Table of Contents

Lab Overview

Important things to note about Python/Jupyter

Welcome to Jupyter. This is a *notebook* development environment for python. If you've used mathematica before, you'll be familiar with the notebook concept.

Each cell (like this one) can either be for *code* or *markdown* (a very basic language for formatting text). This cell is a markdown cell.

Clicking in the `typing` area of a code cell will let you enter edit mode on that cell. Pressing ctrl+Enter will run the code in that cell. Keyboard shortcuts can be found by clicking on help -> keyboard shortcuts.

Please note that any variable created in the code can be viewed by typing `print(variable)`.

The point of this exercise is to introduce you to some of the basic concepts of machine learning and data analytics, and apply knowledge of what you've learned in CE3010 to a real-world problem. We use python for this exercise as it is ubiquitous in the world of data science and machine learning, and is good to get some exposure to. However, you will not be graded on your understanding of python or programming in general. You should be able to gain an intuition of what the code is doing by reading through, and all the code is annotated to help you understand what's going on.

The lab will be structured as follows:

- Lab instructor will go through this jupyter notebook, giving a background to machine learning and taking you through a machine learning problem with the `household_data` dataset
- You will then be asked to perform a similar analysis on the `boole_data` dataset, and write up the results.

You are allowed to copy and paste or otherwise edit some of the code here to apply it to your own datasets if you are not comfortable writing from scratch. As well as this, 99% of errors or anything else you are having trouble with can be solved by a quick google. In particular, the the cross-validated (http://stats.stackexchange.com/) and stack-overflow (http://stackoverflow.com/) stack exchange sites are great resources.

How to load the Jupyter notebook

1. Open Chrome and go to https://notebooks.azure.com (https://notebooks.azure.com)
2. Sign in with your UCC IT account. If you can't remember this, but have a microsoft/hotmail account, you can sign in with this. Otherwise, you'll need to create an account.
3. Click on "Libraries" in the top left
4. Click on "New Library", then click on "From GitHub"
5. The GitHub Repo is https://github.com/lkev/ce3010_lab (https://github.com/lkev/ce3010_lab)
6. Give it the name "CE3010 Lab FirstName LastName" (with your actual name, not FirstName LastName)
7. Give it the ID ce3010-firstname-lastname
8. Click Import

The notebook you'll be using during the class is "Houseing - Partially Filled".

Notebook for homework is "Boole - Partially Filled"

Learning Outcomes

Data analytics allows implicit, previously unknown, and potentially useful information to be extracted from data. It is an interdisciplinary subfield of computer science that combines artificial intelligence, statistics, machine learning and database research. As the volume of data increases, the proportion of it that people understand decreases significantly. Lying hidden in this data is information that is potentially important but has not yet been discovered or articulated. As a result, data analytics allows the useful information within this data to be successfully accessed and analysed. Data analytics and machine learning have many applications in modern engineering sectors, including:

- Building & Energy Analytics
- Industrial Manufacturing
- Engineering Design
- Predictive Maintenance
- Fault Detection & Performance Monitoring
- Self-driving cars
- Banking
- Phone typing
- Medical Diagnosis
- Image recognition

The objectives of this assignment include the following:

- To gain an understanding of the concept of data analytics and its application in buildings for energy performance analysis.
- Get introduced basic machine learning principles and the importance of having a separate test and training set
- Understand the effect of daylight hours, occupancy, heating degree days (HDD) and building opening hours on electrical energy consumption in buildings.
- An introduction to data analysis using python, and the sklearn, numpy and pandas libraries
- To investigate and analyse the energy performance of a UCC building using both correlation and regression analysis
- To predict the future electrical energy performance of the building using the developed regression model

Set Up Libraries

First, we must import the data and relevant libraries

The following code imports various python packages we need to use, and also imports the Housing data from a .CSV file as the variable `house_data`

In [1]:

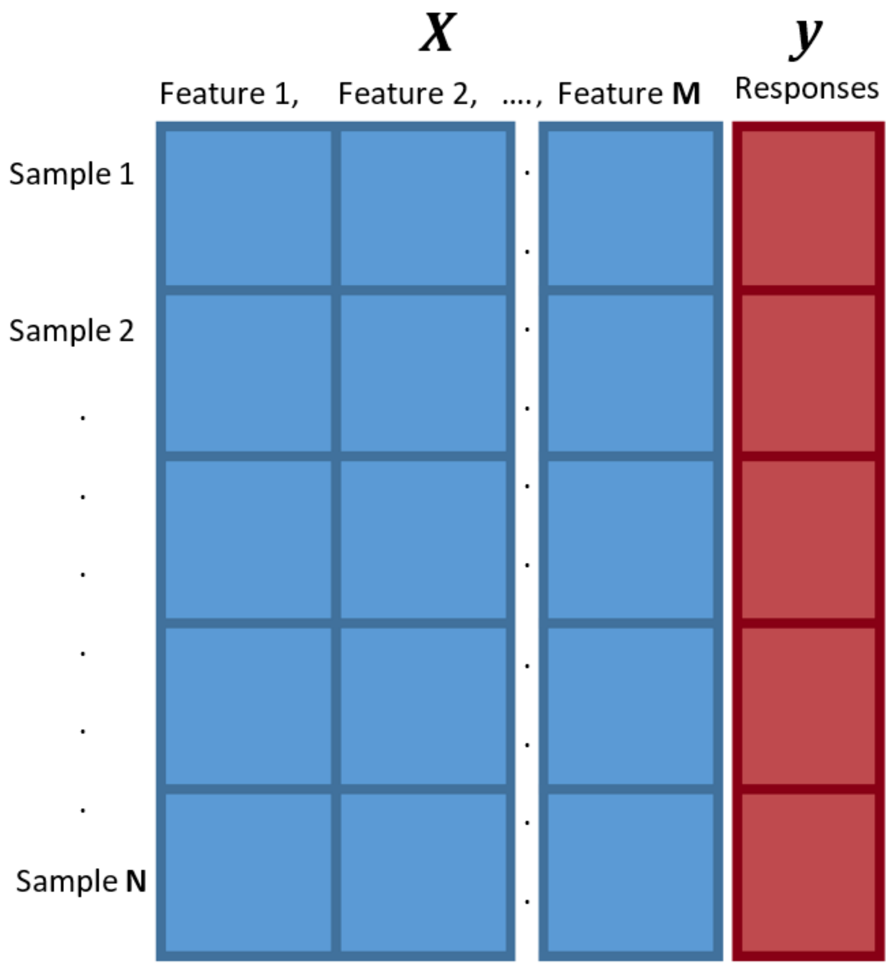
```
# display plots & graphs in browser:
%matplotlib inline

# Various libraries that are required
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import r2_score
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.utils import shuffle

# set the plot styles
sns.set(style="ticks", color_codes=True)

# Import house data
house_data = pd.read_csv("Household Dataset1.csv")
house_data_2 = pd.read_csv("Household Dataset2.csv")

def print_full(x):
    pd.set_option('display.max_rows', len(x))
    pd.set_option('display.max_seq_items', len(x))
    display(x)
    pd.reset_option('display.max_rows')
    pd.reset_option('display.max_seq_items')
```

Background

Introduction

Statistics, machine learning and data science can all be thought of as different sides of the same coin. The fields essentially boil down to different applications or uses for statistics, probability and, to some degree, information theory. As well as this, the different fields may have different terms for the same concepts. At the heart of all of these, however, is **data**. The datasets used in this study have been cleaned and prepared for you for easy manipulation.

Each column in each dataset represents a number of **features** (also known as **independent variables** or **predictors**), as well as a column representing the **response** (also known as the **output** or **dependent variable**). Each row in the dataset then represents an individual entry, comprising the features and associated response for that sample. A diagram of this can be seen below:



In machine learning, the aim is to build a **model** from existing data i.e., existing features and responses. The model identifies some relationship between the features and the responses, so that with any future data you collect, you can make a good **prediction** (also called **estimate** or **hypothesis**) for what the observed response should be.

A good example is house heating: if we have data on a number of houses (the **samples**), we can look at the amount of oil each house used in a year (the **responses**). We then build a model for the relationship between the usage and things such as size, insulation rating, occupancy, etc. (the **features**).

Then, when we want to guess what the usage will be for a house with no existing heating oil data, we can **predict** the amount it should use according to our model.

There are a large number of different statistical models or algorithms that can be used in machine learning to make predictions. When the prediction must be of a quantitative nature (i.e. where the responses are numeric values), we use a technique known as regression. When the responses are qualitative (i.e. when the responses are certain categories or classes), we use classification.

For this assignment, we will be trying to obtain a numeric prediction, so we use regression. Specifically, we will use Ordinary Least Squares Linear Regression, the most basic form of regression.

Data Exploration

In this exercise we will be trying to predict the heating oil usage of homes on a particular day based on a number of factors, including their insulation rating (1-10, higher is better), the temperature on that day, and the age and size of the home.

We have two datasets for houses from two different locations. Dataset 1 includes the heating oil usage for each day, whereas dataset 2 does not.

We will build a regression model from dataset 1 by splitting the data into *training* and *testing* data. The training data is used to build the model, and the testing data is used to see if it generalizes well. We will also explore a feature extraction method to see if we can improve the performance of our model, and verify this using *K-Fold Cross Validation*.

Once we determine the model is performing well, we use it to predict housing heating oil usage for dataset 2. We then compare the two datasets and see why they differ.

Before diving straight into the modelling, it is always a good idea to explore the existing data and see how the features relate to one another. This is a useful sanity check for later on.

Displaying the data

Displaying data to get a good idea of what it looks like should always be the first step of any analysis.

The following shows the columns of `house_data`. Note the features, samples and targets.

Note: The `print_full()` function just prints the data in a better visual format than the standard print function.

In [2]:

```
print('Total number of samples in data:', len(house_data))

#Display first ten rows of the house_data.
house_data.head(10)
```

```
Total number of samples in data: 665
```

Out[2]:

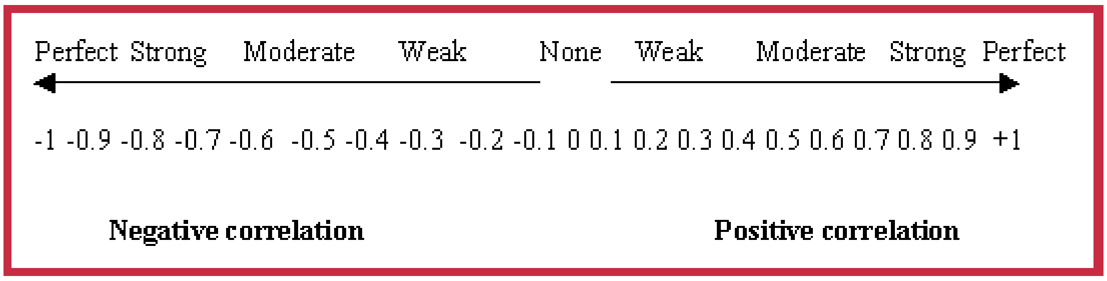
	insulation	temp	age	home_size	oil_usage
0	5.638533	23	23.8	4.524288	132
1	1.271035	6	56.7	4.275943	263
2	8.841273	27	28.0	6.923177	145
3	2.309233	10	45.1	3.025037	196
4	9.515106	27	20.8	2.272948	131
5	6.475489	24	21.5	3.898591	129
6	6.790076	22	23.5	3.898544	131
7	5.724434	31	38.2	6.602018	161
8	6.753981	25	42.5	3.506646	184
9	1.664965	6	51.1	1.535052	225

As can be seen, we have 4 features, `insulation`, `temp`, `age` and `home_size`, and one target, `oil_usage`, with a total of 665 samples.

Correlation Matrix

Correlation analysis is a very quick but powerful technique that can be used to rapidly see how elements in a dataset interact and correlate with each other. This is particularly useful for the optimisation of building energy data as meaningful correlations between dataset attributes (e.g. HDD, footfall, daylight hours etc.) can be found quickly and effectively. In summary, correlation is a statistical measure of how strong the relationships are between the features and responses in a dataset.

Correlation Coefficients between 0 and 1 indicate a **positive correlation**, whereas coefficients between 0 and -1 represent **negative correlation**. A positive correlation coeeficient between two variables means that as one variable rises, the other also rises. A negative correlation means that as one rises, the other falls. The closer a correlation is to 1 or -1, the stronger the correlation, whereas values close to zero indicate little to no statistical relationship between the two variables.



We use the numpy (np) python package to get the correlation coefficients:

```
c = np.coeff(X)
```

This returns a matrix, `c`, of correlation coefficients from an input matrix `X` whose rows are samples and whose columns are features.

In the code below, `corr` is an array of correlation values for each column in the `house_data`.

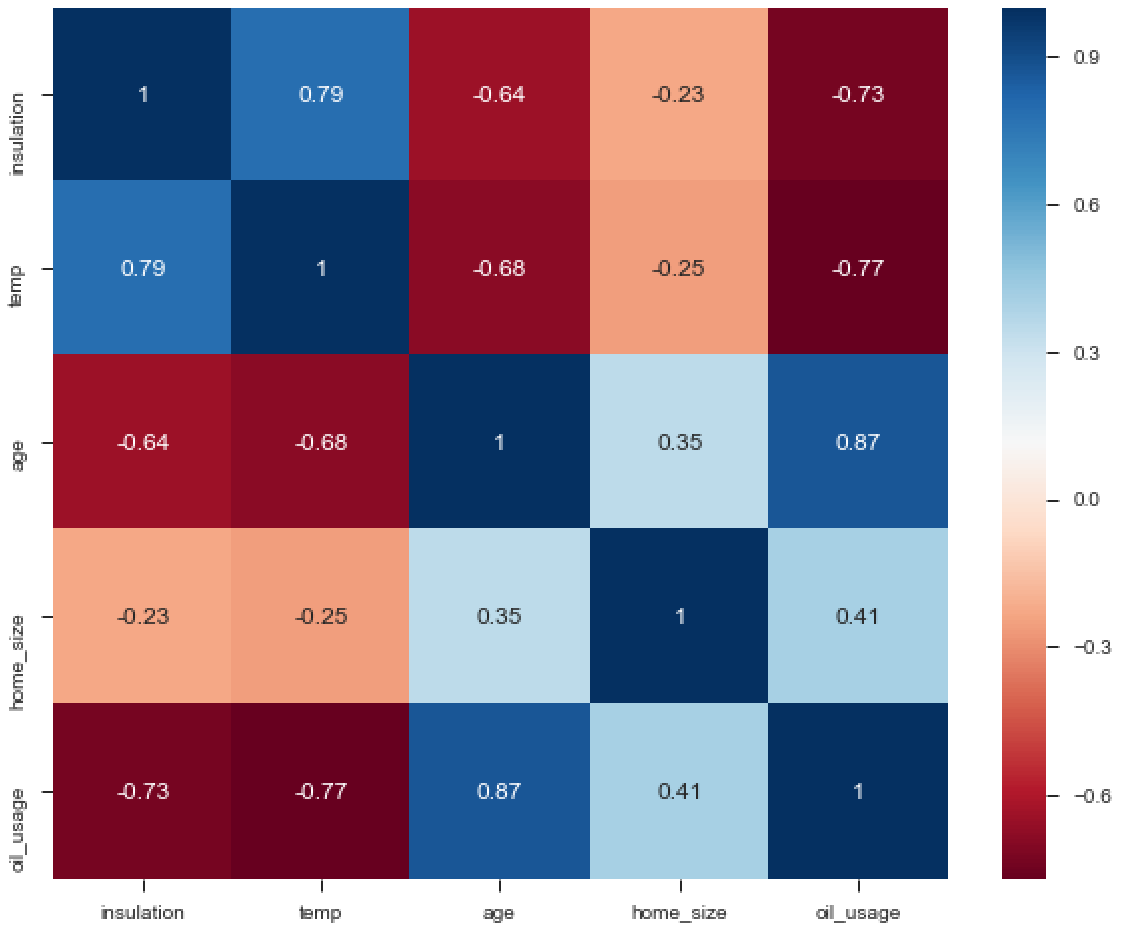
NOTE: `house_data` is imported with the rows and columns the wrong way around for the `np.corr` function. We need to TRANSPOSE the array by using the `.T` method

In [3]:

```
# Create an array of correlation data. Note the .T for transposing
corr = np.corrcoef(house_data.T)

# Create the axis labels for use in the plot
labels = house_data.columns

# set the figure size
plt.figure(figsize=(10, 8))

# Plot a heatmap to easily visualise the relationships
sns.heatmap(corr, xticklabels=labels, yticklabels=labels, annot=True,
            cmap='RdBu')
```

Out[3]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x12b8dee1438>
```



As can be seen, the darker the colour, the stronger the correlation (red for positive, blue for negative). From this, the following are some of the observations we can make from this graphic:

- `insulation` & `oil_usage` are highly negatively correlated (-0.73) – a better insulated house uses proportionately less heating oil, and vice versa
- age and `oil_usage` are highly positively correlated (0.87) – meaning older houses generally use more heating oil, and newer houses use less heating oil.
- `home_size` and `oil_usage` are more weakly correlated correlated - the size of the home has some effect on heating oil usage, but not as much as age, outdoor temperature or insulation

Below, we can see various features plotted against `oil_usage`.

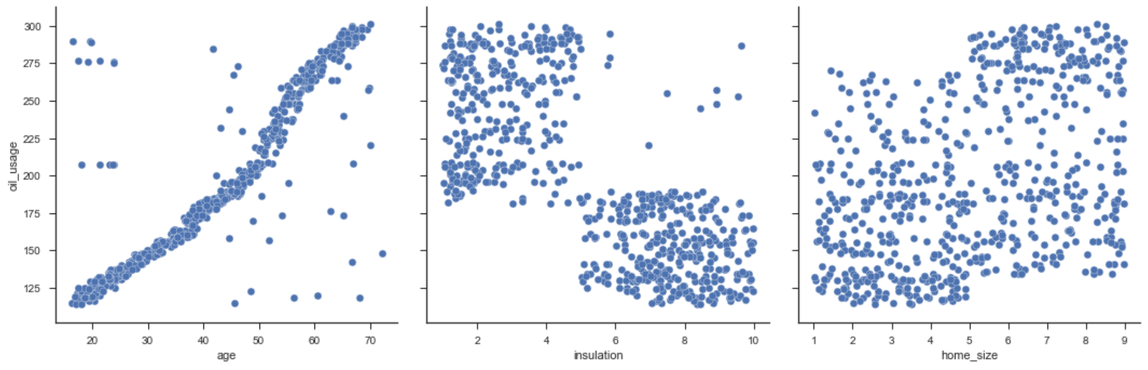
As can be seen, the correlation signs and magnitudes match what is seen in the correlation matrix (i.e. strongly/weakly and positively/negatively correlated)

```
In [4]:
```

```
# plot age, insulation, home_size against oil_usage
sns.pairplot(data=house_data, x_vars=['age', 'insulation', 'home_size'],
             y_vars=['oil_usage'], kind='scatter', size=5)
```

```
Out[4]:
```

```
<seaborn.axisgrid.PairGrid at 0x12b8a5c9be0>
```



Building a Machine Learning Model

Linear Regression

Linear Regression is a simple but powerful technique that is used for numerical prediction. It is a statistical measure that attempts to identify the strength of correlations between one dependent variable (the response) and a series of other changing instances known as independent variables (the features).

Linear regression models are simple models and often provide an adequate and interpretable description of how the inputs affect the outputs. For prediction purposes they can sometimes outperform more elaborate nonlinear models, especially in situations with small numbers of training samples, low signal-to-noise ratio or sparse data.

The equation of a linear regression model is simply the sum of the features for a particular sample, except that weights are applied to each feature before summing them together:

$$\hat{y} = w_0 + w_1x_1 + w_2x_2 + \ldots + w_mx_m$$

Where:

- $\hat{y}$ is the prediction/estimate of the linear model for a particular sample (as opposed to y, the actual value)
- $x_1, x_2, \ldots, x_m$ are the m features of a single sample
- $w_0, w_1, \ldots, w_m$ are the m associated weights applied to each feature

  The w_0 weight is also known as the *intercept*. This is a constant that is added to the equation, and is the value for $\hat{y}$ if all the features x are equal to zero.

Note also that there is no x_0 variable. Sometimes, to make mathematical operations easier, x_0 is set as 1. This is known as a "dummy" variable.

In this exercise, we use the popular `sklearn` package in python to perform linear regression.

Coefficient of Determination (r^2 value)

We evaluate the performance of our model with a particular *scoring metric*, which measures how similar the estimates, $\hat{y}$, are to the actual responses, y. For linear regression, a commonly used metric is the r^2 value, which is essentially a similarity score with values between 0 and 1. Typically, "good" r^2 values are above 0.75.

Over- and Underfitting

When training on the data, a low r^2 value indicates that the model is not entirely accurate. This is known as **underfitting**. However, a high r^2 value does not guarantee that the model will perform well on unseen data as it may **overfit** the data. Examples of these can be seen below:

<img src='over_underfitting.png' /img>

Each point on these graphs represents a training sample, with a single feature, X (on the x axis), and a target value, y, on the y axis. The model itself is shown by the black lines. These are the estimated values of ŷ which will be generated for any unseen values of X.

In the first example, we see that the model is not capturing the true relationship between the feature and target. This leads to a low r^2 score.

The third example shows overfitting. In this case, the model would generate a perfect r^2 score. However, it is obvious that this is not an accurate reflection of the general relationship between the features and targets.

The middle example shows how we would expect to fit the data. This has a lower r^2 score on the training data, but is still a better model overall.

So, how do we evaluate a model to make sure that it generalises well to unseen data (i.e. the future values we will be predicting)? For this, we split our existing data into a *training* set and a *testing* set.

Splitting Data into Training and Testing Sets

To build a successful model that generalises well, we need to split up the initial data we use to build and evaluate the model into **training** and **testing** data.

The training data is used to build the model. The machine learning algorithm will keep iterating over the samples and making target predictions, and compare these to the real value of the targets. On each iteration it will slightly change various parameters specific to the algorithm in use, and keep iterating until the training predictions it makes closely match the actual training responses.

In order to verify that the model performs well on new data that the model has never seen before, we use the testing data to verify. The testing data is fed into the model and a score is given for how similar the test predictions are to the actual test responses.

Typically, a ratio of about 80/20 is used for the train/test split.

The code below gets our features, `X`, and the associated targets, `y`.

The first 80% of the samples are then used for training, and the rest for testing.

In [5]:

```
# Create a matrix of the features (insulation, temp, age, home_size)
X = house_data[['insulation', 'temp', 'age', 'home_size']]

# Create a vector for the dependent variable/target values (oil_usage)
y = house_data['oil_usage']

# Create an array of indices of the training and testing data. The first ~80%
# of the data is used for training, with the rest for testing
train_indices = np.arange(0, len(X) * 0.8)
test_indices = np.arange(len(X) * 0.8, len(X))

# shuffle the data
# X, y = shuffle(X, y)

# get the training samples
X_train = X.iloc[train_indices]
y_train = y.iloc[train_indices]

# and the test samples
X_test = X.iloc[test_indices]
y_test = y.iloc[test_indices]
```

Building the Model

Training

The linear regression model to predict the heating oil consumption of a dwelling is as follows:
$EstimatedHeatingOilUse = w_0 + (w_1 * InsulationRating) + (w_2 * Temperature) + (w_3 * Ag$

We use statistical packages which use specialised optimisation algorithms to search for and find the best weights. The weights are found by feeding the training data into the model. This training data consists of the measured features, X_{train} and responses, y_{train}. The associated weights, w, are found by minimising the difference between the estimations, $\hat{y}_{train}$, and the actual targets, y_{train}.

The code below trains a new linear regression model, `oil_use_model`, on the training data, `X_train` and `y_train`. This results in finding the weights, w.

The intercept, w_0, and weights, w_1, w_2, w_3, w_4, are printed below. Note that weights are sometimes called *regrsssion coefficients*, which is why they are stored here here in the `.coef_` attribute.

In [6]:

```
# create an instance of the sklearn linear regression model
oil_use_model = LinearRegression(fit_intercept=True)

# fit the model to the training data
oil_use_model.fit(X_train, y_train)

intercept = oil_use_model.intercept_
weights = oil_use_model.coef_

# the 'format' function here is just to format to 2 decimal places
print('intercept (w_0):', intercept)
print('weights (w_1, w_2, w_3, w_4):', weights)
```

```
intercept (w_0): 131.462098687
weights (w_1, w_2, w_3, w_4): [-3.76185655 -1.19172767  2.14584295  3.0871
8848]
```

In this instance, our linear regression model, `oil_use_model`, consists of the following equation:
$EstimatedHeatingOilUse = 131.46 - (3.76 * InsulationRating) - (1.19 * Temperature) + ($
$+ (3.09 * SizeOfHome)$

Testing

Once the model is built, we use the weights found previously along with the testing data to verify its real-world performance. As we know, the testing data has a number of samples, again consisting of a number of features, X_{test}, and responses, y_{test}.

We substitute the X_{test} features (home size, insulation, etc.) into our regression model, and multiply by the weights found previously to get an estimated $\hat{y}_{test}$ for each sample.

We can then compare our new $\hat{y}_{test}$ estimates to our actual measured y_{test} responses to calculate a test set r^2 score. This r^2 score is the performance we'd expect our model to have with new, unseen, data.

The code below gets predictions, `y_hat`, using the weights stored in our trained model, `oil_use_model`.

It then calculates the r^2 score using a built in function from `sklearn`.

In [7]:

```
# find predictions using the test data
y_hat = oil_use_model.predict(X_test)

# get the r2 score

score = r2_score(y_test, y_hat)

print(score)
```

```
0.816004971408
```

Visualising the Predictions

The resulting score is ~.82, showing that our model fairly accurately models the data.

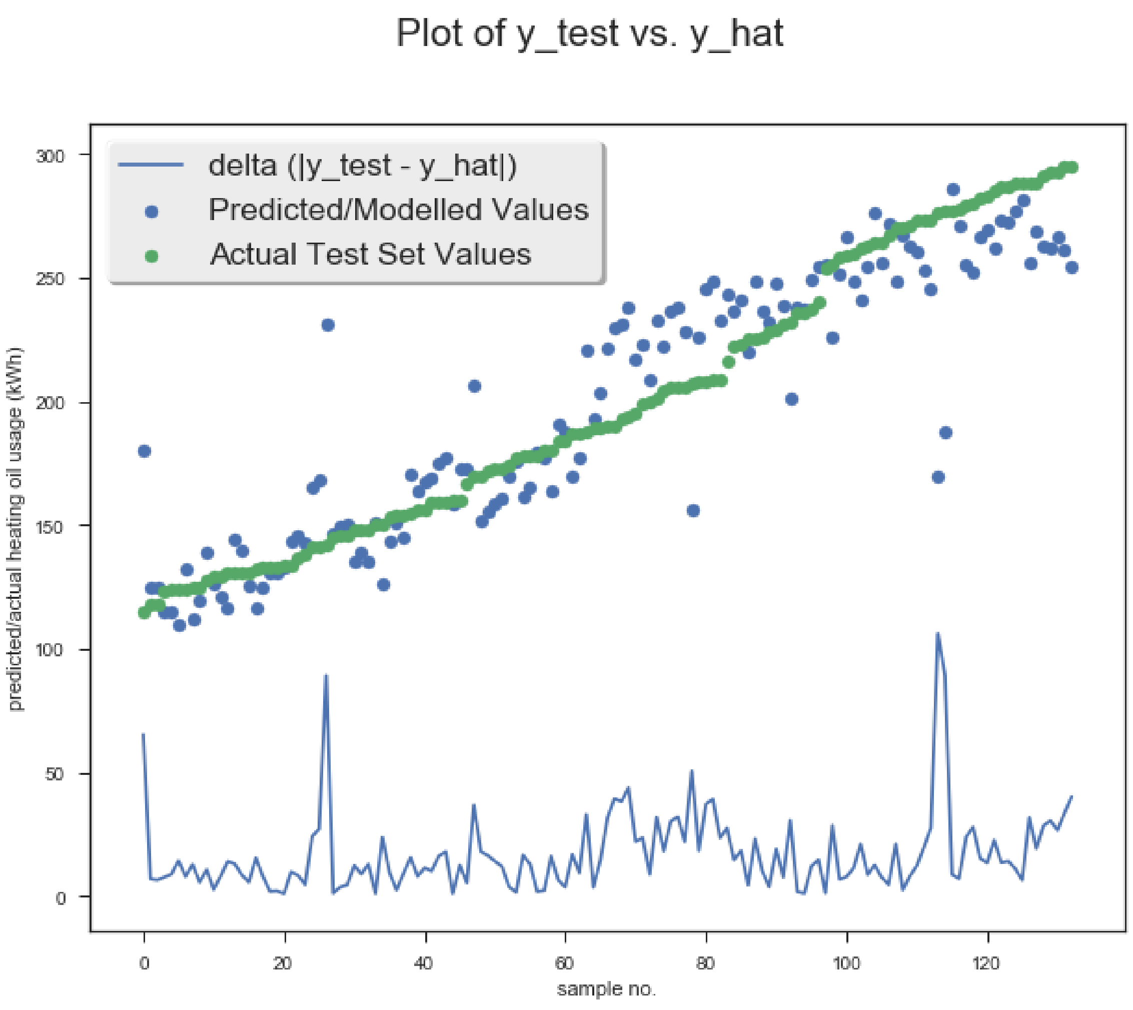
By plotting the predicted $\hat{y}_{test}$ against the actual y_{test}, we can visualise how accurate our model is.

Below we plot `y_hat` against `y_test`. Note that we have re-organised `y_test` here to be in ascending order for easier visualisation. Otherwise the plot looks quite messy (you can play around with this yourself).

In [8]:

```
# Create vector of numbers from 1 to length of y_plot, to serve as the x-axis:
nums = np.arange(len(y_hat))

# sort the values in ascending order
y_test_sorted = np.sort(y_test)
y_hat_sorted = y_hat[np.argsort(y_test)]

# difference between y_hat and y_test
delta = abs(y_hat_sorted - y_test_sorted)

# create the plot
fig, ax = plt.subplots(figsize = (10,8))

# Plot the delta as a line graph
ax.plot(nums, delta, label = 'delta (|y_test - y_hat|)')

# Plot y_hat
ax.scatter(nums, y_hat_sorted, label='Predicted/Modelled Values')

# plot y_test
ax.scatter(nums, y_test_sorted, label='Actual Test Set Values')

# Set up legend & title
fig.suptitle('Plot of y_test vs. y_hat', fontsize='xx-large')
ax.legend(fontsize='x-large', frameon=True, shadow=True)
ax.set_xlabel('sample no.')
ax.set_ylabel('predicted/actual heating oil usage (kWh)')
```

```
Out[8]:

<matplotlib.text.Text at 0x12b90107ef0>
```



Plot of y_test vs. y_hat
delta (|y_test - y_hat|)
Predicted/Modelled Values
Actual Test Set Values
predicted/actual heating oil usage (kWh)
sample no.
0
50
100
150
200
250
300
0
20
40
60
80
100
120

Feature Selection & Cross Validation

A massive part of machine learning is an area called *feature extraction* and *feature selection*.

This involves creating new features from our training data. A simple way to create new features is by introducing **polynomial features** (seen below).

Polynomial Regression

By getting the nth degree polynomials of the features for each sample, we can adapt linear regression to model non-linear behaviours.

For example, the linear regression model for a sample X_n, $X_n = \{x_1, x_2\}$ is:

$$\hat{y} = w_0 + (w_1 * x_1) + (w_2 * x_2)$$

The polynomial features are then:

$$P^2(X_n) = \{x_1, x_2, x_1^2, x_1x_2, x_2^2, x_2x_1\}$$

This would change our linear regression equation to be:

$$\hat{y} = w_0 + (w_1 * x_1) + (w_2 * x_2) + (w_3 * x_1^2) + (w_4 * x_2^2) + (w_5 * 2x_1x_2)$$

Oftentimes, this can lead to better results.

Below, we get the 2nd degree polynomial features for the first two samples in the training data.

This is done with an instance of `PolynomialFeatures` in `sklearn`.

In [9]:

```
base_features = X_train[0:2].values
print('base features:\n', base_features)

poly_transformer = PolynomialFeatures(degree=2, include_bias=False)
poly_features = poly_transformer.fit_transform(base_features)
print('\npoly features:\n', poly_features)
```

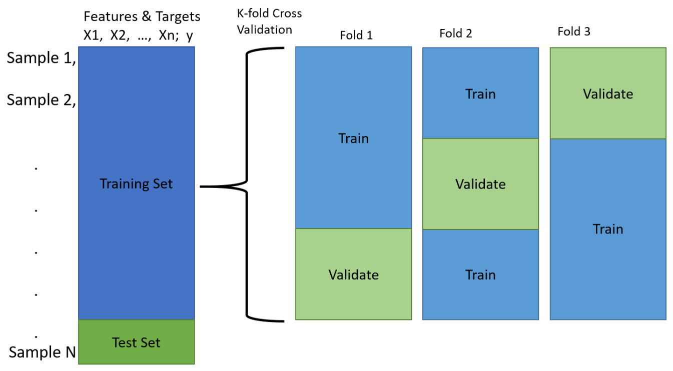
```
base features:
 [[  5.63853313  23.          23.8          4.52428813]
 [  1.27103474   6.          56.7          4.27594252]]

poly features:
 [[  5.63853313e+00   2.30000000e+01   2.38000000e+01   4.52428813e+00
    3.17930558e+01   1.29686262e+02   1.34197088e+02   2.55103485e+01
    5.29000000e+02   5.47400000e+02   1.04058627e+02   5.66440000e+02
    1.07678058e+02   2.04691831e+01]
 [  1.27103474e+00   6.00000000e+00   5.67000000e+01   4.27594252e+00
    1.61552932e+00   7.62620847e+00   7.20676700e+01   5.43487151e+00
    3.60000000e+01   3.40200000e+02   2.56556551e+01   3.21489000e+03
    2.42445941e+02   1.82836844e+01]]
```

Cross Validation

Oftentimes, we want to compare how our model performs on different sets of features. A simple way to do this would be to compare the test set scores of the model trained on each set of features.

However, once again this can lead to overfitting, as the best performing set of features may simply just be well fitted to that particular test set; for other samples that set of features may score comparatively worse than the others. In order to avoid this problem, we use **cross validation**.

Here, we use a strategy called k-fold cross validation. Essentially, this means that the model is trained k times on k different "folds" of the training data, with each fold having a separate training and validation set of its own. This means that for each set of features, k different models are trained, and the average score across these k validation folds are given as the final "cross validation" score.



The best performing set of features are then used to re-train the model with the full set of data, and tested on the test set.

Below, we perform 5-fold cross validation across polynomial features ranging from 1 (i.e. the base set) to 5.

The `KFold().split()` function in `sklearn` allows us to automatically split the data across a number of folds.

it is trained on the training portion of each fold, and scored on the validation portion.

For each `poly_degree`, we store the average of the score across all folds. The best performing set of features are then scored on the test set to get the final score of our model.

In [10]:

```
poly_degrees = [1, 2, 3, 4, 5, 6]
cv_folds = KFold(n_splits=5)

# set up the scores dictionary
scores = {}

# loop through each set of features
for poly_degree in poly_degrees:
    # set up the transformer
    poly_transformer = PolynomialFeatures(
        degree=poly_degree, include_bias=False)

    # generate the new set of features
    X_train_poly = poly_transformer.fit_transform(X_train)

    # empty scores list
    scores[poly_degree] = []

    print('\nscores for poly_degree={}:'.format(poly_degree))

    for i, (train_indices, validation_indices) in enumerate(
            cv_folds.split(X_train, y_train)):

        # get the X and y train and validation set for this fold
        X_train_cv = X_train_poly[train_indices]
        y_train_cv = y_train.iloc[train_indices]

        X_valid_cv = X_train_poly[validation_indices]
        y_valid_cv = y_train.iloc[validation_indices]

        # build the model and make predictions
        oil_use_model = LinearRegression(fit_intercept=True)
        oil_use_model.fit(X_train_cv, y_train_cv)

        y_hat = oil_use_model.predict(X_valid_cv)

        # score it
        fold_score = r2_score(y_valid_cv, y_hat)

        # store the score
        scores[poly_degree].append(fold_score)

        print('fold {}: {:.2f}'.format(i, fold_score))

    average_score = np.mean(scores[poly_degree])
    print('average across folds: {:.2f}'.format(average_score))
```

```
scores for poly_degree=1:
fold 0: 0.90
fold 1: 0.78
fold 2: 0.84
fold 3: 0.77
fold 4: 0.85
average across folds: 0.83

scores for poly_degree=2:
fold 0: 0.92
fold 1: 0.77
fold 2: 0.84
fold 3: 0.80
fold 4: 0.86
average across folds: 0.84

scores for poly_degree=3:
fold 0: 0.97
fold 1: 0.92
fold 2: 0.94
fold 3: 0.78
fold 4: 0.91
average across folds: 0.91

scores for poly_degree=4:
fold 0: 0.97
fold 1: 0.73
fold 2: 0.90
fold 3: 0.80
fold 4: 0.91
average across folds: 0.86

scores for poly_degree=5:
fold 0: 0.77
fold 1: -1.08
fold 2: 0.26
fold 3: 0.42
fold 4: 0.91
average across folds: 0.25

scores for poly_degree=6:
fold 0: -8.33
fold 1: -127.39
fold 2: -0.91
fold 3: -7.51
fold 4: 0.82
average across folds: -28.66
```

Training and Testing Model on Best Feature Set

As can be seen, the best set of features was for `poly_degree`=3. In this case, we now train a model on the full set of training data (as opposed to three separate folds), and test on the held-out test set. This should give us a very confident estimate of how the model will perform on new, unseen data.

In [11]:

```
poly_transformer = PolynomialFeatures(degree=3, include_bias=False)
# generate the new set of features
X_train_poly = poly_transformer.fit_transform(X_train)
X_test_poly = poly_transformer.transform(X_test)

final_oil_use_model = LinearRegression(fit_intercept=True)
final_oil_use_model.fit(X_train_poly, y_train)

y_hat = final_oil_use_model.predict(X_test_poly)

score = r2_score(y_test, y_hat)

print('final score on test set:', score)

print('intercept:\n',  final_oil_use_model.intercept_)
print('weights:\n', final_oil_use_model.coef_)
```

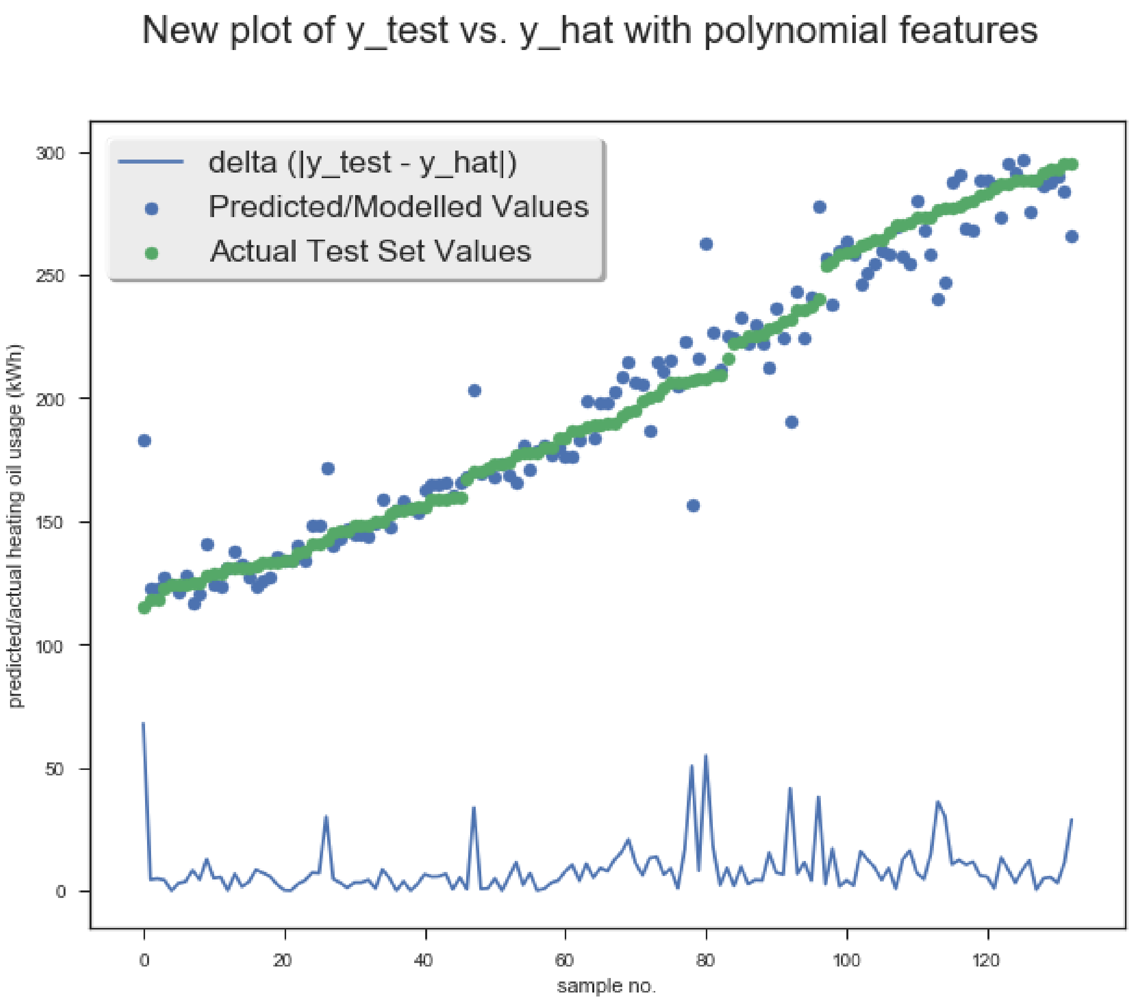
```
final score on test set: 0.93690134861
intercept:
 535.17201354
weights:
 [ -4.88455434e+01  -3.93095627e+00  -2.41597430e+01   2.14782300e+01
  -4.09422461e-01   1.02736256e+00   2.32327369e+00  -3.10631128e+00
  -3.64055177e-01   3.40243267e-01   1.30923659e-01   4.51986962e-01
  -8.94310632e-01   1.77087650e+00   1.19598093e-01  -3.44091011e-02
  -2.47076952e-02  -3.48921271e-02   8.54491953e-03  -2.66669259e-02
   4.69210810e-02  -2.24262575e-02   5.90842836e-02   9.09450646e-03
   4.77291481e-03   2.03306753e-03  -9.79441695e-03  -3.37554388e-03
  -1.46094149e-03  -3.65633018e-03  -2.16803763e-03   6.79358352e-03
   2.06620090e-04  -1.20941152e-01]
```

With an r^2 score of ~.94, the model performs very well and seems to be well suited to modelling on unseen data.

In [12]:

```python
# Create vector of numbers from 1 to length of y_plot, to serve as the x-axis:
nums = np.arange(len(y_hat))

# sort the values in ascending order
y_test_sorted = np.sort(y_test)
y_hat_sorted = y_hat[np.argsort(y_test)]

# difference between y_hat and y_test
delta = abs(y_hat_sorted - y_test_sorted)

# create the plot
fig, ax = plt.subplots(figsize = (10,8))

# Plot the delta as a line graph
ax.plot(nums, delta, label = 'delta (|y_test - y_hat|)')

# Plot y_hat
ax.scatter(nums, y_hat_sorted, label='Predicted/Modelled Values')

# plot y_test
ax.scatter(nums, y_test_sorted, label='Actual Test Set Values')

# Set up legend & title
fig.suptitle('New plot of y_test vs. y_hat with polynomial features',
             fontsize='xx-large')
ax.legend(fontsize='x-large', frameon=True, shadow=True)
ax.set_xlabel('sample no.')
ax.set_ylabel('predicted/actual heating oil usage (kWh)')
```

```
Out[12]:

<matplotlib.text.Text at 0x12b904ff358>
```



Making future predictions

If the r^2 score for the testing data is sufficiently good (~ > 0.75), we can assume the model generalises well to new data.

With this in mind, we can feed the model new, unseen features which we do not have associated responses for. We can then use these features to make predictions using our model:

$$\hat{y}_{new} = w_0 + w_1 x_{1new} + w_2 x_{2new} + \ldots + w_m x_{m\,new}$$

At this point, since we have a good r^2 score on the test set, we can assume that our predictions on a new dataset are accurate. We can then use our predictions to draw conclusions from the data. For example, we can get an average of all X and y from our initial training/testing dataset, and compare to averages of X_{new} and $\hat{y}_{new}$. If the average of $\hat{y}_{new}$ is higher than y, we can make a reasonable hypothesis of why this is the case by looking at the averages of X and X_{new}, and comparing with the correlation matrix found earlier. For example, on the housing dataset, if the average age of houses on the new dataset is higher than on the one used for training and testing, we can infer that the heating oil usage should be higher as well.

Here we'll import a second housing dataset, with houses that are not included in our previous training data.

Note that there is no `oil_usage` column here, so we can make predictions for this based on our previous model!

In [13]:

```
house_data_2.head(10)
```

Out[13]:

	insulation	temp	age	home_size
0	9.805648	27	19.9	2.242131
1	8.749482	31	21.4	1.833833
2	8.506852	23	19.5	1.017444
3	8.812774	27	22.4	4.161662
4	8.902513	24	27.7	4.853260
5	6.724171	28	28.8	8.863458
6	6.630969	30	21.3	4.013038
7	7.939933	24	28.4	8.645317
8	8.275060	27	27.4	3.154570
9	7.759762	25	23.0	2.494612

Input into existing model

Since we now know that the model performs well on unseen data, we can say with some confidence that its predictions will be accurate.

Below, we make predictions from the new data.

Note that we must first transform the features with a 3rd-degree polynomial, as before:

In [14]:

```
# Create a new matrix of features - insulation, age, temp, home_size
X_new = house_data_2[['insulation', 'age', 'temp', 'home_size']]

X_new_poly = poly_transformer.transform(X_new)

# Create a vector of actual target values
y_pred_new = final_oil_use_model.predict(X_new_poly)
```

Compare Predictions against Housing Dataset 1

We've now got predictions for all the new Data. We can save these predictions as a csv for pasting into excel, or manipulate them from within python.

We can sum up and compare the total consumption from each dataset. Why is this higher or lower? In this case, it may be because of increased average temperature, more average occupants per house, or a better insulation rating.

In [16]:

```
# Can export y_pred_new for pasting into the existing Household Dataset1 csv,
# for analysis in excel
# np.savetxt('house_dataset2_predictions.csv', y_pred_new)

# Here, we'll perform analysis in python
# First, we get the average of all features in dataset1 and _new
# The (0) in .mean(0) means go along axis 0, i.e. the columns, instead of axis
# 1, i.e. the rows
print('means for Dataset1: \n', house_data.mean(0), '\n')
print('means for Dataset2: \n', house_data_2.mean(0), '\n',
      'estimated_oil_usage', np.mean(y_pred_new))
```

```
means for Dataset1:
 insulation      5.318111
temp           18.706767
age            42.802556
home_size       5.041455
oil_usage     196.590977
dtype: float64

means for Dataset2:
 insulation     7.872759
temp          26.587719
age           23.071053
home_size      3.867082
dtype: float64
 estimated_oil_usage 139.507550099
```

As can be seen, the estimated heating oil usage in dataset 2 is quite low.

From the correlation matrix in Section 5 and partial regression plots in 6.3.1, we can see that insulation and average age are strongly positively correlated with Heating Oil usage. These are both lower on average for the houses in dataset2, so we see a lower heating oil usage. Temperature is negatively correlated, so it being higher on average in datset2 will also lower the heating oil usage.