05_regression_multiple_regression

May 7, 2025

[ ]:
```
import os
user = os.getenv('USER')
os.chdir(f'/scratch/cd82/{user}/notebooks')
```

0.1 Linear Regression - Multiple Linear Regression (MLR)

In this section: - We look at models with more than a single independant/predictor variable. - We will use a train-test split of the data so as to test our model. - We also look at how to select the best predictor variables using t-test scores and their p-values - We will look at automated methods of reducing the number of predictor variables using *Regularisation*.

The multiple linear regression equation:

$$y = \beta_0 + x_1\beta_1 + x_2\beta_2 + ... + x_n\beta_n + \epsilon$$

Where: - y is the dependent variable that we're trying to predict - x_1, x_2, ..., x_n are the independent variables - β_0 is the y-intercept (bias) - β_1, β_2, ..., β_n are the coefficients (weights) for each feature - ϵ is the error term **In matrix form**:

$$\mathbf{y} = \mathbf{X}\beta + \epsilon$$

$\mathbf{y}$ A (column) vector of responses $\mathbf{X}$ A matrix of independent variables (row for each sample, column for each independant variable) β A vector of coefficients.

Equation to solve for β:

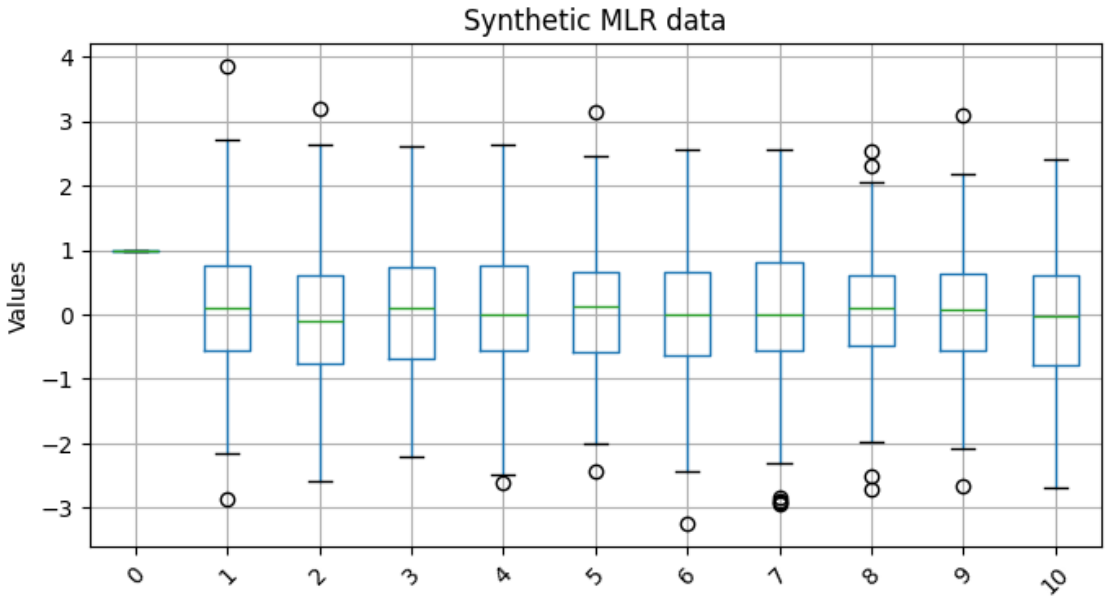
$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

[1]:
```
import numpy as np
import pandas as pd
import statsmodels.api as sm
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from matplotlib import pyplot as plt
```

0.1.1 Generate data

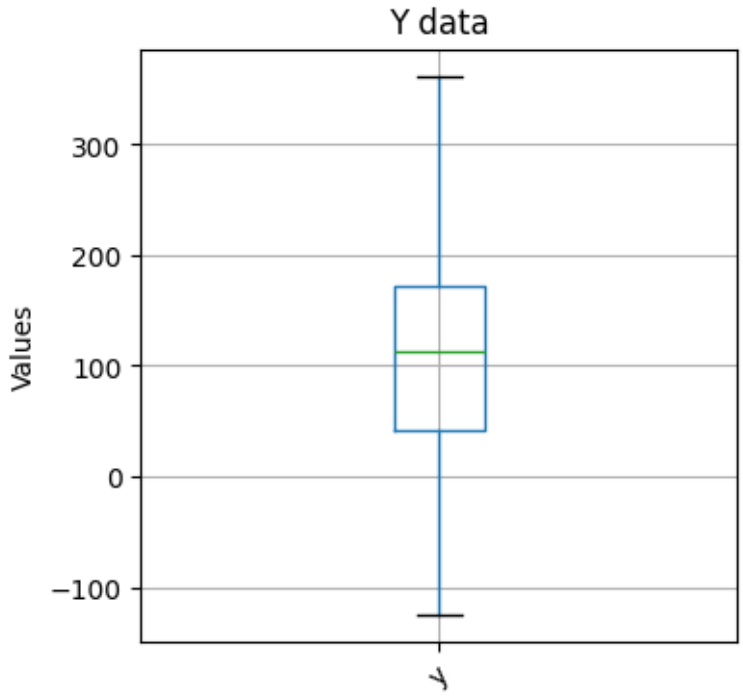
Again, our example is going to use synthetic data i.e. data derived from a known distribution. This time we will use the Skikit-Learn function `make_regression`

[2]:
```
from sklearn.datasets import make_regression
# Generate synthetic data
X, y = make_regression(
    n_samples=200,
    n_features=10,
    n_informative=5,
    noise=5,
    bias=100.0,
    random_state=42)

# Add a constant to the model (intercept)
X_int = sm.add_constant(X)

plt.figure(figsize=(8, 4))

# Create a box and whisker plot for each feature
X_df = pd.DataFrame(X_int)

# Create a box and whisker plot for each feature
X_df.boxplot()
plt.title('Synthetic MLR data')
plt.xticks(rotation=45)
plt.ylabel('Values')
plt.grid(True)
plt.show()
```



Plot the Y data

[3]:
```
plt.figure(figsize=(4, 4))

# Create a box and whisker plot for each feature
# X_df = pd.DataFrame(X_int)
y_df = pd.DataFrame(y, columns=['y'])

# Create a box and whisker plot for each feature
y_df.boxplot()
plt.title('Y data')
plt.xticks(rotation=45)
plt.ylabel('Values')
plt.grid(True)
plt.show()
```



Splitting our input data We will split our dataset into a *training* set and a *testing* set. - This helps prevent our models from being 'over-fit' by the training data. - The split out data means we can validate our model with data that is external to the data the model was trained on. - It helps to produce models that are transferable to new data.

Over-fitting datasets is not such a problem in linear regression methods, however gets more impor-

tant in more complex machine learning methods.

[4]:
```
# Split the data
# from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X_int, y,
    test_size=0.2,
    random_state=42)

print('X_test shape: ', X_test.shape)
print('y_test shape: ', y_test.shape)
```

```
X_test shape:  (40, 11)
y_test shape:  (40,)
```

MLR regression using statsmodels

[5]:
```
import statsmodels.api as sm
import pandas as pd
import numpy as np

# Fit the linear regression model
model = sm.OLS(y_train,X_train)
results_mlr = model.fit()

# Get the R-squared value
r_squared = results_mlr.rsquared
print('R sqrd (extracted):', r_squared)

# Print the summary of the model
print(results_mlr.summary())
```

```
R sqrd (extracted): 0.9977455871802959
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.998
Model:                            OLS   Adj. R-squared:                  0.998
Method:                 Least Squares   F-statistic:                     6594.
Date:                Wed, 07 May 2025   Prob (F-statistic):          9.18e-192
Time:                        09:54:52   Log-Likelihood:                -473.36
No. Observations:                 160   AIC:                             968.7
Df Residuals:                     149   BIC:                             1003.
Df Model:                          10
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         99.3558      0.394    252.193      0.000      98.577     100.134
x1             0.3794      0.382      0.994      0.322      -0.375       1.133
```

```
x2            -0.4513      0.388     -1.162      0.247      -1.219       0.316
x3            10.1054      0.396     25.549      0.000       9.324      10.887
x4            80.6067      0.401    201.075      0.000      79.815      81.399
x5             0.2732      0.399      0.684      0.495      -0.516       1.062
x6            35.2298      0.395     89.179      0.000      34.449      36.010
x7             7.1006      0.364     19.510      0.000       6.381       7.820
x8            40.7213      0.424     96.000      0.000      39.883      41.559
x9             0.3021      0.430      0.702      0.484      -0.548       1.152
x10           -0.2760      0.367     -0.753      0.453      -1.000       0.448
==============================================================================
Omnibus:                        0.373   Durbin-Watson:                   1.927
Prob(Omnibus):                  0.830   Jarque-Bera (JB):                0.500
Skew:                          -0.099   Prob(JB):                        0.779
Kurtosis:                       2.812   Cond. No.                         1.53
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

N.B. Hight absolute t-test scores and small p-values indicate the predictor variable is significant - i.e. the variable is contributing to the predictive power of the model.

[6]:
```
# Make predictions
y_pred_mlr = results_mlr.predict(X_test)
```

[7]:
```
# Evaluate the model on test data
y_test_ave = np.average(y_train)
mse = mean_squared_error(y_test, y_pred_mlr)
r2 = r2_score(y_test, y_pred_mlr)

print(f"Average Y value: {y_test_ave}")
print(f"Mean Squared Error: {mse}")
print(f"Mean Error: {np.sqrt(mse)}")
print(f"R² Score: {r2}")
```

```
Average Y value: 107.79062822578912
Mean Squared Error: 20.319738032770893
Mean Error: 4.507742010449455
R² Score: 0.9965657249753515
```

0.2 Visualisation

Slightly modified the above code to add `residuals` and model `coefficients`.

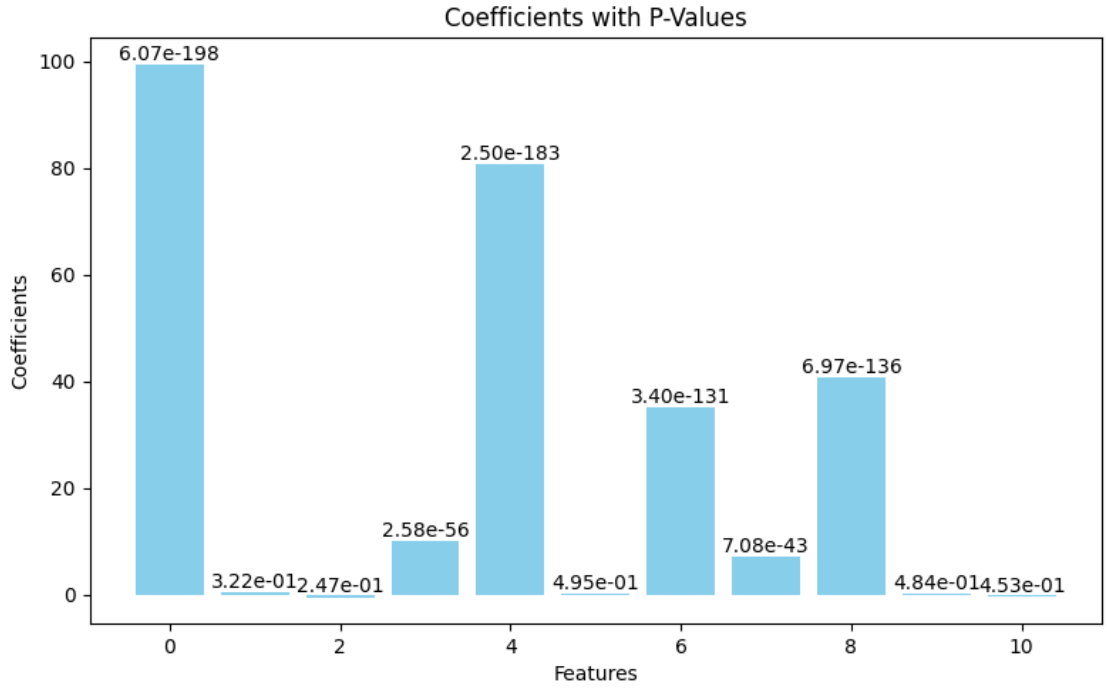
[8]:
```
# added residuals
residuals = y_test - y_pred_mlr

# Evaluate the model
```

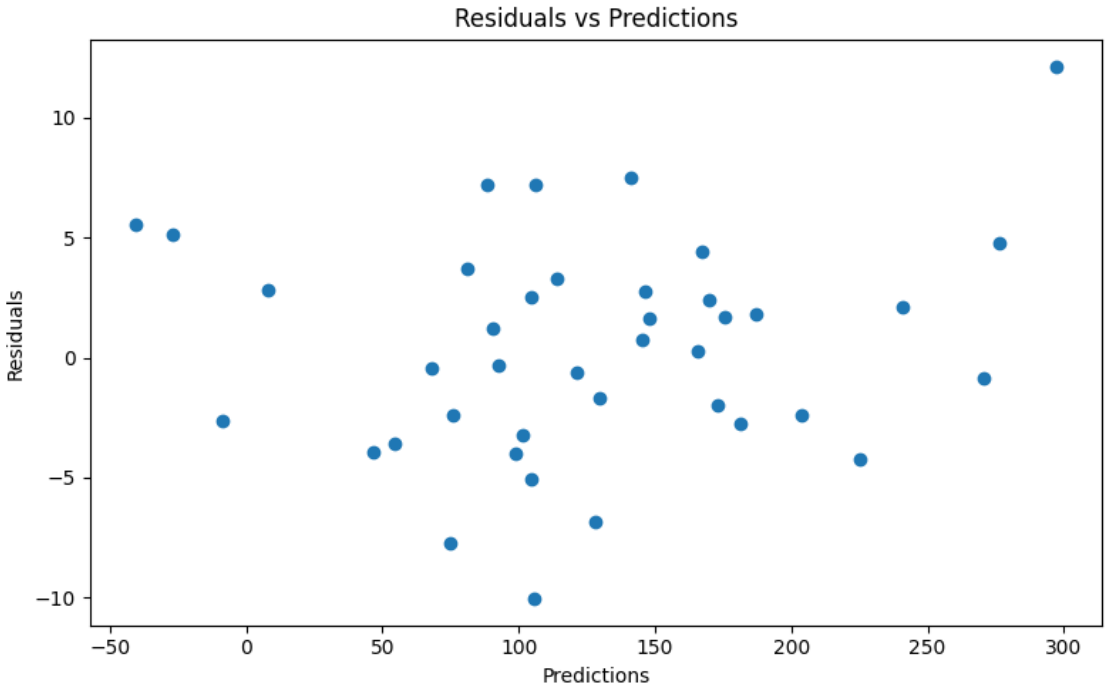
```
mse = mean_squared_error(y_test, y_pred_mlr)
r2 = r2_score(y_test, y_pred_mlr)
# coefficients = model.coef_
coefficients = results_mlr.params
# Create a DataFrame with coefficients and p-values
summary_table = pd.DataFrame({
    'Coefficient': results_mlr.params,
    'P-Value': results_mlr.pvalues
})
print(summary_table)
```

```
    Coefficient        P-Value
0     99.355761  6.073601e-198
1      0.379395   3.217264e-01
2     -0.451331   2.471640e-01
3     10.105365   2.576636e-56
4     80.606689  2.503043e-183
5      0.273211   4.948782e-01
6     35.229828  3.396701e-131
7      7.100571   7.077560e-43
8     40.721302  6.965020e-136
9      0.302116   4.835194e-01
10    -0.275974   4.527504e-01
```

[9]:
```
# Create a figure and a set of subplots
fig, axs = plt.subplots(2, 1, figsize=(8, 10))

# Coefficients plot
# axs[0].bar(X.columns, coefficients)
bars = axs[0].bar(summary_table.index, summary_table['Coefficient'],␣
 ↪color='skyblue')
axs[0].set_title('Feature Coefficients')

# Add p-values on each bar
for bar, p_value in zip(bars, summary_table['P-Value']):
    height = bar.get_height()
    axs[0].text(bar.get_x() + bar.get_width() / 2, height,
            f'{p_value:.2e}', ha='center', va='bottom')

axs[0].set_xlabel('Variables')
axs[0].set_ylabel('Coefficients')
axs[0].set_title('Coefficients with P-Values')

axs[0].set_xlabel('Features')
axs[0].set_ylabel('Coefficients')

# Residuals vs Predictions plot
```

```
axs[1].scatter(y_pred_mlr, residuals)

axs[1].set_title('Residuals vs Predictions')
axs[1].set_xlabel('Predictions')
axs[1].set_ylabel('Residuals')

# Show the plots
plt.tight_layout()
plt.show()
```





The first plot is a bar chart of feature coefficients. This gives a good view of the effect each feature has on the prediction. A feature with a higher coefficient has a larger effect than a feature with a lower coefficient.

The second plot is a scatter plot of residuals versus predictions. Ideally, the residuals should be randomly spread around the centerline. If there are any patterns in the residuals, it signals that

the model could be improved by including non-linear terms or interaction terms.

0.2.1 Regularisation

Regularisation methods use differing penalisation functions, based around 'distance' calculations of the coefficients.

Vector norms Some common distance metrics (vector norms) are list here:

L1 norm (Manhattan distance or Taxicab distance):

$$L_1 = ||\mathbf{v}||_1 = \sum_{i=1}^{n} |v|$$

L2 norm (Euclidean norm):

$$L_2 = ||\mathbf{v}||_2 = \sqrt{\sum_{i=1}^{n} v^2}$$

L infintiy norm:

$$L_\infty = ||\mathbf{v}||_\infty = \max_i |v|$$

0.3 Lasso (*L1*) Regression

Cost Function = RSS + $\times$ (sum of absolute values of coefficients)

$$CostFunction = \sum_{i=1}^{n} (y - \hat{y}_i)^2 + \alpha L_1$$

Where: - α (alpha) is the regularization parameter.

A higher value of α increases the penalty on large coefficients.

Lasso regression needs to solved using a gradient descent type algorithm.

```
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X, y)
lasso_reg.predict([[1.5]])
```

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(penalty="l1")
sgd_reg.fit(X, y.ravel())
sgd_reg.predict([[1.5]])
```

0.4 Ridge (*L2*) Regression

Cost Function = RSS + $\alpha \times$ (sum of squared values of coefficients = (L_2^2))
(N.B. the removal of the square root)

$$CostFunction = \sum_{i=1}^{n}(y - \hat{y}_i)^2 + \alpha L_2^2$$

Where: - α (alpha) is the regularization parameter

A higher value of α increases the penalty on large coefficients.

Ridge regression has a closed form. To solve for β:

$$\beta = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{A})^{-1}\mathbf{X}^T\mathbf{y}$$

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

In **A** the top left entry is 0 so we do not penalise the offset (intercept) value.

```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1, solver="cholesky")
ridge_reg.fit(X, y)
ridge_reg.predict([[1.5]])
```

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(penalty="l2")
sgd_reg.fit(X, y.ravel())
sgd_reg.predict([[1.5]])
```

0.5 Elasticnet (*L1 & L2*) Regression

$$CostFunction = \sum_{i=1}^{n}(y - \hat{y}_i)^2 + r\alpha L_1 + (1-r)\alpha L_2^2$$

In long form:

$$CostFunction = \sum_{i=1}^{n}(y - \hat{y}_i)^2 + r\alpha\sum_{i=1}^{n}|\beta| + (1-r)\alpha\sum_{i=1}^{n}\beta^2$$

Where: - r is the mix ratio of $L1$ to $L2$

```
from sklearn.linear_model import ElasticNet
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_net.fit(X, y)
elastic_net.predict([[1.5]])

# or using statsmodels
import statsmodels.api as sm
# L1_wt == 1.0 then it is Lasso
```

```
# L1_wt == 0.0 then it is Ridge regression
lasso_model = sm.OLS(y, X).fit_regularized(method='elastic_net',
                                           alpha=0.1, L1_wt=1.0)

ridge_model = sm.OLS(y, X).fit_regularized(method='elastic_net',
                                           alpha=0.1, L1_wt=0.0)
```

Selection of coefficents in the original dataset using Lasso

[10]:
```
lasso_model = sm.OLS(y_train, X_train).fit_regularized(method='elastic_net',
                                           alpha=1.0, L1_wt=1.0)

non_zero_coefficients = np.sum(lasso_model.params != 0)
print(f"Number of non-zero coefficients: {non_zero_coefficients}")
```

```
Number of non-zero coefficients: 6
```

[11]:
```
# Make predictions
y_test_pred = lasso_model.predict(X_test)
# print("Predictions:", y_test_pred)

mse = mean_squared_error(y_test, y_test_pred)
r2 = r2_score(y_test, y_test_pred)
print(f"Mean Squared Error: {mse}")
print(f"R² Score: {r2}")
```

```
Mean Squared Error: 31.782233774030566
R² Score: 0.9946284282060299
```

[12]:
```
# Extract non-zero coefficients
non_zero_indices = np.where(lasso_model.params != 0)[0]
selected_predictors = X_train[:, non_zero_indices]  # subset our full dataset␣
 ↪with our selected

# Optionally refit the model with selected predictors
refit_model = sm.OLS(y_train, selected_predictors).fit()

print("Selected predictors:", non_zero_indices)
print("Refitted model summary:", refit_model.summary())
```

```
Selected predictors: [0 3 4 6 7 8]
Refitted model summary:                             OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.998
Model:                            OLS   Adj. R-squared:                  0.998
Method:                 Least Squares   F-statistic:                 1.332e+04
Date:                Wed, 07 May 2025   Prob (F-statistic):          4.65e-201
Time:                        09:56:05   Log-Likelihood:                -475.20
No. Observations:                 160   AIC:                             962.4
```

```
Df Residuals:                      154   BIC:                             980.9
Df Model:                            5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         99.4847      0.384    258.848      0.000      98.725     100.244
x1            10.1402      0.389     26.064      0.000       9.372      10.909
x2            80.7065      0.395    204.490      0.000      79.927      81.486
x3            35.2893      0.384     91.839      0.000      34.530      36.048
x4             7.1670      0.356     20.119      0.000       6.463       7.871
x5            40.6240      0.418     97.078      0.000      39.797      41.451
==============================================================================
Omnibus:                        1.224   Durbin-Watson:                   1.929
Prob(Omnibus):                  0.542   Jarque-Bera (JB):                1.279
Skew:                          -0.204   Prob(JB):                        0.528
Kurtosis:                       2.839   Cond. No.                         1.39
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

Finding the best values in the regularisation As we now have various hyperparameters to choose, the modelling task gets a little more laborious. Tha is where GridSearchCV comes in.

[13]:
```python
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Lasso

# Set up the parameter grid
param_grid = {
    'alpha': [0.1, 0.25, 0.5, 0.75, 1.0, 5.0, 10.0, 20.0],
}

# Initialize the Lasso model
lasso = Lasso()

# Use GridSearchCV to find the best parameters
grid_search = GridSearchCV(estimator=lasso, param_grid=param_grid,
                           cv=5, scoring='r2')
grid_search.fit(X_train, y_train)

# Print the best parameters and the best score
print("Best parameters found: ", grid_search.best_params_)
print("Best cross-validation score (r^2): ", grid_search.best_score_)

# Fit the model with the best parameters
```

```
best_lasso = grid_search.best_estimator_
best_lasso.fit(X_train, y_train)

coefficients = best_lasso.coef_
non_zero_indices = np.where(lasso_model.params != 0)[0]
print("non_zero_indices coefficients:", non_zero_indices)


# Make predictions on the test set
y_pred_gridcv = best_lasso.predict(X_test)

# print("Predictions on test data: ", y_pred_gridcv)
```

```
Best parameters found:  {'alpha': 0.25}
Best cross-validation score (r^2):  0.9972419417539845
non_zero_indices coefficients: [0 3 4 6 7 8]
```

[ ]: