Building Electricity Consumption Prediction Report

Madeleine Moghadasi

Feb 27, 2021

Introduction

In this notebook, I will go through a real-world dataset related to energy consumption. This is a public dataset download from US Energy Information Administration. Data provide the Residential Energy Consumption collected energy-related data for housing units occupied as a primary residence and the households that live in them. Data were collected from 12,083 households selected at random using a complex multistage, area-probability sample design.

Objective

- Build a model that predicts total electric consumption (KWH)

Machine Learning Workflow

Here are some general machine learning structures I intend to implement throughout this project.

1. Data cleaning and formatting
2. Exploratory data analysis
3. Feature engineering and selection
4. Establish a baseline and 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. Conclusion

I will use the standard data science and machine learning libraries:

- numpy, pandas, and scikit-learn
- matplotlib and seaborn for visualization.

In [66]:
```
# Pandas and numpy for data manipulation
import pandas as pd
import numpy as np

# visualization
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set(font_scale = 2)
```

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

from IPython.core.pylabtools import figsize


# 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 RandomForestRegressor, GradientBoostingRegr
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn import tree

# Hyperparameter tuning
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV

#Evaluating model
from sklearn.model_selection import train_test_split

import warnings
warnings.filterwarnings('ignore')
```

1. Data cleaning and formatting

In [67]:
```
# Read in data into a dataframe
data = pd.read_csv("data.csv")

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

Out[67]:

	DOEID	REGIONC	DIVISION	REPORTABLE_DOMAIN	TYPEHUQ	NWEIGHT	HDD65	CDD
0	1	2	4	12	2	2471.68	4742	10
1	2	4	10	26	2	8599.17	2662	1
2	3	1	1	1	5	8969.92	6233	5
3	4	2	3	7	2	18003.64	6034	6
4	5	1	1	1	3	5999.61	5388	7

5 rows × 940 columns

There is a separate csv file that contains details of each column. This is the target's definition. **(KWH)**:

- Total Site Electricity usage, in kilowatt-hours, 2009

Another thing we can see from the dataset is that there are 360 features, and the values in these features are just imputation flags for the rest. Our knowledge comes from documentation that states that all the imputation flag variables begin with the letter 'Z'. As a result, those features will be removed from our dataset.

In [68]:
```
#Iterate thorough columns to find columns names with the letter Z as the
Discarded_feature = [x for x in data.columns if x.startswith('Z')]
```

```
#remove imputation flag variables from dataset
data = data[[c for c in data.columns.values.tolist() if c not in Discard
data.shape
```

Out[68]: (12083, 581)

Data Type and missing values

The next step is to examine the data type. The following list shows the columns that contain categorical variables. Later, one-hot encoding method will be applied to these columns.

In [69]:
```
list(data.select_dtypes(include=['object']).columns)
```

Out[69]: ['METROMICRO', 'UR', 'NOCRCASH', 'NKRGALNC', 'IECC_Climate_Pub']

Missing Values

Documentation indicates that the missing values are coded as "-2" , so this will be changed to np.nan.

In [70]:
```
#replace -2 to not a number
data = data.replace({-2: np.nan})
```

Then I can see how many values are missing in each column. Exploratory Data Analysis is okay with missing values, however, machine-learning methods will require that they filled in. Below is a function which calculates how many values were missing and what percent of total values were missing for each column.

In [71]:
```
# Function to calculate missing values by column
def missing_values_table(df):
        # Total missing values
        mis_val = df.isnull().sum()

        # Percentage of missing values
        mis_val_percent = 100 * df.isnull().sum() / len(df)

        # Make a table with the results
        mis_val_table = pd.concat([mis_val, mis_val_percent], axis=1)

        # Rename the columns
        mis_val_table_ren_columns = mis_val_table.rename(
        columns = {0 : 'Missing Values', 1 : '% of Total Values'})

        # Sort the table by percentage of missing descending
        mis_val_table_ren_columns = mis_val_table_ren_columns[
            mis_val_table_ren_columns.iloc[:,1] != 0].sort_values(
        '% of Total Values', ascending=False).round(1)

        # Print some summary information
        print ("Your dataframe has " + str(df.shape[1]) + " columns.\n"
            "There are " + str(mis_val_table_ren_columns.shape[0]) +
              " columns that have missing values.")

        # Return the dataframe with missing information
        return mis_val_table_ren_columns
```

In [72]:
```
missing_values_table(data)
```

```
Your dataframe has 581 columns.
There are 348 columns that have missing values.
```

Out[72]:

	Missing Values	% of Total Values
AGEHHMEMCAT13	12079	100.0
AGEHHMEMCAT14	12079	100.0
AGEHHMEMCAT12	12077	100.0
AGEHHMEMCAT11	12072	99.9
AGEHHMEMCAT10	12064	99.8
...	...	...
TYPERFR1	19	0.2
REFRIGT1	19	0.2
SIZRFRI1	19	0.2
ICE	19	0.2
PELLIGHT	1	0.0

348 rows × 2 columns

Although we should not discard any information and be careful when removing columns, if the missing values percentage is high, a column is not likely to be useful. We are going to remove columns with more than half of the values missing for this project. All remaining values have to be filled in using an appropriate strategy before applying machine learning.

In [73]:
```
# Get the columns with > 50% missing
missing_df = missing_values_table(data);
missing_columns = list(missing_df[missing_df['% of Total Values'] > 50].
print('We will remove %d columns.' % len(missing_columns))
```

```
Your dataframe has 581 columns.
There are 348 columns that have missing values.
We will remove 210 columns.
```

In [74]:
```
# Drop the columns
data = data.drop(columns = list(missing_columns))
```

In [75]:
```
data.shape
```

Out[75]:
```
(12083, 371)
```

2. Exploratory Data Analysis

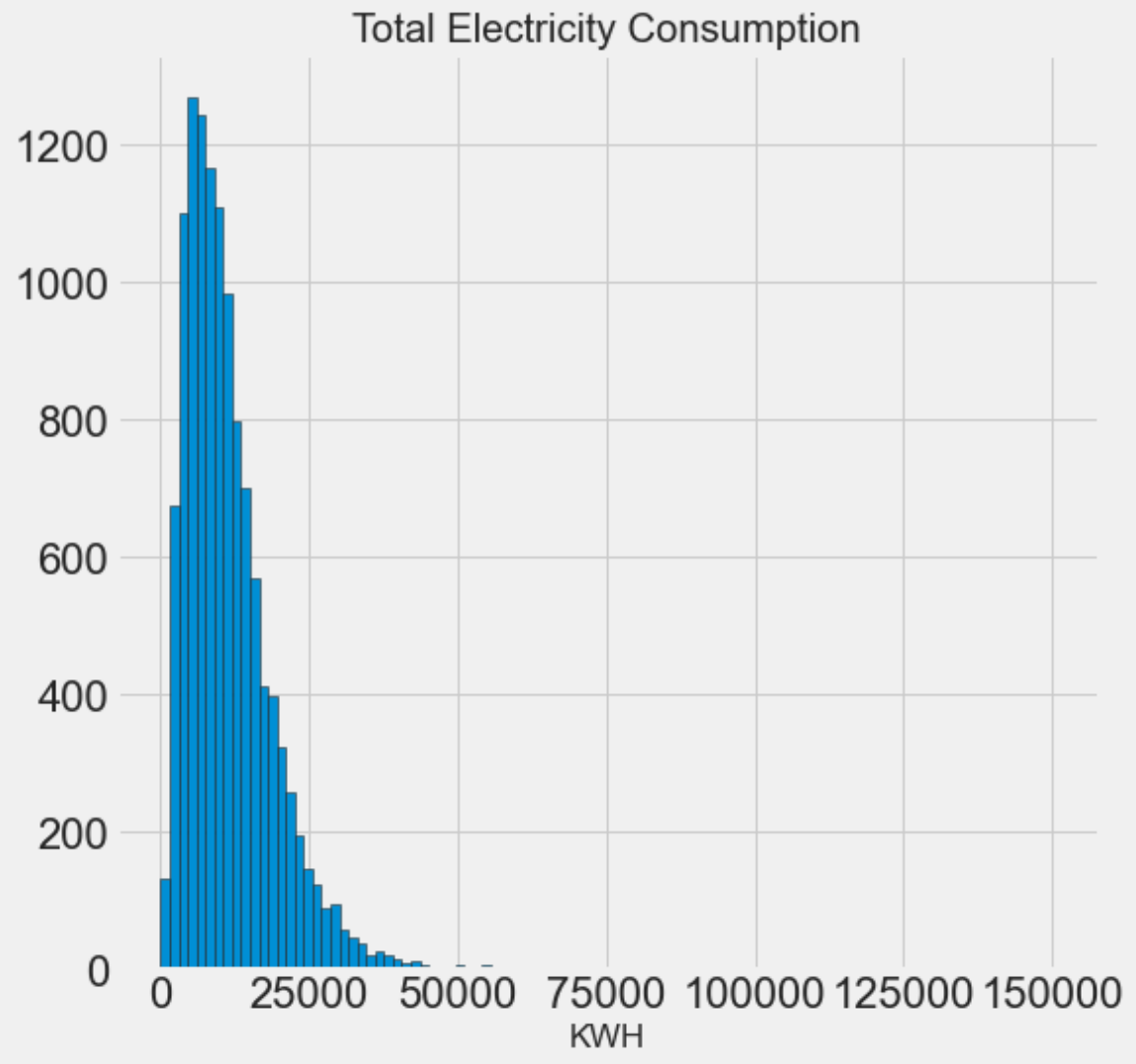
We look for anomalies, patterns, trends, or relationships. Starting with one variable, we will look at total electricity consumption (KWH).

Single Variable Plots

The data should be approximately normally distributed of course with some outliers at the high and low ends.

In [76]:
```
figsize(8,8)
```

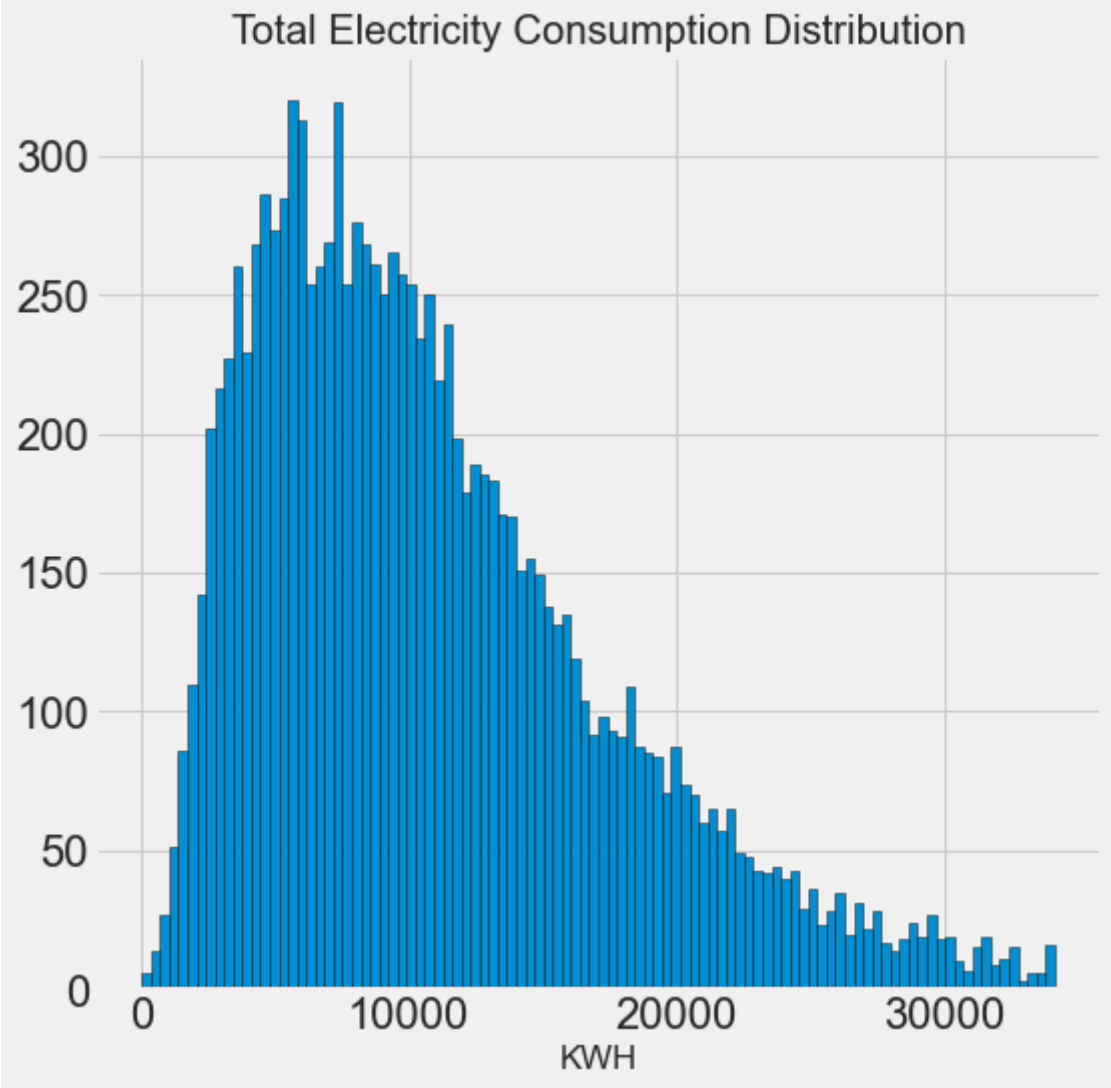
```
plt.style.use('fivethirtyeight')
plt.hist(data["KWH"], bins=100, edgecolor = 'k')
plt.xlabel('KWH');
plt.title('Total Electricity Consumption');
```



In this case, the graph is clearly skewed . A normal distribution is achieved only when the outliers are removed.

In [77]:
```
# Remove outliers that are more than 3 standard deviations from the mean
data = data[np.abs(data["KWH"]-data["KWH"].mean())<=(3.*data["KWH"].std(
```

In [78]:
```
plt.figure(figsize=(8,8))
plt.style.use('fivethirtyeight')
plt.hist(data["KWH"], bins=100, edgecolor = 'k')
plt.xlabel('KWH');
plt.title('Total Electricity Consumption Distribution');
```



It is close to being normally distributed.

Looking for Relationships

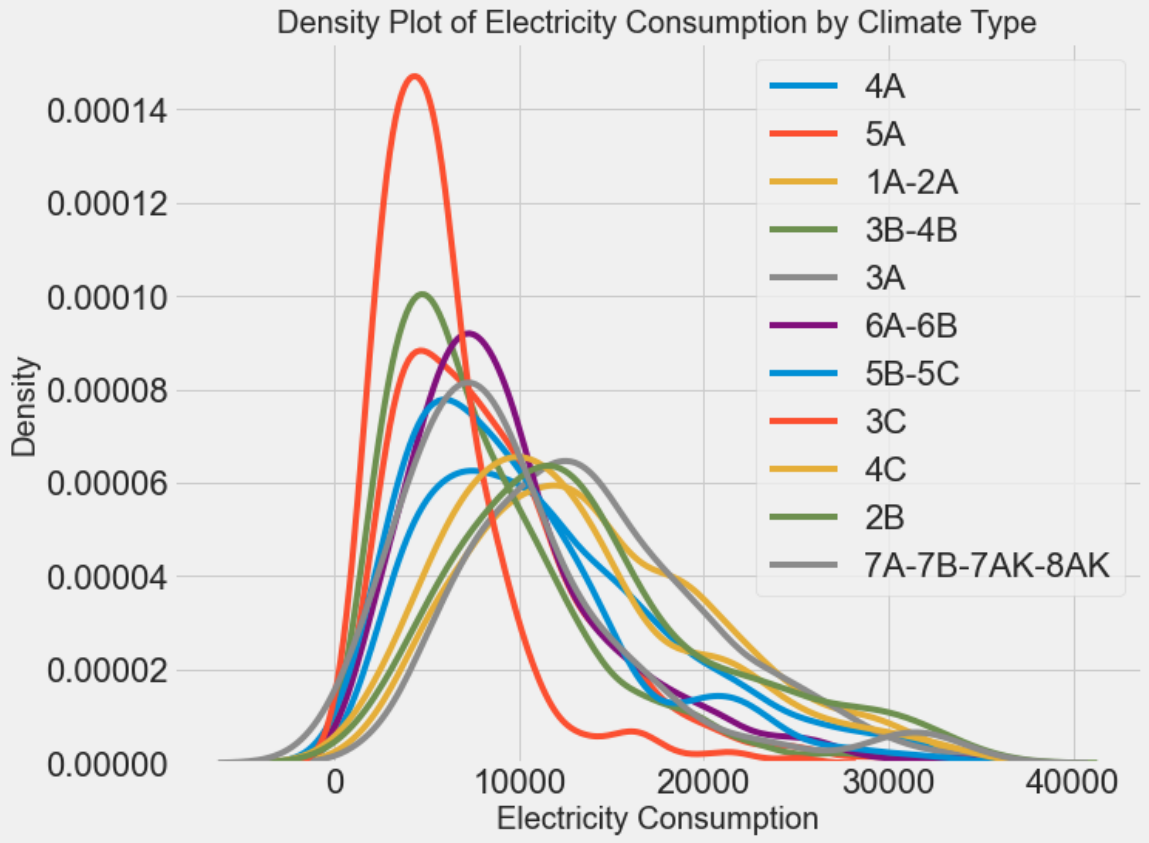
We can use a density plot to examine the effects of categorical variables on the Total Electricity Consumption. The color of the density curves based on a categorical variable allows us to visualize how the distribution varies by class.

The first plot we will make shows the distribution of Total Electricity Consumption by the IECC_Climate_Pub(International Energy Conservation Code climate zone).

In [79]:
```
# Create a list of climate
types = data.dropna(subset=['KWH'] )
types = types['IECC_Climate_Pub'].value_counts()
types = list(types.index)
```

In [80]:
```
# Plot of distribution of total electricity for climate categories
plt.figure(figsize=(10,8))

# Plot each climate
for c_type in types:
    # Select the climate type
    subset = data[data['IECC_Climate_Pub'] == c_type]

    # Density plot of total energy
```

```
    sns.kdeplot(subset['KWH'].dropna(),
                label = c_type, shade = False, alpha = 0.8);

# label the plot
plt.xlabel('Electricity Consumption', size = 20); plt.ylabel('Density',
plt.title('Density Plot of Electricity Consumption by Climate Type', siz
plt.legend();
```



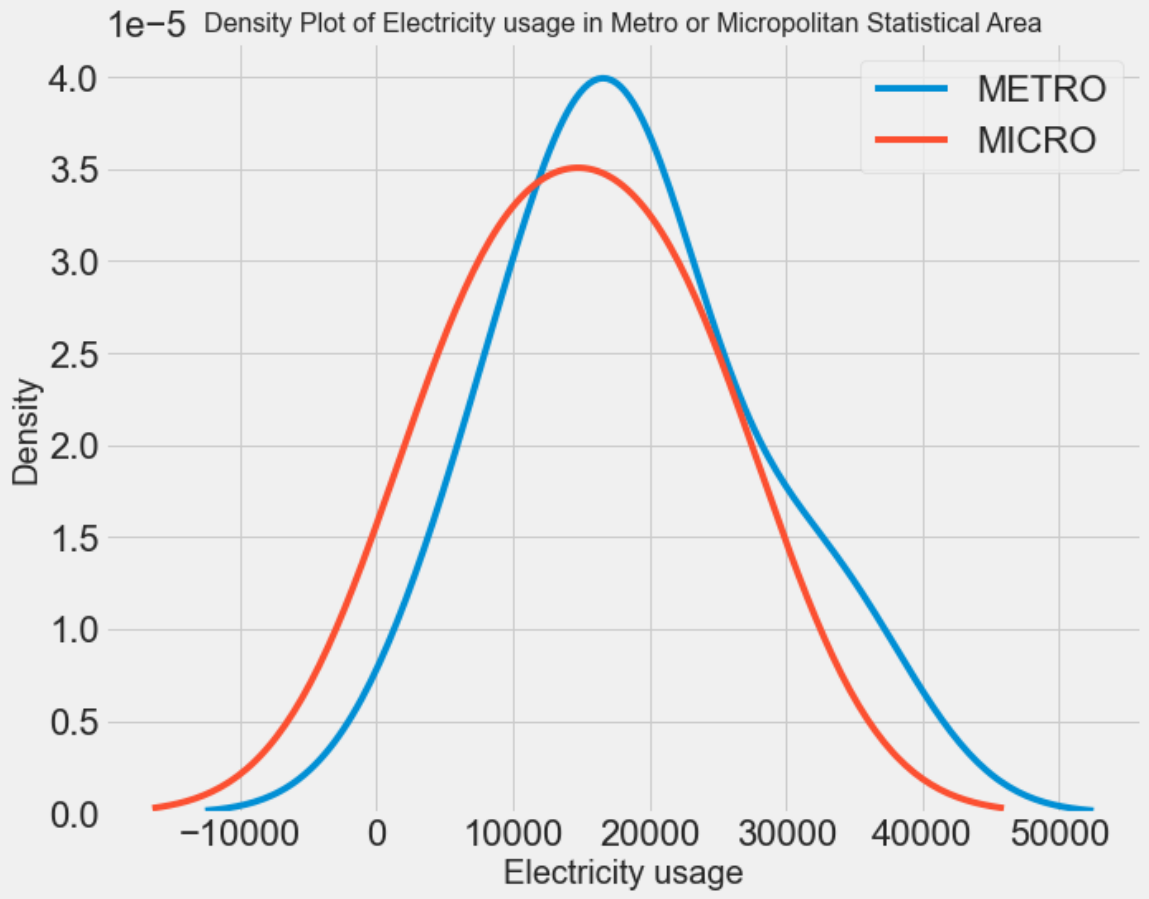
It is clear from the graph that the climate zone does influence electricity consumption. Using this graph, it can be concluded that the climate zone information can be useful in determining electricity consumption. We need to encode the climate zone as a categorical variable in order to use it in a machine learning model.

The same graph can also be made with another categorical variable, but this time we will color it according to metropolitan area.

In [81]:
```
# Create a list of area
areas = data.dropna(subset=['KWH'])
areas= areas['METROMICRO'].value_counts()
areas = list(areas.index)
```

In [82]:
```
# Plot of distribution of total energy for area
plt.figure(figsize=(10,8))

# Plot each borough distribution of total energy
for area in areas:
    # Select the area type
    subset = data[data['METROMICRO'] == area].dropna()

    # Density plot of total energy
    sns.kdeplot(subset['KWH'].dropna(),
                label = area);

# label the plot
```

```
plt.xlabel('Electricity usage', size = 20); plt.ylabel('Density', size =
plt.title('Density Plot of Electricity usage in Metro or Micropolitan St
plt.legend();
```



Electricity consumption distribution does appear to differ somewhat depending on the metropolitan area.Including the metropolitan area as a categorical variable might make sense.

Correlations between Features and Target

Although there can be non-linear relationships between the features and targets, and correlation coefficients do not account for interactions between features, linear relationships are a good way to start exploring trends in the data. These values can then be used to elect which features to include in the model.

This code calculates the correlation coefficient for all variables and electricity.

In [83]:
```
# Find all correlations and sort
correlations_data = data.corr()['KWH'].sort_values()

# Print the most negative correlations
print(correlations_data.head(15), '\n')

# Print the most positive correlations
print(correlations_data.tail(15))
```

```
TYPEHUQ     -0.377784
USENG       -0.323616
UGWATER     -0.316883
WHEATOTH    -0.299609
KOWNRENT    -0.271465
UGCOOK      -0.248316
```

```
UGWARM      -0.245893
HEATOTH     -0.245282
COOLTYPE    -0.230336
GND_HDD65   -0.196057
WSF         -0.195313
PGASHTWA    -0.193854
PGASHEAT    -0.172820
KAVALNG     -0.159496
PERIODNG    -0.157199
Name: KWH, dtype: float64

BTUELRFG       0.593586
KWHRFG         0.593586
TOTALBTURFG    0.593586
TOTALDOLOTH    0.604789
TOTALBTUOTH    0.613930
KWHCOL         0.617897
BTUELCOL       0.617897
TOTALBTUCOL    0.617897
DOLELOTH       0.663862
DOLLAREL       0.853961
KWHOTH         0.856465
BTUELOTH       0.856465
BTUEL          1.000000
KWH            1.000000
USEEL               NaN
Name: KWH, dtype: float64
```

There are several strong negative correlations between the features and the target. The most negative correlations with the target are the type of housing unit, if natural gas is used and whether natural gas used for water heating. Intuitively, these correlations make sense: as the natural gas consumption increases, the electricity usage tends to decrease.

To analyze the response of the features to non-linear relationships, we can take square root and natural log transformations of the elements and calculate their correlation coefficients. We should encode all categorical variables at once to capture all the relationships. In the following code, we take log and square root transformations of the numerical variables, one-hot encode the categorical variables, calculate the correlations between all of the features and the target, and display the top 15 most positive and top 15 most negative correlations.

In [97]:
```
# Select the numeric columns
numeric_subset = data.select_dtypes('number')

# Create columns with square root and log of numeric columns
for col in numeric_subset.columns:
    # Skip the total energy column
    if col == 'KWH':
        next
    else:
        numeric_subset['sqrt_' + col] = np.sqrt(numeric_subset[col])
        numeric_subset['log_' + col] = np.log(numeric_subset[col])

# Select the categorical columns
categorical_subset = data[['METROMICRO', 'UR', "IECC_Climate_Pub"]]

# One hot encode
categorical_subset = pd.get_dummies(categorical_subset)

# Join the two dataframes using concat
# Make sure to use axis = 1 to perform a column bind
features = pd.concat([numeric_subset, categorical_subset], axis = 1)
```

```
# Drop buildings without an energy
features = features.dropna(subset = ['KWH'])

# Find correlations with the energy
correlations = features.corr()['KWH'].dropna().sort_values()
```

In [85]:
```
# Display most negative correlations
correlations.head(15)
```

Out[85]:
```
TYPEHUQ            -0.377784
sqrt_TYPEHUQ       -0.375196
log_TYPEHUQ        -0.366051
USENG              -0.323616
sqrt_USENG         -0.323616
UGWATER            -0.316883
sqrt_UGWATER       -0.316883
sqrt_WHEATOTH      -0.299609
WHEATOTH           -0.299609
log_DOLKEROTH      -0.283594
log_KOWNRENT       -0.278540
sqrt_KOWNRENT      -0.275629
KOWNRENT           -0.271465
UR_U               -0.264617
log_GALLONKERSPH   -0.248734
Name: KWH, dtype: float64
```

In [86]:
```
# Display most positive correlations
correlations.tail(15)
```

Out[86]:
```
log_KWHWTH        0.729688
log_BTUELWTH      0.729694
log_KWHOTH        0.789134
log_BTUELOTH      0.789135
log_DOLLAREL      0.832307
sqrt_KWHOTH       0.848192
sqrt_BTUELOTH     0.848192
DOLLAREL          0.853961
KWHOTH            0.856465
BTUELOTH          0.856465
sqrt_DOLLAREL     0.865426
log_BTUEL         0.918163
sqrt_BTUEL        0.983059
BTUEL             1.000000
KWH               1.000000
Name: KWH, dtype: float64
```

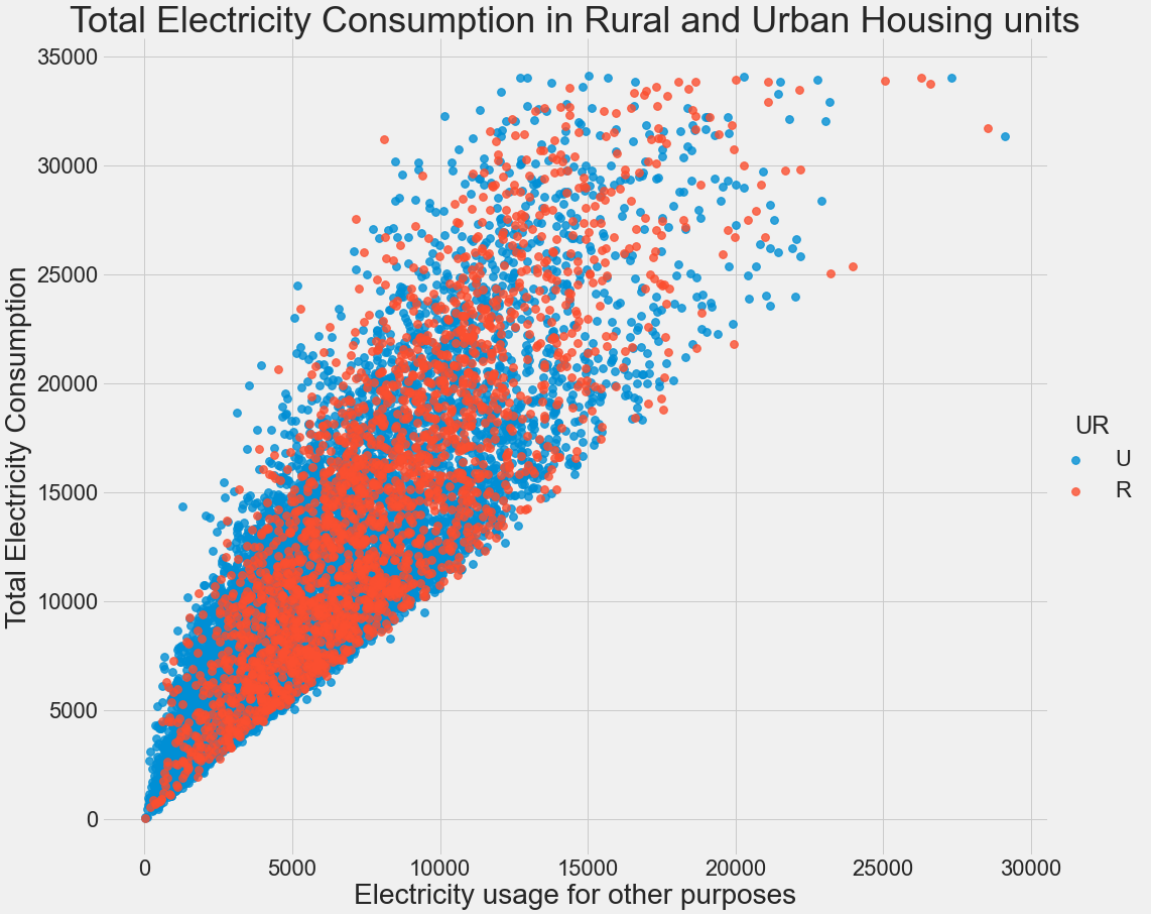
After transforming the features, the strongest relationships are still those related to type of housing unit and natural gas usage . The log and square root transformations do not seem to have resulted in any stronger relationships.

Two-Variable Plots

We are graphing one of the most significant correlations in the dataset which is Electricity usage for other purposes (KWHOTH). We can also incorporate extra variables (Housing units classified as urban or rural) using aspects such as color of the markers color to show how that affects the relationship.

In [87]:
```
figsize(12, 10)

# Extract the area types
features['UR'] = data.dropna(subset = ['KWH'])['UR']
```

```
# Use seaborn to plot a scatterplot of KWH vs urban and rural housing
sns.lmplot('KWHOTH', 'KWH',
           hue = 'UR', data = features,
           scatter_kws = {'alpha': 0.8, 's': 60}, fit_reg = False,
           size = 12, aspect = 1.2);

# Plot labeling
plt.xlabel("Electricity usage for other purposes", size = 28)
plt.ylabel('Total Electricity Consumption', size = 28)
plt.title('Total Electricity Consumption in Rural and Urban Housing unit
```



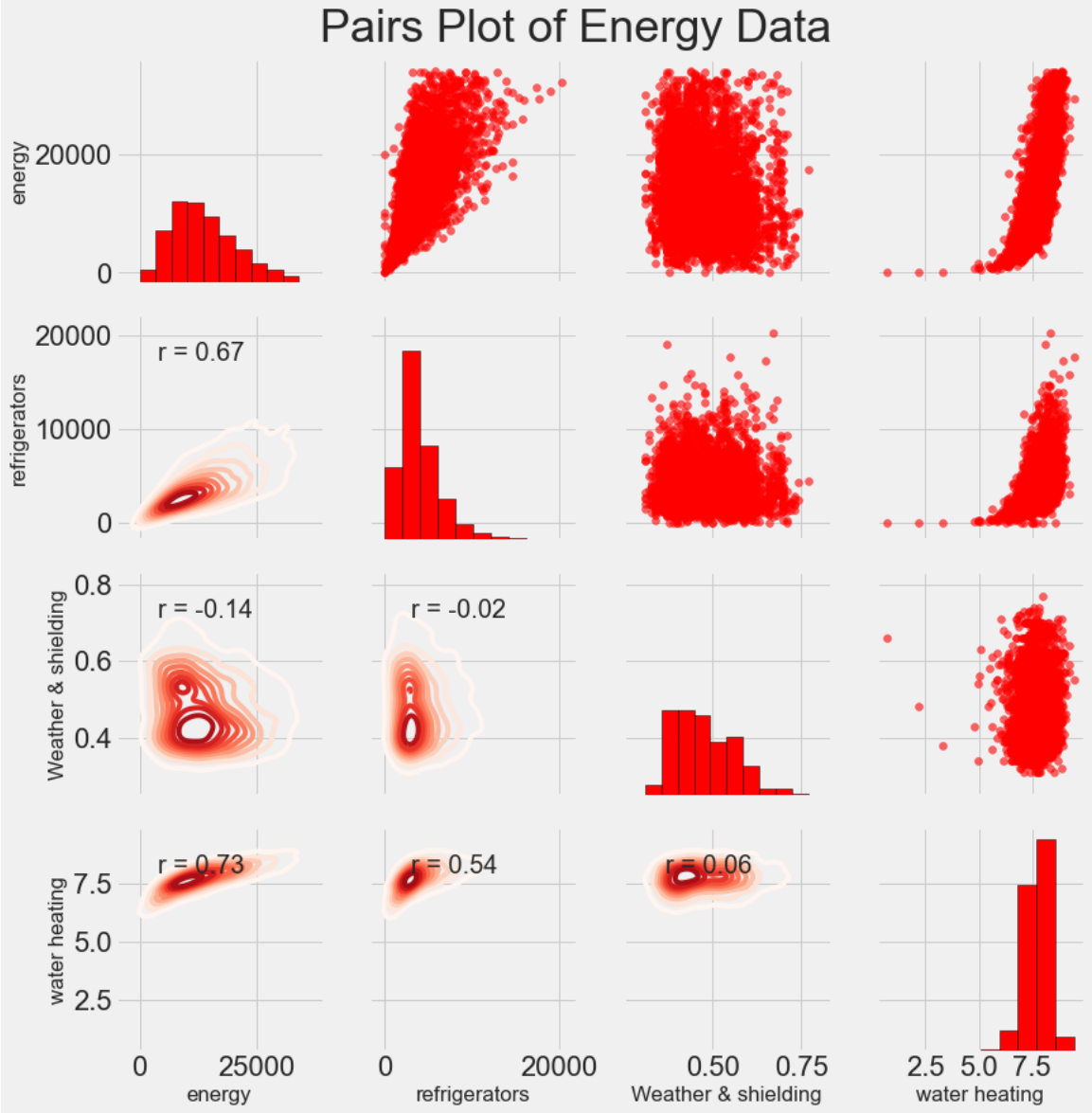
Electricity usage for other purposes (KWHOTH) has a clear relationship with total consumption of electric power. There is a linear relationship between the two variables and this feature seems appropriate for predicting the energy consumption of a building. I observe no difference in consumption between urban and rural areas.

Pairs Plot

As a final plot for exploratory data analysis, we can create pairs plot between several variables. The Pairs Plot shows scatterplots between pairs of variables and histograms of individual variables on the diagonal.

In [88]:
```
# Extract the columns to  plot
plot_data = features[['KWH', 'BTUELRFG',
                      'WSF',
                      'log_KWHWTH']]

# Replace the inf with nan
plot_data = plot_data.replace({np.inf: np.nan, -np.inf: np.nan})
```

```
# Rename columns
plot_data = plot_data.rename(columns = {'KWH': 'energy',
                                        'BTUELRFG': 'refrigerators',
                                        'WSF': 'Weather & shielding',
                                        'log_KWHWTH':'water heating'})

# Drop na values
plot_data = plot_data.dropna()

# Function to calculate correlation coefficient between two columns
def corr_func(x, y, **kwargs):
    r = np.corrcoef(x, y)[0][1]
    ax = plt.gca()
    ax.annotate("r = {:.2f}".format(r),
                xy=(.2, .8), xycoords=ax.transAxes,
                size = 20)

# Create the pairgrid object
grid = sns.PairGrid(data = plot_data, size = 3)

# Upper is a scatter plot
grid.map_upper(plt.scatter, color = 'red', alpha = 0.6)

# Diagonal is a histogram
grid.map_diag(plt.hist, color = 'red', edgecolor = 'black')

# Bottom is correlation and density plot
grid.map_lower(corr_func);
grid.map_lower(sns.kdeplot, cmap = plt.cm.Reds)

# Title for entire plot
plt.suptitle('Pairs Plot of Energy Data', size = 36, y = 1.02);
```



To interpret the relationships in the plot, for example, to find the relationship between total electricity consumption and the weather and shielding, we look at the energy column and find the weather and shielding. At the intersection we see that the energy has a 0.06 correlation coefficient with this varible.

3. Feature Engineering and Selection

Remove Collinear Features

Highly collinear features have a significant correlation coefficient between them. Although variables in a dataset are normally correlated to a small degree, highly correlated variables may be redundant because we only need one of their features to get the necessary information for our model. We can reduce model complexity by removing collinear features and increase model generalization by reducing the number of features in the model. In the following code, we eliminate collinear features by removing one of the two features associated with a given correlation coefficient, based on the threshold we set for the correlation coefficients. The correlation coefficient between two features should be

at least 0.6 and one of the features will be removed if the correlation coefficient exceeds this value.

In [98]:
```
def remove_collinear_features(x, threshold):
    '''
        Remove collinear features in a dataframe with a correlation coef
        greater than the threshold.
    '''

    # Dont want to remove correlations between Total Electricity Consump
    y = x['KWH']
    x = x.drop(columns = ['KWH'])

    # 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:
                drop_cols.append(col.values[0])

    # Drop one of each pair of correlated columns
    drops = set(drop_cols)
    x = x.drop(columns = drops)


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

    return x
```

In [99]:
```
new_features = remove_collinear_features(features, 0.6);
```

In [100…
```
# Remove any columns with all na values
new_features  = new_features.dropna(axis=1, how = 'all')
print('new data shape:', new_features.shape)
print('old data shape:',features.shape)
```

```
new data shape: (11923, 388)
old data shape: (11923, 1118)
```

Our final dataset now has 388 features (one of the columns is the target). Besides encoded categorical variables, we had log and sqrt transformation of variables. While a large number of features may be problematic for linear regression, models such as the random forest perform implicit feature selection and automatically determine which features are important during training.

Split into Training and Testing Sets

We will first extract all the data without a total electricity consumption column (KWH). The training set will include 80% of the households in the training set, and 20% of the households in the test set.

In [101…
```
# Separate out the features and targets
targets = pd.DataFrame(new_features['KWH'])
new_features = new_features.drop(columns='KWH')
```

In [102…
```
# Replace the inf and -inf with nan (required for later imputation)
new_features = new_features.replace({np.inf: np.nan, -np.inf: np.nan})

# Split into 80% training and 20% testing set
X, X_test, y, y_test = train_test_split(new_features, targets, test_size

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

```
(9538, 387)
(2385, 387)
(9538, 1)
(2385, 1)
```

We have 9538 household unit in the training set, and 2385 household in the testing set.

4. Establish Baseline and Comparing Machine Learning Models

Mean Absolute Error

We use mean absolute error to show the average amount our estimate deviates from the target value in the same units.

The function below calculates the mean absolute error between true values and predictions.

In [103…
```
# Function to calculate mean absolute error
def mae(y_true, y_pred):
    return np.mean(abs(y_true - y_pred))
```

Establish A Regression Baseline

Before we implement a machine learning model, we need to calculate a baseline. If our model cannot beat this mark, then machine learning may not be appropriate for the task. For regression tasks, a simple baseline is to predict the mean value of the target in the training data for all the testing examples.

In [104…
```
baseline_guess = np.median(y)

print("Baseline Performance on the test set: MAE = %0.4f" % mae(y_test, 
```

```
Baseline Performance on the test set: MAE = 5151.3187
```

This shows our average estimate on the test set is off by about 5151 . A model that cannot

exceed an average error of 5151 is not fit for our purpose.

Evaluating and Comparing ML Models

Here, we will train, evaluate, and build several machine learning methods for our supervised regression task. Our goal is to determine which model has the greatest potential for further development. The mean absolute error is used to compare the models.

Imputing Missing Values

Because we have already removed features with over 50% missing data in the first part, we will now fill in the remaining missing values. Replacing missing values with the column median is a relatively straightforward way to make substitutions.

In [105…
```
# Create an imputer object with a median filling strategy
imputer = SimpleImputer(strategy='median')

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

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

In [106…
```
# Make sure all values are finite
print(np.where(~np.isfinite(X)))
print(np.where(~np.isfinite(X_test)))
```

```
(array([], dtype=int64), array([], dtype=int64))
(array([], dtype=int64), array([], dtype=int64))
```

Scaling Features

We need to normalize the features because they are in different units, and we do not want the units to affect our algorithm.

As with imputation, when we train the scaling object, we want to use only the training set. When we transform features, we will transform both the training set and the testing set.

In [107…
```
# Create the scaler object with a range of 0-1
scaler = MinMaxScaler(feature_range=(0, 1))

# 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)
```

In [108…
```
# Convert y to one-dimensional array (vector)
y = np.array(y).reshape((-1, ))
y_test = np.array(y_test).reshape((-1, ))
```

Models to Evaluate

The following are five different machine learning models we will compare with the Scikit-Learn library:

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

In order to compare the models, we will be mostly using the Scikit-Learn default values for the model hyperparameters. We are first going to determine the baseline performance of each model, and then we can select the best performing model for further optimization using hyperparameter tuning.

In [109…
```
# 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 tes
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
```

Linear Regression

A **linear regression** is extremely straightforward. This is a good machine learning method to get started with since the results can easily be interpreted. Our model, however, will not be useful if the problem is nonlinear.

In [110…
```
lr = LinearRegression()
lr_mae = fit_and_evaluate(lr)

print('Linear Regression Performance on the test set: MAE = %0.4f' % lr_
```

```
Linear Regression Performance on the test set: MAE = 685918864009.6643
```

Those aren't very good statistics! Later, we will move to more sophisticated machine learning models.

Support Vector Machine

In [111…
```
svm = SVR(C = 1000, gamma = 0.1)
svm_mae = fit_and_evaluate(svm)

print('Support Vector Machine Regression Performance on the test set: MA
```

```
Support Vector Machine Regression Performance on the test set: MAE = 314
3.7950
```

Random Forest

In [112…
```
random_forest = RandomForestRegressor(random_state=60)
random_forest_mae = fit_and_evaluate(random_forest)

print('Random Forest Regression Performance on the test set: MAE = %0.4f
```

```
Random Forest Regression Performance on the test set: MAE = 1265.4577
```

Gradient Boosted

In [113…
```
gradient_boosted = GradientBoostingRegressor(random_state=60)
gradient_boosted_mae = fit_and_evaluate(gradient_boosted)

print('Gradient Boosted Regression Performance on the test set: MAE = %0
```

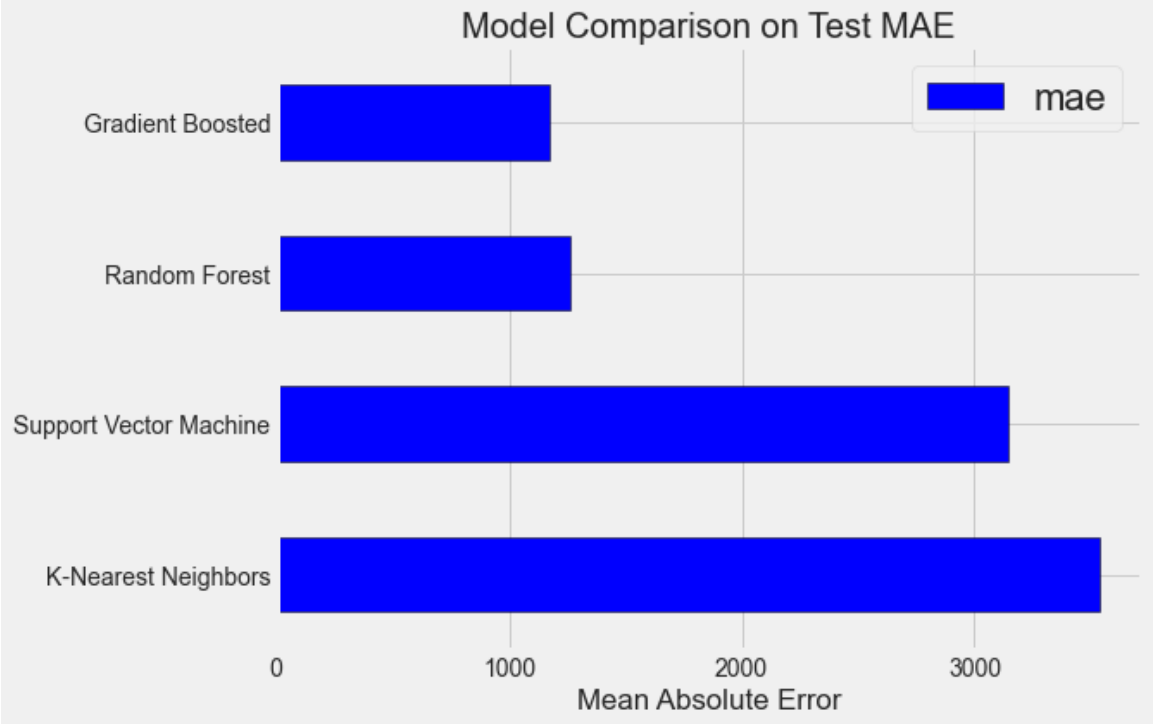
```
Gradient Boosted Regression Performance on the test set: MAE = 1172.6064
```

K-Nearest Neighbors

In [114…
```
knn = KNeighborsRegressor(n_neighbors=10)
knn_mae = fit_and_evaluate(knn)

print('K-Nearest Neighbors Regression Performance on the test set: MAE =
```

```
K-Nearest Neighbors Regression Performance on the test set: MAE = 3537.21
49
```

In [115…
```
plt.style.use('fivethirtyeight')
figsize(8, 6)

# Dataframe to hold the results
model_comparison = pd.DataFrame({'model': ['Support Vector Machine',
                                           'Random Forest', 'Gradient Bo
                                            'K-Nearest Neighbors'],
                                 'mae': [svm_mae, random_forest_mae,
                                         gradient_boosted_mae, knn_mae]}

# Horizontal bar chart of test mae
model_comparison.sort_values('mae', ascending = False).plot(x = 'model',
                                                            color = 'blue

# Plot formatting
plt.ylabel(''); plt.yticks(size = 14); plt.xlabel('Mean Absolute Error')
plt.title('Model Comparison on Test MAE', size = 20);
```



The best performing regression model is the gradient boosting, followed by random forest. The random forest and gradient boosting methods are good for getting started since they are less dependent on the model settings. It was obviously shown that machine learning is applicable as most models outperformed the baseline significantly!

Next, I will optimize the best model using hyperparameter tuning. Given the results here, I will concentrate on using the GradientBoostingRegressor.

5. Perform hyperparameter tuning on the best model to optimize it for the problem

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

- in this method, we choose hyperparameters to evaluate: we define a range of options, and then randomly select combinations to try. 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.

In [116…

```
# Loss function to be optimized
loss = ['ls', 'lad', 'huber']

# Number of trees used in the boosting process
n_estimators = [100, 500, 900, 1100, 1500]

# Maximum depth of each tree
max_depth = [2, 3, 5, 10, 15]

# Minimum number of samples per leaf
min_samples_leaf = [1, 2, 4, 6, 8]

# Minimum number of samples to split a node
```

```
min_samples_split = [2, 4, 6, 10]

# Maximum number of features to consider for making splits
max_features = ['auto', 'sqrt', 'log2', None]

# Define the grid of hyperparameters to search
hyperparameter_grid = {'loss': loss,
                       'n_estimators': n_estimators,
                       'max_depth': max_depth,
                       'min_samples_leaf': min_samples_leaf,
                       'min_samples_split': min_samples_split,
                       'max_features': max_features}
```

Our gradient-boosting regressor was tuned using six different hyperparameters. It is hard to say which of these variables will have the greatest impact on a model ahead of time, so the best way to figure out a combination is to test. Our objective is to find the best combination of hyperparameters. In the code below, we build the Randomized Search Object.

In [57]:
```
# Create the model to use for hyperparameter tuning
model = GradientBoostingRegressor(random_state = 42)

# Set up the random search with 4-fold cross validation
random_cv = RandomizedSearchCV(estimator=model,
                               param_distributions=hyperparameter_grid,
                               cv=4, n_iter=25,
                               scoring = 'neg_mean_absolute_error',
                               n_jobs = -1, verbose = 1,
                               return_train_score = True,
                               random_state=42)
```

In [58]:
```
# Fit on the training data
random_cv.fit(X, y)
```

```
Fitting 4 folds for each of 25 candidates, totalling 100 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent worker
s.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed: 10.4min
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed: 30.0min finished
```

Out[58]:
```
RandomizedSearchCV(cv=4, estimator=GradientBoostingRegressor(random_state
=42),
                   n_iter=25, n_jobs=-1,
                   param_distributions={'loss': ['ls', 'lad', 'huber'],
                                        'max_depth': [2, 3, 5, 10, 15],
                                        'max_features': ['auto', 'sqrt',
'log2',
                                                         None],
                                        'min_samples_leaf': [1, 2, 4, 6,
8],
                                        'min_samples_split': [2, 4, 6, 1
0],
                                        'n_estimators': [100, 500, 900, 1
100,
                                                         1500]},
                   random_state=42, return_train_score=True,
                   scoring='neg_mean_absolute_error', verbose=1)
```

The best gradient boosted model has the following hyperparameters:

In [117…
```
random_cv.best_estimator_
```

Out[117…
```
GradientBoostingRegressor(loss='huber', max_depth=5, min_samples_leaf=8,
                          min_samples_split=6, n_estimators=500,
```

```
random_state=42)
```

It is a good idea to use random search to narrow down the potential hyperparameters. Based on the results of the random search, we could create a grid with hyperparameters that are close to those that worked best in the randomized search. I will only evaluate one setting, the number of trees (n_estimators). Changing a single hyperparameter alone allows us to observe its effect on performance.

Here we will use grid search with a grid that only has the n_estimators hyperparameter. We will evaluate and compare a set of trees then plot our training and testing performance to understand the effect of increasing the number of trees. For the other hyperparameters, we fix the best values returned by the random search.

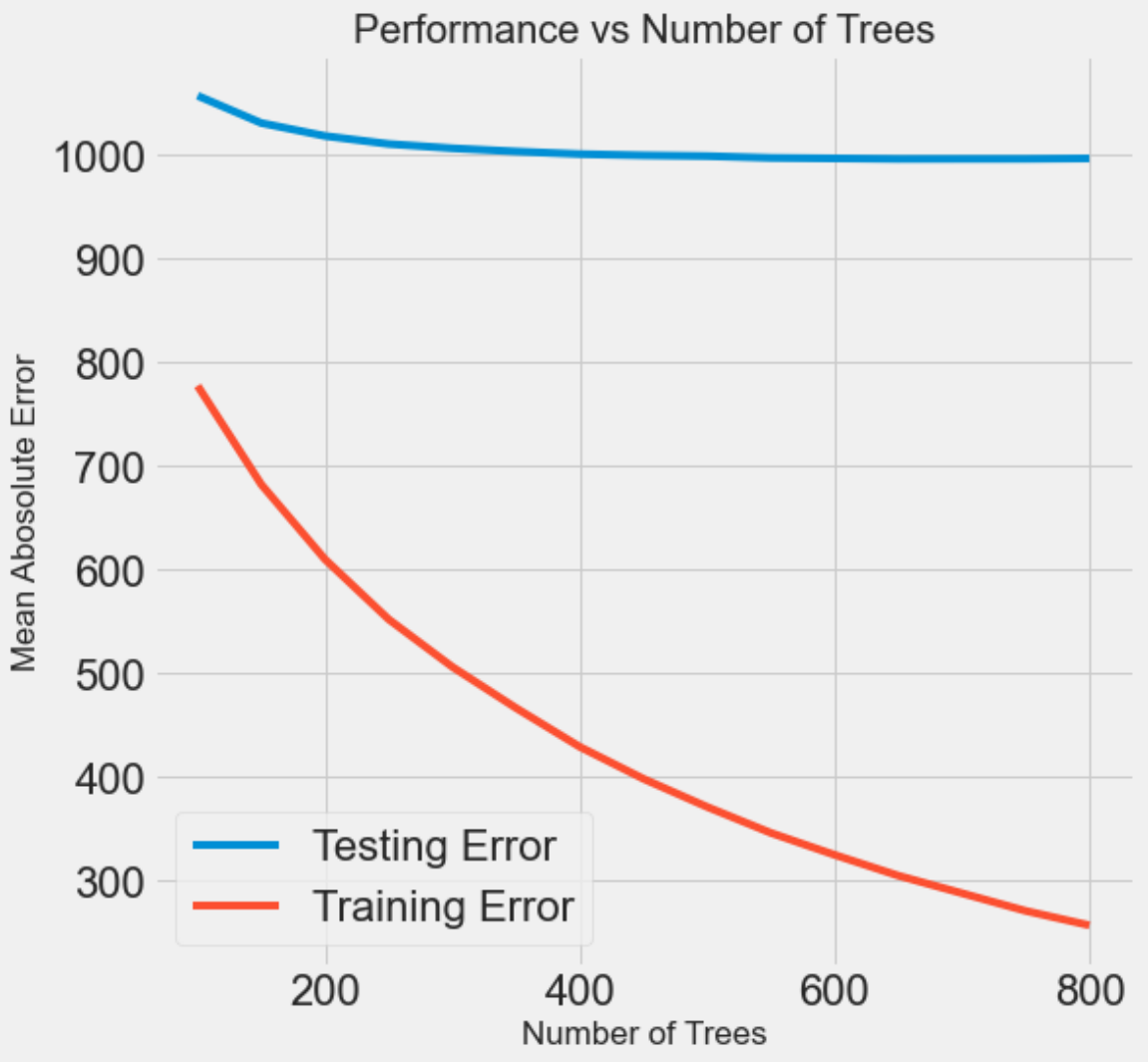
In [118…
```
# Create a range of trees to evaluate
trees_grid = {'n_estimators': [100, 150, 200, 250, 300, 350, 400, 450, 5

model = GradientBoostingRegressor(loss='huber', max_depth=5, max_feature
                          min_samples_leaf=8, min_samples_split=6,random_




# Grid Search Object using the trees range and the random forest model
grid_search = GridSearchCV(estimator = model, param_grid=trees_grid, cv =
                           scoring = 'neg_mean_absolute_error', verbose =
                           n_jobs = -1, return_train_score = True)
```

In [119…
```
# Fit the grid search
grid_search.fit(X, y)
```

```
Fitting 4 folds for each of 15 candidates, totalling 60 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent worker
s.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:  7.2min
[Parallel(n_jobs=-1)]: Done  60 out of  60 | elapsed: 16.3min finished
```

Out[119…
```
GridSearchCV(cv=4,
             estimator=GradientBoostingRegressor(loss='huber', max_depth=
5,
                                                 min_samples_leaf=8,
                                                 min_samples_split=6,
                                                 random_state=42),
             n_jobs=-1,
             param_grid={'n_estimators': [100, 150, 200, 250, 300, 350, 4
00,
                                          450, 500, 550, 600, 650, 700, 7
50,
                                          800]},
             return_train_score=True, scoring='neg_mean_absolute_error',
             verbose=1)
```

In [120…
```
# 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'],
plt.plot(results['param_n_estimators'], -1 * results['mean_train_score']
plt.xlabel('Number of Trees'); plt.ylabel('Mean Abosolute Error'); plt.l
plt.title('Performance vs Number of Trees');
```



In [121…

```
results.sort_values('mean_test_score', ascending = False).head()
```

Out[121…

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_estimators	
11	181.200562	0.152037	0.134584	0.005967	650	{'n_e
12	201.596082	0.203585	0.188223	0.037641	700	{'n_e
13	218.870731	0.215996	0.170098	0.034730	750	{'n_e
14	172.745964	0.264699	0.077485	0.003704	800	{'n_e
10	171.520380	0.116834	0.130122	0.005412	600	{'n_e

We're pretty sure our model is overfitting from this plot! A model's training error is significantly lower than its testing error, meaning that the model is learning the training data very well but then does not generalize to the test data as well. More importantly, overfitting increases with the number of trees. Although both the training and test errors decrease with tree number, the training error decreases faster. A difference will always exist between the training error and the testing error. However, if there is a significant difference, we should try to reduce overfitting by either collecting more training data or by reducing the model's complexity through hyperparameter tuning. For the gradient boosting regressor, some options include reducing the number of trees, reducing the max depth of

each tree, and increasing the minimum number of samples in a leaf node. I'll take the performance model for now regardless of whether it's overfitting to the training set. According to the results, the best model uses 650 trees.

6. Evaluate Final Model on the Test Set

The best model from the hyperparameter tuning will be used to make predictions on the testing set. The performance of the model should give a pretty good indication of how it would perform under real conditions. We can also look at the default model's performance as a comparison.

In [122…
```
# Default model
default_model = GradientBoostingRegressor(random_state = 42)

# Select the best model
final_model = grid_search.best_estimator_

final_model
```

Out[122…
```
GradientBoostingRegressor(loss='huber', max_depth=5, min_samples_leaf=8,
                          min_samples_split=6, n_estimators=650,
                          random_state=42)
```

In [123…
```
default_model.fit(X, y)
```

Out[123…
```
GradientBoostingRegressor(random_state=42)
```

In [142…
```
final_model.fit(X, y)
```

Out[142…
```
GradientBoostingRegressor(loss='huber', max_depth=5, min_samples_leaf=8,
                          min_samples_split=6, n_estimators=650,
                          random_state=42)
```

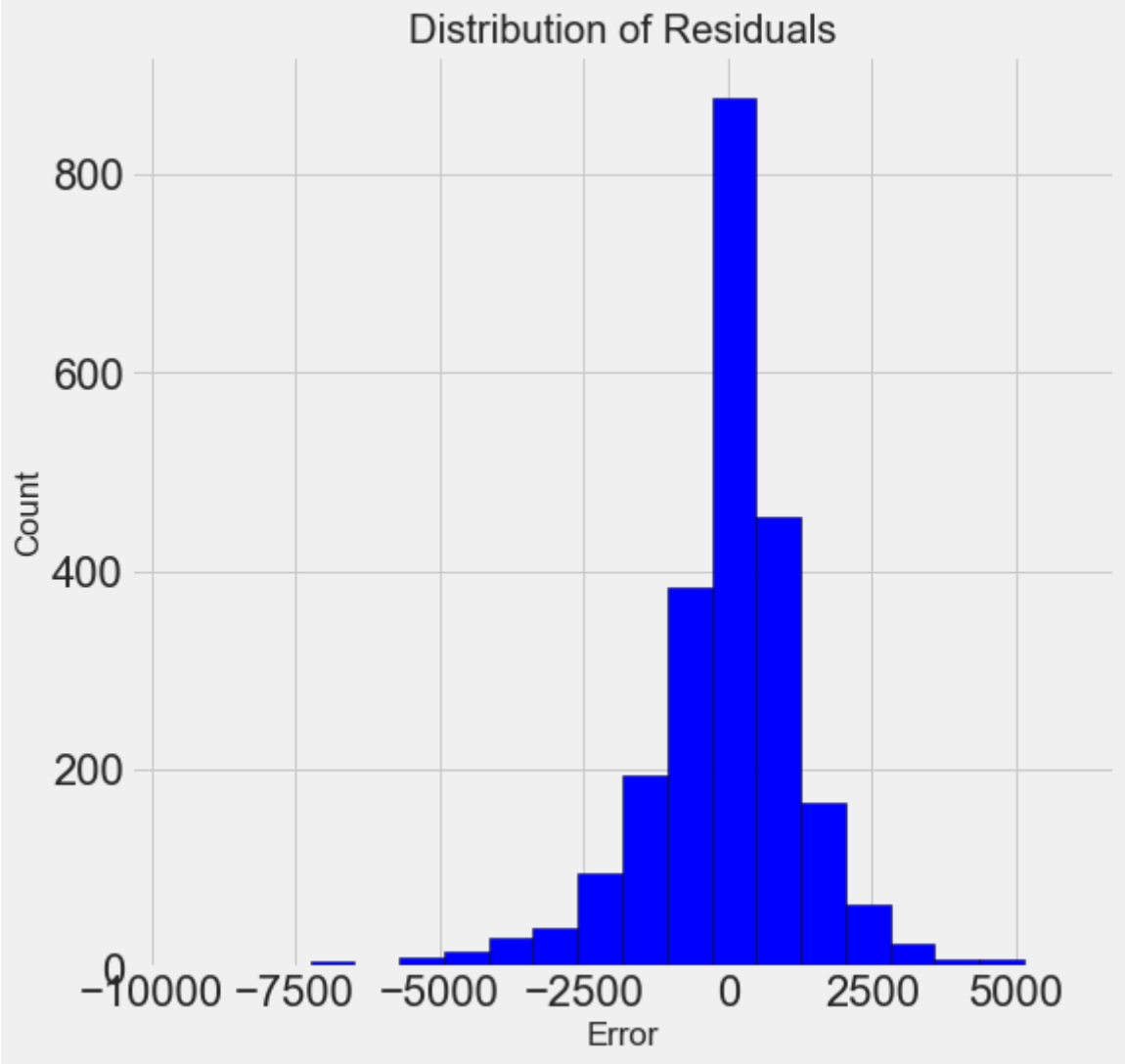
In [143…
```
default_pred = default_model.predict(X_test)
final_pred = final_model.predict(X_test)

print('Default model performance on the test set: MAE = %0.4f.' % mae(y_
print('Final model performance on the test set:   MAE = %0.4f.' % mae(y_
```

```
Default model performance on the test set: MAE = 1172.6188.
Final model performance on the test set:   MAE = 954.5444.
```

The final model does out-perform the baseline model by about 19%.

One of the most useful diagnostic plots is a histogram of the residual values. In an ideal world, the residuals will have a normal distribution, meaning that the model is wrong the same amount in both directions (high and low).

In [126…
```
figsize = (6, 6)

# Calculate the residuals
residuals = final_pred - y_test

# Plot the residuals in a histogram
plt.hist(residuals, color = 'blue', bins = 20,
         edgecolor = 'black')
plt.xlabel('Error'); plt.ylabel('Count')
plt.title('Distribution of Residuals');
```



The residuals are close to being normally distributed.

Feature Importances

Features that are most predictive of the target can be interpreted as feature importances. Scikit-learn makes it pretty straightforward to find the feature importances in an ensemble of trees. In order to visualize and analyze the feature importance dataframes, we will store the feature importance values.

In [144…

```
# Extract the feature importances into a dataframe
feature_results = pd.DataFrame({'new_feature': list(new_features.columns
                                'importance': final_model.feature_import

# Show the top 10 most important
feature_results = feature_results.sort_values('importance', ascending =

feature_results.head(10)
```

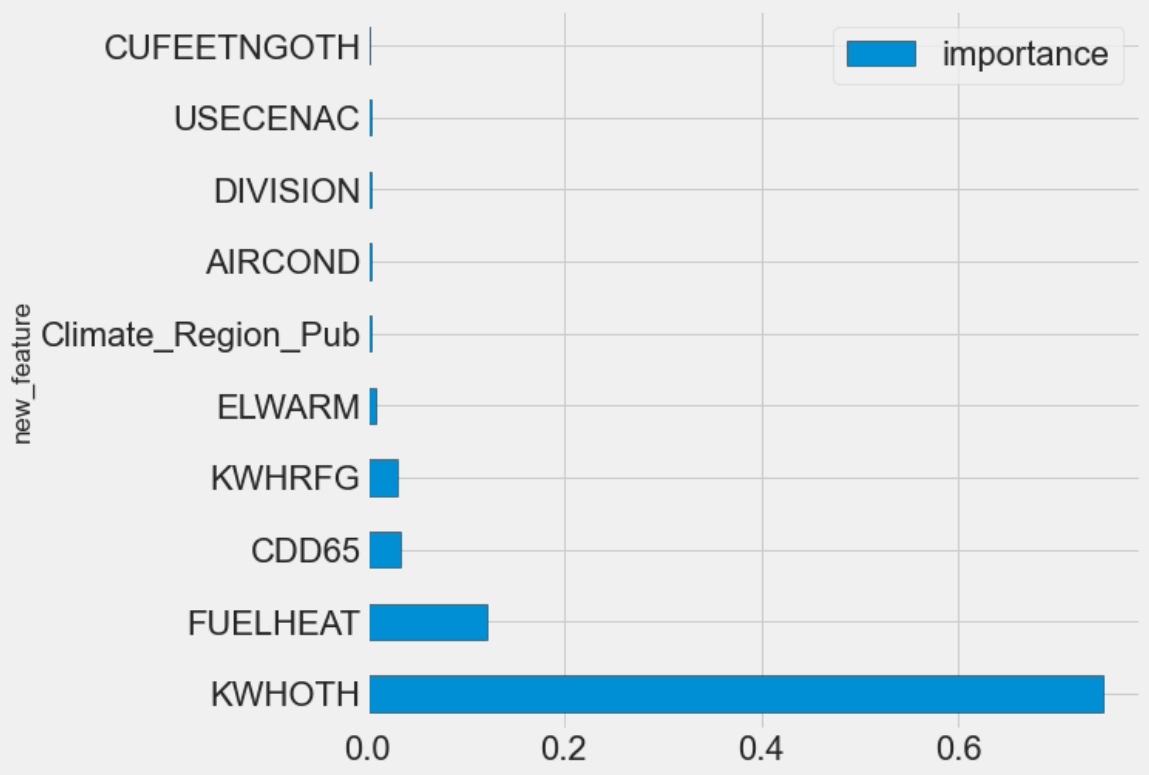
Out[144…

	new_feature	importance
0	KWHOTH	0.747008
1	FUELHEAT	0.121251
2	CDD65	0.033842
3	KWHRFG	0.031654

4	ELWARM	0.009591
5	Climate_Region_Pub	0.004533
6	AIRCOND	0.004469
7	DIVISION	0.003859
8	USECENAC	0.003849
9	CUFEETNGOTH	0.002496

In [145…

```
feature_results.loc[:9, :].plot(x = 'new_feature',
                                y = 'importance',edgecolor = 'k', kind='
```



Feature	Explanation
KWHOTH	Electricity usage for other purposes (all end-uses except SPH, COL, WTH, and RFG)
FUELHEAT	Main space heating fuel
CDD65	Cooling degree days
KWHRFG	Electricity usage for refrigerators
ELWARM	Electricity used for space heating
Climate_Region_Pub	Building America Climate Region
AIRCOND	Air conditioning equipment used
DIVISION	Census Division
USECENAC	Frequency central air conditioner used in summer
CUFEETNGOTH	Natural Gas usage for other purposes (all end-uses except SPH and WTH)

Use Feature Importances for Feature Selection

Given that not every feature is important for finding the target, Let's try using only the 10 most important features in the gradient boosted regressor to see if performance will improved and then re-evaluate it.

In [140…
```
# Extract the names of the most important features
most_important_features = feature_results['new_feature'][:10]

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

# 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:  (9538, 10)
Most important testing  features shape:  (2385, 10)
```

Let's look at reduced set of features in the gradient boosted regressor.

In [141…
```
# Create the model with the same hyperparamters
model_reduced = GradientBoostingRegressor(loss='huber', max_depth=5, max_
                          min_samples_leaf=8, min_samples_split=6,
                          n_estimators=650, 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, mode
```

```
Gradient Boosted Reduced Results: MAE = 1159.4170
```

We will kept all of the features for the final model as the model results are slightly worse with the reduced set of features. It is much faster to train a model with fewer features and often easier to interpret. Because the training time is not significant in this case, keeping all the features is not a major problem, as we can still.

7. Conclusion

Our goal was to find out to build a model to predict total electricity consumption.

The exploration in this notebook leads me to conclude that yes, we should be able to create a model that accurately predicts the total electricity consumption and we have found that the Electricity usage for other purposes (all end-uses except SPH, COL, WTH, and RFG), Main space heating fuel, Cooling degree days, and Electricity usage for refrigerators are the most important factors in determining electricity consumption.

Here are a few highlights:

- The total electricity distribution varies by metropolitan area and to some extent by climate zone.
- The most negative correlations with electricity consumption are the type of housing unit, if natural gas is used and whether natural gas used for water heating.

- A gradient boosted regressor trained on the training data was able to achieve an average absolute error of 954 on a hold-out testing set, which was significantly better than the baseline measure.
- A trained model can accurately determine the total electricity consumption of a new household unit.

This report also specifies areas that will require further attention. Further analysis may involve grouping features based on their characteristics in a pre-processing step.Here are the suggested categories: appliances, building technology, occupant demographics, and energy consumption information. Then create a model that includes all groups and also their combinations. Doing modeling with this way, we will identify key features of all Categories.