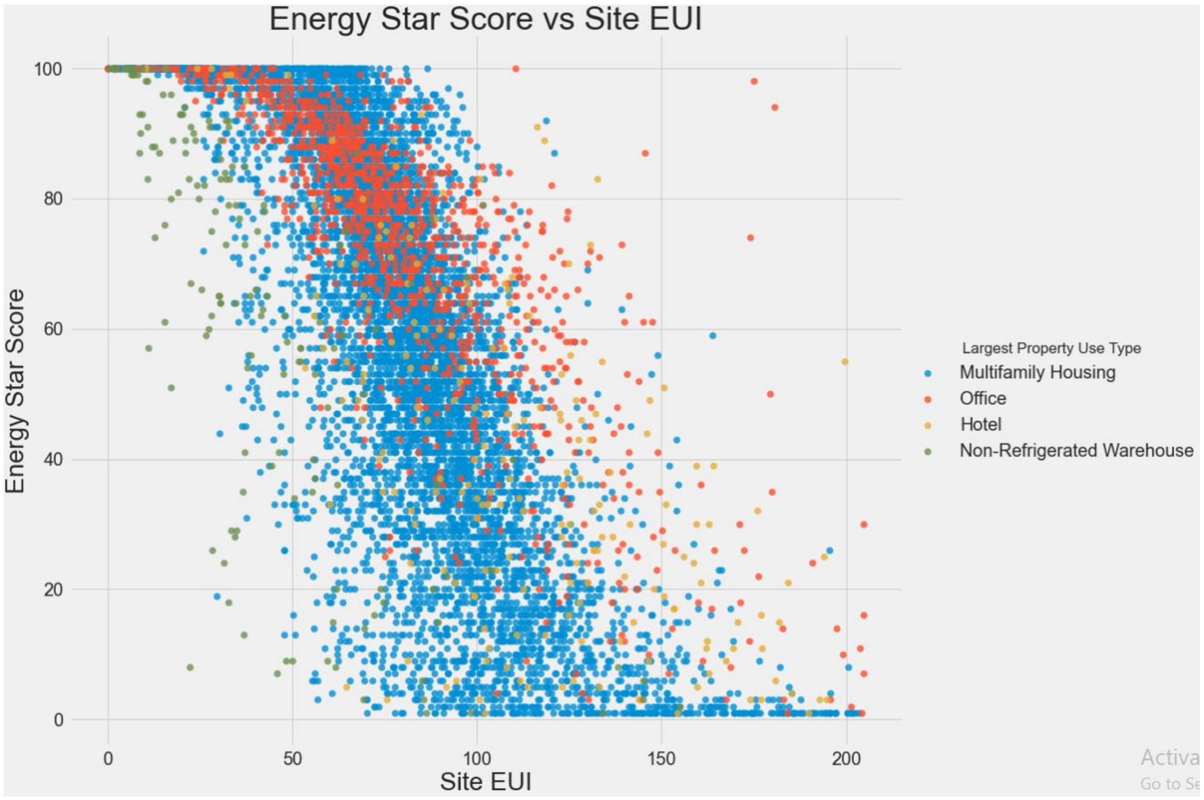
```
# Plot of distribution of scores for building categories
figsize(12, 10)

# Plot each building
for b_type in types:
    # Select the building type
    subset = data[data['Largest Property Use Type'] == b_type]

    # Density plot of Energy Star scores
    sns.kdeplot(subset['score'].dropna(),
               label = b_type, shade = False, alpha = 0.8);

# label the plot
plt.xlabel('Energy Star Score', size = 20); plt.ylabel('Density', size = 20);
plt.title('Density Plot of Energy Star Scores by Building Type', size = 28);
```



2.4 Two variable plots

In order to visualize the relationship between two variables, we use a scatterplot. We can also include additional variables using aspects such as color of the markers or size of the markers. Here we will plot two numeric variables against one another and use color to represent a third categorical variable.

```
figsize(12, 10)

# Extract the building types
features['Largest Property Use Type'] = data.dropna(subset = ['score'])['Largest Property Use Type']

# Limit to building types with more than 100 observations (from previous code)
features = features[features['Largest Property Use Type'].isin(types)]

# Use seaborn to plot a scatterplot of Score vs Log Source EUI
sns.lmplot('Site EUI (kBtu/ft²)', 'score',
    hue = 'Largest Property Use Type', data = features,
    scatter_kws = {'alpha': 0.8, 's': 60}, fit_reg = False,
    size = 12, aspect = 1.2);

# Plot labeling
plt.xlabel("Site EUI", size = 28)
plt.ylabel('Energy Star Score', size = 28)
plt.title('Energy Star Score vs Site EUI', size = 36);
```



3 Feature Engineering & selection

3.1 Remove collinear Features

Highly collinear features have a significant correlation coefficent between them. For example, in our dataset, the Site EUI and Weather Norm EUI are highly correlated because they are just slightly different means of calculating the energy use intensity.

```
def remove_collinear_features(x, threshold):
    '''
    Objective:
        Remove collinear features in a dataframe with a correlation coefficient
        greater than the threshold. Removing collinear features can help a model
        to generalize and improves the interpretability of the model.

    Inputs:
        threshold: any features with correlations greater than this value are removed

    Output:
        dataframe that contains only the non-highly-collinear features
    '''

    # Dont want to remove correlations between Energy Star Score
    y = x['score']
    x = x.drop(columns = ['score'])

    # Calculate the correlation matrix
    corr_matrix = x.corr()
    iters = range(len(corr_matrix.columns) - 1)
    drop_cols = []

    # Iterate through the correlation matrix and compare correlations
    for i in iters:
        for j in range(i):
            item = corr_matrix.iloc[j:(j+1), (i+1):(i+2)]
            col = item.columns
            row = item.index
            val = abs(item.values)

            # If correlation exceeds the threshold
            if val >= threshold:
                # Print the correlated features and the correlation value
                # print(col.values[0], "|", row.values[0], "|", round(val[0][0], 2))
                drop_cols.append(col.values[0])

    # Drop one of each pair of correlated columns
    drops = set(drop_cols)
    x = x.drop(columns = drops)
    x = x.drop(columns = ['Weather Normalized Site EUI (kBtu/ft²)',
                          'Water Use (All Water Sources) (kgal)',
                          'log_Water Use (All Water Sources) (kgal)',
                          'Largest Property Use Type - Gross Floor Area (ft²)'])

    # Add the score back in to the data
    x['score'] = y

    return x

# Remove the collinear features above a specified correlation coefficient
features = remove_collinear_features(features, 0.6);
```

3.2 Split the data into training & testing sets

In machine learning, we always need to separate our features into two sets:

1. **Training set** (X_train, y_train) which we provide to our model during training along with the answers so it can learn a mapping between the features and the target.
2. **Testing set** (X_test, y_test) which we use to evaluate the mapping learned by the model. The model has never seen the answers on the testing set, but instead, must make predictions using only the features. As we

know the true answers for the test set, we can then compare the test predictions to the true test targets to ghet an estimate of how well our model will perform when deployed in the real world.

```
# Extract the buildings with no score and the buildings with a score
no_score = features[features['score'].isna()]
score = features[features['score'].notnull()]

print(no_score.shape)
print(score.shape)

# Separate out the features and targets
features = score.drop(columns='score')
targets = pd.DataFrame(score['score'])

# Replace the inf and -inf with nan (required for later imputation)
features = features.replace({np.inf: np.nan, -np.inf: np.nan})

# Split into 70% training and 30% testing set
X, X_test, y, y_test = train_test_split(features, targets, test_size = 0.3, random_state = 42)

print(X.shape)
print(X_test.shape)
print(y.shape)
print(y_test.shape)
```

3.3 Establish a baseline

It's important to establish a naive baseline before we beginning making machine learning models. If the models we build cannot outperform a naive guess then we might have to admit that machine learning is not suited for this problem. This could be because we are not using the right models, because we need more data, or because there is a simpler solution that does not require machine learning. Establishing a baseline is crucial so we do not end up building a machine learning model only to realize we can't actually solve the problem.

3.4 Performance metric : MAE

There are a number of metrics used in machine learning tasks and it can be difficult to know which one to choose. Most of the time it will depend on the particular problem and if you have a specific goal to optimize for. I like Andrew Ng's advice to use a single real-value performance metric in order to compare models because it simplifies the evaluate process. Rather than calculating multiple metrics and trying to determine how important each one is, we should use a single number. In this case, because we doing regression, the **mean absolute error** is an appropriate metric. This is also interpretable because it represents the average amount our estimate if off by in the same units as the target value.

```
# Function to calculate mean absolute error
def mae(y_true, y_pred):
    return np.mean(abs(y_true - y_pred))

baseline_guess = np.median(y)

print('The baseline guess is a score of %0.2f' % baseline_guess)
print("Baseline Performance on the test set: MAE = %0.4f" % mae(y_test, baseline_guess))
```

4 Evaluating & comparing machine learning models

4.1 Imputing missing values

Standard machine learning models cannot deal with missing values, and which means we have to find a way to fill these in or disard any features with missing values. Since we already removed features with more than 50% missing values in the first part, here we will focus on filling in these missing values, a process known as imputation). There are a number of methods for imputation but here we will use the relatively simple method of replacing missing values with the median of the column. (Here is a more thorough discussion on imputing missing values)

```
!pip install impyute
from impyute.imputation.cs import mice
from sklearn.ensemble import RandomForestRegressor,GradientBoostingRegressor

imputer = IterativeImputer(GradientBoostingRegressor())
# Train on the training features
imputer.fit(train_features)

# Transform both training data and testing data
X = imputer.transform(train_features)
X_test = imputer.transform(test_features)

# Create an imputer object with a median filling strategy
#imputer = Imputer(strategy='median')

# Train on the training features
#imputer.fit(train_features)

# Transform both training data and testing data
#X = imputer.transform(train_features)
#X_test = imputer.transform(test_features)
```

4.2 Scaling features

The final step to take before we can build our models is to scale the features. This is necessary because features are in different units, and we want to normalize the features so the units do not affect the algorithm. Linear Regression and Random Forest do not require feature scaling, but other methods, such as support vector machines and k nearest neighbors, do require it because they take into account the

Euclidean distance between observations. For this reason, it is a best practice to scale features when we are comparing multiple algorithms.

```
# Create the scaler object with a range of 0-1
#scaler = MinMaxScaler(feature_range=(0, 1))
from sklearn.preprocessing import StandardScaler
scaler = MinMaxScaler()
# Fit on the training data
scaler.fit(X)

# Transform both the training and testing data
X = scaler.transform(X)
X_test = scaler.transform(X_test)

# Convert y to one-dimensional array (vector)
y = np.array(train_labels).reshape((-1, ))
y_test = np.array(test_labels).reshape((-1, ))
```

4.3 Machine learning models to evaluate

We will compare five different machine learning models using the great Scikit-Learn library:

1. Linear Regression
2. Support Vector Machine Regression
3. Random Forest Regression
4. Gradient Boosting Regression
5. K-Nearest Neighbors Regression

```
# Function to calculate mean absolute error
def mae(y_true, y_pred):
    return np.mean(abs(y_true - y_pred))

# Takes in a model, trains the model, and evaluates the model on the test set
def fit_and_evaluate(model):

    # Train the model
    model.fit(X, y)

    # Make predictions and evalute
    model_pred = model.predict(X_test)
    model_mae = mae(y_test, model_pred)

    # Return the performance metric
    return model_mae
```

```
lr = LinearRegression()
lr_mae = fit_and_evaluate(lr)
print('Linear Regression Performance on the test set: MAE = %0.4f' % lr_mae)

from sklearn.model_selection import cross_val_score
scoreslr = cross_val_score(lr, X, y, cv=10)
print("Linear Regression")
print(scoreslr)
#Bias Variance Error Calculation
print("Bias Errors (1-Accuracy/R2)")
print(1-np.mean(scoreslr))
print("Variance Errors")
print(np.var(scoreslr))

svm = SVR(C = 1000, gamma = 0.1)
svm_mae = fit_and_evaluate(svm)

print('Support Vector Machine Regression Performance on the test set: MAE = %0.4f' % svm_mae)

from sklearn.model_selection import RandomizedSearchCV
import scipy.stats as stats

# defining parameter range
param_grid = {"C": stats.uniform(10, 1000),
            "gamma": stats.uniform(0.1, 10)}

grid = RandomizedSearchCV(SVR(), param_grid, verbose = 3)

# fitting the model for grid search
grid.fit(X, y)
```

4.4 Model Optimization

In machine learning, optimizing a model means finding the best set of hyperparameters for a particular problem.

Hyperparameters -

First off, we need to understand what model hyperparameters are in contrast to model parameters :

- Model hyperparameters are best thought of as settings for a machine learning algorithm that are tuned by the data scientist before training. Examples would be the number of trees in the random forest, or the number of neighbors used in K Nearest Neighbors Regression.
- Model parameters are what the model learns during training, such as the weights in the linear regression.

We as data scientists control a model by choosing the hyperparameters, and these choices can have a significant effect on the final performance of the model (although usually not as great of an effect as getting more data or engineering features).

Tuning the model hyperparameters controls the balance of under vs over fitting in a model. We can try to correct for under-fitting by making a more complex model, such as using more trees in a random forest or more layers in a deep neural network. A model that underfits has high bias, and occurs when our model does not have enough capacity (degrees of freedom) to learn the relationship between the features and the target. We can try to correct for overfitting by limiting the complexity of the model and applying regularization. This might mean decreasing the degree of a polynomial regression, or adding dropout layers to a deep neural network. A model that overfits has high variance and in effect has memorized the training set. Both underfitting and overfitting lead to poor generalization performance on the test set.

4.5 Hyperparameter tuning

We can choose the best hyperparameters for a model through random search and cross validation.

- Random search refers to the method in which we choose hyperparameters to evaluate: we define a range of options, and then randomly select combinations to try. This is in contrast to grid search which evaluates every single combination we specify. Generally, random search is better when we have limited knowledge of the best model hyperparameters and we can use random search to narrow down the options and then use grid search with a more limited range of options.
- Cross validation is the method used to assess the performance of the hyperparameters. Rather than splitting the training set up into separate training and validation sets which reduces the amount of training data we can use, we use K-Fold Cross Validation. This means dividing the training data into K folds, and then going through an iterative process where we first train on K-1 of the folds and then evaluate performance on the kth fold. We repeat this process K times so eventually we will have tested on every example in the training data with the key that each iteration we are testing on data that we did not train on. At the end of K-fold cross validation, we take the average error on each of the K iterations as the final

performance measure and then train the model on all the training data at once. The performance we record is then used to compare different combinations of hyperparameters.

```
# defining parameter range
param_grid = {"n_estimators": np.arange(10, 5000)}

random_cv = RandomizedSearchCV(GradientBoostingRegressor(), param_grid, verbose = 3)

# fitting the model for grid search
random_cv.fit(X, y)

# Get the results into a dataframe
results = pd.DataFrame(grid_search.cv_results_)

# Plot the training and testing error vs number of trees
figsize(8, 8)
plt.style.use('fivethirtyeight')
plt.plot(results['param_n_estimators'], -1 * results['mean_test_score'], label = 'Testing Error')
plt.plot(results['param_n_estimators'], -1 * results['mean_train_score'], label = 'Training Error')
plt.xlabel('Number of Trees'); plt.ylabel('Mean Abosolute Error'); plt.legend();
plt.title('Performance vs Number of Trees');
# Create a list of buildings with more than 100 measurements
types = data.dropna(subset=['score'])
types = types['Largest Property Use Type'].value_counts()
types=list(types[types.values>100].index)
```

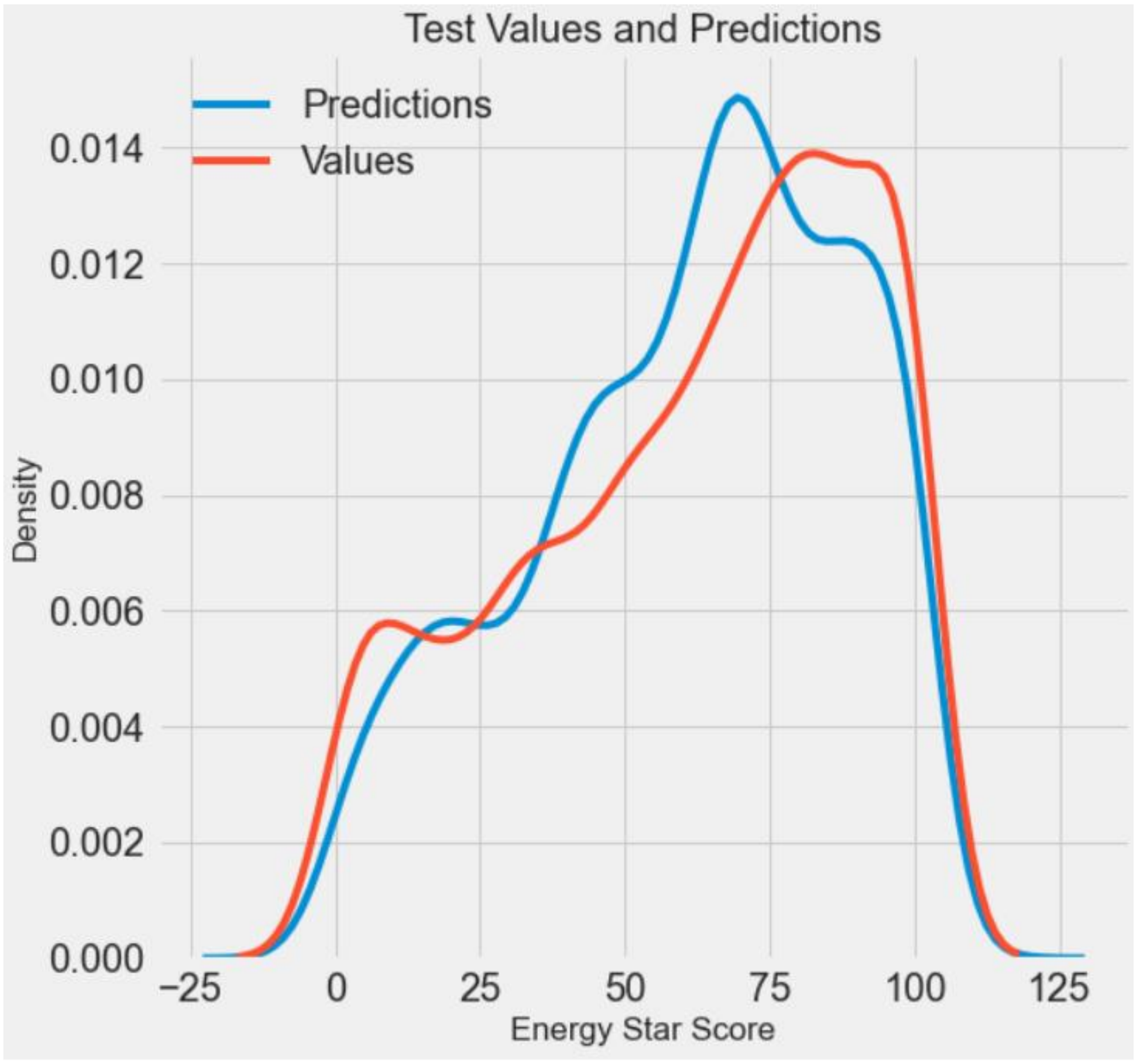
4.6 Evaluate the final model

We will use the best model from hyperparameter tuning to make predictions on the testing set. Remember, our model has never seen the test set before, so this performance should be a good indicator of how the model would perform if deployed in the real world.

For comparison, we can also look at the performance of the default model. The code below creates the final model, trains it (with timing), and evaluates on the test set.

```
# Default model
default_model = GradientBoostingRegressor(random_state = 42)

# Select the best model
final_model = grid_search.best_estimator_

final_model

figsize(8, 8)
```

```
# Density plot of the final predictions and the test values
sns.kdeplot(final_pred, label = 'Predictions')
sns.kdeplot(y_test, label = 'Values')

# Label the plot
plt.xlabel('Energy Star Score'); plt.ylabel('Density');
plt.title('Test Values and Predictions');
```



4.7 Recreate the final model

```
# Create an imputer object with a median filling strategy
#imputer = Imputer(strategy='median')

# Train on the training features
#imputer.fit(train_features)

# Transform both training data and testing data
#X = imputer.transform(train_features)
#X_test = imputer.transform(test_features)

# Sklearn wants the labels as one-dimensional vectors
y = np.array(train_labels).reshape((-1,))
y_test = np.array(test_labels).reshape((-1,))
# Function to calculate mean absolute error
def mae(y_true, y_pred):
    return np.mean(abs(y_true - y_pred))
model = GradientBoostingRegressor(loss='lad', max_depth=5, max_features=None,
                                  min_samples_leaf=6, min_samples_split=6,
                                  n_estimators=800, random_state=42)


model.fit(X, y)
#  Make predictions on the test set
model_pred = model.predict(X_test)

print('Final Model Performance on the test set: MAE = %0.4f' % mae(y_test, model_pred))
```

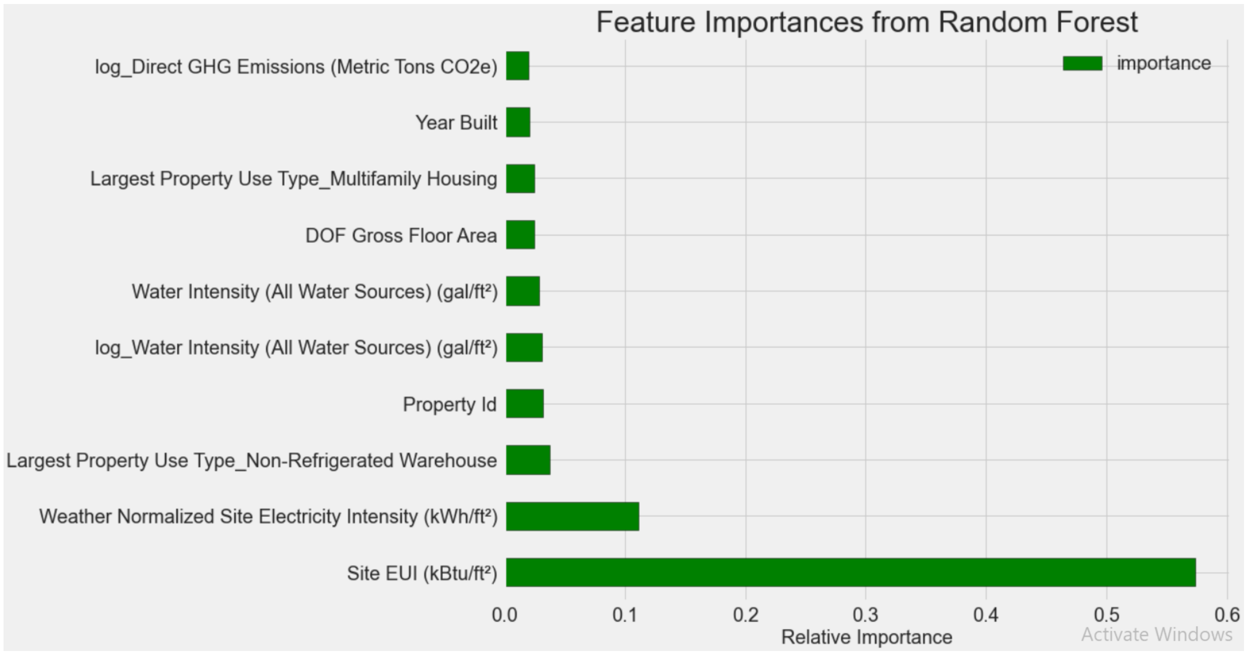
4.8 Interprete the model

Machine learning is often criticized as being a black-box: we put data in on one side and it gives us the answers on the other. While these answers are often extremely accurate, the model tells us nothing about how it actually made the predictions. This is true to some extent, but there are ways in which we can try and discover how a model "thinks" such as the Locally Interpretable Model-agnostic Explainer (LIME). This attempts to explain model predictions by learning a linear regression around the prediction, which is an easily interpretable model!

We will explore several ways to interpret our model:

- Feature importance
- Locally Interpretable Model-agnostic Explainer (LIME)
- Examining a single decision tree in the ensemble.

```
# Extract the feature importances into a dataframe
feature_results = pd.DataFrame({'feature': list(train_features.columns),
                    'importance': model.feature_importances_})

# Show the top 10 most important
feature_results = feature_results.sort_values('importance', ascending = False).reset_index(drop=True)

feature_results.head(10)

figsize(12, 10)
plt.style.use('fivethirtyeight')

# Plot the 10 most important features in a horizontal bar chart
feature_results.loc[:9, :].plot(x = 'feature', y = 'importance',
                     edgecolor = 'k',
                     kind='barh', color = 'green');
plt.xlabel('Relative Importance', size = 20); plt.ylabel('')
plt.title('Feature Importances from Random Forest', size = 30);
```



5 Conclusions

We set out to answer the question: Can we build a model to predict the Energy Star Score of a building and what variables provide us the most information about the score? Given the exploration in this notebook, I conclude that yes, we can create a model to accurately infer the Energy Star Score of a building and we have determined that the Site Energy Use Intensity, the Electricity Intensity, the floor area, the natural gas use, and the Building Type are the most useful measures for determining the energy star score.

The final part of the machine learning pipeline might be the most important: we need to compress everything we have learned into a short summary highlighting only the most crucial findings. Personally, I have difficulty avoiding explaining all the technical details because I enjoy all the work. However, the person you're presenting to probably doesn't have much time to listen to all the details and just wants to hear the takeaways. Learning to extract the most important elements of data science or machine learning project is a crucial skill, because if our results aren't understood by others, then they will never be used!

I encourage you to come up with your own set of conclusions, but here are my top 2 designed to be communicated in 30 seconds:

1. Using the given building energy data, a machine learning model can predict the Energy Star Score of a building to within 10 points.
2. The most important variables for determining the Energy Star Score are the Energy Use Intensity, Electricity Use Intensity, and the Water Use Intensity
3. Offices, residence halls, and non-refrigerated warehouses have higher energy star scores than senior care communities and hotels with scores of multi-family housing falling in between.
4. If provided with data for a new building, a trained model can accurately infer the Energy Star Score.
5. The Site Energy Use Intensity, the Electricity Intensity, and the natural gas usage are all negatively correlated with the Energy Star Score.

If anyone asks for the details, then we can easily explain all the implementation steps, and present our (hopefully) well-documented work. Another crucial aspect of a machine learning project is that you have commented all your code and made it easy to follow! You want someone else (or yourself in a few months) to be able to look at your work and completely understand the decisions you made. Ideally you should write code with the intention that it will be used again. Even when we are doing projects by ourselves, it's good to practice proper documentation and it will make your life much easier when you want to revisit a project.

This report also identified areas for follow-up. These include finding an objective measurement of overall building energy performance and determining why the Energy Star Score distributions vary between building types. That’s all for now!