Introduction: Machine Learning Project Part 3

In this project, we are walking through a complete machine learning problem dealing with a real-world dataset. Using building energy data, we are trying to build a model to predict the Energy Star Score of a building, making this a supervised regression, machine learning task.

In the first two parts of this project, we implemented the first 6 steps of the machine learning pipeline:

1. Data cleaning and formatting
2. Exploratory data analysis
3. Feature engineering and selection
4. Compare several machine learning models on a performance metric
5. Perform hyperparameter tuning on the best model to optimize it for the problem
6. Evaluate the best model on the testing set
7. Interpret the model results to the extent possible
8. Draw conclusions and write a well-documented report

In this notebook, we will concentrate on the last two steps, and try to peer into the black box of the model we built. We know is it accurate, as it can predict the Energy Star Score to within 9.1 points of the true value, but how exactly does it make predictions? We'll examine some ways to attempt to understand the gradient boosting machine and then draw conclusions (I'll save you from having to write the report and instead link to the report I completed for the company who gave me this assignment as a job screen.)

Imports

We will use a familiar stack of data science and machine learning libraries.

In [ ]:
```
%%capture
%pip install lime
```

In [ ]:
```
# 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
pd.set_option('display.max_columns', 60)

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

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

from IPython.core.pylabtools import figsize

# Seaborn for visualization
import seaborn as sns

sns.set(font_scale = 2)

# Imputing missing values
from sklearn.preprocessing import MinMaxScaler
from sklearn.impute import SimpleImputer

# Machine Learning Models
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import GradientBoostingRegressor

from sklearn import tree

# LIME for explaining predictions
import lime
import lime.lime_tabular
```

Read in Data

In [ ]:
```
# Read in data into dataframes
train_features = pd.read_csv('data/training_features.csv')
test_features = pd.read_csv('data/testing_features.csv')
train_labels = pd.read_csv('data/training_labels.csv')
test_labels = pd.read_csv('data/testing_labels.csv')
```

Recreate Final Model

In [ ]:
```
# Create an imputer object with a median filling strategy
#imputer = Imputer(strategy='median')
imputer = SimpleImputer(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,))
```

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

In [ ]:
```
# ‘lad’ (least absolute deviation)
# The 'loss' parameter of GradientBoostingRegressor must be a str among {'huber', 'absolute_error', 'quant
model = GradientBoostingRegressor(loss='absolute_error', max_depth=5, max_features=None,
                                  min_samples_leaf=6, min_samples_split=6,
                                  n_estimators=800, random_state=42)

model.fit(X, y)
```

Out[ ]:
```
▼                    GradientBoostingRegressor
GradientBoostingRegressor(loss='absolute_error', max_depth=5,
                          min_samples_leaf=6, min_samples_split=6,
                          n_estimators=800, random_state=42)
```

In [ ]:
```
#  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))
```

```
Final Model Performance on the test set: MAE = 9.0839
```

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

Feature Importances

One of the basic ways we can interpret an ensemble of decision trees is through what are known as the feature importances. These can be interpreted as the variables which are most predictive of the target. While the actual details of the feature importances are quite complex (here is a Stack Overflow question on the subject, we can use the relative values to compare the features and determine which are most relevant to our problem.

Extracting the feature importances from a trained ensemble of trees is quite easy in scikit-learn. We will store the feature importances in a dataframe to analyze and visualize them.

In [ ]:
```
# 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)
```

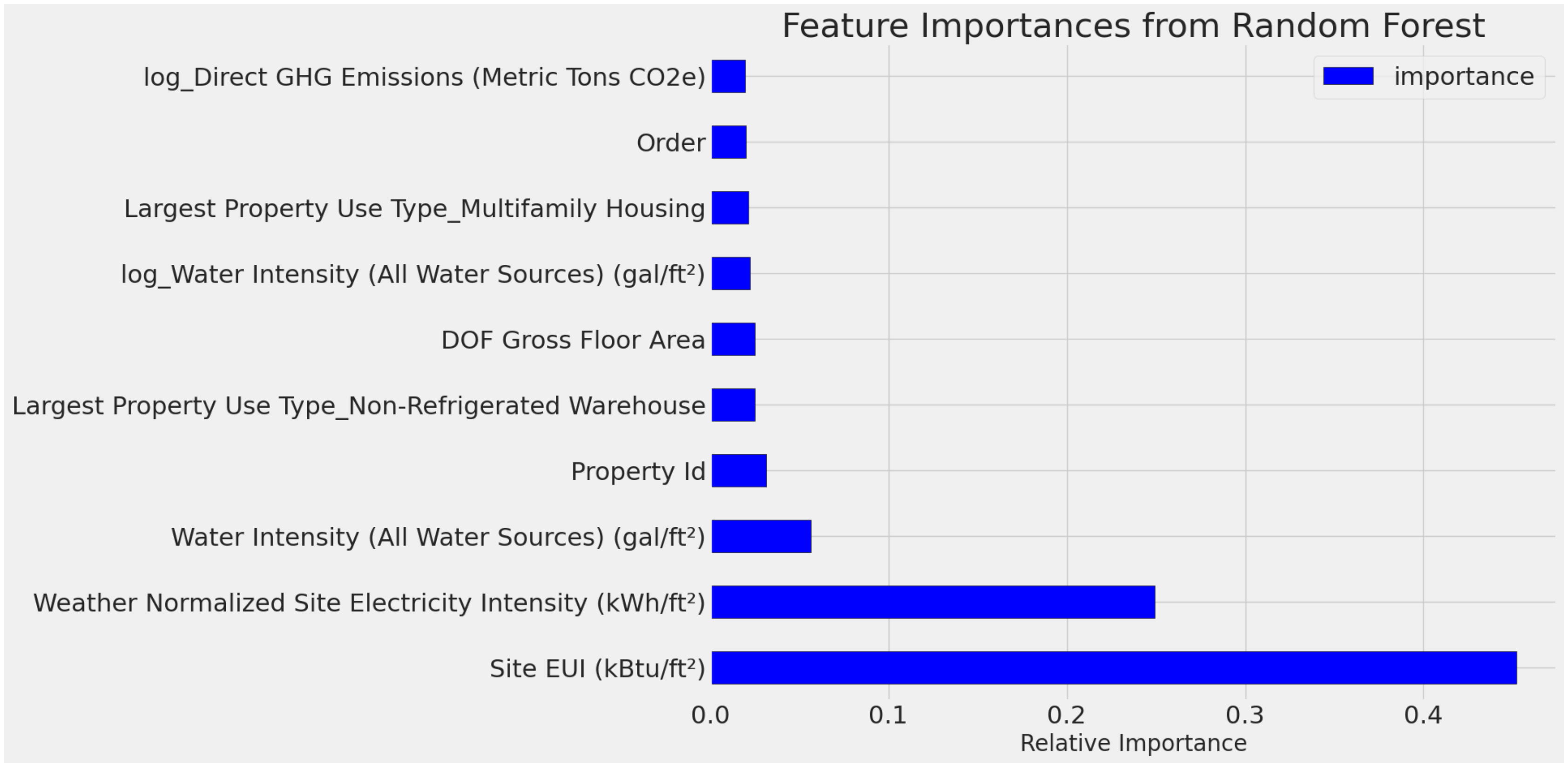
Out[ ]:

	feature	importance
0	Site EUI (kBtu/ft²)	0.452163
1	Weather Normalized Site Electricity Intensity ...	0.249107
2	Water Intensity (All Water Sources) (gal/ft²)	0.056662
3	Property Id	0.031396
4	Largest Property Use Type_Non-Refrigerated War...	0.025153
5	DOF Gross Floor Area	0.025003
6	log_Water Intensity (All Water Sources) (gal/ft²)	0.022335
7	Largest Property Use Type_Multifamily Housing	0.021462
8	Order	0.020169
9	log_Direct GHG Emissions (Metric Tons CO2e)	0.019410

The Site Energy Use Intensity, `Site EUI (kBtu/ft²)`, and the Weather Normalized Site Electricity Intensity, `Weather Normalized Site Electricity Intensity (kWh/ft²)` are the two most important features by quite a large margin. After that, the relative importance drops off considerably which indicates that we might not need to retain all of the features to create a model with nearly the same performance.

Let's graph the feature importances to compare visually.

In [ ]:
```python
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 = 'blue');
plt.xlabel('Relative Importance', size = 20); plt.ylabel('')
plt.title('Feature Importances from Random Forest', size = 30);
```



Use Feature Importances for Feature Selection

Given that not every feature is important for finding the score, what would happen if we used a simpler model, such as a linear regression, with the subset of most important features from the random forest? The linear regression did outperform the baseline, but it did not perform well compared to the model complex models. Let's try using only the 10 most important features in the linear regression to see if performance is improved. We can also limit to these features and re-evaluate the random forest.

In [ ]:
```
# Extract the names of the most important features
most_important_features = feature_results['feature'][:10]

# Find the index that corresponds to each feature name
indices = [list(train_features.columns).index(x) for x in most_important_features]

# Keep only the most important features
```

```
X_reduced = X[:, indices]
X_test_reduced = X_test[:, indices]

print('Most important training features shape: ', X_reduced.shape)
print('Most important testing  features shape: ', X_test_reduced.shape)
```

```
Most important training features shape:  (6622, 10)
Most important testing  features shape:  (2839, 10)
```

In [ ]:
```
lr = LinearRegression()

# Fit on full set of features
lr.fit(X, y)
lr_full_pred = lr.predict(X_test)

# Fit on reduced set of features
lr.fit(X_reduced, y)
lr_reduced_pred = lr.predict(X_test_reduced)

# Display results
print('Linear Regression Full Results: MAE =    %0.4f.' % mae(y_test, lr_full_pred))
print('Linear Regression Reduced Results: MAE = %0.4f.' % mae(y_test, lr_reduced_pred))
```

```
Linear Regression Full Results: MAE =    13.4651.
Linear Regression Reduced Results: MAE = 14.5095.
```

Well, reducing the features did not improve the linear regression results! It turns out that the extra information in the features with low importance do actually improve performance.

Let's look at using the reduced set of features in the gradient boosted regressor. How is the performance affected?

In [ ]:
```
# Create the model with the same hyperparamters
model_reduced = GradientBoostingRegressor(loss='absolute_error', max_depth=5, max_features=None,
                                  min_samples_leaf=6, min_samples_split=6,
                                  n_estimators=800, random_state=42)

# Fit and test on the reduced set of features
model_reduced.fit(X_reduced, y)
model_reduced_pred = model_reduced.predict(X_test_reduced)

print('Gradient Boosted Reduced Results: MAE = %0.4f' % mae(y_test, model_reduced_pred))
```

```
Gradient Boosted Reduced Results: MAE = 9.3735
```

The model results are slightly worse with the reduced set of features and we will keep all of the features for the final model. The desire to reduce the number of features is because we are always looking to build the most parsimonious model: that is, the simplest model with adequate performance. A model that uses fewer features will be faster to train and generally easier to interpret. In this case, keeping all of the features is not a major concern because the training time is not significant and we can still make interpretations with many features.

Locally Interpretable Model-agnostic Explanations

We will look at using LIME to explain individual predictions made the by the model. LIME is a relatively new effort aimed at showing how a machine learning model thinks by approximating the region around a prediction with a linear model.

We will look at trying to explain the predictions on an example the model gets very wrong and an example the model gets correct. We will restrict ourselves to using the reduced set of 10 features to aid interpretability. The model trained on the 10 most important features is slightly less accurate, but we generally have to trade off accuracy for interpretability!

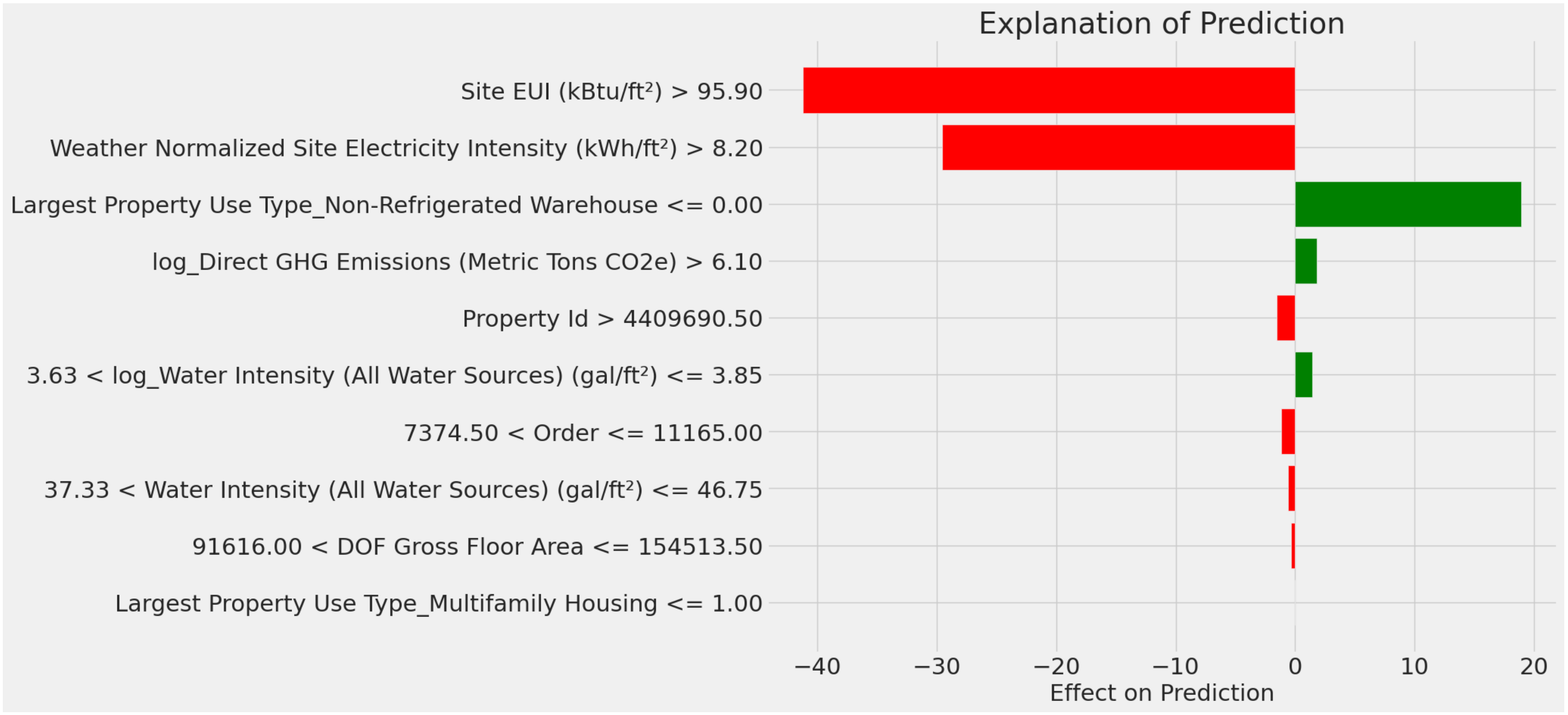
In [ ]:
```
# Find the residuals
residuals = abs(model_reduced_pred - y_test)

# Exact the worst and best prediction
wrong = X_test_reduced[np.argmax(residuals), :]
right = X_test_reduced[np.argmin(residuals), :]
```

In [ ]:
```
# Create a lime explainer object
explainer = lime.lime_tabular.LimeTabularExplainer(training_data = X_reduced,
                                                   mode = 'regression',
                                                   training_labels = y,
                                                   feature_names = list(most_important_features))
```

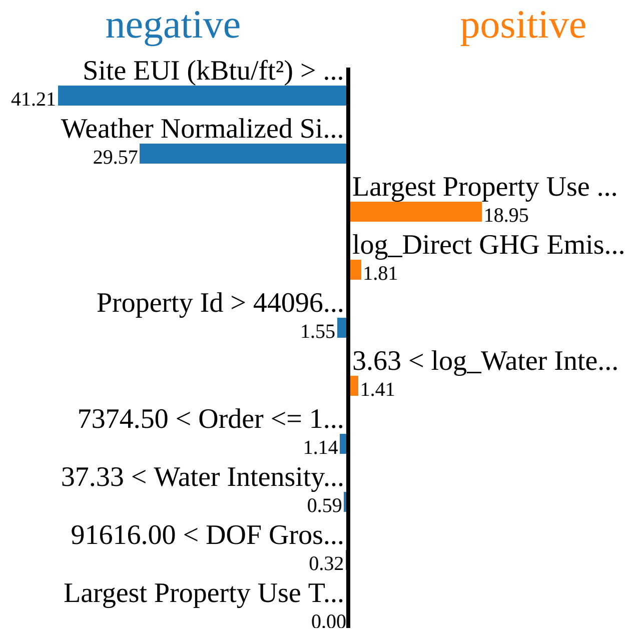
In [ ]:
```
# Display the predicted and true value for the wrong instance
print('Prediction: %0.4f' % model_reduced.predict(wrong.reshape(1, -1)))
print('Actual Value: %0.4f' % y_test[np.argmax(residuals)])

# Explanation for wrong prediction
wrong_exp = explainer.explain_instance(data_row = wrong,
                                       predict_fn = model_reduced.predict)
```

```
# Plot the prediction explaination
wrong_exp.as_pyplot_figure();
plt.title('Explanation of Prediction', size = 28);
plt.xlabel('Effect on Prediction', size = 22);
```

```
Prediction: 16.3665
Actual Value: 96.0000
```



```
In [ ]: wrong_exp.show_in_notebook(show_predicted_value=False)
```



Feature	Value
Site EUI (kBtu/ft²)	167.90
Weather Normalized Site Electricity Intensity (kWh/ft²)	21.50
Largest Property Use Type_Non-Refrigerated Warehouse	0.00
log_Direct GHG Emissions (Metric Tons CO2e)	6.79
Property Id	4508351.00
log_Water Intensity (All Water Sources) (gal/ft²)	3.85
Order	8298.00
Water Intensity (All Water Sources) (gal/ft²)	46.75
DOF Gross Floor Area	137189.00

In this example, our gradient boosted model predicted a score of 12.86 and the actual value was 100.

The plot from LIME is showing us the contribution to the final prediction from each of the features for the example. We can see that the Site EUI singificantly decreased the prediction because it was above 95.50. The Weather Normalized Site Electricity Intensity on the other hand, increased the prediction because it was lower than 3.80.

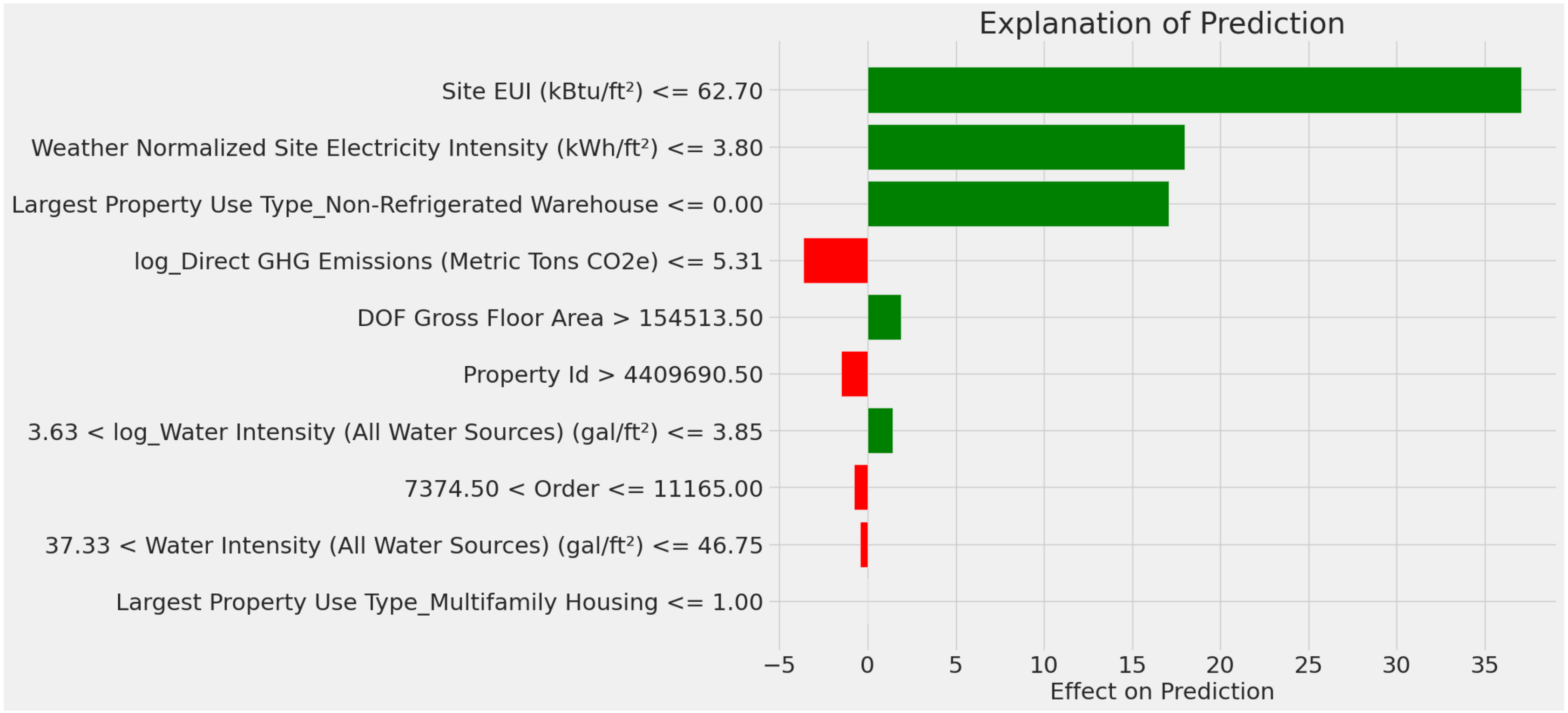
We can interpret this as saying that our model thought the Energy Star Score would be much lower than it actually was because the Site EUI was high. However, in this case, the score was 100 despite the high value of the EUI. While this significant mistake (off by 88 points!) might initially have been confusing, now we can see that in reality, the model was reasoning through the problem and just arrived at the incorrect value! A human going over the same process probably would have arrived at the same conclusion (if they had the patience to go through all the data).

Now we can go through the same process with a prediction the model got correct.

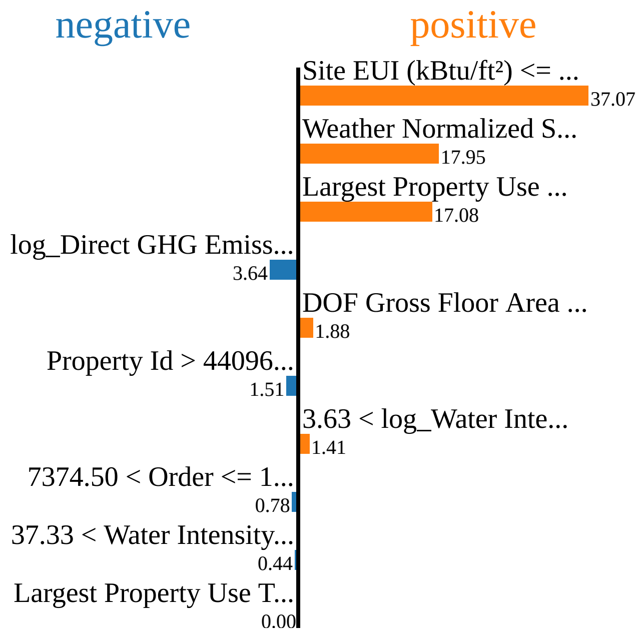
In [ ]:
```python
# Display the predicted and true value for the wrong instance
print('Prediction: %0.4f' % model_reduced.predict(right.reshape(1, -1)))
print('Actual Value: %0.4f' % y_test[np.argmin(residuals)])

# Explanation for wrong prediction
right_exp = explainer.explain_instance(right, model_reduced.predict, num_features=10)
```

```
right_exp.as_pyplot_figure();
plt.title('Explanation of Prediction', size = 28);
plt.xlabel('Effect on Prediction', size = 22);
```

```
Prediction: 100.0000
Actual Value: 100.0000
```



```
In [ ]: right_exp.show_in_notebook(show_predicted_value=False)
```



Feature	Value
Site EUI (kBtu/ft²)	12.30
Weather Normalized Site Electricity Intensity (kWh/ft²)	3.50
Largest Property Use Type_Non-Refrigerated Warehouse	0.00
log_Direct GHG Emissions (Metric Tons CO2e)	1.13
DOF Gross Floor Area	166044.00
Property Id	4951645.00
log_Water Intensity (All Water Sources) (gal/ft²)	3.85
Order	10094.00
Water Intensity (All Water Sources) (gal/ft²)	46.75

The correct value for this case was 100 which our gradient boosted model got right on!

The plot from LIME again shows the contribution to the prediciton of each of feature variables for the example. For instance, because the Site EUI was less than 62.70, that contributed significantly to a higher estimate of the score. Likewise, the year built being less than 1927 also positively contributed to the final prediction.

Observing break down plots like these allow us to get an idea of how the model makes a prediction. This is probably most valuable for cases where the model is off by a large amount as we can inspect the errors and perhaps engineer better features or adjust the hyperparameters of the model to improve predictions for next time. The examples where the model is off the most could also be interesting edge cases to look at manually. The model drastically underestimated the Energy Star Score for the first building because of the elevated Site EUI. We might therefore want to ask why the building has such a high Energy Star Score even though it has such a high EUI. A process such as this where we try to work with the machine learning algorithm to gain understanding of a problem seems much better than simply letting the model make predictions and completely trusting them! Although LIME is not perfect, it represents a step in the right direction towards explaining machine learning models.

Examining a Single Decision Tree

One of the coolest parts about a tree-based ensemble is that we can look at any individual estimator. Although our final model is composed of 800 decision trees, and looking at a single one is not indicative of the entire model, it still allows us to see the general idea of how a decision tree works. From there, it is a natural extension to imagine hundreds of these trees building off the mistakes of previous trees to make a final prediction (this is a significant oversimplification of how gradient boosting regression works!)

We will first extract a tree from the forest and then save it using `sklearn.tree.export_graphviz`. This saves the tree as a `.dot` file which can be converted to a png using command line instructions in the Notebook.

In [ ]:
```
# Extract a single tree
single_tree = model_reduced.estimators_[105][0]

tree.export_graphviz(single_tree, out_file = 'images/tree.dot',
                     rounded = True,
                     feature_names = most_important_features,
                     filled = True)

single_tree
```

Out[ ]:
```
▼                          DecisionTreeRegressor
DecisionTreeRegressor(criterion='friedman_mse', max_depth=5, min_samples_leaf=6,
                      min_samples_split=6,
                      random_state=RandomState(MT19937) at 0x7F6F91DCE040)
```

In [ ]:
```
# Convert to a png from the command line
# This requires the graphviz visualization library (https://www.graphviz.org/)


# !dot -Tpng images/tree.dot -o images/tree.png
```

image

That's one entire tree in our regressor of 800! It's a little difficult to make out because the maximum depth of the tree is 5. To improve the readability, we can limit the max depth in the call to export our tree.

In [ ]:
```
tree.export_graphviz(single_tree, out_file = 'images/tree_small.dot',
                     rounded = True, feature_names = most_important_features,
                      filled = True, max_depth = 3)
```

image

Now we can take a look at the tree and try to intrepret it's decisions! The best way to think of a decision tree is as a series of yes/no questions, like a flowchart. We start at the top, called the root, and move our way down the tree, with the direction of travel determined by the answer to each equation.

For instance, here the first question we ask is: is the `Site EUI` less than or equal to 15.95? If the answer is yes, then we move to the left and ask the question: is the `Weather Normalized Site Electricity Intensity` less than or equal to 3.85? If the answer to the first question was no, we move to the right and ask the question is the `Weather Normalized Site Electricity Intensity` less than or equal to 26.85?

We continue this iterative process until we reach the bottom of the tree and end up in a leaf node. Here, the value we predict corresponds to the value shown in the node (the values in this tree appear to be the actual predictions divided by 100).

Each node has four different pieces of information:

1. The question: based on this answer we move right or left to the next node a layer down in the tree
2. The friedman_mse: a measure of the error for all of the examples in a given node
3. The samples: number of examples in a node
4. The value: the prediction of the target for all examples in a node

We can see that as we increase the depth of the tree, we will be better able to fit the data. With a small tree, there will be many examples in each leaf node, and because the model estimates the same value for each example in a node, there will probably be a larger error (unless all of the examples have the same target value). Constructing too large of a tree though can lead to overfitting. We can control a number of hyperparameters that determine the depth of the tree and the number of examples in each leaf. We saw how to select a few of these hyperaparameters in the second part when we performed optimimation using cross validation.

Although we clearly cannot examine every tree in our model, looking at a single one does give us some idea how our model makes predictions. In fact, this flowchart based method seems much like how a human makes decisions, answering one question about a single value at a time. Decision tree based ensembles simply take the idea of a single decision tree and combine the predictions of many individuals in order to create a model with less variance than a single estimator. Ensembles of trees tend to be very accurate, and also are intuitive to explain!

Make Conclusions and Document Findings

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

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.

Presenting your Work

One of the best aspects about Jupyter Notebooks is that they can be directly downloaded as a pdf or html and then shared with others. This project was originally an "assignment" given to me by a company as an evaluation for a data science position (I was offered the job, but then the CTO of the company quit so they couldn't take anyone on for a while!). My final report can be seen here (it differs from this project a little because I did it in a compressed time frame and made some different modeling choices). After finishing up the Jupyter Notebook, I downloaded it as `.tex`, made a few small changes in texStudio, and then compiled it to a pdf. The final version looks a little better than the default Jupyter Notebook pdf and it's wortwhile to learn a little latex to create more professional reports!

With that in mind, I think it's time for us to wrap up this project! I had a great time writing these notebooks and doing the analysis, and I hope you enjoyed reading this and now feel more confident in implementing your own machine learning project. If you want, you can start by modifying this project and trying to beat my models! Remember, don't feel like you have to go it alone: there are plenty of amazing machine learning resources out there, and coming from someone who is entirely self-taught thanks to the great data science community, the best way to learn is to get involved and start putting your work out there!

Resources for data science and machine learning:

- Hands-On Machine Learning with Scikit-Learn and Tensorflow
  - Jupyter Notebooks for this book are available online for free!
- An Introduction to Statistical Learning
- Kaggle: The Home of Data Science and Machine Learning
- Datacamp: Good Beginner Tutorials for Practicing Coding with a Focus on Data Science
- Dataquest: Hands on Lessons for Data Science Programming

```
In [ ]:
```