Analysis of Nutrition data through the use of Machine Learning techniques

Timothy Ryall

April 20, 2023

Contents

1 Introduction

In this report we will preform an analysis on a data set gathered from "The Australian Food Composition Database" using several machine learning techniques and algorithms.

There are many problems we could tackle with this data set however this report will focus on predicting the 'Energy (KJ)' per 100g of a food is based off its nutrient composition. This choice was based off this being a real and interesting problem that is worth investigating. Currently the true energy content of a food is calculated through a very complicated and potentially costly process [4]. Thus, being able to estimate the energy content of a food based off a small amount of nutritional information is a potentially very useful technique. However these techniques could be applied to different target variables to achieve similar analysis.

Since 'Energy (KJ)' is a numerical variable this problem will be formulated as a regression problem.

2 Data set analysis and processing

2.1 Data set information

This report will use data contained within "The Australian Food Composition Database - Release 2" specifically the file "Rel_2_Nutrient_File.xlsx". This file contains data on 1616 foods, and the component nutrients contained within them. We will focus on the sheet: "All solids & liquids per 100g", which as the title implies contains data on solid and liquid foods, and their nutrients.

This data has 1616 observations, with 323 features, however we will remove the first 3 columns from our data as they are not directly related to nutrients, and therefore should not be used for analysis (they are: 'Public Food Key', 'Classification' and 'Food Name').

Within this data we can observe our target feature as: 'Energy with dietary fibre, equated (kJ)', which we will now just refer to as 'Energy (KJ)'. We can also note that all the remaining features within this data set (including our target feature) are numerical. This will be important when considering what model to use which we will discuss later.

Within this section we will use the package *pandas* to import and manipulate our data. And the packages *seaborn* and *matplotlib* to visualise our data.

2.2 Data cleaning

To begin processing our data we will first import our data into python as a data frame. We will then make the adjustments to our data that was mentioned in the previous section.

```
df = pd.read_csv("report/Rel_2_Nutrient_file.csv")

# remove the 4 columns mentioned in 'Data set information' Section
df = df.drop(columns=df.columns[:3]) # removing: Key, Classifcation, Name

# rename Energy column
df = df.rename(columns={'Energy with dietary fibre, equated \n(kJ)': 'Energy \
    n(kJ)'})
```

Listing 1: Importing and inital adjustments to data

2.2.1 Removing redundant features

There are a number of features within the data set which are functions of other features. There is another data file, "Core_nutrient_details.xlsx", which contains a list of 'core' nutrients (none of the nutrients in the list direct linear functions of each other). So to minimise redundant features we should only use the features contained within that list.

```
# read in the core nutrient file
core_nutrients_df = pd.read_csv("report/Core_Nutrient_details.csv")
# only use the component column
core_nutrients = core_nutrients_df['Component']

# make a list of the features that are core nutrients
columns_to_keep = []
for col in df.columns:
    for nutrient in core_nutrients:
        if nutrient in col: # if column is a core nutrient
            columns_to_keep.append(col)
            break

# drop the features that are not core nutrients
df = df.drop(columns=[col for col in df.columns if col not in columns_to_keep
    ])
```

Listing 2: Removing non-core Nutrients

2.2.2 Removing features with majority null or zero

If a majority ($> 50\%$) of the observations have a zero or null value for a feature, we will remove the feature as it likely will not contain enough data to help predict our target.

```
# when a feature has > 50% NaN or zero values we remove it
# count the number of non-null, non-zero values in each column
obs_counts = df.apply(lambda x: x[x.notnull() & (x != 0)].count())

# define the threshold value for the minimum number of observations
min_obs = df.shape[0] * 0.5 # number of observations * 50%
# filter the columns based on the minimum number of observations
selected_columns = obs_counts[obs_counts >= min_obs].index

# update dataframe with only the selected columns
df = df[selected_columns]
```

Listing 3: Removing features with majority null or zero values

2.2.3 Removing null values

The next step is to make sure there are no Null values present in our data. To do this we will make the assumption that if a value is not given for a particular observation (it is left Null), then it is meant to be 0. So we must replace all Null values with 0 in our data.

```
# replace all NaN values with assumed 0
df = df.fillna(0)
```

Listing 4: Replacing Null values with 0

2.2.4 Converting to numeric data

Some of the data contains commas which will result in the numbers being interpreted as strings. So, remove these to can convert to numerical data.

```
# replace commas with nothing in all columns
df = df.replace(',', '', regex=True)
# convert all columns to numeric type
df = df.apply(pd.to_numeric, errors='coerce')
```

Listing 5: Converting to numerical data

2.3 Feature selection

It is common to use Principal component analysis (PCA) to reduce the dimensionality of the input data. However due to the fact that this has not yet been covered we will use a more heuristic approach.

We will select features for the model based on their correlation with the target feature, 'Energy'. Applying *Cohen's* interpretation of correlation, we can approximate that features with an absolute correlation > 0.3 can be considered to have moderate correlation [1]. So, we will select a feature to use for our model if they have an absolute correlation of > 0.3.

```
# calculate the correlation between Energy and every other column
corr_with_energy = df.corrwith(df['Energy (kJ)']).abs()
# define the threshold value
threshold = 0.3
# filter the correlation coefficients to remove elements below the threshold
corr_with_energy = corr_with_energy[corr_with_energy >= threshold]

# plot the correlation coefficients as a bar chart
corr_with_energy.plot(kind='bar')
plt.title('Correlation with Energy')
plt.show()

# plot the heat map
corelation_map(df[corr_with_energy.index])

def corelation_map(features):
    # calculate the correlation between columns
    corr_matrix = features.corr(numeric_only = True).abs()

    # plot the correlation matrix as a heatmap
    sns.heatmap(corr_matrix, annot=True)

    # display the heatmap
    plt.show()
```

Listing 6: Selecting higher correlation parameters to include in model

Through this processes, we have now selected 11 input features and 1 target feature (Energy), to make up our model. The list of features selected can be seen in the bellow graphs. The features on the graph have been abbreviated for readability. A list of the full feature names be found in Appendix A.

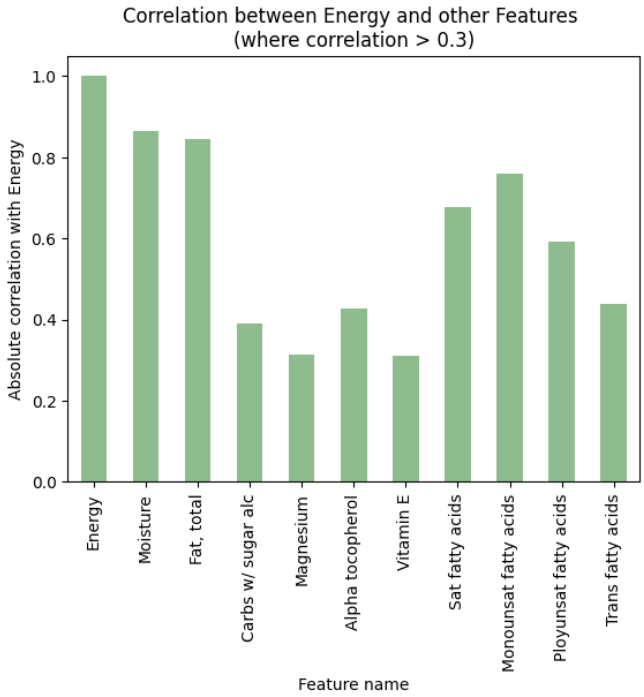


Figure 1: Correlations between Energy and the other features in the model

2.4 Feature evaluation

We can look at the correlation between each pair of features in our model to check there is no redundancy. No feature appears to have an abnormally strong correlation (> 0.95) which implies that it is likely that none of the features are functions of each other.

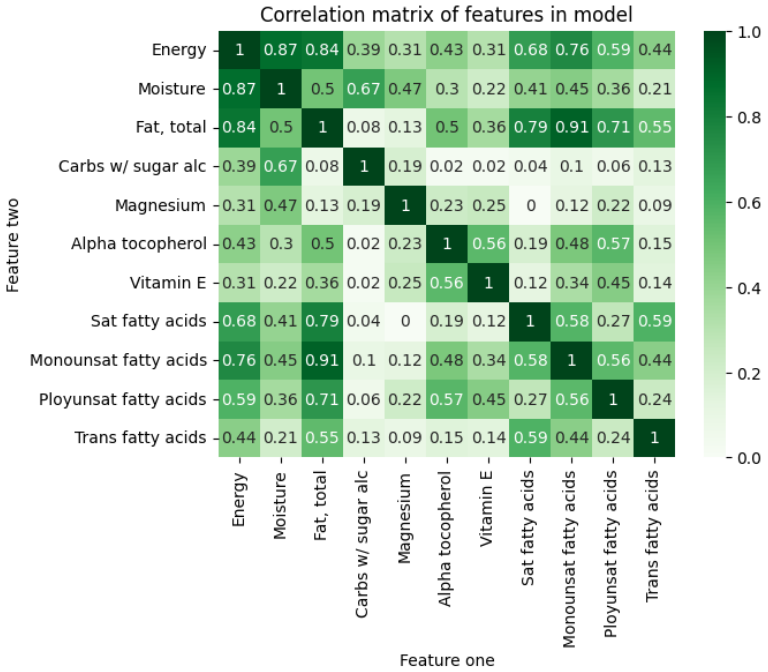


Figure 2: Correlations between Energy and the other features in the model

3 Model selection

Now that we have defined the features we intend to use we can being to construct our models for predicting our target feature, Energy.

Since our target feature, Energy (KJ), is numerical we will be considering a regression problem when developing models to predict our target. We can also note that all our input features are also numerical.

3.1 Linear regression

The logical first choice when both the target feature and all the input features are numerical is to use a linear regression model. A linear regression model has n **numerical** inputs and one **numerical** output and thus is perfect for our case for predicting the numerical feature, Energy. (It should be noted that there are ways to use linear regressing with categorical inputs and outputs but this will not be covered here).

For this report we will first consider a simple model with just 1 input feature, i.e. a *simple linear regression*. Then we will extend this idea to take into account all 10 of our input features, i.e. a *multiple linear regression*.

3.2 Polynomial regression

We can further build on the idea of linear regressions by adding more 'inputs' to our model, with each new input being a polynomial product of our original input features. By doing this we increase the complexity of our model and hope to capture more information in the training data for use in our prediction. This model is useful for our particular data set as it is likely that there are certain interactions between different combinations of nutrients which cannot be captured in a normal linear regression, but can be captured within the cross terms of a polynomial regression.

The reason why this model will be formulated in addition to the original because it will likely be more accurate in its predictions than the linear regression due to the aforementioned reasons.

The reasons behind this models effectiveness for this particular problem are similar to the Linear regression above. Similarly, for this report we will first consider a simple model with just 1 input feature, i.e. a *simple polynomial regression*. Then we will extend this idea to take into account all 11 of our input features, i.e. a *multiple polynomial regression*.

3.3 Models that were not chosen

Two examples of models that are commonly used in machine learning but were not chosen are K Nearest Neighbors and Decision Trees. There are a few reasons why these methods would not be idea for this problem however, the major issue is their limited ability to extrapolate data. Since both methods rely heavily on the local bounds of the training data set to make their predictions, if we wish to predict outside this region (i.e. extrapolate).

For example, with K-NN if we make a prediction outside the domain of the training data there may not be enough neighbouring data points to make an accurate prediction. This will result in the prediction being made as if was at the edge of the training domain, and not further outside it.

This is a major issue that we do not want present in our model. As when testing our model on non-training data it is very likely that we could see higher quantities of certain nutrients (input features) than what was observed in training, due to the real world nature of our problem (nutrient quantities can be highly varied between foods).

4 Linear regression

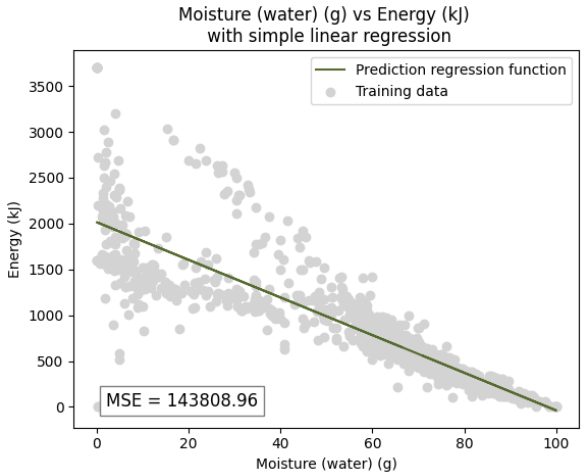
4.1 Simple linear regression

To begin we will start with a simple model using a 1 feature linear regression, with 2 parameters: the intercept term θ_0 and the first input's coefficient θ_1. Therefore our model will be of the form are our parameters:

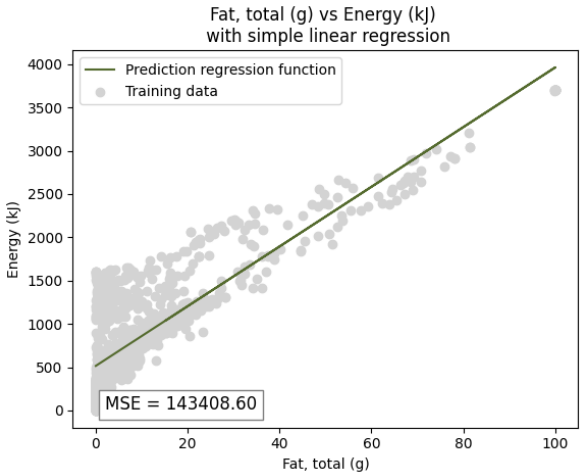
$$\hat{y} = \theta_0 + \theta_1 x_1$$

Where $\hat{y}$ (output)is our predicted value for Energy (KJ), x_1 (input) is our single input feature, and $\underline{\theta} = (\theta_0, \theta_1)^T$. Understanding of how this model works is assumed so will not be detailed here however Chapter 3 in [3] provides a good overview.

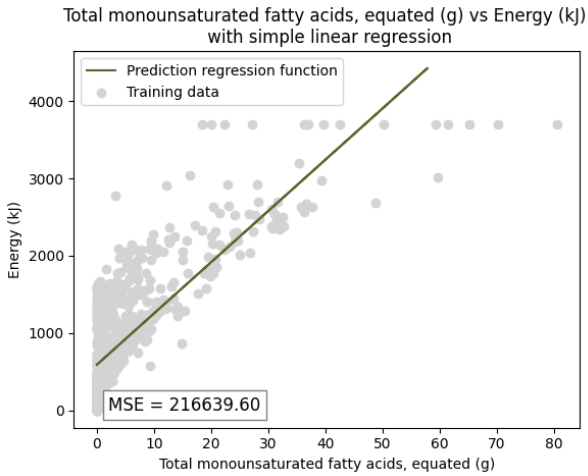
Bellow we have conducted several simple linear regressions for our target, Energy (KJ), each one using a different input feature from our list of input features (Appendix A). The four examples shown bellow are using the four input features with the largest correlation with (therefore the most accurate at predicting) the target feature (see Figure 1).



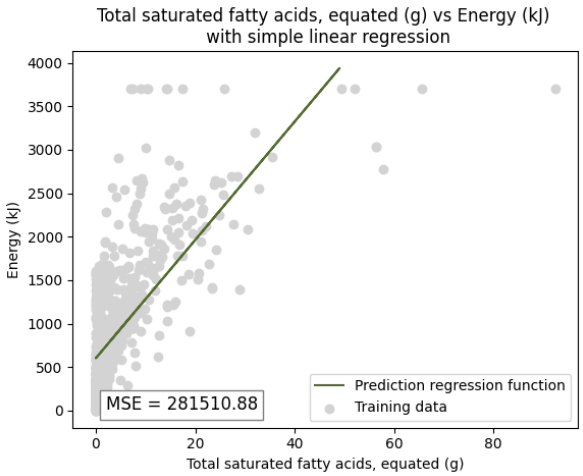
(a) Correlation with Energy (kJ): 0.87



(b) Correlation with Energy (kJ): 0.84



(c) Correlation with Energy (kJ): 0.76



(d) Correlation with Energy (kJ): 0.68

Figure 4: Example simple linear regressions with the four features of highest correlation with the target

```python
for input_feature in df.columns: # for each input feature
    df = df.sample(frac=1) # randomly shuffle data
    X = df.loc[:, [input_feature]] # observations for our input
    y = df['Energy (kJ)'] # observations for our output

    # Assign 70% training data 30% testing data for each data set
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

    # train and fit the linear regression model
    reg = LinearRegression()
    reg.fit(X_train, y_train)

    # apply the linear model to our test data
    y_hat = reg.predict(X_test)
    mse  = mean_squared_error(y_test, y_hat) # calculate MSE
```

Listing 7: Modeling simple linear regressions for each of the input features. Here we have used the package sklearn for the predefined functions used.

For our simple linear regressions, and for all following models in this report we will use Mean Square Error (MSE) of our model against a set of test data (that our model was not trained on) as our primary measure of accuracy. This is due to the fact that we are using only numerical data an dealing with a regression problem. When using numerical data it is standard and common practice to use MSE as an accuracy measure.

On average our lowest error simple linear regression model (least MSE) is when our input feature is 'Moisture (water) (g)'. So this is the model we will use to evaluate our simple linear regression. By running 1000 sample trials of different test train splits on this model it was found that the average MSE is 124510.10.

$$\text{MSE} = 124510.10$$

4.2 Multiple Linear Regression

We can now extend on the simple linear regression and now formulate a multiple linear regression model which incorporates all of our input features. For this model we will have 11 parameters: the intercept term θ_0 and the 10 input coefficient's $\theta_1, \theta_2, \ldots, \theta_{10}$. Therefore our model will be of the form:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_{10} x_{10}$$

Where $\hat{y}$ (output)is our predicted value for Energy (kJ), $\underline{x} = (x_1, x_2, \ldots, x_{10})^T$ (input) are our inputs, and $\underline{\theta} = (\theta_0, \theta_1, \ldots, \theta_{10})^T$ are our parameters. Understanding of how this model works is assumed so will not be detailed here however Chapter 3 in [3] provides a good overview.

We will not be able to provide a nice graph for this as it is hard to visualise 11 dimension however the process is very similar to the simple linear regression.

```
df = df.sample(frac=1) # randomly shuffle data
X = df.drop(columns=df.columns[0]) # observations for inputs (drop target)
y = df['Energy (kJ)'] # observations for our output

# Scale the data to standarise inputs
scaler = StandardScaler()
scaler.fit(X)
X = scaler.transform(X)

all_mse = []
for _ in range(1000):
    # Assign 70% training data 30% testing data for each data set
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

    # train and fit the linear regression model
    reg = LinearRegression()
    reg.fit(X_train, y_train)

    # apply the linear model to our test data
    y_hat = reg.predict(X_test)
    mse  = mean_squared_error(y_test, y_hat) # calculate MSE
    all_mse.append(mse) # add mse to lsit

print(np.mean(all_mse)) # average MSE
```

Listing 8: Modeling multiple linear regression using all of the input features. Here we have used the package sklearn for the predefined functions

Following a similar process to the simple linear regressions we get that our average MSE is equal to 18925.41.

$$\text{MSE} = 18925.41$$

5 Polynomial regression

5.1 Simple polynomial regression

To illustrate the model we again begin with a simple model using a 1 feature polynomial. When doing a polynomial regression we have to choose the maximum degree polynomial which we wish to go up to, we will denote this by p where p is a hyperparameter. For a Simple polynomial

regression of max degree 2 we have 3 parameters: the intercept term θ_0 and the linear term coefficient θ_1 and the squared term coefficient θ_2. Therefore our model will be of the form are our parameters:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$$

Where $\hat{y}$ (output)is our predicted value for Energy (KJ), x_1 (input) is our single input feature, and $\underline{\theta} = (\theta_0, \theta_1, \theta_2)^T$. Understanding of how this model works is assumed so will not be detailed here however Chapter 3 in [3] provides a good overview.

Bellow we have conducted several simple polynomial regressions with maximum degree = 2 for our target, Energy (KJ), each one using a different input feature from our list of input features (Appendix A). The four examples shown bellow are using the four input features with the largest correlation with (therefore the most accurate at predicting) the target feature (see Figure 1).

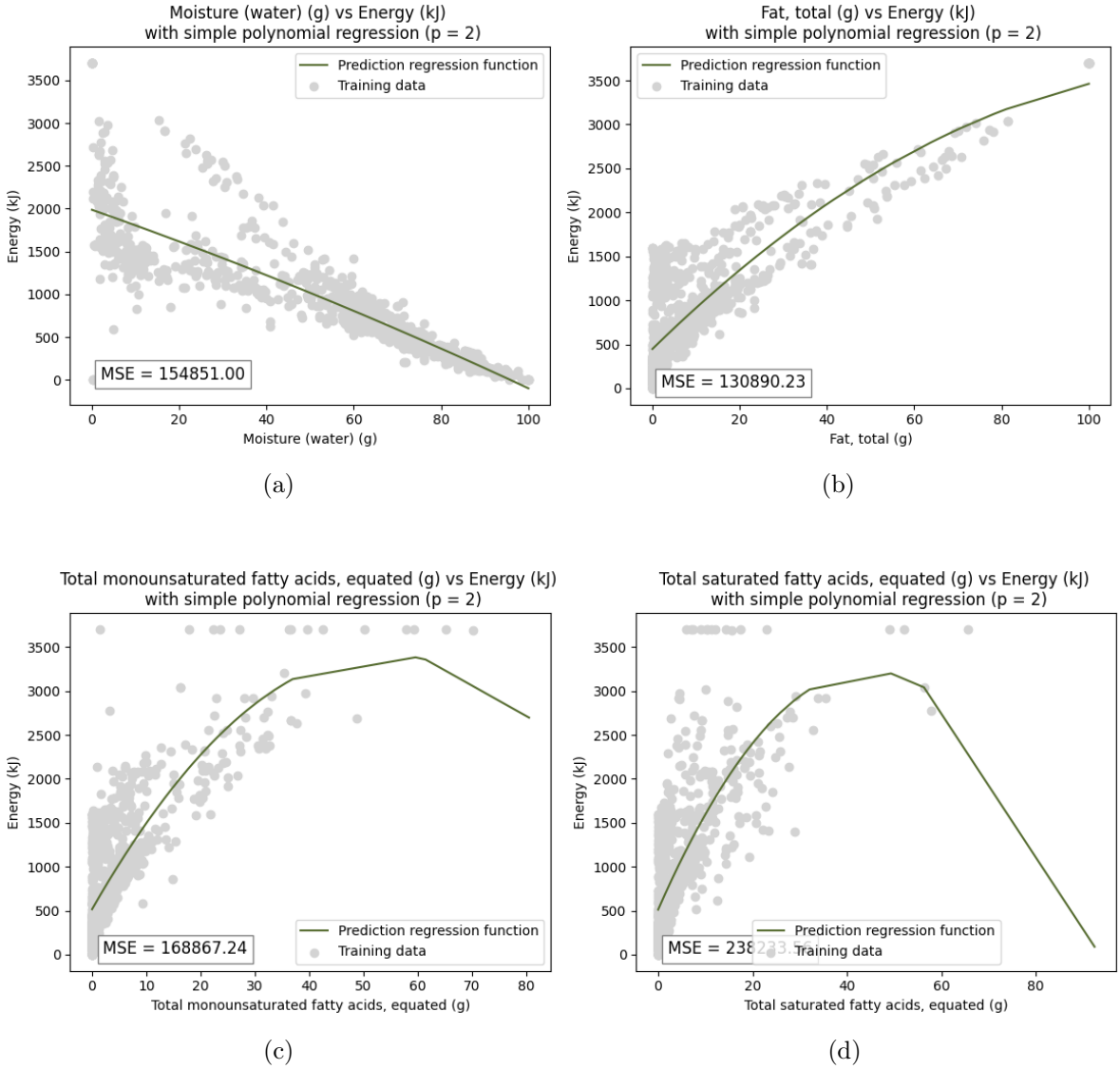


Figure 6: Example simple polynomial regressions with the four features of highest correlation with the target

```
for input_feature in df.columns: # for each input feature
    df = df.sample(frac=1) # randomly shuffle data
    X = df.loc[:, [input_feature]] # observations for our input
    y = df['Energy (kJ)'] # observations for our output

    # Assign 70% training data 30% testing data for each data set
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
    X_test = np.sort(X_test)

    # Create a polynomial feature transformer with degree 2
    transformer = PolynomialFeatures(degree=degree, include_bias=False)
    X_train_poly = transformer.fit_transform(X_train) # adujst X to add
    polynomial terms
    X_test_poly = transformer.fit_transform(X_test)

    # train and fit the linear regression model
    reg = LinearRegression()
    reg.fit(X_train_poly, y_train)

    # apply the linear model to our test data
    y_hat = reg.predict(X_test_poly)
    mse  = mean_squared_error(y_test, y_hat) # calculate MSE
```

Listing 9: Modeling simpe polynomial regressions for each of the input features. Here we have used the package sklearn for the predefined functions used.

On average our lowest error simple polynomial regression model (least MSE) is when our input feature is 'Moisture (water) (g)'. So this is the model we will use to evaluate our simple polynomial regression. By running 1000 sample trials of different test train splits on this model it was found that the average MSE is 118422.08.

$$\text{MSE} = 118422.08$$

5.2 Multiple polynomial Regression

We can now extend on the simple polynomial regression and now formulate a multiple polynomial regression model which incorporates all of our input features. In this model we will have $(p+1)(2\times 10+p)/2+1$ one term for each unique polynomial product (up to degree p) of our 10 input features e.g. (x_3^2 or $x_2x_4x_8^2$)

For this model, for a polynomial of degree p we will have $(p+1)(2\times 10+p)/2+1$ parameters: the intercept term θ_0 and polynomial coefficient's $\theta_1, \theta_2, \ldots$. Therefore our model will be of the form:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots \theta_{(p+1)(2\times 10+p)} x_{10}^p$$

Where $\hat{y}$ (output)is our predicted value for Energy (kJ), $\underline{x} = (x_1, x_2, \ldots, x_{10})^T$ (input) are our inputs, and $\underline{\theta} = (\theta_0, \theta_1, \ldots)^T$ are our parameters. Understanding of how this model works is assumed so will not be detailed here however Chapter 3 in [3] provides a good overview.

We will not be able to provide a nice graph for this as it is hard to visualise 11 dimension however the process is very similar to the simple polynomial regression.

```
df = df.sample(frac=1) # randomly shuffle data
X = df.drop(columns=df.columns[0]) # observations for inputs (drop target)
y = df['Energy (kJ)'] # observations for our output

```

```
# Scale the data to standarise inputs
scaler = StandardScaler()
scaler.fit(X)
X = scaler.transform(X)

all_mse = []
degree = 2
for _ in range(1000):
    # Assign 70% training data 30% testing data for each data set
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

    # Create a polynomial feature transformer with degree 2
    transformer = PolynomialFeatures(degree=degree, include_bias=False)
    X_train = transformer.fit_transform(X_train) # adujst X to add polynomial
    terms
    X_test = transformer.fit_transform(X_test)

    # train and fit the linear regression model
    reg = LinearRegression()
    reg.fit(X_train, y_train)

    # apply the linear model to our test data
    y_hat = reg.predict(X_test)
    mse  = mean_squared_error(y_test, y_hat) # calculate MSE
    all_mse.append(mse) # add mse to lsit

print(np.mean(all_mse)) # average MSE
```

Listing 10: Modeling multiple polynomial regression using all of the input features and p equal to 2. Here we have used the package sklearn for the predefined functions

Following a similar process to the simple polynomial regressions we get that our average MSE for $(p = 2)$ is equal to 7028.20.

$$\text{MSE} = 7028.20$$

5.3 Evaluation of current models

Up to this point we have developed 4 different types of models. To roughly compare their effectiveness we will compare their respective testing MSE. Note that this is not a definite metric of how "effective" each model is (as it may preform badly on data outside of this data set), however it gives us a good idea, given the information we have access to.

We can see in the graph bellow that as the number of parameters within the model increases, the MSE decreases. This makes logical sense as if we add an extra parameter to the model, we have two options we can set it to 0, which will yeild the same MSE, or we can set it to a value which will give a new MSE. If the new MSE is greater than the old MSE then we can just set the parameter to 0 to ensure we keep the lowest MSE we can. Thus, whenever we increase the number of parameters we will expect to see a decrease in MSE, as we now just have additional information to help explain the data.

Initially, the idea to simply have as many parameters as possible to fit the data perfectly may seem like a good plan, however this leads to a major issue. This issue revolves around the models ability to generalise beyond the given data. If we fit the model too perfectly to the training data, the models ability to predict new data maybe be negatively impacted, we call

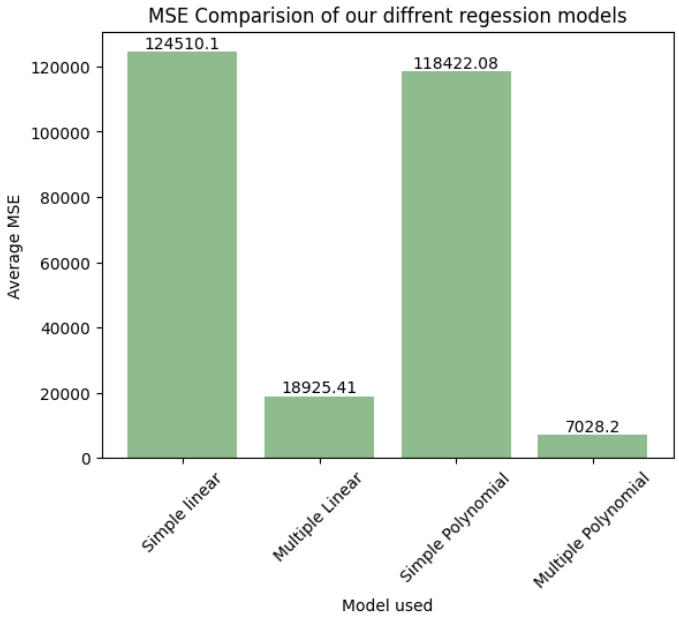


Figure 7: Comparison between MSE of each of the regression models developed

this over fitting. In the following section we will discuss how to improve our model to help it to generalise beyond the given data.

The ability for a model to generalise, and preform well on unseen data is the most important feature of the model, as the aim of any ML model is to be able to predict new data, it is not important how well we can predict our training data since if we want to know the Energy for a training data point we can just look it up.

6 Generalising the model

We can see from Figure 7 that out of the models that we have currently developed, multiple polynomial regression is best suited for our data due to its low MSE, and thus we will be using this model for remainder of this report.

We have now chosen the type of model that we wish to implement, the next job is to refine and optimise it to the best of our abilities. Our main focus, as mentioned in the previous section, will be to improve the models ability to generalise beyond the training data. That is, we wish to minimise the error of the model when it is used on new, unseen data, we call this error.

6.1 Generalisation Gap

There are many ways to measure how good a model is able to 'generalise' however the *Training Error–Generalisation Gap* is a commonly used measure of how well a model is able to generalise beyond the training data.

This gap is defined as the difference between the error when the model is applied to the training data (E_{train}) and the error when the model is applied to new, unseen data (E_{new}),

i.e. $E_{train} - E_{new}$. A large gap indicates that there is likely over fitting, and the model is not generalising well. Whereas a small gap indicates that the models accuracy on training data is similar to its accuracy on unseen data. So, to produce a model that is able to generalise well, we wish to minimise the Generalisation Gap. (Note it is very rare we can evaluate E_new thus E_test, the error in the testing data is used as an estimate)

There are many ways that we could use to attempt to minimise the Generalisation Gap, however it is generality the case that a model's 'complexity' is inversely proportional to a models ability to generalise. That is, as we make a model more complex, the larger the generalisation gap becomes [6]. This is logical as if we increase the complexity of a model too much, we will likely over fit and thus will not able to generalise well.

There are many things that can influence a models complexity however a major factor, as mentioned in Section 5.3, is the number and or weight of the parameters within the model. This makes sense as if we have fewer, or less influential parameters, the model produced will have less complexity. There are two methods that we will use to attempt to minimise the generation gap following the above logic. Namely: L_1 regularisation and L_2 regularisation.

Note: Another way to alter the complexity would be to alter the degree of the polynomial. This was attempted and resulted in a generalisation gap of $> 10^{16}$ for any polynomial greater than degree 2 and thus this avenue will not be pursued.

7 Regularisation

Full understanding of how regularisation works is assumed so will not be detailed here however Chapter 5 in [3] provides a good overview. The idea behind regularisation is that we want to keep our parameters small, or remove them, if they are not making an impactful difference on our model. That is we only want to increase model complexity if it makes a meaningful improvement to our predictions.

In this report we will only consider explicit regularisation (see [3] for more details). To implement explicit regularisation we add a penalty term to our cost function (our cost function is the function we optimise to train our parameters, see [3] for more details) which discourages overly large parameter values. That is, with this penalty term, we will only have large parameter values (impactful on model) if the decrease in MSE is significant enough to warrant it i.e. we maintain small parameter values unless the data heavily conveniences us otherwise. This therefore aims to minimise the complexity of our model.

7.1 L1 Regularisation

L_1 regularisation, also know as *LASSO* regularisation, adds the penalty term $+\lambda||\underline{\theta}||_1$ (where $||\quad||_1 = |\theta_0| + |\theta_1| + \ldots$ is the taxicab norm (see [5])) to our cost function. This regularisation method tends to result in 'sparse' models, which are models where only a few parameters are non-zero.

Here λ is a hyper parameter which we call our regularisation parameter. High values of λ will result in heavy regularisation i.e. parameter values will be strongly encouraged to stay as low as possible. Where as low values of λ will result in light regularisation i.e. parameter values will be only be lightly encouraged to stay low. So as we increase λ our model complexity will decrease.

We can now retrain our original multiple polynomial regression with now the addition of L_1 Regularisation. We test several different values of λ in an aim to determine which value will minimise our generalisation gap.

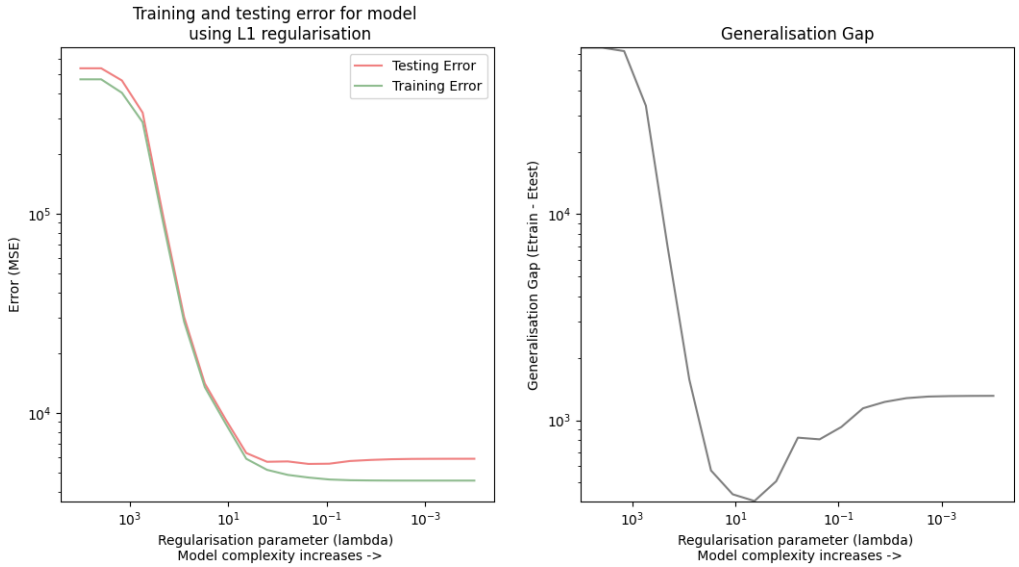


Figure 8: Comparison of Testing and Training Error for different λ values using L_1 regularisation (note that x-axis' are inverted for easier understating of increasing model complexity)

```
lambdas = np.logspace(-4, 4, num=20)
test_mse_per_lambda = []
train_mse_per_lambda = []
for lambda_l1 in lambdas:
    # train and fit the regression model
    reg = Lasso(alpha=lambda_l1, max_iter=10000) # alpha is our regularisation
     param
    reg.fit(X_train, y_train)

    # apply the regularisation model to our data
    y_hat_test = reg.predict(X_test)
    y_hat_train = reg.predict(X_train)

    # calculate MSE
    test_mse = mean_squared_error(y_test, y_hat_test)
    train_mse = mean_squared_error(y_train, y_hat_train)

    # add MSE to array
    test_mse_per_lambda.append(np.mean(test_current_lambda_mse))
    train_mse_per_lambda.append(np.mean(train_current_lambda_mse))
```

Listing 11: Modeling multiple polynomial regression using L1 regularisation

We can note that when L_1 regularisation it appears a value between (0.1, 10) for λ would minimise the generalisation gap. Thus if we want our model to generalise to unseen data a value within this range would be ideal. These results will be further be discussed in Section 7.3

7.2 L2 Regularisation

L_1 regularisation, also know as *Ridge* regularisation, adds the penalty term $+\lambda||\underline{\theta}||_2^2$ (where $||\quad||_2$ is the standard L_2 norm, or Euclidean norm (see [2])) to our cost function. This

regularisation method tends to push all parameters to small values. λ is defined the same as in L_1.

We can now retrain our original multiple polynomial regression with now the addition of L_1 Regularisation. We test several different values of λ in an aim to determine which value will minimise our generalisation gap.

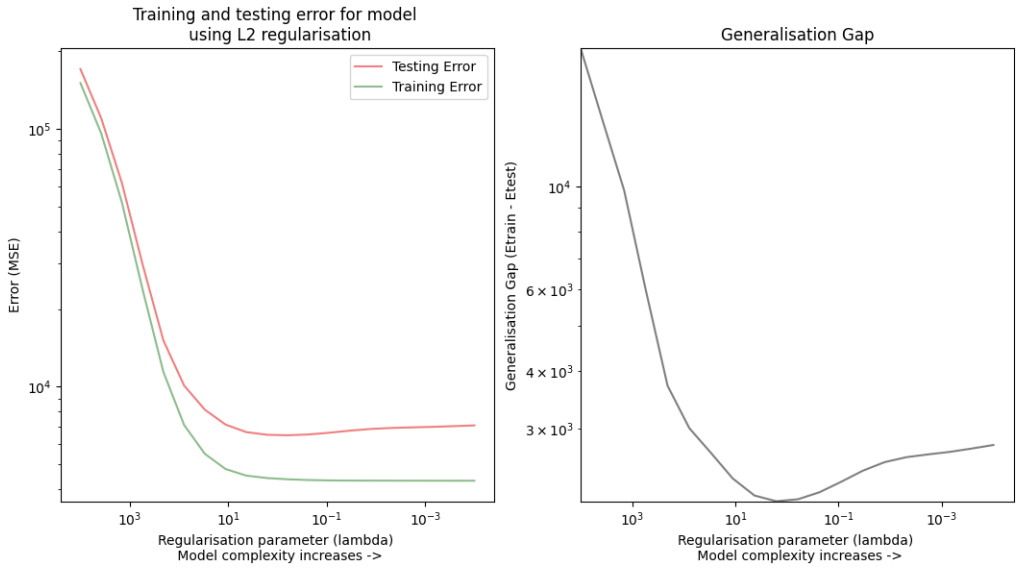


Figure 9: Comparison of Testing and Training Error for different λ values using L_1 regularisation (note that x-axis' are inverted for easier understanding of increasing model complexity)

```
lambdas = np.logspace(-4, 4, num=20)
test_mse_per_lambda = []
train_mse_per_lambda = []
for lambda_l1 in lambdas:
    # train and fit the regression model
    reg = Ridge(alpha=lambda_l1, max_iter=10000) # alpha is our regularisation
     param
    reg.fit(X_train, y_train)

    # apply the regularisation model to our data
    y_hat_test = reg.predict(X_test)
    y_hat_train = reg.predict(X_train)

    # calculate MSE
    test_mse = mean_squared_error(y_test, y_hat_test)
    train_mse = mean_squared_error(y_train, y_hat_train)

    # add MSE to array
    test_mse_per_lambda.append(np.mean(test_current_lambda_mse))
    train_mse_per_lambda.append(np.mean(train_current_lambda_mse))
```

Listing 12: Modeling multiple polynomial regression using L2 regularisation

We can note that when L_2 regularisation it also appears a value between (0.1, 10) for λ would minimise the generalisation gap. Thus if we want our model to generalise to unseen data a value within this range would be ideal. These results will be further be discussed in Section 7.3

7.3 Evaluation of regularised models

When looking at both models we can see that as we decrease λ i.e. increase model complexity, the training error is consistently . This makes sense as by decreasing regularisation (increased complexity) the model is allowed to have a better fit to the training data.

However when looking at the testing data we can see that the testing error decreases consistently up until we hit the point of minimum generation gap and then the testing error begins to increase.

We can consider this section of decreasing error as when the model is 'under fitting', where we need to increase the model complexity in order to capture all the information that the data is providing.

When we start to see the error increasing this means that the model has begun to 'over fit' as we have increase the complexity too far and now the model is too well molded to just the training data.

The section where our training error is minimised, and our generalisation gap is the smallest (some λ between (0.1, 10)) is our so called 'perfect fit' where our model is best suited to generalise to unseen data.

So we know that to ensure that our model is able to generalise to unseen data (which is the goal for all models), we should choose a regularisation parameter between (0.1 , 10) however we still have quite a few questions. Namely: which regularisation method should we use (L_1 or L_2)? and what is the testing MSE of our new model after regularisation? To answer these questions we will use Hold-out validation.
a method called K-fold cross validation.

8 Hold-out validation

A logical thought after seeing the previous information may be to simply just choose what ever λ minimises the test error (E_{test}), however if we do this, then we have just trained our model on our test data (meaning it is now just apart of our training data) which means that we have no true test data left to evaluate our models performance.

So as a work around to this problem we can set aside a second 'test data set' called our hold-out validation set. Which we can use to optimise λ since our models error against this set $E_{hold-out}$ will also approximate E_{new}.

So we will split out data into 3 sets: training, hold-out and testing. We will preform the same process as before where we train our data on the training set for a specific λ. Then we will calculate our $E_{hold-out}$. We will repeat this for many different λ values and choose the value of λ that minimises $E_{hold-out}$ and use this lambda for our final model (i.e. is able to generalise to unseen data the best). Then we can finally evaluate the final model's performance on the testing set.

```
# Assign 60% training data
X_train, X_other, y_train, y_other = train_test_split(X, y, test_size=0.4)

# Assign 20% testing data 20% hold out data from orignal set
```

```
X_ho, X_test, y_ho, y_test = train_test_split(X, y, test_size=0.5)

# Create a polynomial feature transformer with degree 2
transformer = PolynomialFeatures(degree=2, include_bias=False)
X_train = transformer.fit_transform(X_train) # adujst X to add polynomial terms
X_test = transformer.fit_transform(X_test)
X_ho = transformer.fit_transform(X_ho)

lambdas = np.logspace(-1, 1, num=30)
ho_mse_per_lambda = []
train_mse_per_lambda = []
for lambda_l1 in lambdas:
    # train and fit the linear regression model
    reg = Lasso(alpha=lambda_l1, max_iter=10000) # alpha is our regularisation param
    reg.fit(X_train, y_train)

    # apply the model to our test data
    y_hat_ho = reg.predict(X_ho)
    y_hat_train = reg.predict(X_train)

    ho_mse = mean_squared_error(y_ho, y_hat_ho) # calculate MSE
    train_mse = mean_squared_error(y_train, y_hat_train)

    #print(f'lambda: {lambda_l1}: {np.mean(current_lambda_mse)}') # average MSE
    ho_mse_per_lambda.append(ho_mse)
    train_mse_per_lambda.append(train_mse)

for i in range(len(ho_mse_per_lambda)):
    if ho_mse_per_lambda[i] == min(ho_mse_per_lambda): # if minimum E_ho
        # train and fit the best regression model
        reg = Lasso(alpha=lambdas[i], max_iter=10000) # alpha is our regularisation param
        reg.fit(X_train, y_train)

        # apply the linear model to our test data
        y_hat_test = reg.predict(X_test)
        test_mse = mean_squared_error(y_test, y_hat_test) # calculate MSE
```

Listing 13: Using a hold-out validation data set to determine the optimal value of lambda for L1 regularisation

This same method was preformed for L_2 regularisation as well. Using this method we can determine that for L_1 regularisation, a λ value of $\lambda = 0.303$ was found to minimise E_new and resulted in a testing error (MSE) of: 6125.52.

$$L_1 \text{ MSE} = 6125.52$$

For L_2 regularisation, a λ value of $\lambda = 1.487$ was found to minimise E_new and resulted in a testing error (MSE) of: 6375.50.

$$L_2 \text{ MSE} = 6375.50$$

We can note that L_1 appears to have better performance when applied to our polynomial model. This is likely because many of the polynomial terms provide no additional information and thus the those corresponding coefficient parameters should be set to 0. This means that the space model as produced by L_1 would be favoured.

Note that these outcomes are just the case for this particular run of the experiment however due to the random nature of test train splits there may be different outcomes for different trials. (We did run the above experiment multiple times and noticed little deviation in results so these are likely fairly accurate)

9 K-fold cross validation

K-fold cross validation is a technique which allows us analyse the expected value and variance in the testing error generated by a model. Full understanding of how K-fold cross validation works is assumed so will not be detailed here however Chapter 5 in [3] provides a good overview.

However, in summary, we split our data up into k batches, and then run through the training and testing of the model k times. Each time we will set a different one of our batches as our test data and the rest of the batches as training data, each iteration we will calculate the test error (MSE). After this process is completed we will have a sample of k test errors which we can apply statistical functions to.

```
from sklearn.model_selection import KFold
# define the number of folds for cross-validation
num_folds = 5
kf = KFold(n_splits=num_folds)

# loop through each fold
for train_index, test_index in kf.split(X):
    # get the training and testing data for this fold
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train and fit the linear regression model
    reg = LinearRegression()
    reg.fit(X_train, y_train)

    # apply the linear model to our test data
    y_hat = reg.predict(X_test)
    mse  = mean_squared_error(y_test, y_hat) # calculate MSE
    allmse.append(mse)
print(f'Mean: {np.mean(allmse)} SD: {np.sqrt(np.var(allmse))}')
```

Listing 14: Using k-fold cross validation on our simple linear regression model with 5 folds. Noting: KFold is from the package sklearn

This process was repeated for all our developed models and the results are shown in the graph and table below.

10 Model evaluation and Conclusion

After developing and refining several models, we can see, based on the experiments and analysis conducted, that our 'Multiple polynomial regression with L_1 regularisation' is best fit for predicting our target feature, Energy. This is because it has both: The lowest mean testing error (meaning the model will be able to generalise and therefore predict new unseen data with the highest accuracy). And the lowest standard deviation in the testing error (meaning that when applied to different data sets its prediction ability is consistent, i.e. predict unseen data more consistently).

Model	Mean of testing error	Standard deviation of testing error
Simple linear	124510.10	20471.11
Multiple Linear	18925.41	4852.92
Simple Polynomial	118422.08	37505.93
Multiple Polynomial	7028.20	2196.84
Polynomial L1 Reg	6125.52	989.26
Polynomial L2 Reg	6375.50	2287.14

Table 1: Table of results from the application of k-fold cross validation to our regression models ($k = 5$)

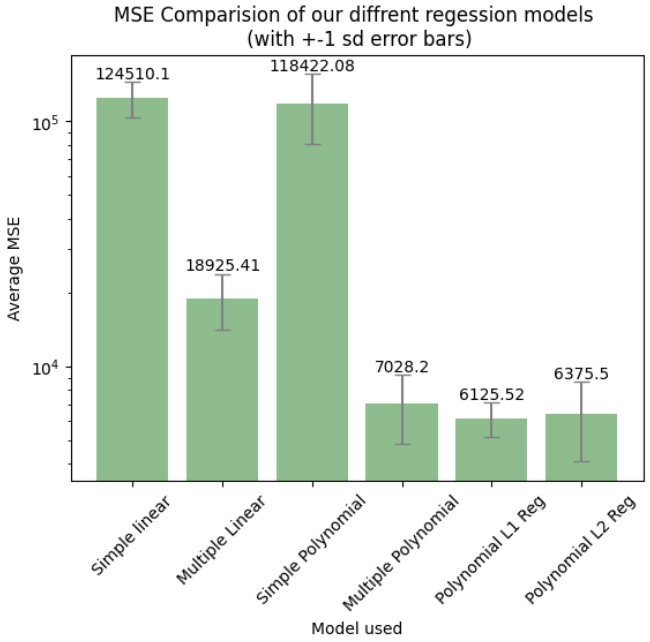


Figure 10: Comparison between average MSE of each of the regression models with ±1 standard deviation error bars (from the use of 5-fold cross validation). Number above each bar is the bar height (avg MSE).

This is a fairly good prediction however it up to the user of the model if this is accurate enough for their purposes.

Finally, we must keep in mind that these conclusions are the result of testing on a subset of the population data, and it is not possible to know how models will preform when given unseen data. However, given the information we have and under the assumption this data is a good representation of the population, these conclusions can be considered reasonable and valid.

References

[1] Dennis E Hinkle, William Wiersma, and Stephen G Jurs. *Applied statistics for the behavioral sciences*. 5th. Houghton Mifflin, 2003. ISBN: 9780618124053.

[2] David C. Lay and Judith J. McDonald. *Linear Algebra and Its Applications*. 5th. Pearson, 2016. URL: https://www.pearson.com/us/higher-education/program/Lay-Linear-Algebra-and-Its-Applications-5th-Edition/PGM252646.html.

[3] Andreas Lindholm et al. *MACHINE LEARNING: A First Course for Engineers and Scientists*. Cambridge University Press, 2022. URL: https://web.stanford.edu/~boyd/cvxbook/.

[4] Sally D Poppitt and Ann M Prentice. "Energy density and its role in the control of food intake: Evidence from metabolic and community studies". In: *Appetite* 26.2 (1996), pp. 153–174. DOI: 10.1006/appe.1996.0012.

[5] Walter Rudin. *Principles of Mathematical Analysis*. 3rd. McGraw-Hill, 1976. URL: https://www.mheducation.com/highered/product/principles-mathematical-analysis-rudin/M0070856133.html.

[6] Chiyuan Zhang et al. "Understanding deep learning requires rethinking generalization". In: *Proceedings of the International Conference on Learning Representations (ICLR)*. 2017. URL: https://openreview.net/forum?id=Sy8gdB9xx.

A Full list of features used in model

Target feature:
Energy (kJ)

Input features:
Moisture (water) (g)
Fat, total (g)
Available carbohydrate, with sugar alcohols (g)
Magnesium (Mg) (mg)
Alpha tocopherol (mg)
Vitamin E (mg)
Total saturated fatty acids, equated (g)
Total monounsaturated fatty acids, equated (g)
Total polyunsaturated fatty acids, equated (g)
Total trans fatty acids, imputed (mg)