STAT40800 Data Programming with Python Project

Fanahan McSweeney (Student No: 20203868)

December 6, 2020

Assessing the Performance of Regression Algorithms in Prediciting the Total Number of COVID-19 Cases Reported by a Given Date per 10,000 Population in a Region

1 Abstract

An extensive explanatory data analysis was performed on a number of variables that were related to counties in Ireland, and countries, in an effort to identify variables that could be used to predict the total number of reported COVID-19 cases per 10,000 population in a given region. For the given data, population density and Province were found to be the strongest indicators of the total number of reported cases per 10,000 population for Irish counties, while number of deaths of children under 10 per 10,000 population, income group and continent were identified as the strongest indicators of the total number of reported cases per 10,000 population for countries.

Various regression models were created and tested on a random subset of the data in order to find the best regression technique for predicting the total number of reported COVID-19 cases per 10,000 population with the available data. K-fold cross validation was used to evaluate the performance of each model using a range of input parameters in order to identify optimum parameter values. The final optimised models were tested on the randomly selected test set and the root mean squared error for the predictions of each model were calculated to compare their performace. Random forest regression was found to be the most effective model for predciting the total number of reported cases per 10,000 population for counties in Ireland, while the ridge regression model slightly outperformed the other three tested models for the countries data.

Unexpected trends were identified in the relationship between a number of the variables in the countries data. The number of deaths of children under 10 per 10,000 population was found to have a negative relationship with the total number of reported cases per 10,000, and higher income countries were also found to have a higher mean total number of reported cases per 10,000 than lower income countries. It was concluded that these trends were likely a result of the greater volume of air travel in developed countries, and the greater amount of testing carried out in developed countries, in comparison to underdeveloped and lower income countries.

Proposed future areas of investigation included carrying out analysis of excess death figures recorded during the COVID-19 pandemic with respect to the other variables analysed in this project, and carrying out similar analysis and modelling of data from future pandemics with special consideration for the most influential indicator variables observed in this project.

2 Introduction

2.1 Overview

Over the past year, the coronavirus disease COVID-19 has impacted every country across the globe. The highly infectious disease is primarily spread between people through respiratory droplets produced when exhaling, sneezing or coughing. The majority of people who contract the virus will only develop a mild or moderate illness, while a small percentage of people who contract it will suffer serious respiratory illness and potentially die, in particular the elderly and those with underlying health conditions. As such, cases of COVID-19 and deaths related to the disease have been reported in virtually every country on the planet.

The severity of the disease and it's risk to public health has meant that most countries have implemented various measures to combat the spread of the virus, such as lockdowns, enforced requirements for wearing personal protective equipment in public and closure of non-essential businesses and services. As a result, the COVID-19 pandemic has had a significant economic impact worldwide, with various economies struggling due to rising unemployment.

From a data science perspective, an incredible amount of data relating to the current pandemic is already available, such as information regarding numbers of reported cases and COVID-19 related deaths and infection rates. Conveniently, a lot of the available data is broken down into individual countries, and closer to home there is also a substantial amount of data related to the pandemic in Ireland which is broken down by individual counties.

2.2 Project Motivation

The primary aim of this project is to find a suitable regression model to predict the total number of COVID-19 cases per 10,000 population for a given region.

This would give an insight into some key indicator variables that could be used to estimate the total number of COVID-19 cases per 10,000 population for a region where this figure is unknown. Also, any identified indicator variables could potentially be looked at in relation to other diseases and pandemics that occur in the future when attempting to predict the spread and impact of the disease.

Specifically, this project will look at two separate cases:

- Number of reported COVID-19 cases per 10,000 population for a county in Ireland, as of 14th October 2020
- Number of reported COVID-19 cases per 10,000 population for a given country, as of 17th October 2020

In both cases, a number of different machine learning regression methods will be assessed to determine which method is most effective. Namely, this project look at the following regression models:

- Multiple linear regression
- Lasso regression
- Ridge regression
- Random Forest Regression

2.3 Data

As this project investigates two separate cases, the first looking at counties in Ireland and the second looking at countries, a different selection of the provided data sets will be used in each case.

2.3.1 Counties in Ireland

Based on the data sets available for this project, I have chosen to look at the following variables to assess as potential predictor variables in a regression model used to predict the number of reported COVID-19 cases per 10,000 population for a county in Ireland (as of 14th October 2020):

- Total population
- Population density
- Number of deaths of children under 10 per 10,000 population (pre-COVID)
- Percentage of population unemployed (pre-COVID)
- Province

To obtain the data for the above variables, the following data files can be used:

- ***'unemployment_ireland_by_month_by_region_by_age.csv'*** - the average number of people unemployed in 2019 can be obtained from this file.

- ***'deaths_ireland_2010_to_2019.csv'*** - the average number of deaths of children under the age of 10 between 2010 and 2019 can be obtained from this file.

- ***'cases_by_county_ireland.csv'*** - this file can be used to obtain population data, land area, population density and the total number of confirmed COVID-19 cases for each county.

2.3.2 Countries

When looking at potential predictor variables for a regression model used to predict the number of reported COVID-19 cases per 10,000 population for a country (as of 17th October 2020), I have chosen again to look at similar variables to those described for the counties data above, along with some additional variables that can be obtained from the provided data sets:

- Total Population
- Population Density
- Number of deaths of children under 10 per 10,000 population (pre-COVID)
- Percentage of population unemployed (pre-COVID)
- Continent
- Income Group
- Number of days since first reported case of COVID-19

To obtain the data for the above variables, the following data files can be used:

- ***'daily-cases-by-country.csv'*** - this file can be used to obtain population data, continent, the date of the first recorded COVID-19 case and the total number of confirmed COVID-19 cases for each country.

- ***'country_income_groups.csv'*** - the income group of each country can be obtained from this file.

- ***'unemployment_by_country_by_year.csv'*** - the percentage of people unemployed in 2019 can be obtained from this file.

In addition to the data files above, the World Bank API can be used to obtain some additional data for each country which can also be used, namely total land area and number of deaths of children under the age of 10.

2.4 Assumptions

Due to a number of limitations in the data files available for this project, as well as having limited available information relating to the exact descriptions of certain variables and their units in the data files, a number of assumptions have been made:

- The total number of reported COVID-19 cases will be considered up to the latest date where this information is available in the provided data files, which is the 14th of October 2020 for Irish counties and the 17th of October 2020 for countries.
- The *Shape_Area* column in the 'cases_by_county_ireland.csv' file is assumed to contain the total land area of each county, with units m^2.
- The figures in the *ConfirmedCovidCases* and *PopulationProportionCovidCases* columns in the 'cases_by_county_ireland.csv' file are assumed to be cumulative figures.
- Pre-COVID-19 figures for the numbers of deaths of children under 10 per 10,000 population in Irish counties will be estimated using the average number of deaths of children under the age of 10 between the years 2010 and 2019.
- Pre-COVID-19 figures for the numbers of deaths of children under 10 per 10,000 population in countries will be estimated using figures from 2016 (most recent year where that this information is available for all of the countries included in the available data).
- Pre-COVID-19 figures for the percentage of population unemployed in Irish counties will be estimated using the average of the monthly unemployment figures from the year 2019.
- Pre-COVID-19 figures for the percentage of population unemployed in countries will be estimated using the unemployment figures from the year 2019.

2.5 Report Structure

The remaining report is structured as follows:

- **Background** - this section will provide some brief background information on the exploratory data analysis and statistical analysis methods to be used for this project, including a brief description of each of the regression algorithms to be assessed.
- **Analysis** - this section will cover the processing of the available data files, the exploratory data analysis and statistical analysis performed as part of the project. All relevant code, tables, figures and graphs required for the analysis of the data will also be included in the section.
- **Results** - the results obtained from the analysis section will be summarised and discussed in further detail in this section.
- **Conclusions** - finally, the results obtained will be discussed further and conclusions drawn on these results and any findings from the data analysis.

3 Background

3.1 Exploratory Data Analysis Methods

Exploratory data analysis methods can be used to investigate the strength of the relationship between different variables, both visually and numerically. A number of these methods will be used as part of this project to determine which variables will be most suitable to use as indicator variables for regression analysis.

3.1.1 Scatter plots

Scatter plots are used to plot two variables against each other with the data points displayed as dots. They can be used to quickly determine if a relationship exists between the variables and the nature of their relationship.

Pairs plots are a very effective way of visualising the relationship between multiple variables simulaneously. These are effectively a grid of scatter plots which plot every combination of two variables against each other. They can be used to quickly visulaise a numerical data set and determine which variables potentially have a relationship.

3.1.2 Correlation plots

Depending on the nature of the data being analysed, it is not always clear from a scatter plot whether a significant relationship exists between variables or the relative strength of the relationship. The correlation between two variables is a measure of the strength of the linear relationship between them.

A correlation matrix is a table which contains the correlation coefficient between every combination of numerical variables in a data set. In cases where a large number of variables are being analysed, it can be particularly useful to use a correlation plot to display this data. These plots are used to display the information from a correlation matrix but with added visual elements such as colours correspodning to the magnitude of each coeffiecient value. This helps to quickly identify which varibles are most strongly related to one another.

3.1.3 Data transformation

When modelling data using regression, it is generally preferred to use variables that are normally distributed when possible. This can reduce the likelihood of some assumptions of linear regression models not being satisfied, including homoscedasticity and normality of errors.

In cases where the distribution of a variable is significantly skewed, it can often be beneficial to transform the data so that its ditribution approximately conforms to normaility. Log transformations are possibly the most frequently used transformation method, and can be used in cases where a variable's distribution is heavily right-skewed.

Histograms and desnisty plots can be used to visualize the distribution of a variable and determine whether a transformation may be valid.

3.1.4 Box plots

While the above methods can be used to analyse numeric variables, different methods must be used when looking variables that can only take on one of a limited number of values, also known as categorical variables. Box plots can be a useful way to convey the relationship between a categorical variable and a numeric variable.

A box plot of a numeric variable can give an insight into the distrbution of the variable by visually displaying some key summary statistics related to the distribution. Most importantly, these plots display the median, first quartile and third quartile of a variable in the form of a box with an intersecting line. This gives a quick indication of the center of a variables distribution and how much variance there is in the data.

To observe the relationship between a categorical variable and a numeric variable, the numeric variable data can be split up based on each unique category of the categorical variable. Box plots of the distributions for each category can then be used to quickly determine whether there is a significant difference in the distribution of the numeric variable between some or all of the categories of the categorical variable.

3.2 Data Validation and Testing

3.2.1 Train, Validation and Test Data

When evaluating the performance of a model, it is critical that the data used to test the model is unbiased and independent from the data used to train the model. For this reason, before building a model it is common to take a random sample of data from the data set being used, known as test data. The test data is not used when buiding the model and only used to evaluate the final model. Typically, between ten and twenty-five percent of the total data is reserved for test data.

It is possible to use the full set of remaining data as training data for building the model. However, it can be useful to take a further random sample of the remaining data, known as validation data, and use it for testing and tuning the model. Again, between ten and twenty-five percent of the total data is typically reserved for validation data.

The remaining data is known as training data, and is used to fit the parameters of the model. It is important that this data is independent from the test data to avoid overfitting.

3.2.2 K-Fold Cross Validation

K-fold cross validation is a model validation technique that takes all data outside of the test data set, and partitions it into a specified number of groups, known as folds. This method is used to create multiple combinations of training and validation data sets, equal to the number of specified folds. In each case, a single fold is used as validation data and the remaining folds are used as training data. Each time, the training data is fit to a model and the validation data is used to test the model. This way, the performance of the model can be assessed across the full range of the available data.

Cross validation can be particularly useful when dealing with relatively small datasets, as there is higher probability of data in a single validation set being biased or skewed from the full data set, due to the small number of samples. Once the required performance is achieved from the model used k-fold cross validation, the final model can be fitted to the full set of training data (i.e. all data outside of the test data set).

3.3 Statistical Analysis - Regression Modelling

Regression is a method of mathematically formulating a relationship between variables which can be used for prediction purposes. A number of different regression methods exist, several of which will be tested as part of this project.

3.3.1 Multiple Linear Regression

Multiple linear regression is a statistical regression model used to predict the outcome of a response variable based on inputs from a number of explanatory or indicator variables.

The ordinary least-squares regression method takes a single explanatory variable and predictor variable and estimates the best fit linear relationship between. Mulitple linear regression is an extension of ordinary least squares regression where more than one explanatory variable is used.

The model parameters, or $\hat{\beta}$ parameters, corresponding to each explanatory variable are computed by minimizing the following cost function:

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2$$

where n is the number of observations and p is the number of explanatory variables.

The formaula for the model can be written as follows:

$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip} + \epsilon_i$

where, for $i = n$ observations,

- y_i is the i^{th} observation of the response variable,
- x_i is the i^{th} observation of one of the explanatory variables,
- β_0 is the y-intercept term (i.e. model's predicted value when all explanatory variables have a value of 0),
- β_j is the slope parameter coeffieicent for each of the p explanatory variables,
- ϵ_i is the error term for the i^{th} observation

3.3.2 Lasso Regression

Lasso (Least Absolute Shrinkage and Selection Operator) regression is another form of linear regression. Unlike multiple linear regression, the lasso method aims to reduce the number of predictor variables used in a model by shrinking the coefficients of less significant explanatory variables to zero.

This is achieved by adding a penalty term into the cost function used for multiple linear regression, so that the $\hat{\beta}_{\text{lasso}}$ parameters for the model are calculated by minimizing the following function:

$$\hat{\beta}_{lasso} = \underset{\beta}{\operatorname{argmin}} \left[\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right]$$

where the magnitude of the shrinkage is determined by the value of λ.

This can be particularly beneficial in reducing the size of models where a large number of explanatory variables are provided.

3.3.3 Ridge Regression

Ridge regression is a similar method to lasso regression, however it uses a different penalty term to penalize the model for the sum of squared values of the weights, as opposed to penalizing the sum of the absolute values of the weights with lasso regression.

Thus, the $\hat{\beta}_{\text{ridge}}$ parameters for the model can be calculated by minimizing the following function:

$$\hat{\beta}_{ridge} = \underset{\beta}{\operatorname{argmin}} \left[\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right]$$

Again, ridge regression can be used to reduce the complexity in a model in comparison to standard multiple linear regression.

3.3.4 Random Forest Regression

Decision tree regression uses a series of sequential decisions that converge to a specific value. This form of modelling creates a ‘tree’ of yes/no decision nodes with branch out to further decision nodes, eventually converging on a single ‘leaf’ node corresponding to a predicted value.

Random forest regression is similar to decision tree regression, but instead of using a single decision tree it uses multiple decision trees that are created using random subsets of the explanatory variables in the model.

For each prediction, all of the decision trees in the model are evaluated, and the results are combined using an averaging method to generate a single output prediction value.

3.4 Model Evaluation

3.4.1 Root Mean Squared Error

The Root Mean Squared Error (RMSE) is a measure of the distance between values predicted from a model and the actual values. The square of the error is used to ensure only the magnitude of the error is considered, and not its sign. The average of each error value is then calculated by summing up the squared error value for each prediction and dividing by the total number of predictions. Taking the root of the mean squared error returns a value is that on the same scale as the dependent variable in the model, making it easier to interpret.

The mathematical equation for the Root Mean Squared Error can be written as follows:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (y_i - \hat{y_i})^2}{n}}$$

A small RMSE value indicates that the model is a good predictor of the tested data, while a large MSE value is an indication of a poor model.

3.4.2 Normalised Root Mean Squared Error

While the Root Mean Squared Error is a useful metric for comparing the performance of multiple models for a single data set, it is often difficult to compare the performance of models from different data sets, in particular where the models have different dependent variables, or where a data transform has been performed on the dependent variable for one of the models.

In such cases, a better metric to use is the Normalised Root Mean Squared Error (NRMSE). This is an estimate of the RMSE value that would be obtained for the same data used if the dependent variable had been normalised prior to modelling.

If a non-normalised dependent variable is used during the modelling, it is not possible to accurately calulcate the RMSE for a model with the equivalent normalised dependent variable. However, there are various different methods of calculating the NRMSE that are recognised in literature, which can be used to estimate this value.

For this project, the standard deviation method will be used for calculating the NRMSE, which can be described using the following mathematical equation:

$$NRMSE = \frac{RMSE}{\sigma}$$

where σ is the standard deviation in the actual values.

3.4.3 R-Squared

The coefficient of determination, denoted R-squared, is a figure that indicates the proportion of variance in the dependent variable in a model that is explained by the predictor variables in the model. The R-squared value is obtained by calculating the ratio between the sum of squares of residuals in a model and the total sum of squares, and subtracting the resulting value from 1.

The mathematical equation for the coefficient of determination can be written as follows:

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y_i})^2}{\sum_{i=1}^{n} (y_i - \bar{y_i})^2}$$

3.4.4 Adjusted R-Squared

One issue with using the R-squared equation above on regression models with multiple predictor variables is that it does not decrease when additional predictor variables are added to the model, regardless of whether they contribute any useful information to the model.

To combat this issue, adjusted R-squared can be used to better evaluate the performance of models with multiple predictor variables. Contrary to standard R-squared, adjusted R-squared takes the degrees of freedom of the sum of squares of residuals and the total sum of squares into account, thus penalizing the model for any predictor variables that do not contribute meaningfully to the model.

The equation for the adjusted R-squared can be written as follows:

$$R^2 adjusted = 1 - \frac{(1 - R^2)(n - 1)}{n - p - 1}$$

where R^2 is the sample coefficient of determination value, n is the total data sample size, and p is the number of predictors in the model.

4 Analysis

As discussed in the introduction, this project will look at two separate cases, the first looking at data for counties in Ireland and the second looking at data for countries all over the world. These cases will be analysed separately.

Before any analysis is carried out, is is useful to import any relevant libraries that will be required. The path name where all the data files are stored must also be specified before attempting to load any of the files into the notebook.

[1]:
```
# Import libraries
import pandas as pd
import numpy as np
from pandas import DataFrame, Series

import wbdata as wbd

import datetime

from matplotlib import cm
import matplotlib.pyplot as plt
import seaborn as sns

from mpl_toolkits.mplot3d import art3d
from matplotlib.patches import Ellipse

from sklearn.model_selection import KFold

from sklearn import linear_model
from sklearn.linear_model import Ridge
from sklearn import svm
from sklearn.ensemble import RandomForestRegressor

from IPython.display import Markdown, display

# Define path name
path = 'Data Files/'
```

4.1 Irish Counties Data Analysis

4.1.1 Data Preprocessing

Unemployment Data

Monthly unemployment data for each county in Ireland can be obatined from the 'unemployment_ireland_by_month_by_region_by_age.csv' data file. On inspection of this data file, it is clear that a number of pre-processing steps are required to ensure the data is in a usable format.

Firstly, the first three rows and final five rows are not required, and can be dropped from the dataset. As we are are interested solely in unemployment figures from the year 2019, the last nine columns relating to figures from 2020 can also be dropped.

Due to the format of the file, the first three columns of the data set don't have suitably defined headings, and so these columns can be renamed to make the data easier to read.

There are also a number of empty entries containing a blank space. As a result, these were not replaced with *NaN* values when the data file was loaded in. Therefore, these must be manually replaced with *NaN* values.

[2]:
```
# read in csv file with first 3 rows removed
unemployIE1 = pd.read_csv(path +␣
↪'unemployment_ireland_by_month_by_region_by_age.csv', skiprows=[0,1,2])

# get length of data frame and num cols in data frame
df_len = len(unemployIE1)
df_numcols = len(unemployIE1.columns)

# drop last 5 rows
unemployIE1.drop(unemployIE1.index[range(df_len-5,df_len)], inplace = True)

# drop last 9 columns (only looking at 2019 figures)
unemployIE1.drop(unemployIE1.
↪columns[range(df_numcols-9,df_numcols)],axis=1,inplace=True)

# rename first 3 columns to give them sensible names
unemployIE1.rename(columns={
    unemployIE1.columns[0]:'AgeRange',
    unemployIE1.columns[1]:'Sex',
    unemployIE1.columns[2]:'County'}, inplace=True)

# replace all ' ' characters with NaN
unemployIE1.replace(' ', np.nan, inplace=True)
```

The first 2 columns of the data set only contain a single value at the start of each group, rather that explicitly indicating the corresponding value on each individual row. The forward fill method can be used to forward fill all *NaN* values in these columns, so that each row of data has an an *AgeRange* and *Sex* value explicitly stated.

One useful observation that can be made from the data set is that the total figure for each individual county is given in rows where the the value in the *County* column includes the word 'County'. There is one exception to this, which is Tipperary, which is split into 'Tipperary North' and 'Tipperary South'. To overcome this issue, string replace methods can be used to replace these values with 'Tipperary County'.

This project is not focusing on gender specific data, so columns with 'Male' and 'Female' values in their *Sex* columns can also be filtered out. The monthly figures in the data are also imported as strings rather than integers, so the data types of these columns must be converted.

All rows without 'County' in their *County* column can then be filtered out of the data frame. Finally the data can be grouped by *County* and *AgeRange* with grouped values summed to combine the two Tipperary categories and also to combine the age range categories for each county.

[3]:
```
# create copy of previous data set
unemployIE2 = unemployIE1.copy()

# forward fill all NaN values in first 2 columns
unemployIE2['AgeRange'] = unemployIE2['AgeRange'].fillna(method='ffill')
unemployIE2['Sex'] = unemployIE2['Sex'].fillna(method='ffill')

# delete all remaining rows containing NaN values
unemployIE2 = unemployIE2.dropna()

# filter out rows that don't contain 'County', 'North Tipperary' or 'South␣
↪Tipperary' in the County column
unemployIE2 = unemployIE2[unemployIE2['County'].str.contains("County|North␣
↪Tipperary|South Tipperary")]

# filter out rows that don't contain 'Both sexes' in the Sex column
unemployIE2 = unemployIE2[unemployIE2['Sex'].str.contains("Both sexes")]

# change data type of 4th to final column from string to int
unemployIE2[unemployIE2.columns[range(3,len(unemployIE2.columns))]] =␣
↪unemployIE2[unemployIE2.columns[range(3,len(unemployIE2.columns))]].
↪astype(int)

# replace North and South Tipperary with 'Tipperary County' to match all other␣
↪counties
unemployIE2['County'] = unemployIE2['County'].replace(['North Tipperary',␣
↪'South Tipperary'], 'Tipperary County')

# group dataset by age range and county, and sum values together (this will␣
↪combine Tipp entries)
unemployIE2 = unemployIE2.groupby(['County']).sum().reset_index()
```

The desired unemployment figure from this data file is the average number of people unemployed in 2019. The mean value of all monthly figures from 2019 can be used to estimate this. After the average unemployment figures are determined, the monthly figure columns are no longer of use and can be dropped from the data frame.

To ensure that the data frame can be merged with other data frames relating to county data, the word ‘County’ can be removed from each county name. The spelling of Laois in the current data also differs to the spelling used in the other data files, so its spelling can also be corrected avoid issues when merging data frames later.

The first 3 rows of the processed data are displayed below.

[4]:
```
# create copy of previous data set
unemployIE = unemployIE2.copy()

# create new column which is mean of all unemployment figures (could use sum or␣
 ↪equaivalent case where we are looking at deaths etc.)
unemployIE['AvgUnemployed'] = unemployIE.mean(axis=1)

# re-calculate number of columns
df_numcols = len(unemployIE.columns)

# now drop all monthly figure columns (only looking at average figures)
unemployIE.drop(unemployIE.columns[range(1,df_numcols-1)],axis=1,inplace=True)

# drop the word 'County' from each county
unemployIE['County'] = unemployIE['County'].str.replace(' County', '')

# replace the word 'Laoighis' with the spelling 'Laois' to match other datasets
unemployIE['County'] = unemployIE['County'].str.replace('Laoighis', 'Laois')

# create list of county names
countyList = unemployIE.County.unique()

unemployIE.head(3)
```

[4]:
```
   County  AvgUnemployed
0  Carlow    3364.833333
1   Cavan    3244.916667
2   Clare    4606.166667
```

Number of Deaths of Children Under 10 Years

The 'deaths_ireland_2010_to_2019.csv' data files contains information on the number of deaths in all Irish regions for various different age groups between 2010 and 2019.

It has a similar structure to the 'unemployment_ireland_by_month_by_region_by_age.csv' file in that its first two columns only contain a single value at the start of each group of data, rather that explicitly indicating the corresponding value for each individual row. Again, blank spaces in the data frame can be replaced with *NaN* values, but a further replacement must also be made for this data to replace all instances of the characters '..' with the value 0.

```
[5]: # read in csv file with first 2 rows removed
     deathsIE1 = pd.read_csv(path + 'deaths_ireland_2010_to_2019.csv',␣
      ↪skiprows=[0,1])

     # get length of data frame
     df_lastrow = len(deathsIE1)-1

     # drop last row
     deathsIE1.drop(deathsIE1.index[df_lastrow], inplace = True)

     # rename first 3 columns to give them sensible names
     deathsIE1.rename(columns={
         deathsIE1.columns[0]:'AgeRange',
         deathsIE1.columns[1]:'Sex',
         deathsIE1.columns[2]:'County'}, inplace=True)

     # replace all ' ' characters with NaN
     deathsIE1.replace(' ', np.nan, inplace=True)

     # replace all '..' characters with 0
     deathsIE1.replace('..', 0, inplace=True)
```

The same forward fill method used for the unemployment data frame can be used on the first two columns to ensure each row has the corresponding value for these columns explicitly stated.

Unfortunately, there are several counties in this data set for which a single figure for toal number of deaths in the county is not provided, specifically for Dublin, Waterford, Cork, Limerick, Tipperary, and Galway. However, the main regions that can be combined to give the total number of deaths figure for each county are relatively easy to identify. String replacement can again be used to replace the value in the *County* column for each of these identifed regions with their associated county's name.

After these replacement steps are complete, all rows with a value for *County* that isn't contained in the list of County names from the unemployment data frame can be dropped.

```
[6]: # create copy of previous data set
     deathsIE2 = deathsIE1.copy()

     # forward fill all NaN values in first 2 columns
```

```
deathsIE2['AgeRange'] = deathsIE2['AgeRange'].fillna(method='ffill')
deathsIE2['Sex'] = deathsIE2['Sex'].fillna(method='ffill')

# delete all rows with NaN values
deathsIE2 = deathsIE2.dropna()

# need to rename regions to match county name where they contribute to total␣
 ↪number of county
# replace Dublin sub regions with 'Dublin'
deathsIE2['County'] = deathsIE2['County'].replace(['Dublin City', 'Fingal',␣
 ↪'Dun Laoghaire Rathdown', 'South Dublin'], 'Dublin')
# replace Waterford sub regions with 'Waterford'
deathsIE2['County'] = deathsIE2['County'].replace(['Waterford County',␣
 ↪'Waterford City'], 'Waterford')
# replace Cork sub regions with 'Cork'
deathsIE2['County'] = deathsIE2['County'].replace(['Cork County', 'Cork City'],␣
 ↪'Cork')
# replace Limerick sub regions with 'Limerick'
deathsIE2['County'] = deathsIE2['County'].replace(['Limerick County', 'Limerick␣
 ↪City'], 'Limerick')
# replace Tipperary with 'Tipp Town' and replace Tipp sub regions with␣
 ↪'Tipperary'
deathsIE2['County'] = deathsIE2['County'].replace(['Tipperary'], 'Tipp Town')
deathsIE2['County'] = deathsIE2['County'].replace(['North Tipperary', 'South␣
 ↪Tipperary'], 'Tipperary')
# replace Galway sub regions with 'Galway'
deathsIE2['County'] = deathsIE2['County'].replace(['Galway County', 'Galway␣
 ↪City'], 'Galway')

# remove all rows where County name isn't in list of counties (from␣
 ↪unemployment ireland .csv file)
deathsIE2 = deathsIE2[deathsIE2['County'].isin(countyList)]
```

Again, we are not interested in data related to specific genders so rows with values other than 'Both Sexes' in their *Sex* column can also be dropped. Also, the data type of the columns containing the number of deaths each year need to be converted from strings to integers, as done below.

Another indicator variable which could be interesting to include in the model would be the province associated with each county. This is not specified in the available data files but it can manually be added to the current data frame.

Only three of the age range groups included in the data set are required to determine the number of deaths of children under the age of 10. It is useful to rename the *AgeRange* value for these groups so that they are easier to understand.

The data can then be grouped by *County*, *Province* and *AgeRange* and values summed to combine the individual regions renamed above to form single entries for each county.

```
[7]: # create copy of previous data set
deathsIE3 = deathsIE2.copy()

# filter out rows that don't contain 'Both sexes' in the Sex column
deathsIE3 = deathsIE3[deathsIE3['Sex'].str.contains("Both sexes")]

# change data type of 4th to final column from string to int
deathsIE3[deathsIE3.columns[range(3,len(deathsIE3.columns))]] =␣
 ↪deathsIE3[deathsIE3.columns[range(3,len(deathsIE3.columns))]].astype(int)

# create list of provinces corresponding to current order of counties in data␣
 ↪frame
provinceList = ( ['Leinster']*15 + ['Munster']*10 + ['Connacht']*6 +␣
 ↪['Ulster']*3 ) * len(deathsIE3.AgeRange.unique())
deathsIE3.insert(2, 'Province', provinceList)

# rename some of the AgeRange groups so that they order correctly
deathsIE3['AgeRange'] = deathsIE3['AgeRange'].str.replace('1 - 4 years', '01 -␣
 ↪04 years')
deathsIE3['AgeRange'] = deathsIE3['AgeRange'].str.replace('5 - 9 years', '05 -␣
 ↪09 years')
deathsIE3['AgeRange'] = deathsIE3['AgeRange'].str.replace('Under 1 year', '00 -␣
 ↪01 years')

# group dataset by age range, province and county, and sum values together
deathsIE3 = deathsIE3.groupby(['County','Province','AgeRange']).sum().
 ↪reset_index()
```

To estimate the mean number of deaths of childern under 10 in each county, the mean number of yearly deaths for each individual group between 2010 and 2019 can first be calculated. Then all rows other than the three group that make up the under 10 years bracket can be dropped from the data frame. Finally the data can be grouped by *County* and *Province* and the values summed to get the mean number of deaths of children under 10 per county.

The yearly figure columns can now be dropped as they are no longer required. The first 3 rows of the processed data are displayed below.

```
[8]: # create copy of previous data set
deathsIE = deathsIE3.copy()

# create new column which is mean of all deaths
deathsIE['AvgDeathsU10'] = deathsIE.mean(axis=1)

# filter out rows for age ranges >10 years
deathsIE = deathsIE[deathsIE['AgeRange'].str.contains("00 - 01 years|01 - 04␣
 ↪years|05 - 09 years")]
```

```
# group dataset by county and province to, and sum values together to get sum␣
↪of age ranges under 10
deathsIE = deathsIE.groupby(['County','Province']).sum()
# reset index to flatten dataframe after grouping
deathsIE=deathsIE.reset_index()

# calculate number of columns
df_numcols = len(deathsIE.columns)

# now drop all yearly figure columns (only looking at average figures)
deathsIE.drop(deathsIE.columns[range(2,df_numcols-1)],axis=1,inplace=True)

deathsIE.head(3)
```

```
[8]:    County  Province  AvgDeathsU10
     0  Carlow  Leinster           3.4
     1   Cavan    Ulster           4.2
     2   Clare   Munster           7.8
```

Number of COVID-19 Cases per County

The remaining Irish counties data required before beginning exploratory data analysis can be obtained from the 'cases_by_county_ireland.csv' file. This data file has a more regular structure than the previous two files, with each column filled appropriately and without any unnecessary rows included in the files.

As the *ConfirmedCovidCases* column is assumed to contain cumulative figures, only figures reported on the most recent date, the 14th of October 2020, are required, and so all other rows can be dropped. There are various additional columns in the data set that are not required as part of this project, so these can also be dropped from the data set.

[9]:
```
# read in the csv file
casesIE1 = pd.read_csv(path + 'cases_by_county_ireland.csv')

# filter out rows that don't contain '2020/10/14' in the TimeStamp column (most␣
↪up to date value of total confirmed cases on 2020/10/14)
casesIE1 = casesIE1[casesIE1['TimeStamp'].str.contains("2020/10/14")]

# drop unneeded columns
casesIE1 = casesIE1.drop(['OBJECTID', 'ORIGID', 'TimeStamp',␣
↪'ConfirmedCovidDeaths', 'ConfirmedCovidRecovered', 'Shape__Length',␣
↪'IGEasting', 'IGNorthing', 'UGI'], axis=1)

# replace NaN values with 0
casesIE1 = casesIE1.fillna(0)
```

As the *Shape__Area* column is assumed to contain the total land area of each county in metres squared, it is helpful to rename it to *AreaKM2* and divide the data by 100,000 to convert the values to land area in kilometres squared. Renaming the *CountyName* column to *County* will also simplify the merging process when combining mulitple data sets later.

When comparing the *PopulationCensus16* and *ConfirmedCovidCases* figures with the *PopulationProportionCovidCases* figures in the data set, it is clear that the values provided are the proportion of cases per 100,000 population. To convert the proportion values to represent the proportion of cases per 10,000 population, the *PopulationProportionCovidCases* data can simply be divided by 10.

From the data available, it is also possible to calculate the population density of each county by dividing the population data by the area data. The first 3 rows of the processed data are displayed below.

[10]:
```
# rename shape area, county and population proportion cases columns
casesIE = casesIE1.rename(columns = {'Shape__Area':'AreaKM2','CountyName':
↪'County','PopulationProportionCovidCases':'PopProportionCovidCases'})

# divide area km2 column by 1000000 to convert from m2 to km2
casesIE['AreaKM2'] = casesIE['AreaKM2'] / 1000000
```

```
# divide population proportion covid cases by 10 to get proportion into cases␣
↪per 10K
casesIE['PopProportionCovidCases'] = casesIE['PopProportionCovidCases'] / 10

# create new population density column
casesIE['PopulationDensity'] = casesIE['PopulationCensus16'] /␣
↪casesIE['AreaKM2']

# reset index values
casesIE.reset_index(inplace=True, drop=True)

casesIE.head(3)
```

```
[10]:   County  PopulationCensus16      Lat    Long  ConfirmedCovidCases  \
     0  Carlow               56932  52.7168 -6.8367                  339
     1   Cavan               76176  53.9878 -7.2937                 1451
     2   Clare              118817  52.8917 -8.9889                  964

        PopProportionCovidCases      AreaKM2  PopulationDensity
     0                59.544720  2432.351721          23.406154
     1               190.479941  5575.009911          13.663832
     2                81.133171  8722.541706          13.621832
```

Merging the Data

To combine the *unemployIE*, *deathsIE* and *casesIE* data frames into a single data frame, the *merge* function from pandas can be used to join the data frames based on their *County* columns.

[11]:
```
# merge casesIE and deathsIE dataframes on their County columns
dataIE = pd.merge(casesIE, deathsIE, how='inner', on='County')
# merge the newly created data set with the unemployIE dataframe on their␣
 ↪County columns
dataIE = pd.merge(dataIE, unemployIE, how='inner', on='County')
```

With the data frames combined, it is now possible to create further columns required for the exploratory data analysis using data from a combination of different columns, namely the average number of yearly deaths of children under 10 years of age per 10,000 population and the average percentage of the population unemployed in 2019.

The first 3 rows of the final data set, which will be used for the exploratory data analysis, are printed below.

[12]:
```
# create new columns corresponding to AvgDeathsU10PerPop (per 10000) and␣
 ↪AvgUnemployedPerPop (percent)
dataIE['AvgDeathsU10PerPop'] = dataIE['AvgDeathsU10'] /␣
 ↪dataIE['PopulationCensus16'] * 10000
dataIE['AvgUnemployedPerPop'] = dataIE['AvgUnemployed'] /␣
 ↪dataIE['PopulationCensus16'] * 100

dataIE.head(3)
```

[12]:
```
   County  PopulationCensus16      Lat     Long  ConfirmedCovidCases  \
0  Carlow               56932  52.7168  -6.8367                  339
1   Cavan               76176  53.9878  -7.2937                 1451
2   Clare              118817  52.8917  -8.9889                  964

   PopProportionCovidCases       AreaKM2  PopulationDensity  Province  \
0                59.544720   2432.351721          23.406154  Leinster
1               190.479941   5575.009911          13.663832    Ulster
2                81.133171   8722.541706          13.621832   Munster

   AvgDeathsU10  AvgUnemployed  AvgDeathsU10PerPop  AvgUnemployedPerPop
0           3.4    3364.833333            0.597204             5.910267
1           4.2    3244.916667            0.551355             4.259762
2           7.8    4606.166667            0.656472             3.876690
```

4.1.2 Exploratory Data Analysis

A Note on Test Data

The *dataIE* dataframe above contains all the training, validation and testing data that will be used when looking at the Irish counties. Therefore, it is important that any data that will be used to test and evaluate the final regression models is not included when performing exploratory data analysis.

This ensures that any decisions made while performing exploratory data analysis are not biased due to the test values being included. The test data must effectively be treated as unknown and independent data until this analysis has been completed and the final regression models have been created, so that they can be evaluated fairly.

For this project, 20 percent of the total available data will be reserved for testing, with the remaining data used for training and validation. In order to randomly split the data, a randomly oredered vector, *train_val_select* can be used to reorder the data, and using a combinatopn of this vector and the *iloc* function, it is possible to filter a specified amount of values out of the dataset.

This technique will be used to only include the training and validation data during all explanatory data analysis steps.

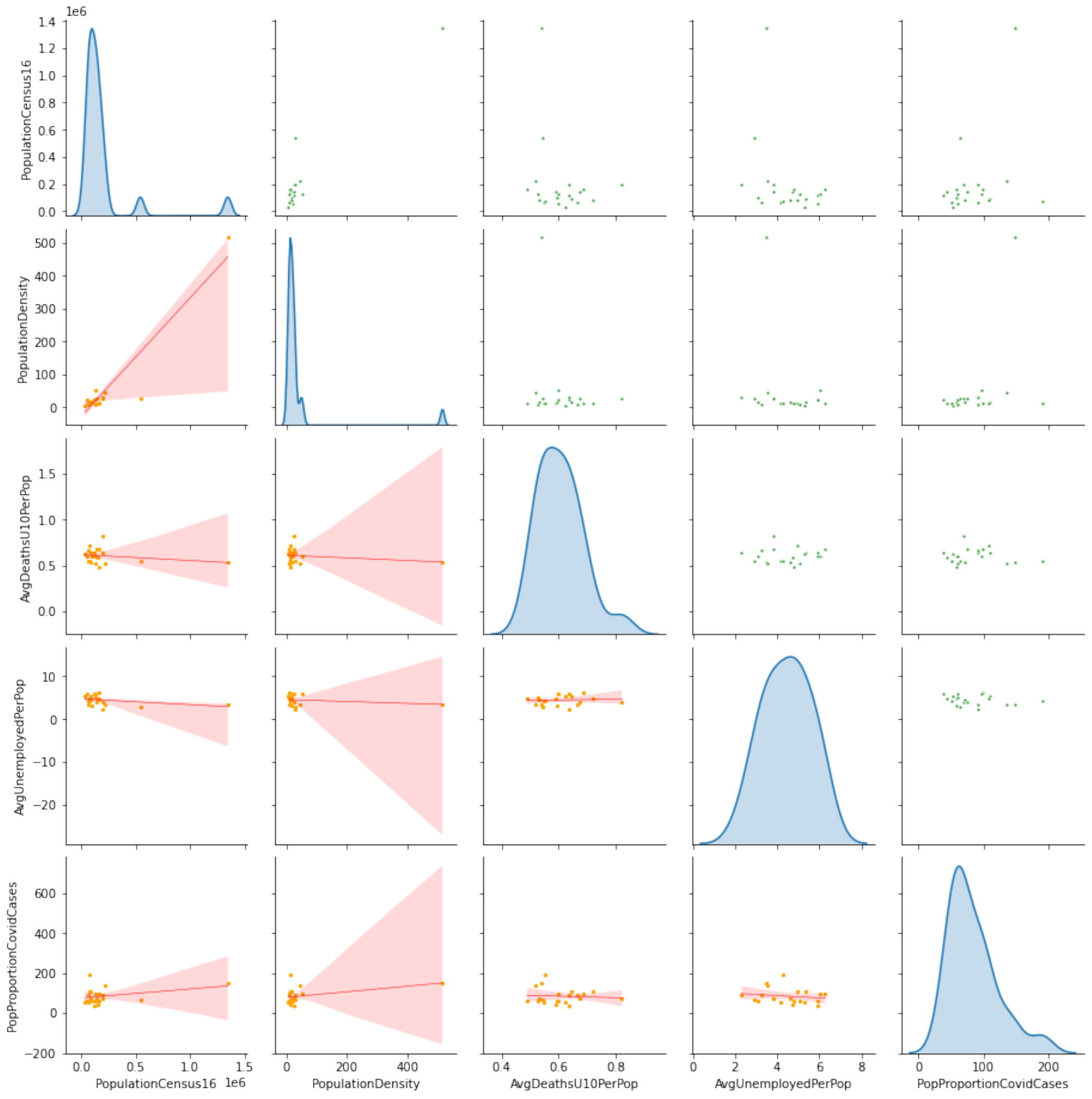
[13]:
```
# set seed value so results are repeatable
np.random.seed(123)

# split train/validation and test data 80:20
train_val_size = int(np.round(dataIE.shape[0]*0.8))

# create vector from 0 to length y, and randomise the order
train_val_select = np.random.permutation(range(dataIE.shape[0]))
```

Continuous Variables

Scatter plots can be used to visualize the relationship between all of the continuous or numeric values in our data frame.

Firstly, a separate data frame containing all numeric values of interest can be created by specifying the relevant column names. A pairs plot can then be used to create a matrix of scatter plots comparing each pair combination of variables in the numeric data frame. The pairs plot below is configured to display standard scatter plots in the upper right portion, and scatter plots with a fitted linear regression line in the lower left portion of the pairs plot. The diagonal elements of the pairs plot have also been configured to display density plots, which can be used the visualize the distribution of each variable.

```
[14]: # create data frame containing all numeric variables to be analysed
dataIE_num = dataIE.loc[:, ['PopulationCensus16', 'PopulationDensity',␣
 ↪'AvgDeathsU10PerPop', 'AvgUnemployedPerPop', 'PopProportionCovidCases']]

# create pairs plot of numeric data with test values filtered out
pairsIE = sns.pairplot(dataIE_num.iloc[train_val_select[:train_val_size],:].
 ↪reset_index(drop=True), kind="scatter", diag_kind="kde",␣
 ↪plot_kws=dict(alpha=0.8, marker=".", linewidth=1, color='green'))
pairsIE.map_lower(sns.regplot, scatter_kws={'alpha':1, 'color':'orange',␣
 ↪'marker':'.','s':5}, line_kws={'color':'red', 'linewidth':0.5})
plt.show()
```



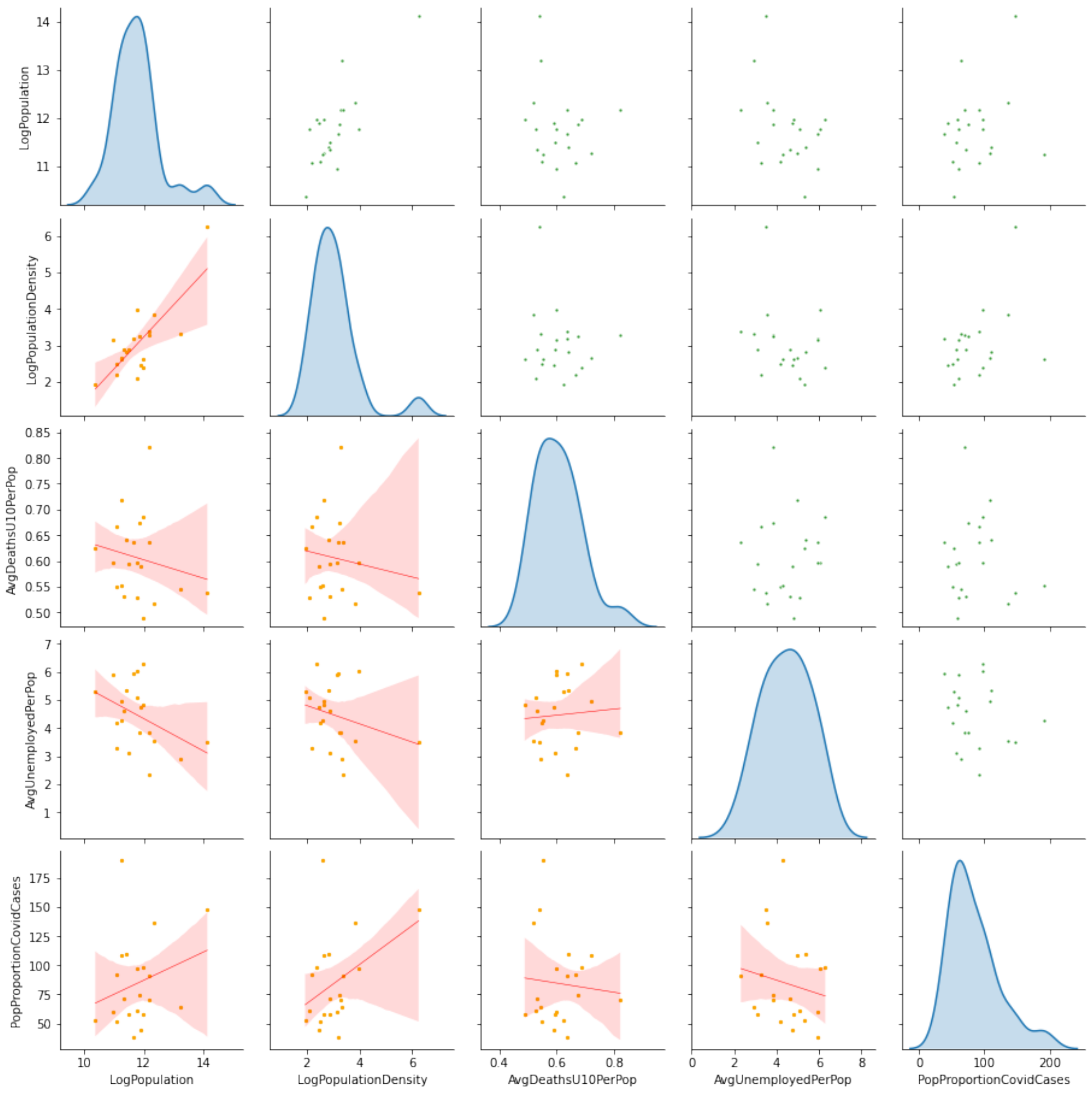
Looking at the density plots for the population and population density variables above, both appear to have heavily right-skewed distributions, with the majority of data points in their related scatter plots located on the left-hand side of the plot and very few data points scattered furter towards the right-hand side.

In such cases, it is often beneficial to perform a log transformation on the related variable so that its data distribution will fall closer to that of a normal distribution.

After performing log transforms on these variables, the pairs plot can be recreated as done previously.

[15]:
```
# perform log transform on population and population density columns
dataIE_num['PopulationCensus16'] = np.log(dataIE_num['PopulationCensus16'])
dataIE_num['PopulationDensity'] = np.log(dataIE_num['PopulationDensity'])
```

```
# rename log transformed columns
dataIE_num.rename(columns={'PopulationCensus16':
 ↪'LogPopulation','PopulationDensity':'LogPopulationDensity'}, inplace=True)

# recreate pairs plot of numeric data with test values filtered out
pairsIE = sns.pairplot(dataIE_num.iloc[train_val_select[:train_val_size],:].
 ↪reset_index(drop=True), kind="scatter", diag_kind="kde",␣
 ↪plot_kws=dict(alpha=0.8, marker=".", linewidth=1, color='green'))
pairsIE.map_lower(sns.regplot, scatter_kws={'alpha':1, 'color':'orange',␣
 ↪'marker':'.','s':5}, line_kws={'color':'red', 'linewidth':0.5})
plt.show()
```

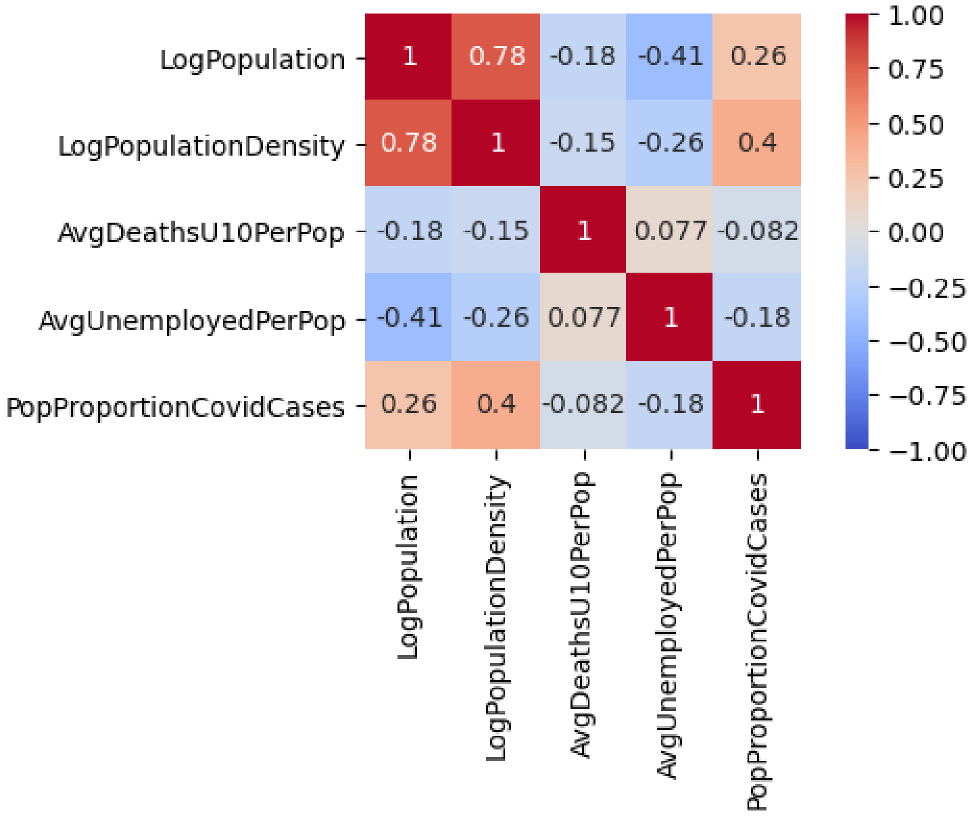


In the pairs plot above, the log transformed distrbutions of the population and population desnity variables appear to have a distribution much closer to a normal distribution than the original distributions.

Correlation plots can also be used to visualise the strength of the linear relationship between variables in a given data set. The correlation plot below shows the correlation coefficients between each pair of variables in the counties data.

The plot uses heatmap colouring to vary the colour in each square with respect to its corresponding correlation coefficient, with strong positive correlation coefficients (with values close to 1) displayed in dark red squares and strong negative coefficients (with values close to -1) displayed in dark blue squares. This helps to visualise the relative strength of each correlation coefficient and determine which variables have a strong and weak relationship.

[16]:
```
# get correlation matrix for the data with test values filtered out
corr_IE = (dataIE_num.iloc[train_val_select[:train_val_size],:].
 ↪reset_index(drop=True)).corr(method='pearson')
# plot correlation plot heatmap
fig, ax = plt.subplots(figsize=(6,3), dpi= 100)
sns.heatmap(corr_IE, vmin=-1, vmax=1, square=True, annot=True, ␣
 ↪cmap='coolwarm', ax=ax)
plt.show()
```

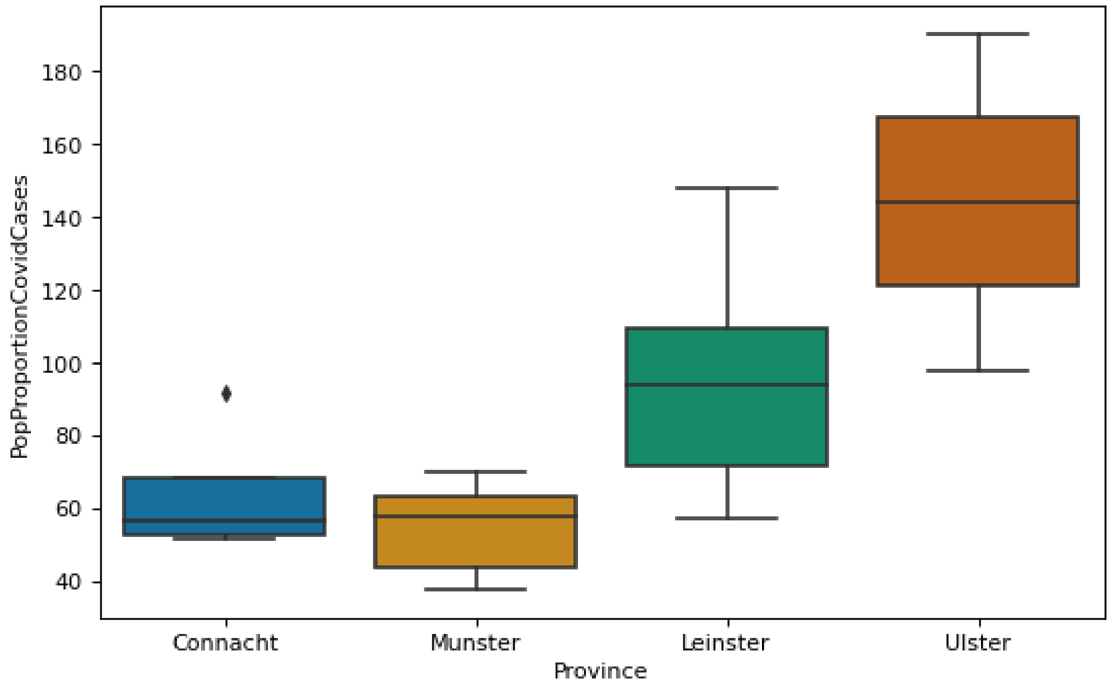


	LogPopulation	LogPopulationDensity	AvgDeathsU10PerPop	AvgUnemployedPerPop	PopProportionCovidCases
LogPopulation	1	0.78	-0.18	-0.41	0.26
LogPopulationDensity	0.78	1	-0.15	-0.26	0.4
AvgDeathsU10PerPop	-0.18	-0.15	1	0.077	-0.082
AvgUnemployedPerPop	-0.41	-0.26	0.077	1	-0.18
PopProportionCovidCases	0.26	0.4	-0.082	-0.18	1

Categorical Variables

Different methods must be used when investigating the relationship between a categorical variable and a continuous variable. Below, a box plot of the number of reported COVID-19 cases per 10,000 population grouped by province shows that there appears to be significant relationship between these two variables.

```
[17]: # gete order of PopulationProportionCovidCases median data grouped by province
order = (dataIE.iloc[train_val_select[:train_val_size],:].
 ↪reset_index(drop=True)).groupby("Province")["PopProportionCovidCases"].
 ↪median().sort_values(ascending=False)[::-1].index

# create boxplot of PopProportionCovidCases grouped by province, with test data␣
 ↪excluded
plt.figure(figsize=(8,5), dpi= 80)
sns.boxplot(y="PopProportionCovidCases", x="Province", data=(dataIE.
 ↪iloc[train_val_select[:train_val_size],:].reset_index(drop=True)),␣
 ↪palette="colorblind", order = order)
plt.show()
```



4.1.3 Statistical Analysis - Regression Modelling

Standardisation

When numerous variables of different units and scales are used as predictor variabes for a model, there is a possibilty that variables will not contribute equally to the model. This can be resolved by standardising the data by subtracting the mean and dividing by standard deviation.

The first 3 rows of the standardised data frame containing all numeric predictor variables for the model are printed below.

[18]:
```
# create new data frame with standardised numerical data
X_IE_sd = (dataIE_num - dataIE_num.mean())/dataIE_num.std()

# drop PopulationProportionCovidCases (don't need to include dependent variable)
X_IE_sd = X_IE_sd.drop(['PopProportionCovidCases'], axis=1)

# reset index values
X_IE_sd.reset_index(inplace=True, drop=True)

X_IE_sd.head(3)
```

[18]:
```
   LogPopulation  LogPopulationDensity  AvgDeathsU10PerPop  \
0      -0.997717              0.204512           -0.200954
1      -0.623498             -0.445439           -0.765318
2      -0.052204             -0.449156            0.528590

   AvgUnemployedPerPop
0             1.256035
1            -0.235669
2            -0.581885
```

Predictor and Response Variable Matrices

The X matrix corresponding to the predictor or independent variables for the model can be created by combining the standardised data frame above with the *Province* categorical variable. Variables which don't appear to have a significant relationship with the response variable *PopulationProportionCovidCases*, as identified during exploratory data analysis, can also be omitted from the X matrix at this point. In particular, *AvgDeathsU10PerPop* is extremely weakly correlated with the predictor variable and there is no evidence of a significant relationship between the variables, so this will not be considered when creating the regression models.

The y matrix corresponding to the response or dependant variable for the model is simply the *PopulationProportionCovidCases* column from the *dataIE* data frame.

[19]:
```
# copy standardised values to X matrix
X_IE = X_IE_sd.copy()
# drop AvgDeathsU10PerPop column, not a required predictor variable
X_IE = X_IE.drop(['AvgDeathsU10PerPop'], axis=1)
# add Province column to X matrix
X_IE['Province'] = dataIE.Province
# convert province column into indicator columns
X_IE = pd.get_dummies(data=X_IE, drop_first=True)

# create y matrix
y_IE = dataIE.PopProportionCovidCases.copy()
```

Test, Training and Validation Data

In order to test the test each regression model using unbiased data, a random sample of the total data can be separated from the remaining data and reserved for testing. As specified earlier, 20 percent of the total available data will be reserved for testing and will be split using the *train_val_select* order vector created earlier, with the remaining data used for training and validation.

Cross validation will be used to validate each regression model, and again the aim for this project is to use 20 percent of the total available data for vallidation purposes. As the training and validation data makes up 80 percent of the total data, k-fold cross validation with 4 folds can be used to use 20 percent of the data for validation with the remaining 60 percent used for training the model.

[20]:
```
# create train/validation and test sets
X_train_val = X_IE.iloc[train_val_select[:train_val_size],:].
 ↪reset_index(drop=True)
X_test = X_IE.iloc[train_val_select[train_val_size:],:].reset_index(drop=True)
y_train_val = y_IE[train_val_select[:train_val_size]].reset_index(drop=True)
y_test = y_IE[train_val_select[train_val_size:]].reset_index(drop=True)

# use 4 fold cross validation (60% training, 20% validation of total data)
cv_IE = KFold(n_splits=4)
```

Regression Model Training and Testing

Four different variations of regression models will be built and tested to model the Irish counties data.

The default parameters will be used for the multiple linear regression models. K-fold cross validation will be performed initally for this regression technique using the validation and training sets created above. After cross validation is complete, a final multiple linear regression model will be fitted to the full set of training and validation data available, and this model will be used to predict the values from the test data set, *y_test*, based on their corresponding x values in *X_test*.

For the remaining three regression techniques (lasso regression, ridge regression, random forest regression) cross validation will be used to assess the performance of each model across a range of parameters to determine the the optimum parameter values for each model, based on the validation and training data sets. After the optimum parameters have been determined, a final model using for each of these regression techniques will be created using the obtained parameters value, by fitting to the full set of training and validation data available. Again, these models will be used to predict the values from the test data set.

In order to produce consistent plots and report the same values for each regression model, a function can be used.

The function below creates two variations of the same plot to help visualise the results from the cross validation performed on each model, the first a 3D plot separating the results from each cross validation iteration, and the second a 2D plot displaying all results on a single plot but with different colours and markers used to distinguish between values from each cross validation iteration.

This function also calculates the Root Mean Squared Error (RMSE) value for each cross validation iteration, and returns the average RMSE value. This value can be used to evaluate the perfromance of each model, and in particular will assist with identifying the parameters that optimise each model.

[21]:
```
# function to perform cross validation
def modelCrossVal(model, cv, show_plot=True):

    # create vectors of colours and markers for the plot
    colours=['b','orange','g','r']
    markers=['o','*','+','x']

    # if show_plot is True, set up plot figure
    if show_plot==True:
        # create new figure
        fig = plt.figure(figsize=(10,5), dpi= 80)

        # create first subplot (3d plot)
        ax = fig.add_subplot(121,projection='3d')
        ax.set_xlabel('Cross Validation Iteration')
        ax.set_ylabel('Actual Y Value')
        ax.set_zlabel('Predicted Y Value')

        # create second plot (2d plot)
        ax2 = fig.add_subplot(122)
        ax2.set_xlabel('Actual Y value')
        ax2.set_ylabel('Predicted Y value')

    # create empty list to hold Root Mean Squared Error values
    RMSE_reg=[]

    # set index value to zero before starting for loop
    ind=0

    # train and validate model for each cross validation iteration
    for train_index, val_index in cv.split(X_train_val):

        # create X and y training and validation sets for current cross␣
↪validation iteration
        X_train, X_val = X_train_val.iloc[list(train_index)], X_train_val.
↪iloc[list(val_index)]
        y_train, y_val = y_train_val.iloc[list(train_index)], y_train_val.
↪iloc[list(val_index)]

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

        # precit validation values from
        y_val_pred = model.predict(X_val)

        # if show_plot is True, plot predicted y vals vs actual y vals
        if show_plot==True:
            # get max/min plot range values and add 10% either side
            min_y = np.min((np.min(y_val), np.min(y_val_pred)))
            max_y = np.max((np.max(y_val), np.max(y_val_pred)))
            y_range_10pc = (max_y - min_y) / 10
            min_y = min_y - y_range_10pc
            max_y = max_y + y_range_10pc

            # add points and line to 3d plot
            ax.plot(ind * np.ones(len(y_val)), y_val, y_val_pred, '.',␣
↪color=colours[ind])
            ax.plot(ind * np.ones(2), [min_y,max_y], [min_y,max_y], ls="--",␣
↪color=colours[ind])

            # add points to 2d plot
            ax2.plot(y_val, y_val_pred, '.',color=colours[ind],␣
↪marker=markers[ind])

        # calculate RMSE for current model and append to list
        RMSE_reg.append(np.sqrt(np.mean((y_val_pred - y_val)**2)))

        # increment index value
        ind=ind+1

    # if show_plot is True, plot line on 2d plot and adjust subplot spacing
    if show_plot==True:
        ax2.plot([min_y,max_y], [min_y,max_y], ls="--", alpha=0.5)
        plt.subplots_adjust(wspace=0.5, top=0.75)
        plt.show()

    # return average Root Mean Squared Error value from all cross validation␣
 ↪iterations
    return np.mean(RMSE_reg)
```

After cross validation is complete, a final regression model can be fitted for each regression technique. Before attempting to predict the unknown test data using these models, it is possible to obtain some useful information from them, including the coefficients of the model, the R-squared value and the adjusted R-squared value.

The coefficients of the model can be useful to report when evaluating a regression model. These will give an insight into which predictor variables have the largest influence on the response variable.

Again, a function can be created to extract the intercept term and the model coefficients for each of the predictor variables, and return the values in a data frame. As some models may not have such coefficients available (e.g. random forest regression models), the function will also confirm check that these coefficients exist before attempting to read them from the model, to avoid causing an error.

The R-squared and adjusted R-squared values can also be calculated for the model at this stage. These metrics can be used to evaluate how well of each of the final regression models fit their fitted data and compare them to one another.

[22]:
```
# fit data to model, and return R-squared, adjusted R-squared and model␣
 ↪coefficients
def modelCoefsR2(model, X_train, y_train):
    # fit training model to data
    model.fit(X_train,y_train)

    # check if model has coefficients (i.e. not SVR)
    if hasattr(model,'coef_'):

        # create list including all predictor variable names
        names = X_train.columns.tolist()
        # insert 'Intercept' at start of the list
        names.insert(0,'Intercept')

        # create list of coefficients associated with each predictor
        coefs = model.coef_.tolist()
        # insert intercept coefficient at start of the list
        coefs.insert(0,model.intercept_)

        # create data frame, add lists to the data frame
        df_coefs = DataFrame()
        df_coefs['Parameter'] = names
        df_coefs['Coefficient'] = coefs

    # if coeeficients are not available, assign a string to 'df_coef'
    else:
        df_coefs = 'There are no coefficients available for this regression␣
↪model.'

    # get predicted y values for training data
    y_pred = model.predict(X_train)

    # calculate R2 and R2 adjusted
    R2 = 1 - np.sum((y_pred - y_train)**2) / np.sum((np.mean(y_train) -␣
↪y_train)**2)
    R2_adj = 1 - ((1 - R2) * (X_train.shape[0] - 1)) / (X_train.shape[0] -␣
↪X_train.shape[1] - 1)

    return df_coefs, R2, R2_adj
```

Finally, a function can be created to plot the same graphs and return the same statistics for each final model, to standardise the testing of the final regression models.

The function below creates a plot which compares the y values predicted by the model with the actual y values from the test data set.

This function again calculates the Root Mean Squared Error and Normalised Root Mean Squared Error values for the given model, which can be used to evaluate how well it performs at predicting the unknown test data.

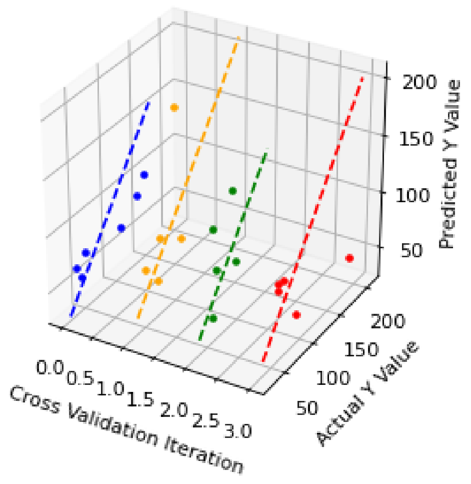
[23]:
```
# function to test final models
def modelTest(model, X_train, y_train, X_test, y_test):

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

    # predict y values from test X values
    y_pred = model.predict(X_test)

    # get max/min plot range values and add 10% either side
    min_y = np.min((np.min(y_test), np.min(y_pred)))
    max_y = np.max((np.max(y_test), np.max(y_pred)))
    y_range_10pc = (max_y - min_y) / 10
    min_y = min_y - y_range_10pc
    max_y = max_y + y_range_10pc

    # plot actual y values vs predicted y values
    fig = plt.figure(figsize=(3.5,3.5), dpi= 80)
    plt.plot(y_test, y_pred,'x')
    plt.plot([min_y,max_y], [min_y,max_y], ls="--")
    plt.xlabel('Actual Y value')
    plt.ylabel('Predicted Y value')

    # calculate RMSE and NRMSE
    RMSE = np.sqrt(np.mean((y_pred - y_test)**2))
    NRMSE = RMSE / (y_test).std()

    return RMSE, NRMSE
```

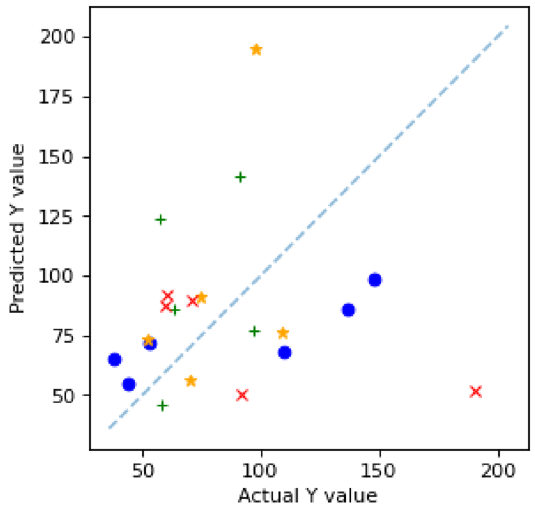
Multiple Linear Regression Model

Cross Validation

After creating a multiple linear regression model, the cross validation function defined above can be used to perform cross validation on the model and plot the results, as well as returning the mean RMSE value.

[24]:
```
# Create multiple linear regression model
mlr = linear_model.LinearRegression()

# Use model prediction plot function with linear model
mlr_RMSE = modelCrossVal(mlr, cv_IE, True)

# Print RMSE for linear model
print("\nMean RMSE for Mulitple Linear Regression Model:", mlr_RMSE,'\n')
```





```
Mean RMSE for Mulitple Linear Regression Model: 48.012526007754914
```

Model Coefficients, R^2 and adjusted R^2

The final multiple linear regression model can be fitted with the full set of training and validation data, *X_train_val* and *y_train_val*. The *modelCoefsR2* function defined above can be used to report the model coefficients, R-squared value and adjusted R-squared value for the model.

[25]:
```
# get coefficients, R2 and R2 adjusted values
mlr_coefs, mlrR2, mlrR2adj = modelCoefsR2(mlr, X_train_val, y_train_val)

# print R2 and R2 adjusted
print("\nR-squared value for final multiple regression model:", np.
 ↪round(mlrR2,3))
print("\nAdjusted R-squared value for final multiple regression model:", np.
 ↪round(mlrR2adj,3),"\n")

# print the model coefficients
display(Markdown(mlr_coefs.to_markdown()))
```

```
R-squared value for final multiple regression model: 0.686

Adjusted R-squared value for final multiple regression model: 0.552
```

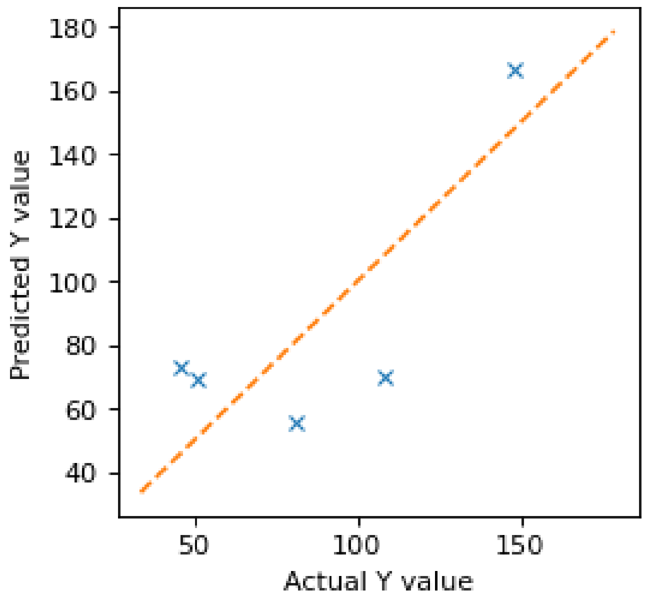
	Parameter	Coefficient
0	Intercept	76.699
1	LogPopulation	-6.92585
2	LogPopulationDensity	19.3971
3	AvgUnemployedPerPop	-9.56871
4	Province_Leinster	5.83705
5	Province_Munster	-18.3672
6	Province_Ulster	84.3863

Predicting the Unknown Test Data

The *modelTest* function also defined above can then be used to predict the values in the test data, and plot the results. It will also return the root mean squared error calculated for the tested values, which can be used when evaluating the performance of the final models.

[26]:
```
# Test model on unknown test data
mlr_pred_RMSE, mlr_pred_NRMSE = modelTest(mlr, X_train_val, y_train_val,␣
 ↪X_test, y_test)

# print RMSE
print("\nRMSE for model predictions of the test data:", np.
 ↪round(mlr_pred_RMSE,3), '\n')
```

```
RMSE for model predictions of the test data: 26.631
```



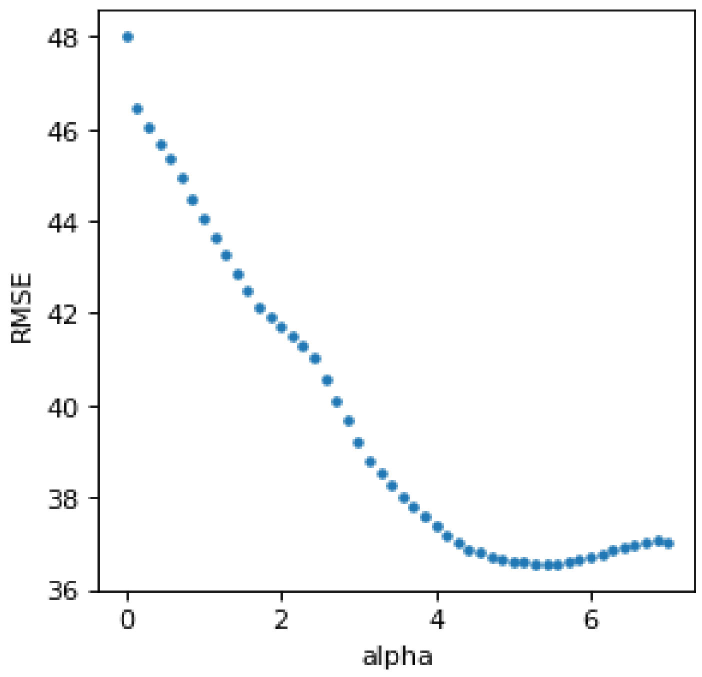
Lasso Regression Model

Cross Validation

For lasso regression, choosing different values for the α parameter can affect the performance of the model. The cross validation function can be used to evaluate the performance of the model for a range of different α values and return the value which minimizes the RMSE value outputted from the function. Plotting the RMSE value obtained for each α value will also help identify what is the best value for the parameter.

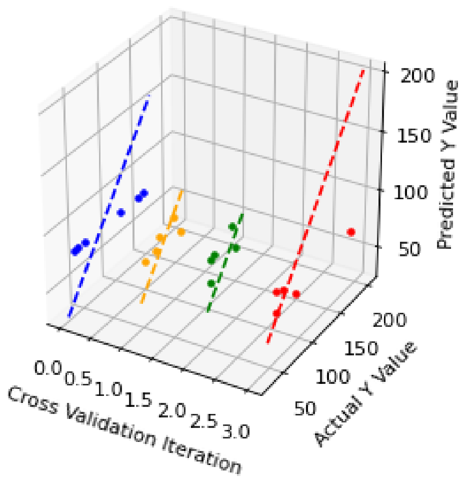
[27]:
```
# create vector of 50 evenly spaces points between 0 and 7
alpha_vals = np.linspace(0.0001, 7, num=50)

# create empty list to hold all RMSE values
RMSEs = []

# set start values for min RMSE and best alpha values
min_RMSE = 9999
best_alpha = 0

# Loop through all alpha values
for i in alpha_vals:

    # create Lasso regression model using current alpha value
    lasso = linear_model.Lasso(alpha=i)

    # calculate RMSE of current model, and append value to list
    lasso_RMSE = modelCrossVal(lasso, cv_IE, False)
    RMSEs.append(lasso_RMSE)

    # if current RMSE is less than the minimum RMSE, set current RMSE to min␣
 ↪RMSE and current alpda to best alpha
    if lasso_RMSE < min_RMSE:
        min_RMSE = lasso_RMSE
        best_alpha = i

# plot RMSE vs alpha values
plt.figure(figsize=(4,4), dpi= 80)
plt.plot(alpha_vals, RMSEs, '.')
plt.xlabel('alpha')
plt.ylabel('RMSE')

# print alpha value that minimizes RMSE of model
print("\nOptimum alpha value:", np.round(best_alpha,3))
print("\nMinimum RMSE value with optimised alpha value:", np.round(min_RMSE,3),␣
 ↪"\n")
```

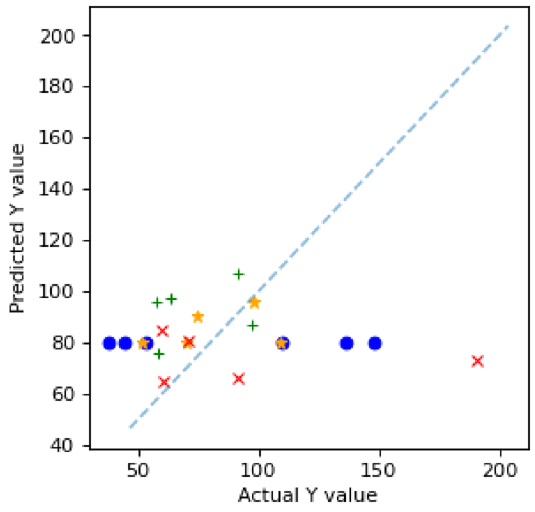
```
Optimum alpha value: 5.429

Minimum RMSE value with optimised alpha value: 36.558
```



Once an optimum α is chosen, the cross validation function can be used again using the optimum value in the lasso regression model to plot the results from the cross validation and return the mean RMSE value.

[28]:
```
# Create lasso regression model
lasso = linear_model.Lasso(alpha=best_alpha)

# Use model prediction plot function with lasso model
lasso_RMSE = modelCrossVal(lasso, cv_IE, True)

# Print MSE for lasso model
print("\nRMSE for Lasso Regression Model:", lasso_RMSE, '\n')
```





```
RMSE for Lasso Regression Model: 36.558214830348845
```

Model Coefficients, R^2 and adjusted R^2

The final lasso regression model can be fitted with the full set of training and validation data, *X_train_val* and *y_train_val*. The *modelCoefsR2* function defined above can be used to report the model coefficients, R-squared value and adjusted R-squared value for the model.

[29]:
```
# get coefficients, R2 and R2 adjusted values
lasso_coefs, lassoR2, lassoR2adj = modelCoefsR2(lasso, X_train_val, y_train_val)

# print R2 and R2 adjusted
print("\nR-squared value for final lasso regression model:", np.
 ↪round(lassoR2,3))
print("\nAdjusted R-squared value for final lasso regression model:", np.
 ↪round(lassoR2adj,2),"\n")

# print the model coefficients
display(Markdown(lasso_coefs.to_markdown()))
```

```
R-squared value for final lasso regression model: 0.277

Adjusted R-squared value for final lasso regression model: -0.03
```

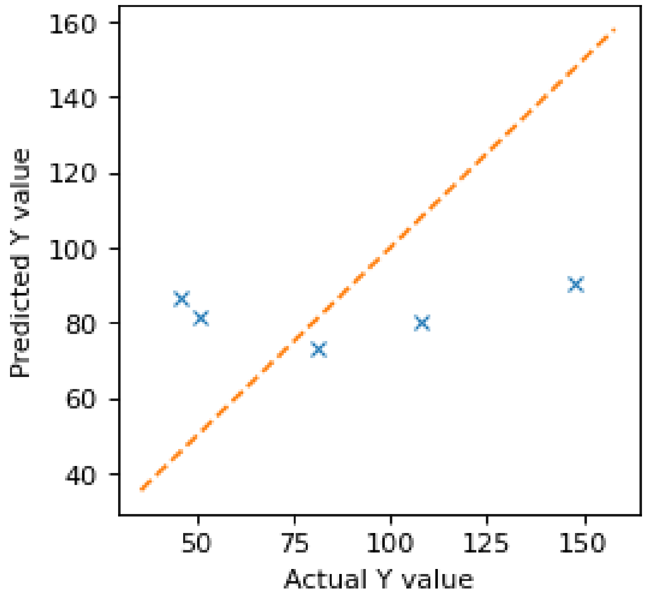
	Parameter	Coefficient
0	Intercept	84.8073
1	LogPopulation	0
2	LogPopulationDensity	9.50658
3	AvgUnemployedPerPop	-0
4	Province_Leinster	0
5	Province_Munster	-7.18058
6	Province_Ulster	8.03196

Predicting the Unknown Test Data

The *modelTest* function can be used to predict the values in the test data, and plot the results. It will also return the root mean squared error calculated for the tested values, which can be used when evaluating the performance of the final models.

[30]:
```
# Test model on unknown test data
lasso_pred_RMSE, lasso_pred_NRMSE = modelTest(lasso, X_train_val, y_train_val,␣
↪X_test, y_test)

# print RMSE
print("\nRMSE for model predictions of the test data:", np.
↪round(lasso_pred_RMSE, 3), '\n')
```

```
RMSE for model predictions of the test data: 36.816
```



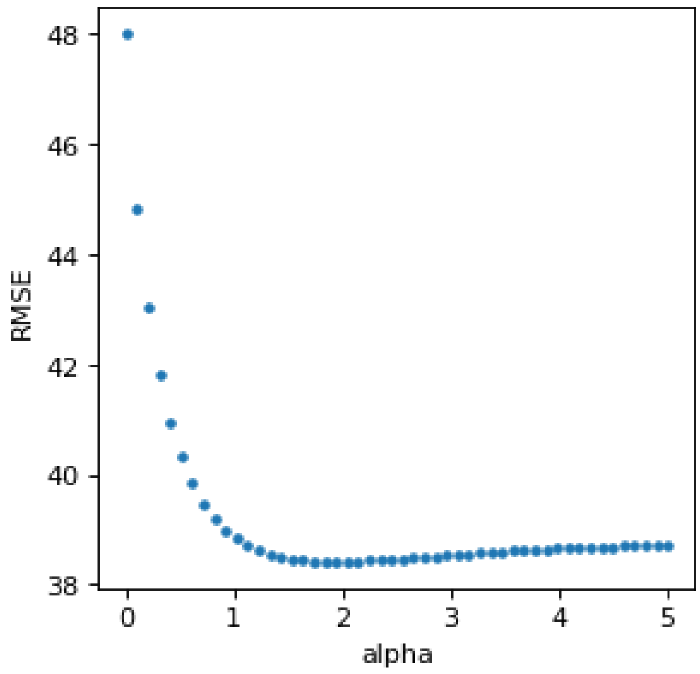
Ridge Regression Model

Cross Validation

Similar to lasso regression, choosing different values for the α parameter for ridge regression can also affect the performance of the model. The cross validation function can be used to evaluate the performance of the model for a range of different α values and return the value which minimizes the RMSE value outputted from the function. Plotting the RMSE value obtained for each α value will also help identify what is the best value for the parameter.

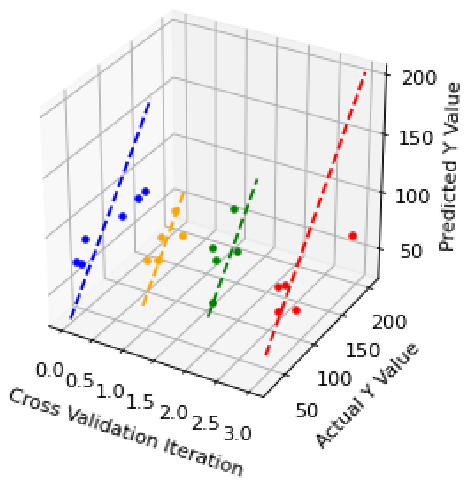
[31]:
```
# create vector of 50 evenly spaces points between 0 and 5
alpha_vals=np.linspace(0.0,5,num=50)

# create empty list to hold all RMSE values
RMSEs = []

# set start values for min RMSE and best alpha values
min_RMSE = 9999
best_alpha = 0

# Loop through all alpha values
for i in alpha_vals:

    # create Ridge regression model using current alpha value
    ridge = Ridge(alpha=i)

    # calculate RMSE of current model, and append value to list
    ridge_RMSE = modelCrossVal(ridge, cv_IE, False)
    RMSEs.append(ridge_RMSE)

    # if current RMSE is less than the minimum RMSE, set current RMSE to min␣
 ↪RMSE and current alpda to best alpha
    if ridge_RMSE < min_RMSE:
        min_RMSE = ridge_RMSE
        best_alpha = i

# plot RMSE vs alpha values
plt.figure(figsize=(4,4), dpi= 80)
plt.plot(alpha_vals, RMSEs, '.')
plt.xlabel('alpha')
plt.ylabel('RMSE')

# print alpha value that minimizes RMSE of model
print("\nOptimum alpha value:", np.round(best_alpha,3))
print("\nMinimum RMSE value with optimised alpha value:", np.round(min_RMSE,3),␣
 ↪"\n")
```

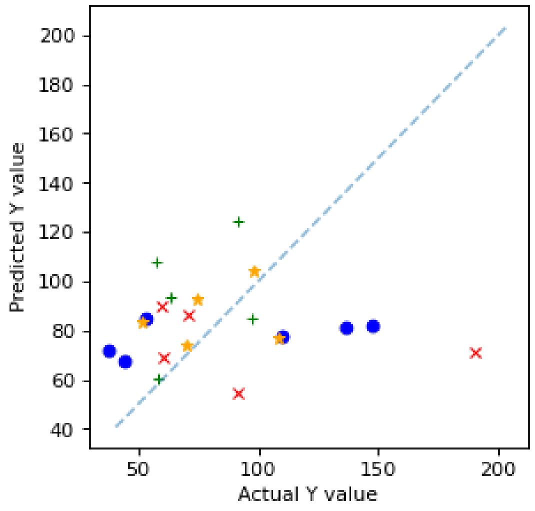
```
Optimum alpha value: 1.939

Minimum RMSE value with optimised alpha value: 38.415
```



Once an optimum α is chosen, the cross validation function can be used again using the optimum value in the ridge regression model to plot the results from the cross validation and return the mean RMSE value.

[32]:
```
# Create ridge regression model
ridge = Ridge(alpha=best_alpha)

# Use model prediction plot function with ridge model
ridge_RMSE = modelCrossVal(ridge, cv_IE, True)

# Print RMSE for ridge model
print("\nRMSE for Ridge Regression Model:", ridge_RMSE, '\n')
```





```
RMSE for Ridge Regression Model: 38.414956124257124
```

Model Coefficients, R^2 and adjusted R^2

The final ridge regression model can be fitted with the full set of training and validation data, *X_train_val* and *y_train_val*. The *modelCoefsR2* function defined above can be used to report the model coefficients, R-squared value and adjusted R-squared value for the model.

[33]:
```
# get coefficients, R2 and R2 adjusted values
ridge_coefs, ridgeR2, ridgeR2adj = modelCoefsR2(ridge, X_train_val, y_train_val)

# print R2 and R2 adjusted
print("\nR-squared value for final ridge regression model:", np.
 ↪round(ridgeR2,3))
print("\nAdjusted R-squared value for final ridge regression model:", np.
 ↪round(ridgeR2adj,2),"\n")

# print the model coefficients
display(Markdown(ridge_coefs.to_markdown()))
```

```
R-squared value for final ridge regression model: 0.567

Adjusted R-squared value for final ridge regression model: 0.38
```

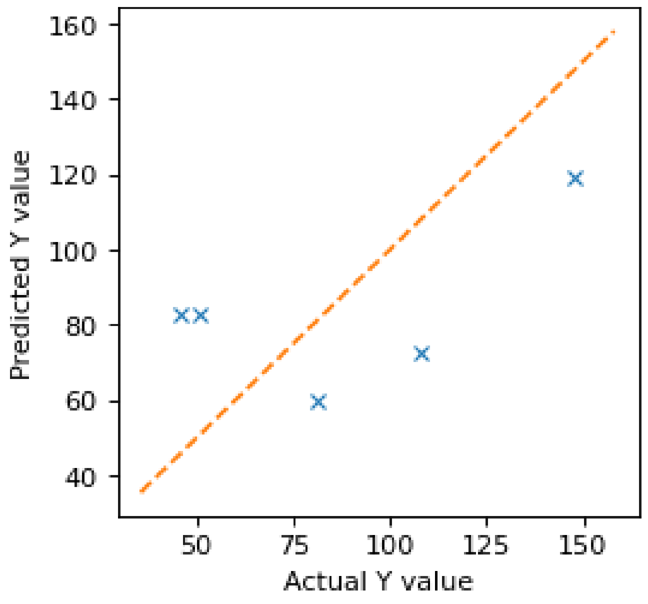
	Parameter	Coefficient
0	Intercept	84.2702
1	LogPopulation	0.214909
2	LogPopulationDensity	12.713
3	AvgUnemployedPerPop	-4.89288
4	Province_Leinster	1.73713
5	Province_Munster	-21.5334
6	Province_Ulster	35.9524

Predicting the Unknown Test Data

The *modelTest* function can be used to predict the values in the test data, and plot the results. It will also return the root mean squared error calculated for the tested values, which can be used when evaluating the performance of the final models.

[34]:
```
# Test model on unknown test data
ridge_pred_RMSE, ridge_pred_NRMSE = modelTest(ridge, X_train_val, y_train_val,␣
 ↪X_test, y_test)

# print RMSE
print("\nRMSE for model predictions of the test data:", np.
 ↪round(ridge_pred_RMSE, 3), '\n')
```

```
RMSE for model predictions of the test data: 31.43
```



Random Forest Regression Model

Cross Validation

Two parameters will be considered for this project when assessing the performance of random forest regression models, the number of estimators and maximum depth. The cross validation function can be used to evaluate the performance of the model for a range of different number of estimators and maximum depth values and return the value which minimizes the RMSE value outputted from the function. Nested for loops can be used to test every combination of given number of estimators and maximum depth values.

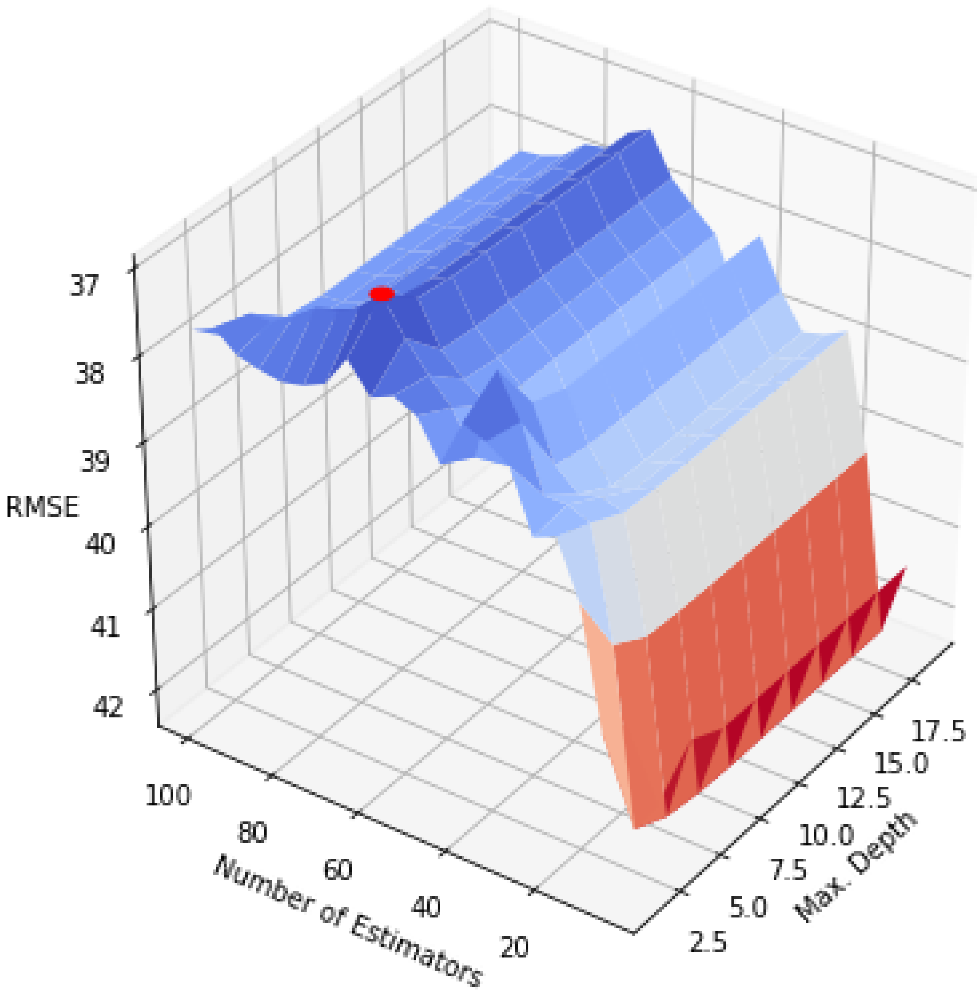
[35]:
```
# create vectors of integers for max depth and n_estimators
depth_vals = range(1,21,2)
est_vals = range(5,105,5)

# create empty list to hold all RMSE values
RMSEs = [[0 for x in depth_vals] for x in range(len(est_vals))]

# set start values for min RMSE and best depth and num. estimators values
min_RMSE = 9999
best_depth = 0
best_n_est = 0

# create i/j index values to track current iteration number of for loops
i = 0
j = 0

# Loop through all n_estimators values
for n_est in est_vals:
    # Loop through all max_depth values
    for max_d in depth_vals:
        # create Random Forest regression model using current max_depth and␣
 ↪n_estimator value
        rf = RandomForestRegressor(n_estimators=n_est, max_depth=max_d,␣
 ↪random_state=123)

        # calculate RMSE of current model, and write value to list
        rf_RMSE = modelCrossVal(rf, cv_IE, False)
        RMSEs[i][j]=rf_RMSE

        # if current RMSE < minimum RMSE...
        if rf_RMSE < min_RMSE:
            # set current RMSE to min RMSE and current n_est/depth to best␣
 ↪n_est/depth
            min_RMSE = rf_RMSE
            best_n_est = n_est
            best_depth = max_d
        # increment j index
        j = j + 1

    # increment i index
    i = i + 1
    # reset j index
    j = 0

# print max depth value that minimizes RMSE of model
print("\nOptimum Number of Estimators:", best_n_est)
print("\nOptimum Max. depth value:", best_depth)
print("\nMinimum RMSE value with optimised parameters", np.round(min_RMSE, 3),␣
↪"\n")
```

```
Optimum Number of Estimators: 65

Optimum Max. depth value: 3

Minimum RMSE value with optimised parameters 36.962
```

To visualise the RMSE values obtained for every combination of given maximum depth values and number of estimators values, a 3D surface plot can be used. The surface plot below also includes a red dot at the obtained values for the optimum maximum depth and number of estimators. The Z-axis on the plot has been inverted to make it easier to identify the point in the surface where the minimum RMSE is located, which would equate to the highest point on this plot. Indeed, the optimum point plotted on the surface does appear to fall on the highest peak, meaning that it produces the lowest RMSE value.

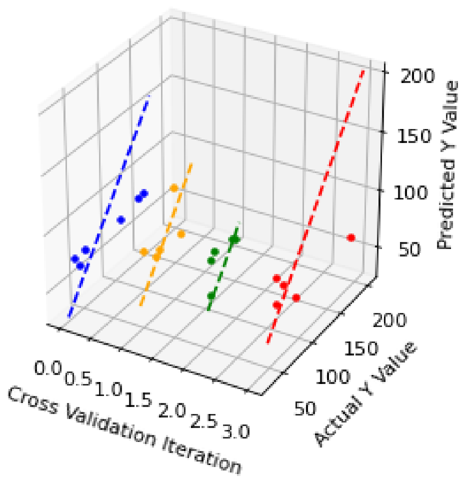
[36]:
```
# create grid of all combinations of max depth values and n_estimators values
X,Y = np.meshgrid(depth_vals, est_vals)
# Convert RMSEs 2D list from cross validation testing to a numpy array
Z = np.array(RMSEs)

# create 3D plot
fig = plt.figure(figsize=(7,7))
ax = fig.add_subplot(111,projection='3d')
ax.set_xlabel('Max. Depth')
ax.set_ylabel('Number of Estimators')
ax.set_zlabel('RMSE')

# plot surface with max_depth on X-axis, n_estimators on Y-axis and RMSEs on␣
↪Z-axis
ax.plot_surface(X, Y, Z, alpha=1, cmap=cm.coolwarm, linewidth=0)

# add point to the plot at the optimum max_depth value and n_estimators value
optimised_point = Ellipse((best_depth, best_n_est), 0.75, 4, ec='r', fc="r")
```

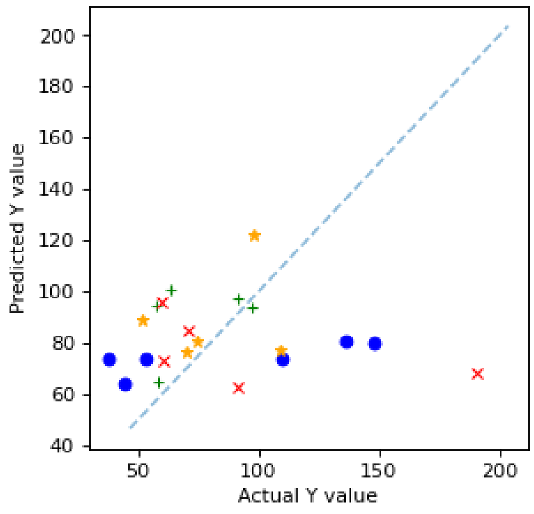
```
ax.add_patch(optimised_point)
art3d.pathpatch_2d_to_3d(optimised_point, z=min_RMSE, zdir="z")

# Set view angle of plot and reverse Z-axis
ax.elev = 35
ax.azim = -145
ax.invert_zaxis()

plt.show()
```



Once optimum maximum depth and number of estimators values are chosen, the cross validation function can be used again using the optimum values in the random forest regression model to plot the results from the cross validation and return the mean RMSE value.

[37]:
```
# Create random forest regression model
rf = RandomForestRegressor(n_estimators=best_n_est, max_depth=best_depth,␣
 ↪random_state=13)

# Use model prediction plot function with random forest model
rf_RMSE = modelCrossVal(rf, cv_IE, True)

# Print RMSE for random forest model
print("\nRMSE for Random Forest Regression Model:", rf_RMSE,'\n')
```





```
RMSE for Random Forest Regression Model: 37.63496932112645
```

Model Coefficients, R^2 and adjusted R^2

The final random forest regression model can be fitted with the full set of training and validation data, *X_train_val* and *y_train_val*. The *modelCoefsR2* function defined above can be used to report the R-squared value and adjusted R-squared value for the model.

[38]:
```python
# get coefficients, R2 and R2 adjusted values
rf_coefs, rfR2, rfR2adj = modelCoefsR2(rf, X_train_val, y_train_val)

# print R2 and R2 adjusted
print("\nR-squared value for final random forest regression model:", np.
 ↪round(rfR2,3))
print("\nAdjusted R-squared value for final random forest regression model:",␣
 ↪np.round(rfR2adj,2),"\n")

# print the model coefficients
print(rf_coefs)
```

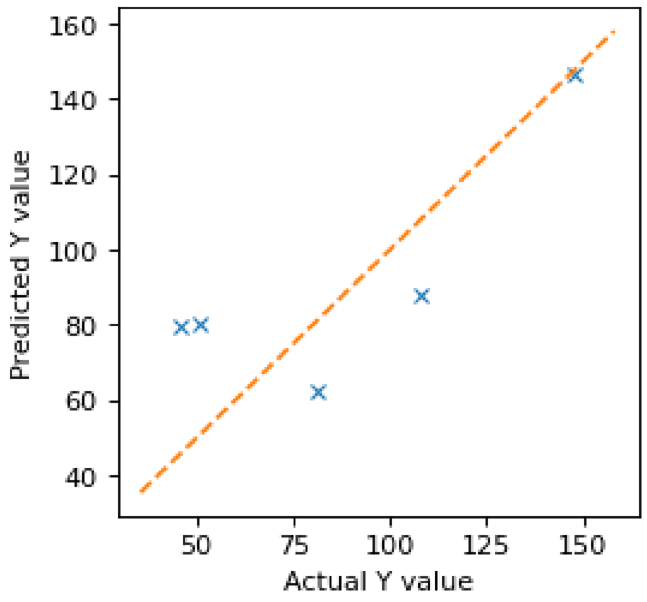
```

R-squared value for final random forest regression model: 0.785

Adjusted R-squared value for final random forest regression model: 0.69

There are no coefficients available for this regression model.
```

Predicting the Unknown Test Data

The *modelTest* function can be used to predict the values in the test data, and plot the results. It will also return the root mean squared error calculated for the tested values, which can be used when evaluating the performance of the final models.

[39]:
```
# Test model on unknown test data
rf_pred_RMSE, rf_pred_NRMSE = modelTest(rf, X_train_val, y_train_val, X_test,␣
 ↪y_test)

# print RMSE
print("\nRMSE for model predictions of the test data:", np.round(rf_pred_RMSE,␣
 ↪3), '\n')
```

```
RMSE for model predictions of the test data: 23.751
```



Base Case Model Comparison

While figures obtained above can be used to compare each regression model against each other, it can also be useful to calculate the root mean squared error for a base model which simply returns the mean value of the given training data for all predictions.

[40]:
```
test_RMSE_IE = np.sqrt(np.mean((np.mean(y_train_val) - y_test)**2))
print("RMSE for base model predictions:", np.round(test_RMSE_IE,3))
```

```
RMSE for base model predictions: 38.04
```

4.2 Countries Data Analysis

4.2.1 Data Preprocessing

COVID-19 Cases an Population Data

The 'daily-cases-by-country.csv' data file contains a lot of useful information relating to various different countries. The population, continent, total number of reported COVID-19 cases, and number of days since the first confirmed case for each country can be obtined from the file. However, several preprocessing steps need to be carried out to get the data in the required format.

After reading the file into a data frame, columns that are not required and rows that contain *NaN* values can be dropped from the data frame.

The World Bank API will be used later to obtain additional data relating to the countries in the data. However, not all countries in the given data files are available in the World Bank data. To find which countries are common to both, the unique values in the *countryterritorycode* column can be compared with the country ids for all available countries in the World Bank API. Countries that are not common to both can then be dropped from the data frame.

```
[41]: # read in csv file
countryCases1 = pd.read_csv(path + 'daily-cases-by-country.csv')

# drop unused columns and columns with NaN values
countryCases1 = countryCases1.drop(['day', 'month', 'year', 'geoId',␣
 ↪'Cumulative_number_for_14_days_of_COVID-19_cases_per_100000'], axis=1)
countryCases1 = countryCases1.dropna()

# get list of unique countries in dataset
countries_countryCases = countryCases1['countryterritoryCode'].unique()

# get list of all country 'id's in world bank data
countries_wbd = [country['id'] for country in wbd.get_country()]

# create set of all countries that feature in both the csv file and world bank␣
 ↪data
countries_common = set(countries_countryCases) & set(countries_wbd)
countries_common = sorted(countries_common)

# filter out rows of countries not in wbd
countryCases1 = countryCases1[countryCases1['countryterritoryCode'].
 ↪isin(countries_common)]
```

Next, columns where no deaths and no cases were reported can be filtered out of the dataframe. To calculate the total number of reported cases and total number deaths for each country, a second dataframe (*countryCasesSum*) can be created which groups the original data frame by country and sums the values for each country.

Then, the original data frame (*countryCases2*) can also be grouped by country, but in this case only the last row for each country will be kept. As rows where no deaths or cases occurred were dropped from the data frame, the last row for each country corresponds to the row where the first case was reported for the related country.

[42]:
```
# filter out rows where both cases and deaths are 0
countryCases2 = countryCases1.query('cases != 0 | deaths != 0')

# group by countries and get sum of values
countryCasesSum = countryCases2.
↪groupby(['countriesAndTerritories','countryterritoryCode']).sum().
↪reset_index()
# drop population and countriesAndTerritories columns
countryCasesSum = countryCasesSum.drop(['popData2019',␣
↪'countriesAndTerritories'], axis=1)
# rename cases and deaths columns to include total in names
countryCasesSum = countryCasesSum.rename(columns={'cases':'totalCases',␣
↪'deaths':'totalDeaths'})
# drop population and countriesAndTerritories columns
countryCasesSum = countryCasesSum.drop('totalDeaths', axis=1)

# group by countries and only take last value to get date for each
countryCases2 = countryCases2.groupby(['countriesAndTerritories']).last().
↪reset_index()
```

To calculate the number of days since the first reported COVID-19 case for each country, the *dateRep* column must be converted from string type to datetime. Subtracting the date of the first recorded case for each country from the latest date considered in the data set, the 17th of October 2020, and converting the result to an integer returns the number of days since the first confirmed case. Any columns that are no longer needed can then be dropped.

The *countryCasesSum* and *countryCases2* data frames can then be merged to get all of the relevant data into a single data frame. The first 3 rows of this data frame are printed below.

[43]:
```
# create copy of country cases data frame
countryCases = countryCases2.copy()

# convert dates to datetime
countryCases['dateRep'] = pd.to_datetime(countryCases['dateRep'], dayfirst=True)

# create new column of number of days since first case was reported (from␣
↪17-Oct-20), convert to integer
countryCases['DaysSinceFirstCase'] = pd.to_datetime("2020-10-17") -␣
↪countryCases['dateRep']
countryCases['DaysSinceFirstCase'] = countryCases['DaysSinceFirstCase'].dt.days.
↪astype('int64')

# drop unneeded columns
countryCases = countryCases.drop(['dateRep','cases','deaths'], axis=1)
```

```
# merge casesIE and deathsIE dataframes on their County columns
countryCases = pd.merge(countryCases, countryCasesSum, how='inner',␣
 ↪on='countryterritoryCode')

countryCases.head(3)
```

```
[43]:   countriesAndTerritories countryterritoryCode  popData2019 continentExp  \
0             Afghanistan                  AFG   38041757.0         Asia
1                 Albania                  ALB    2862427.0       Europe
2                 Algeria                  DZA   43053054.0       Africa

   DaysSinceFirstCase  totalCases
0                 235       40073
1                 222       16501
2                 234       53988
```

World Bank Indicators Data

Additional country data can be obtained using the World Bank API. For this project, the total land area and number of yearly deaths of children under 10 years of age will be considered, figures which are not available in the given data files.

These values can be obtained by using indicators for the number of deaths of children under 5 (*SH.DTH.MORT*), the number of deaths of children between 5 and 9 years of age (*SH.DTH.0509*), and the total land area (*AG.LND.TOTL.K2*), for a given year. 2016 was chosen for the year as it is the most recent year where data is available for the majority of the countries in our data set.

The number of deaths under 10 for each country can be obtained by summing the deaths under 5 and deaths between 5 and 9 years old. Columns that are no longer required can then be dropped from the data frame. The first 3 rows of the data frame are printed below.

```
[44]: # get list of metrics from world bank api for all countries in our dataset
allMetrics = wbd.get_dataframe({'SH.DTH.MORT':'DeathsU5',
                                'SH.DTH.0509':'Deaths5_9',
                                'AG.LND.TOTL.K2':'LandArea',
                                }, countries_common, data_date = datetime.
 ↪datetime(2016,1,1))

# create new column for deaths under 10 (sum deaths under 5 and deaths 5 to 9)
allMetrics['DeathsU10'] = allMetrics['DeathsU5'] + allMetrics['Deaths5_9']

allMetrics = allMetrics.reset_index()

# add country territory code as a column, and drop country
allMetrics.insert(0, 'countryterritoryCode', countries_common)

# drop unneeded columns
allMetrics = allMetrics.drop(['country','DeathsU5', 'Deaths5_9'], axis=1)

allMetrics.head(3)
```

```
[44]:   countryterritoryCode   LandArea  DeathsU10
      0                  ABW      180.0        NaN
      1                  AFG   652860.0    83090.0
      2                  AGO  1246700.0   109107.0
```

Income Groups Data

The income group of each country will also be considered for this project. This information is given in the 'country_income_groups.csv' file. After loading the file into a data frame, countries that are not in common to the World Bank data can be dropped, as well as any unrequired columns.

To simplify the merging of this data frame with other data frames, it will also help to rename the *Country Code* column to match the other data frames. The first 3 rows of the resulting data frame are shown below.

```
[45]: incomeGroups = pd.read_csv(path + 'country_income_groups.csv')

      # filter out counties not in our list of european countries
      incomeGroups = incomeGroups[incomeGroups['Country Code'].isin(countries_common)]

      # rename Country Code column to match other dataframes
      incomeGroups = incomeGroups.rename(columns={'Country Code':
       ↪'countryterritoryCode'})

      # drop unneeded columns
      incomeGroups = incomeGroups.drop(['Region','TableName'], axis=1)

      incomeGroups.head(3)
```

```
[45]:   countryterritoryCode           IncomeGroup
      0                  ABW           High income
      1                  AFG            Low income
      2                  AGO   Lower middle income
```

Unemployment Data

Unemplyment figures for each country can be obtained from the 'unemployment_by_country_by_year.csv' data file. Again, countries that are not available in the World Bank data can be dropped. Only the figures of 2019 will be considered for this project, so all columns except for *Country Code* and *2019* can be filtered out of the data frame.

Finally, the *Country Code* column can be renamed to match the other data frames, and the *2019* column can be renamed to a more logical name for our analysis, *UnemployedPercent*. The first 3 rows of the resulting data frame are printed below.

[46]:
```
# read in csv files while skipping first 4 rows
unemployWorld = pd.read_csv(path + 'unemployment_by_country_by_year.
 ↪csv',skiprows=[0,1,2,3])

# filter out rows of countries not in wbd
unemployWorld = unemployWorld[unemployWorld['Country Code'].
 ↪isin(countries_common)]

# filter out the required columns
unemployWorld = unemployWorld.filter(['Country Code','2019'])

# rename Country Code and 2019 columns
unemployWorld = unemployWorld.rename(columns={'Country Code':
 ↪'countryterritoryCode', '2019':'UnemployedPercent'})

unemployWorld.head(3)
```

[46]:
```
  countryterritoryCode  UnemployedPercent
0                  ABW                NaN
1                  AFG             11.118
2                  AGO              6.886
```

Merging the Data

The data frames above can all be merged into a single data frame on their *countryterritoryCode* columns. Any rows with *NaN* entries can then be dropped.

With the information from the four separate data frames now combined into a single data frame, it is possible to create three new columns which will be condsidered during the exploratory data analysis, namely the total number of reported cases per 10,000 population, total number of pre COVID-19 deaths of children under 10 per 10,000 population, and population density.

The first 3 rows of the final data frame, which will be used for exploratory data analysis, are printed below.

```
[47]: # merge countryCases and allMetrics dataframes on their countryterritoryCode␣
       ↪columns
      dataWorld = pd.merge(countryCases, allMetrics, how='inner',␣
       ↪on='countryterritoryCode')
      # merge the newly created data set with the incomeGroups dataframe on their␣
       ↪countryterritoryCode columns
      dataWorld = pd.merge(dataWorld, incomeGroups, how='inner',␣
       ↪on='countryterritoryCode')

      # merge newly created data set with the unemployWorld dataframe, drop NaN rows
      dataWorld = pd.merge(dataWorld, unemployWorld, how='inner',␣
       ↪on='countryterritoryCode')
      dataWorld = dataWorld.dropna()

      # create new columns where values are divided by population (per 10000)
      dataWorld['totalCasesPerPop'] = dataWorld['totalCases'] /␣
       ↪dataWorld['popData2019'] * 10000
      dataWorld['DeathsU10PerPop'] = dataWorld['DeathsU10'] /␣
       ↪dataWorld['popData2019'] * 10000
      dataWorld['PopDensity'] = dataWorld['LandArea'] / dataWorld['popData2019']
      dataWorld.head(3)
```

```
[47]:   countriesAndTerritories countryterritoryCode  popData2019 continentExp  \
      0             Afghanistan                  AFG   38041757.0         Asia
      1                 Albania                  ALB    2862427.0       Europe
      2                 Algeria                  DZA   43053054.0       Africa

         DaysSinceFirstCase  totalCases   LandArea  DeathsU10          IncomeGroup  \
      0                 235       40073   652860.0    83090.0           Low income
      1                 222       16501    27400.0      366.0  Upper middle income
      2                 234       53988  2381740.0    26490.0  Lower middle income

         UnemployedPercent  totalCasesPerPop  DeathsU10PerPop  PopDensity
      0             11.118         10.533951        21.841788    0.017162
      1             12.331         57.646885         1.278635    0.009572
      2             11.704         12.539877         6.152874    0.055321
```

4.2.2 Exploratory Data Analysis

A Note on Test Data

The *dataWorld* dataframe above contains all the training, validation and testing data that will be used when looking at countries. As with the county data, it is important that any data that will be used to test and evaluate the final regression models is not included when performing exploratory data analysis.

This ensures that any decisions made while performing exploratory data analysis are not biased due to the test values being included. The test data must effectively be treated as unknown and independent data until this analysis has been completed and the final regression models have been created, so that they can be evaluated fairly.

For this project, 20 percent of the total available data will be reserved for testing, with the remaining data used for training and validation. In order to randomly split the data, a randomly oredered vector, *train_val_select* can be used to reorder the data, and using a combinatopn of this vector and the *iloc* function, it is possible to filter a specified amount of values out of the dataset.

This technique will be used to only include the training and validation data during all explanatory data analysis steps.

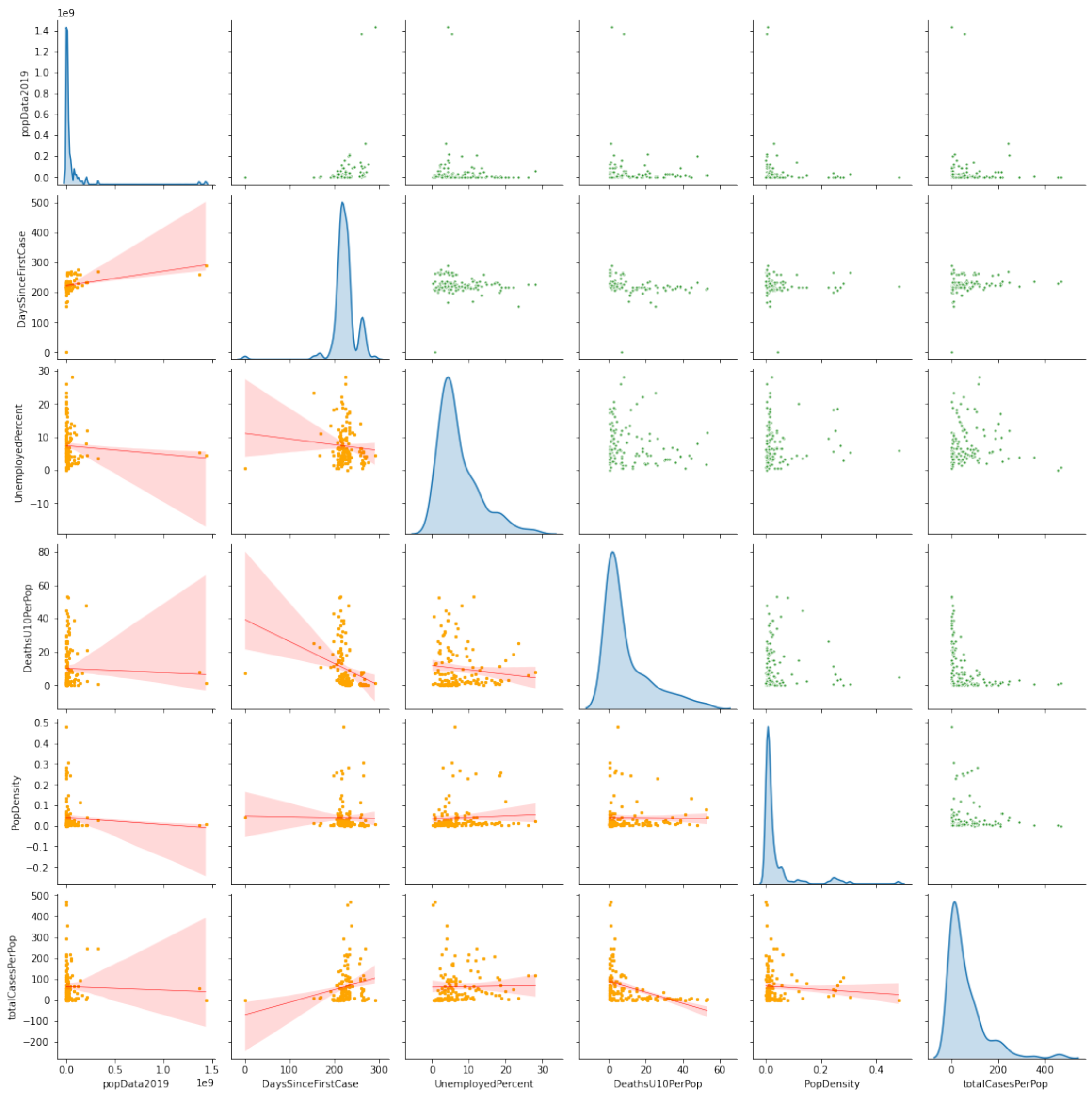
[48]:
```
# set seed value so results are repeatable
np.random.seed(123)

# split train/validation and test data 80:20
train_val_size = int(np.round(dataWorld.shape[0]*0.8))

# create vector from 0 to length y, and randomise the order
train_val_select = np.random.permutation(range(dataWorld.shape[0]))
```

Continuous Variables

Scatter plots can be used to visualize the relationship between all of the continuous or numeric values in our data frame.

Firstly, a separate data frame containing all numeric values of interest can be created by specifying the relevant column names. A pairs plot can then be used to create a matrix of scatter plots comparing each pair combination of variables in the numeric data frame. The pairs plot below is configured to display standard scatter plots in the upper right portion, and scatter plots with a fitted linear regression line in the lower left portion of the pairs plot. The diagonal elements of the pairs plot have also been configured to display density plots, which can be used the visualize the distribution of each variable.

```
[49]: # create data frame containing all numeric variables to be analysed
dataWorld_num = dataWorld.loc[:, ['popData2019', 'DaysSinceFirstCase',␣
 ↪'UnemployedPercent', 'DeathsU10PerPop', 'PopDensity', 'totalCasesPerPop']]

# create pairs plot of numeric data with test values filtered out
pairsWorld = sns.pairplot(dataWorld_num.iloc[train_val_select[:train_val_size],:
 ↪].reset_index(drop=True), kind="scatter", diag_kind="kde",␣
 ↪plot_kws=dict(alpha=0.8, marker=".", linewidth=1, color='green'))
pairsWorld.map_lower(sns.regplot, scatter_kws={'alpha':1, 'color':'orange',␣
 ↪'marker':'.','s':5}, line_kws={'color':'red', 'linewidth':0.5})
plt.show()
```



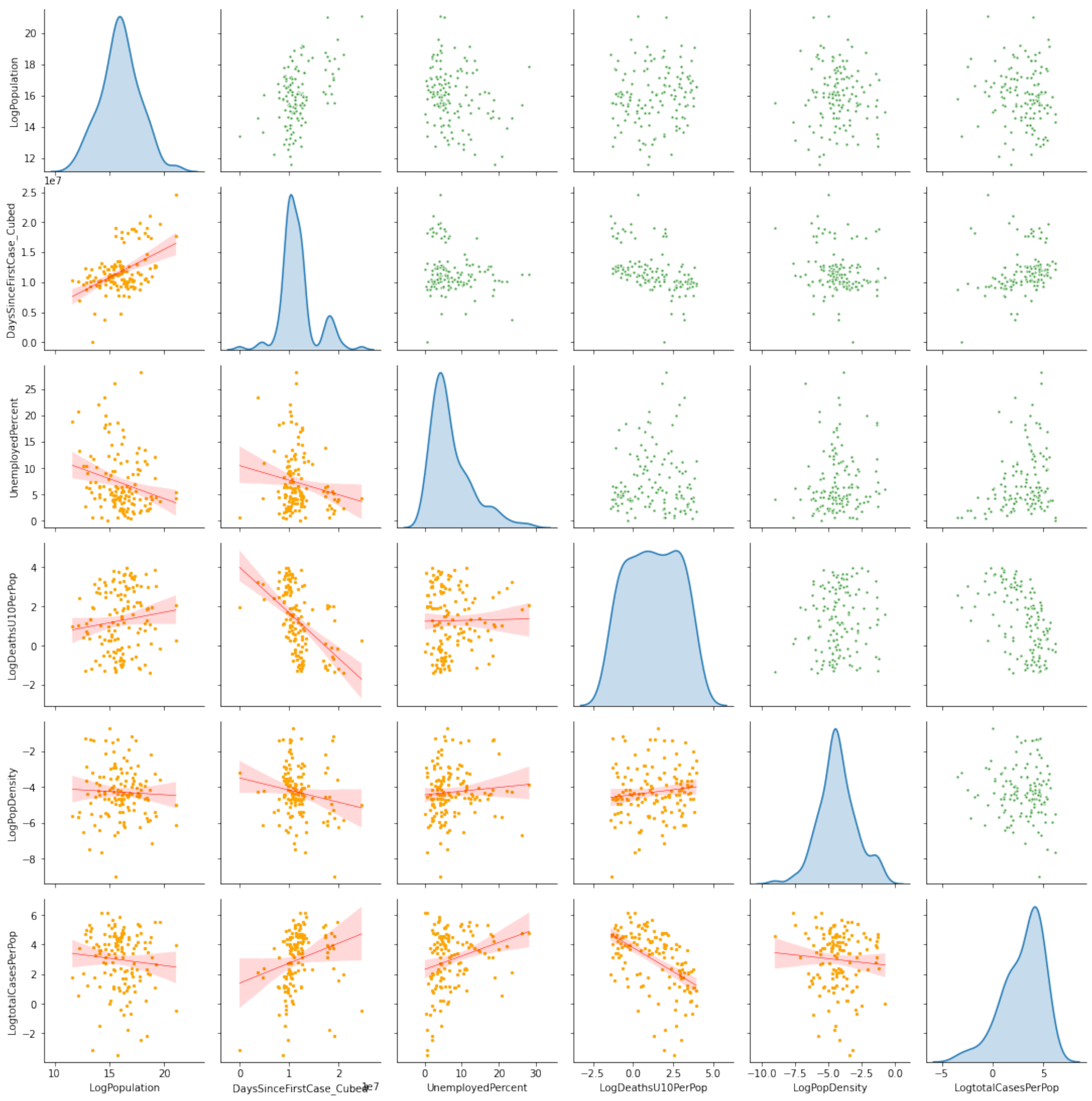
Looking at the density plots above, a number of the variables appear to have heavily right-skewed distributions, with the majority of data points in their related scatter plots located on the left-hand side of the plot and very few data points scattered furter towards the right-hand side. This can be seen in the distributions of the population, deaths under 10 per 10,000 population and population density variables, as well as the total confirmed cases per 10,000 population variable which will be considered as the dependent variable for our regression models.

In such cases, it is often beneficial to perform a log transformation on the related variable so that its data distribution will fall closer to that of a normal distribution.

Also, the distribution of the *DaysSinceFirstCase* column is significantly left-skewed, with the majority of data points bunched to the right of the plot but with a small number of values located on the far left. To gets its distribution closer to a normal distribution, a power transform can be used. In this case, the *DaysSinceFirstCase* variable will be taken to the third power.

After performing log transforms on these variables, the pairs plot can be recreated as done previously.

[50]:
```
# perform log transform on population and population density columns
dataWorld_num['popData2019'] = np.log(dataWorld_num['popData2019'])
dataWorld_num['DeathsU10PerPop'] = np.log(dataWorld_num['DeathsU10PerPop'])
dataWorld_num['PopDensity'] = np.log(dataWorld_num['PopDensity'])
dataWorld_num['totalCasesPerPop'] = np.log(dataWorld_num['totalCasesPerPop'])
dataWorld_num['DaysSinceFirstCase'] = dataWorld_num['DaysSinceFirstCase']**3

# rename log/power transformed columns
dataWorld_num.rename(columns={'DaysSinceFirstCase':'DaysSinceFirstCase_Cubed',␣
↪'popData2019':'LogPopulation', 'DeathsU10PerPop':'LogDeathsU10PerPop',␣
↪'PopDensity':'LogPopDensity', 'totalCasesPerPop':'LogtotalCasesPerPop'},␣
↪inplace=True)

# recreate pairs plot of numeric data with test values filtered out
pairsWorld = sns.pairplot(dataWorld_num.iloc[train_val_select[:train_val_size],:
↪].reset_index(drop=True), kind="scatter", diag_kind="kde",␣
↪plot_kws=dict(alpha=0.8, marker=".", linewidth=1, color='green'))
pairsWorld.map_lower(sns.regplot, scatter_kws={'alpha':1, 'color':'orange',␣
↪'marker':'.','s':5}, line_kws={'color':'red', 'linewidth':0.5})
plt.show()
```

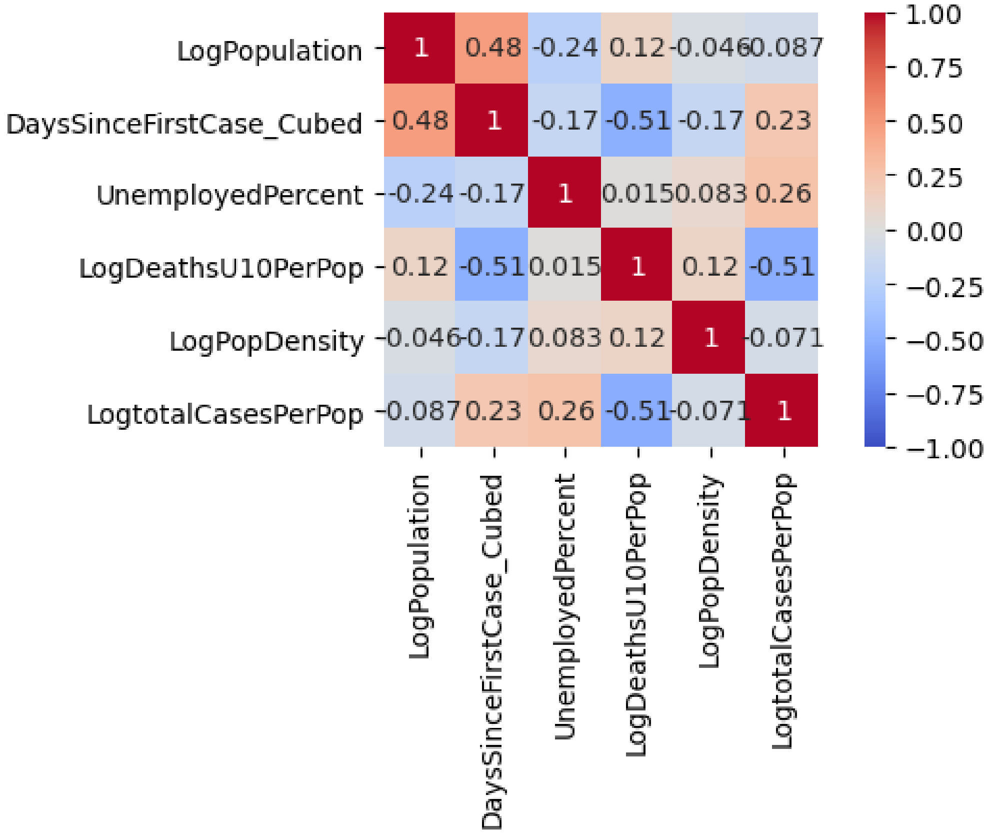


In the pairs plot above, the log transformed and power transformed distrbutions appear to have a less skewed distribution that is much closer to a normal distribution than their original distributions.

Correlation plots can also be used to visualise the strength of the linear relationship between variables in a given data set. The correlation plot below shows the correlation coefficients between each pair of variables in the counties data.

```
[51]: # get correlation matrix for the data with test values filtered out
corr_World = (dataWorld_num.iloc[train_val_select[:train_val_size],:].
 ↪reset_index(drop=True)).corr(method='pearson')
# plot correlation plot heatmap
fig, ax = plt.subplots(figsize=(6,3), dpi= 100)
```

```
sns.heatmap(corr_World, vmin=-1, vmax=1, square=True, annot=True, ␣
↪cmap='coolwarm', ax=ax)
plt.show()
```

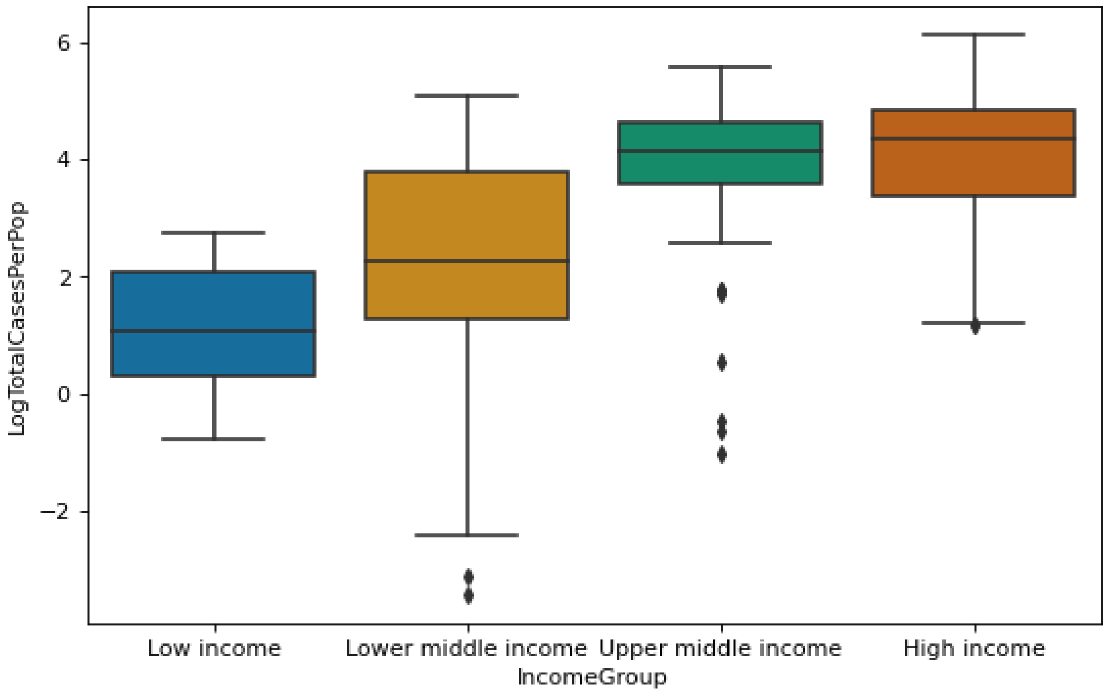


Categorical Variables

There are two categorical variables that will be analysed for the countries data, income group and continent. The first box plot below shows the log of the number of reported COVID-19 cases per 10,000 population grouped by the income group of each country.

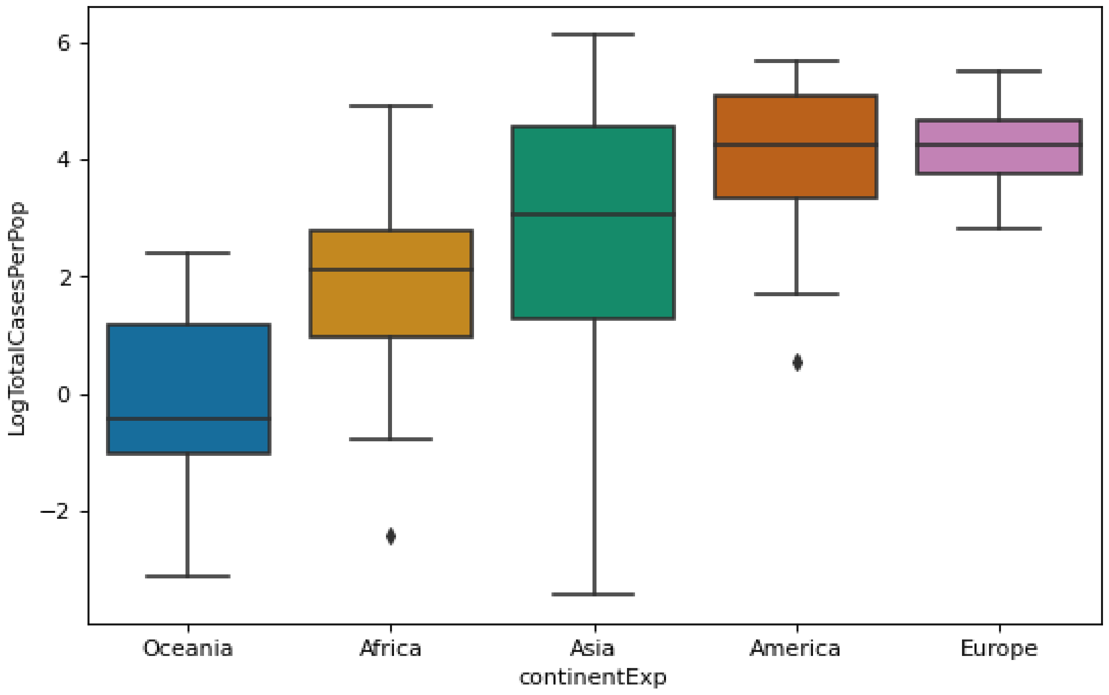
There appears to be a relationship between these two variables, with the median values between most groups differing significantly

[52]:
```
# create data frame including totalCasesPerPop and both categorical variables
dataWorld_cat = dataWorld.loc[:,␣
↪['IncomeGroup','continentExp','totalCasesPerPop']]
# perform log transform on totalCasesPerPop and rename
dataWorld_cat['totalCasesPerPop'] = np.log(dataWorld_cat['totalCasesPerPop'])
dataWorld_cat.rename(columns={'totalCasesPerPop':'LogTotalCasesPerPop'},␣
↪inplace=True)

# get order of LogTotalCasesPerPop median data grouped by IncomeGroup
ig_order = dataWorld_cat.groupby("IncomeGroup")["LogTotalCasesPerPop"].median().
↪sort_values(ascending=False)[::-1].index
plt.figure(figsize=(8,5), dpi= 80)
sns.boxplot(y=dataWorld_cat.LogTotalCasesPerPop, x=dataWorld_cat.IncomeGroup,␣
↪palette="colorblind", order = ig_order)
plt.show()
```



A similar analysis of the relationship between the log of total number of reported COVID-19 cases per 10,000 population and continent using box plots again appears to show a significant relationship between the variables.

[53]:
```
# get order of LogTotalCasesPerPop median data grouped by IncomeGroup
c_order = dataWorld_cat.groupby("continentExp")["LogTotalCasesPerPop"].median().
 ↪sort_values(ascending=False)[::-1].index
plt.figure(figsize=(8,5), dpi= 80)
sns.boxplot(y=dataWorld_cat.LogTotalCasesPerPop, x=dataWorld_cat.continentExp,␣
 ↪palette="colorblind", order = c_order)
plt.show()
```



4.2.3 Statistical Analysis - Regression Modelling

Standardisation

When numerous variables of different units and scales are used as predictor variabes for a model, there is a possibilty that variables will not contribute equally to the model. This can be resolved by standardising the data by subtracting the mean and dividing by standard deviation.

The first 3 rows of the standardised data frame containing all numeric predictor variables for the model are printed below.

[54]:
```
# create new data frame with standardised numerical data
X_World_sd = (dataWorld_num - dataWorld_num.mean())/dataWorld_num.std()

# drop LogtotalCasesPerPop (don't need to include dependent variable)
X_World_sd = X_World_sd.drop(['LogtotalCasesPerPop'], axis=1)

# reset index values
X_World_sd.reset_index(inplace=True, drop=True)

X_World_sd.head(3)
```

[54]:
```
   LogPopulation  DaysSinceFirstCase_Cubed  UnemployedPercent  \
0       0.793693                  0.339420           0.747601
1      -0.697293                 -0.251688           0.972167
2       0.865014                  0.291544           0.856089

   LogDeathsU10PerPop  LogPopDensity
0            1.192348       0.171300
1           -0.637293      -0.251116
2            0.375591       1.018206
```

Predictor and Response Variable Matrices

The X matrix corresponding to the predictor or independent variables for the model can be created by combining the standardised data frame above with the *IncomeGroup* and *continentExp* categorical variables.

The y matrix corresponding to the response or dependant variable for the model is simply the *LogtotalCasesPerPop* column from the *dataWorld_num* data frame.

[55]:
```
# copy standardised values to X matrix
X_World = X_World_sd.copy()
# add IncomeGroup and continentExp columns to X matrix
X_World['IncomeGroup'] = dataWorld.IncomeGroup
X_World['Continent'] = dataWorld.continentExp
# convert categorical variable columns into indicator columns
X_World = pd.get_dummies(data=X_World, drop_first=True)

# create y matrix
y_World = dataWorld_num.LogtotalCasesPerPop.copy()
```

Test, Training and Validation Data

In order to test the test each regression model using unbiased data, a random sample of the total data can be separated from the remaining data and reserved for testing. As specified earlier, 20 percent of the total available data will be reserved for testing and will be split using the *train_val_select* order vector created earlier, with the remaining data used for training and validation.

Cross validation will be used to validate each regression model, and again the aim for this project is to use 20 percent of the total available data for vallidation purposes. As the training and validation data makes up 80 percent of the total data, k-fold cross validation with 4 folds can be used to use 20 percent of the data for validation with the remaining 60 percent used for training the model.

[56]:
```
# create train/validation and test sets
X_train_val = X_World.iloc[train_val_select[:train_val_size],:].
 ↪reset_index(drop=True)
X_test = X_World.iloc[train_val_select[train_val_size:],:].
 ↪reset_index(drop=True)
y_train_val = y_World.iloc[train_val_select[:train_val_size]].
 ↪reset_index(drop=True)
y_test = y_World.iloc[train_val_select[train_val_size:]].reset_index(drop=True)

# use 4 fold cross validation (60% training, 20% validation of total data)
cv_World = KFold(n_splits=4)
```

Regression Model Training and Testing

Four different variations of regression models will be built and tested to model the countries data, the was done with the Irish counties data.

The default parameters will be used for the multiple linear regression models. K-fold cross validation will be performed initally for this regression technique using the validation and training sets created above. After cross validation is complete, a final multiple linear regression model will be fitted to the full set of training and validation data available, and tested on the test data.

For the remaining three regression techniques (lasso regression, ridge regression, random forest regression) cross validation will be used to assess the performance of each model across a range of parameters to determine the the optimum parameter values for each model, based on the validation and training data sets. After the optimum parameters have been determined, a final model using for each of these regression techniques will be created using the obtained parameters value, by fitting to the full set of training and validation data available. Again, these models will be used to predict the values from the test data set.

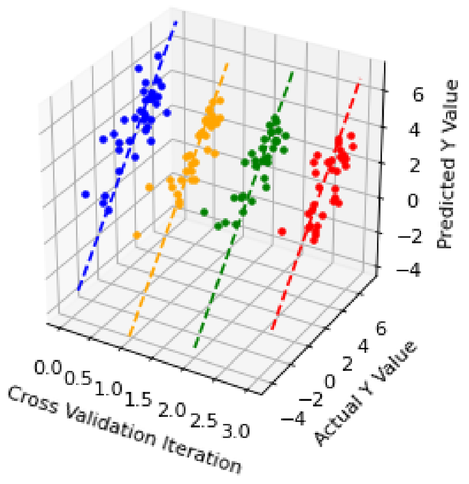
The functions defined in the Irish counties regression analysis section which were used to perform cross validation on the models, return the coefficients and R-squared values of the final fitted models and to test the final models on the test data can all be reused for the countries data.

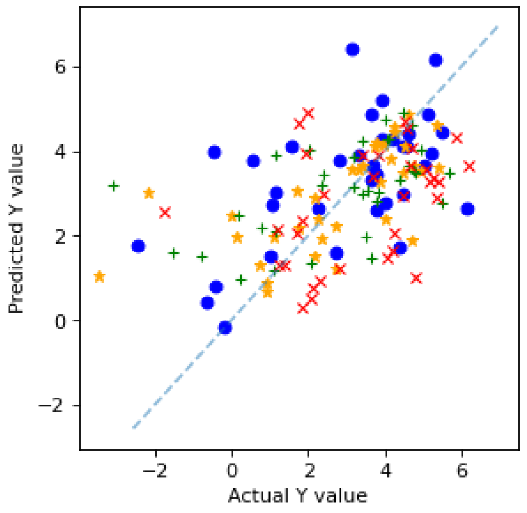
Multiple Linear Regression Model

Cross Validation

After creating a multiple linear regression model, the cross validation function defined earlier can be used to perform cross validation on the model and plot the results, as well as returning the mean RMSE value.

[57]:
```
# Create multiple linear regression model
mlr2 = linear_model.LinearRegression()

# Use model prediction plot function with linear model
mlr2_RMSE = modelCrossVal(mlr2, cv_World, True)

# Print RMSE for linear model
print("\nMean RMSE for Mulitple Linear Regression Model:", mlr2_RMSE,'\n')
```





```
Mean RMSE for Mulitple Linear Regression Model: 1.7346112336106878
```

Model Coefficients, R^2 and adjusted R^2

The final multiple linear regression model can be fitted with the full set of training and validation data, *X_train_val* and *y_train_val*. The *modelCoefsR2* function defined above can be used to report the model coefficients, R-squared value and adjusted R-squared value for the model.

[58]:
```
# get coefficients, R2 and R2 adjusted values
mlr2_coefs, mlr2R2, mlr2R2adj = modelCoefsR2(mlr2, X_train_val, y_train_val)

# print R2 and R2 adjusted
print("\nR-squared value for final multiple regression model:", np.
 ↪round(mlr2R2,3))
print("\nAdjusted R-squared value for final multiple regression model:", np.
 ↪round(mlr2R2adj,3),"\n")

# print the model coefficients
display(Markdown(mlr2_coefs.to_markdown()))
```

```
R-squared value for final multiple regression model: 0.39

Adjusted R-squared value for final multiple regression model: 0.331
```

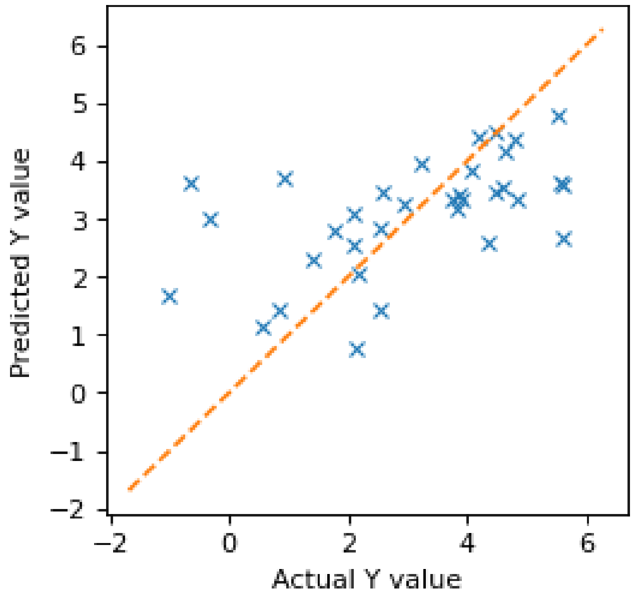
	Parameter	Coefficient
0	Intercept	3.64459
1	LogPopulation	0.148994
2	DaysSinceFirstCase_Cubed	-0.117514
3	UnemployedPercent	0.499812
4	LogDeathsU10PerPop	-1.06397
5	LogPopDensity	-0.120451
6	IncomeGroup_Low income	-0.529041
7	IncomeGroup_Lower middle income	-0.791033
8	IncomeGroup_Upper middle income	-0.295761
9	Continent_America	-0.906325
10	Continent_Asia	-0.673666
11	Continent_Europe	-0.221018
12	Continent_Oceania	-0.61563

Predicting the Unknown Test Data

The *modelTest* function also defined above can then be used to predict the values in the test data, and plot the results. It will also return the root mean squared error calculated for the tested values, which can be used when evaluating the performance of the final models.

[59]:
```
# Test model on unknown test data
mlr2_pred_RMSE, mlr2_pred_NRMSE = modelTest(mlr2, X_train_val, y_train_val,␣
 ↪X_test, y_test)

# print RMSE
print("\nRMSE for model predictions of the test data:", np.
 ↪round(mlr2_pred_RMSE,3), '\n')
```

```
RMSE for model predictions of the test data: 1.518
```



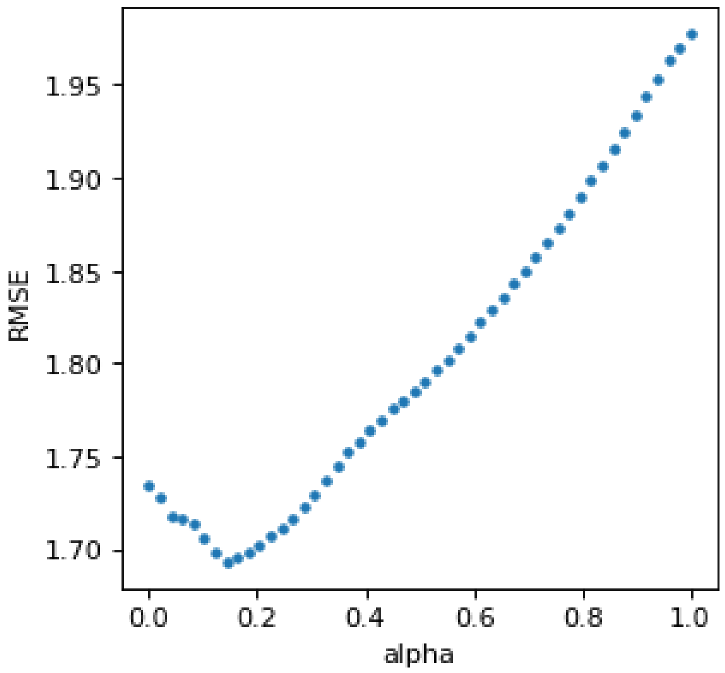
Lasso Regression Model

Cross Validation

For lasso regression, choosing different values for the α parameter can affect the performance of the model. The cross validation function can be used to evaluate the performance of the model for a range of different α values and return the value which minimizes the RMSE value outputted from the function. Plotting the RMSE value obtained for each α value will also help identify what is the best value for the parameter.

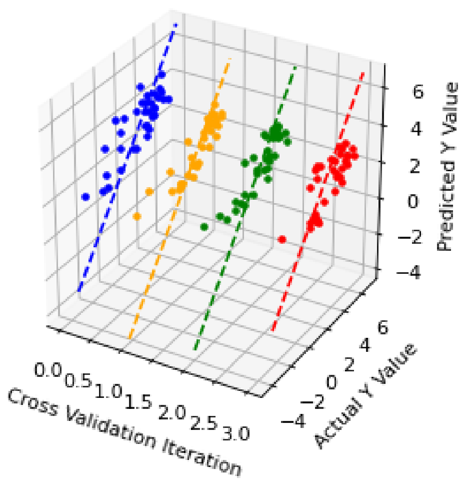
[60]:
```
# create vector of 50 evenly spaces points between 0 and 1
alpha_vals = np.linspace(0.0001, 1, num=50)

# create empty list to hold all RMSE values
RMSEs = []

# set start values for min RMSE and best alpha values
min_RMSE = 9999
best_alpha = 0

# Loop through all alpha values
for i in alpha_vals:

    # create Lasso regression model using current alpha value
    lasso2 = linear_model.Lasso(alpha=i)

    # calculate RMSE of current model, and append value to list
    lasso_RMSE = modelCrossVal(lasso2, cv_World, False)
    RMSEs.append(lasso_RMSE)

    # if current RMSE is less than the minimum RMSE, set current RMSE to min␣
 ↪RMSE and current alpda to best alpha
    if lasso_RMSE < min_RMSE:
        min_RMSE = lasso_RMSE
        best_alpha = i

# plot RMSE vs alpha values
plt.figure(figsize=(4,4), dpi= 80)
plt.plot(alpha_vals, RMSEs, '.')
plt.xlabel('alpha')
plt.ylabel('RMSE')

# print alpha value that minimizes RMSE of model
print("\nOptimum alpha value:", np.round(best_alpha,3))
print("\nMinimum RMSE value with optimised alpha value:", np.round(min_RMSE,3),␣
 ↪"\n")
```

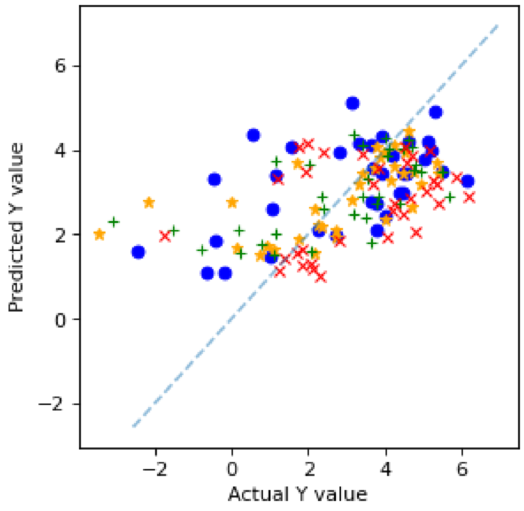
```
Optimum alpha value: 0.143

Minimum RMSE value with optimised alpha value: 1.693
```



Once an optimum α is chosen, the cross validation function can be used again using the optimum value in the lasso regression model to plot the results from the cross validation and return the mean RMSE value.

```
[61]: # Create lasso regression model
lasso2 = linear_model.Lasso(alpha=best_alpha)

# Use model prediction plot function with lasso model
lasso2_RMSE = modelCrossVal(lasso2, cv_World, True)

# Print MSE for lasso model
print("\nRMSE for Lasso Regression Model:", lasso2_RMSE, '\n')
```





```
RMSE for Lasso Regression Model: 1.6932882016507445
```

Model Coefficients, R^2 and adjusted R^2

The final lasso regression model can be fitted with the full set of training and validation data, *X_train_val* and *y_train_val*. The *modelCoefsR2* function defined above can be used to report the model coefficients, R-squared value and adjusted R-squared value for the model.

[62]:
```
# get coefficients, R2 and R2 adjusted values
lasso2_coefs, lasso2R2, lasso2R2adj = modelCoefsR2(lasso2, X_train_val,␣
↪y_train_val)

# print R2 and R2 adjusted
print("\nR-squared value for final lasso regression model:", np.
↪round(lasso2R2,3))
print("\nAdjusted R-squared value for final lasso regression model:", np.
↪round(lasso2R2adj,2),"\n")

# print the model coefficients
display(Markdown(lasso2_coefs.to_markdown()))
```

```
R-squared value for final lasso regression model: 0.322

Adjusted R-squared value for final lasso regression model: 0.26
```

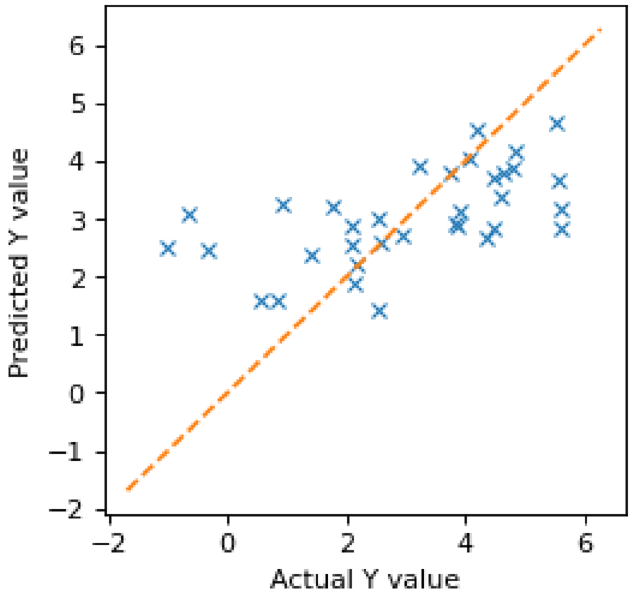
	Parameter	Coefficient
0	Intercept	2.99023
1	LogPopulation	0
2	DaysSinceFirstCase_Cubed	0
3	UnemployedPercent	0.374334
4	LogDeathsU10PerPop	-0.867337
5	LogPopDensity	-0
6	IncomeGroup_Low income	-0
7	IncomeGroup_Lower middle income	-0
8	IncomeGroup_Upper middle income	-0
9	Continent_America	-0
10	Continent_Asia	-0
11	Continent_Europe	0
12	Continent_Oceania	-0

Predicting the Unknown Test Data

The *modelTest* function can be used to predict the values in the test data, and plot the results. It will also return the root mean squared error calculated for the tested values, which can be used when evaluating the performance of the final models.

```
[63]: # Test model on unknown test data
      lasso2_pred_RMSE, lasso2_pred_NRMSE = modelTest(lasso2, X_train_val,␣
       ↪y_train_val, X_test, y_test)

      # print RMSE
      print("\nRMSE for model predictions of the test data:", np.
       ↪round(lasso2_pred_RMSE, 3), '\n')
```

```
RMSE for model predictions of the test data: 1.506
```



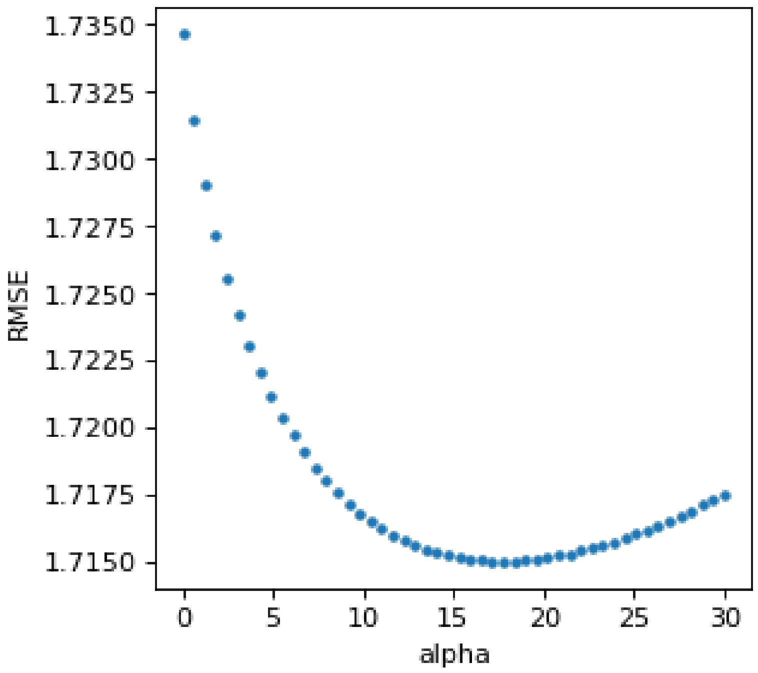
Ridge Regression Model

Cross Validation

Similar to lasso regression, choosing different values for the α parameter for ridge regression can also affect the performance of the model. The cross validation function can be used to evaluate the performance of the model for a range of different α values and return the value which minimizes the RMSE value outputted from the function. Plotting the RMSE value obtained for each α value will also help identify what is the best value for the parameter.

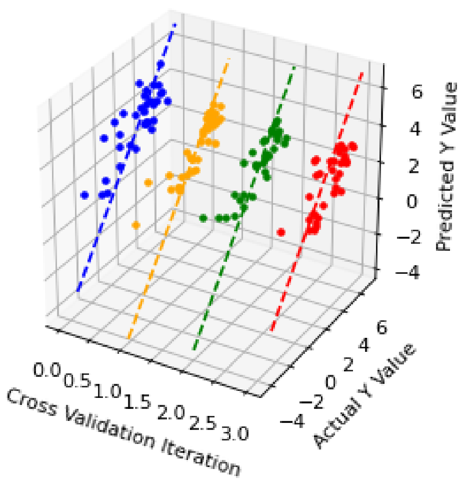
[64]:
```
# create vector of 50 evenly spaces points between 0 and 30
alpha_vals=np.linspace(0.0,30,num=50)

# create empty list to hold all RMSE values
RMSEs = []

# set start values for min RMSE and best alpha values
min_RMSE = 9999
best_alpha = 0

# Loop through all alpha values
for i in alpha_vals:

    # create Ridge regression model using current alpha value
    ridge2 = Ridge(alpha=i)

    # calculate RMSE of current model, and append value to list
    ridge_RMSE = modelCrossVal(ridge2, cv_World, False)
    RMSEs.append(ridge_RMSE)

    # if current RMSE is less than the minimum RMSE, set current RMSE to min␣
 ↪RMSE and current alpda to best alpha
    if ridge_RMSE < min_RMSE:
        min_RMSE = ridge_RMSE
        best_alpha = i

# plot RMSE vs alpha values
plt.figure(figsize=(4,4), dpi= 80)
plt.plot(alpha_vals, RMSEs, '.')
plt.xlabel('alpha')
plt.ylabel('RMSE')

# print alpha value that minimizes RMSE of model
print("\nOptimum alpha value:", np.round(best_alpha,3))
print("\nMinimum RMSE value with optimised alpha value:", np.round(min_RMSE,3),␣
 ↪"\n")
```

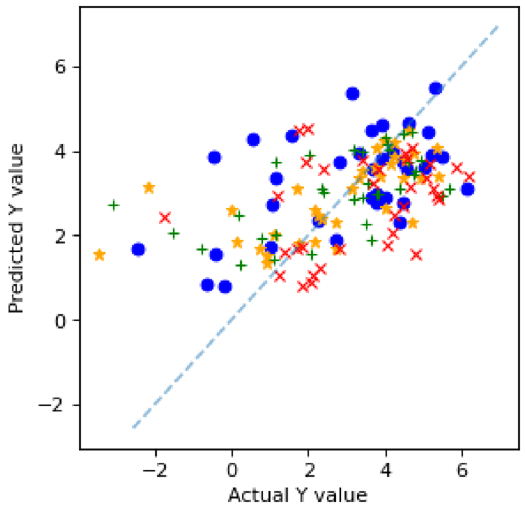
```
Optimum alpha value: 17.755

Minimum RMSE value with optimised alpha value: 1.715
```



Once an optimum α is chosen, the cross validation function can be used again using the optimum value in the ridge regression model to plot the results from the cross validation and return the mean RMSE value.

[65]:
```python
# Create ridge regression model
ridge2 = Ridge(alpha=best_alpha)

# Use model prediction plot function with ridge model
ridge2_RMSE = modelCrossVal(ridge2, cv_World, True)

# Print RMSE for ridge model
print("\nRMSE for Ridge Regression Model:", ridge2_RMSE, '\n')
```





```
RMSE for Ridge Regression Model: 1.7149793898781707
```

Model Coefficients, R^2 and adjusted R^2

The final ridge regression model can be fitted with the full set of training and validation data, *X_train_val* and *y_train_val*. The *modelCoefsR2* function defined above can be used to report the model coefficients, R-squared value and adjusted R-squared value for the model.

```
[66]: # get coefficients, R2 and R2 adjusted values
ridge2_coefs, ridge2R2, ridge2R2adj = modelCoefsR2(ridge2, X_train_val,␣
 ↪y_train_val)

# print R2 and R2 adjusted
print("\nR-squared value for final ridge regression model:", np.
 ↪round(ridge2R2,3))
print("\nAdjusted R-squared value for final ridge regression model:", np.
 ↪round(ridge2R2adj,2),"\n")

# print the model coefficients
display(Markdown(ridge2_coefs.to_markdown()))
```

```
R-squared value for final ridge regression model: 0.366

Adjusted R-squared value for final ridge regression model: 0.31
```

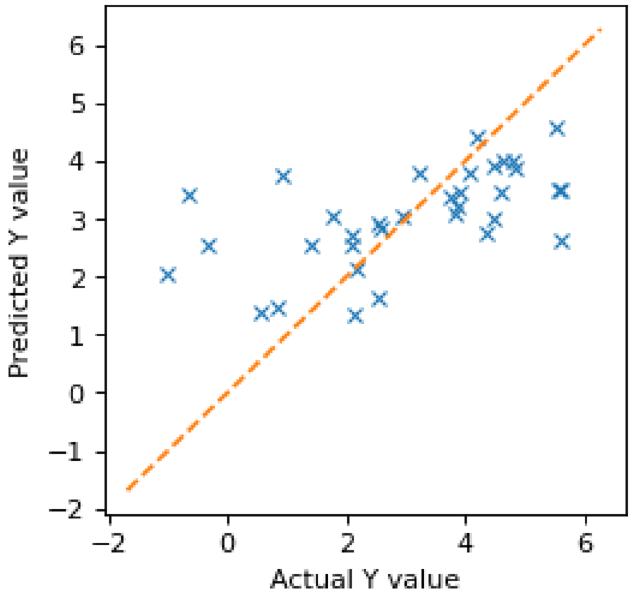
	Parameter	Coefficient
0	Intercept	3.19442
1	LogPopulation	0.0214455
2	DaysSinceFirstCase_Cubed	0.0644816
3	UnemployedPercent	0.456544
4	LogDeathsU10PerPop	-0.850662
5	LogPopDensity	-0.0689772
6	IncomeGroup_Low income	-0.1037
7	IncomeGroup_Lower middle income	-0.305885
8	IncomeGroup_Upper middle income	-0.163868
9	Continent_America	-0.360643
10	Continent_Asia	-0.282784
11	Continent_Europe	0.130655
12	Continent_Oceania	-0.0734973

Predicting the Unknown Test Data

The *modelTest* function can be used to predict the values in the test data, and plot the results. It will also return the root mean squared error calculated for the tested values, which can be used when evaluating the performance of the final models.

```
[67]: # Test model on unknown test data
ridge2_pred_RMSE, ridge2_pred_NRMSE = modelTest(ridge2, X_train_val,␣
 ↪y_train_val, X_test, y_test)

# print RMSE
print("\nRMSE for model predictions of the test data:", np.
 ↪round(ridge2_pred_RMSE, 3), '\n')
```

```
RMSE for model predictions of the test data: 1.504
```



Random Forest Regression Model

Cross Validation

Two parameters will be considered for this project when assessing the performance of random forest regression models, the number of estimators and maximum depth. The cross validation function can be used to evaluate the performance of the model for a range of different number of estimators and maximum depth values and return the value which minimizes the RMSE value outputted from the function. Nested for loops can be used to test every combination of given number of estimators and maximum depth values.

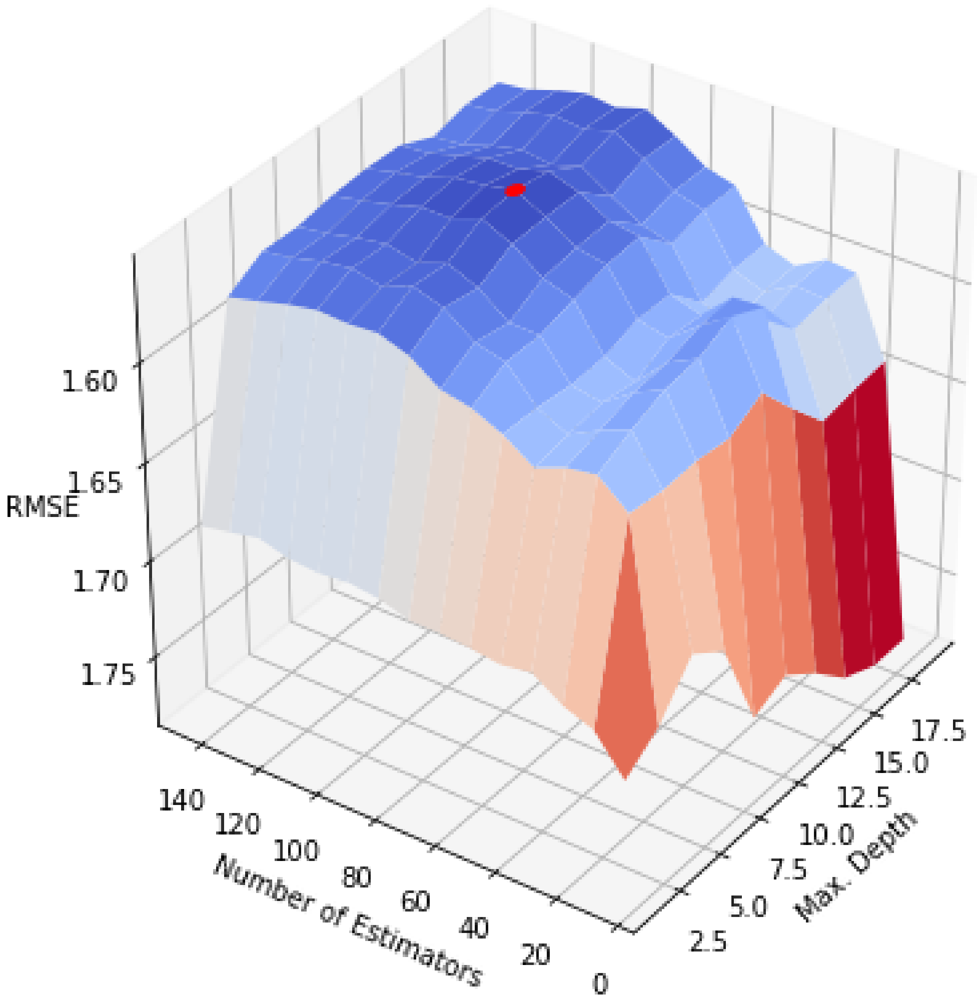
[68]:
```
# create vectors of integers for max depth and n_estimators
depth_vals = range(1,21,2)
est_vals = range(5,155,10)

# create empty list to hold all RMSE values
RMSEs = [[0 for x in depth_vals] for x in range(len(est_vals))]

# set start values for min RMSE and best depth and num. estimators values
min_RMSE = 9999
best_depth = 0
best_n_est = 0

# create i/j index values to track current iteration number of for loops
i = 0
j = 0

# Loop through all n_estimators values
for n_est in est_vals:
    # Loop through all max_depth values
    for max_d in depth_vals:
        # create Random Forest regression model using current max_depth and␣
 ↪n_estimator value
        rf2 = RandomForestRegressor(n_estimators=n_est, max_depth=max_d,␣
 ↪random_state=123)

        # calculate RMSE of current model, and write value to list
        rf_RMSE = modelCrossVal(rf2, cv_World, False)
        RMSEs[i][j]=rf_RMSE

        # if current RMSE < minimum RMSE...
        if rf_RMSE < min_RMSE:
            # set current RMSE to min RMSE and current n_est/depth to best␣
 ↪n_est/depth
            min_RMSE = rf_RMSE
            best_n_est = n_est
            best_depth = max_d
        # increment j index
        j = j + 1

    # increment i index
    i = i + 1
    # reset j index
    j = 0

# print max depth value that minimizes RMSE of model
print("\nOptimum Number of Estimators:", best_n_est)
print("\nOptimum Max. depth value:", best_depth)
print("\nMinimum RMSE value with optimised parameters", np.round(min_RMSE, 3),␣
 ↪"\n")
```

```
Optimum Number of Estimators: 95

Optimum Max. depth value: 11

Minimum RMSE value with optimised parameters 1.552
```

To visualise the RMSE values obtained for every combination of given maximum depth values and number of estimators values, a 3D surface plot can be used. The surface plot below also includes a red dot at the obtained values for the optimum maximum depth and number of estimators. The Z-axis on the plot has been inverted to make it easier to identify the point in the surface where the minimum RMSE is located, which would equate to the highest point on this plot. Indeed, the optimum point plotted on the surface does appear to fall on the highest peak, meaning that it produces the lowest RMSE value.

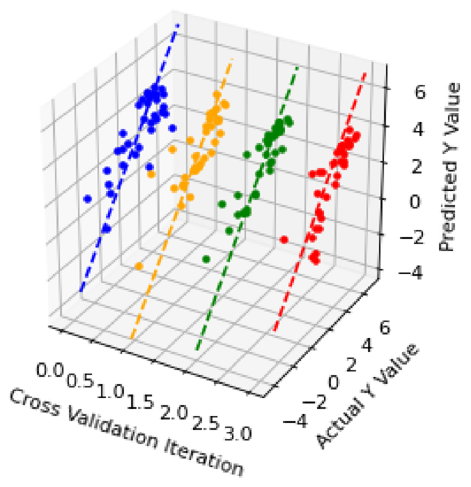
[69]:
```
# create grid of all combinations of max depth values and n_estimators values
X,Y = np.meshgrid(depth_vals, est_vals)
# Convert RMSEs 2D list from cross validation testing to a numpy array
Z = np.array(RMSEs)

# create 3D plot
fig = plt.figure(figsize=(7,7))
ax = fig.add_subplot(111,projection='3d')
ax.set_xlabel('Max. Depth')
ax.set_ylabel('Number of Estimators')
ax.set_zlabel('RMSE')

# plot surface with max_depth on X-axis, n_estimators on Y-axis and RMSEs on␣
 ↪Z-axis
ax.plot_surface(X, Y, Z, alpha=1, cmap=cm.coolwarm, linewidth=0)

# add point to the plot at the optimum max_depth value and n_estimators value
optimised_point = Ellipse((best_depth, best_n_est), 0.75, 4, ec='r', fc="r")
```

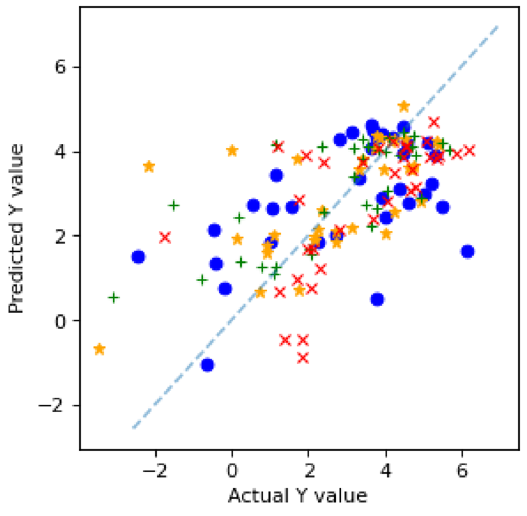
```python
ax.add_patch(optimised_point)
art3d.pathpatch_2d_to_3d(optimised_point, z=min_RMSE, zdir="z")

# Set view angle of plot and reverse Z-axis
ax.elev = 35
ax.azim = -145
ax.invert_zaxis()

plt.show()
```



Once optimum maximum depth and number of estimators values are chosen, the cross validation function can be used again using the optimum values in the random forest regression model to plot the results from the cross validation and return the mean RMSE value.

```
[70]: # Create random forest regression model
      rf2 = RandomForestRegressor(n_estimators=best_n_est, max_depth=best_depth,␣
       ↪random_state=13)

      # Use model prediction plot function with random forest model
      rf2_RMSE = modelCrossVal(rf2, cv_World, True)

      # Print RMSE for random forest model
      print("\nRMSE for Random Forest Regression Model:", rf2_RMSE,'\n')
```





```
RMSE for Random Forest Regression Model: 1.5700934489600795
```

Model Coefficients, R^2 and adjusted R^2

The final random forest regression model can be fitted with the full set of training and validation data, *X_train_val* and *y_train_val*. The *modelCoefsR2* function defined above can be used to report the R-squared value and adjusted R-squared value for the model.

[71]:
```
# get coefficients, R2 and R2 adjusted values
rf2_coefs, rf2R2, rf2R2adj = modelCoefsR2(rf2, X_train_val, y_train_val)

# print R2 and R2 adjusted
print("\nR-squared value for final random forest regression model:", np.
 ↪round(rf2R2,3))
print("\nAdjusted R-squared value for final random forest regression model:",␣
 ↪np.round(rf2R2adj,2),"\n")

# print the model coefficients
print(rf2_coefs)
```

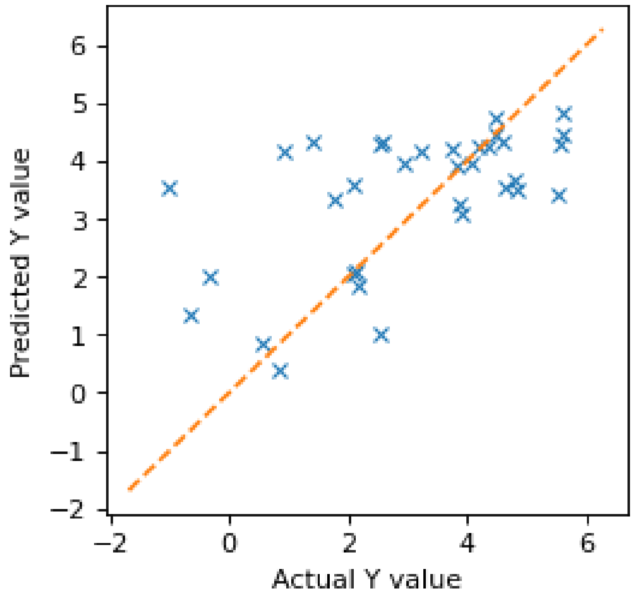
```

R-squared value for final random forest regression model: 0.905

Adjusted R-squared value for final random forest regression model: 0.9

There are no coefficients available for this regression model.
```

Predicting the Unknown Test Data

The *modelTest* function can be used to predict the values in the test data, and plot the results. It will also return the root mean squared error calculated for the tested values, which can be used when evaluating the performance of the final models.

[72]:
```
# Test model on unknown test data
rf2_pred_RMSE, rf2_pred_NRMSE = modelTest(rf2, X_train_val, y_train_val,␣
 ↪X_test, y_test)

# print RMSE
print("\nRMSE for model predictions of the test data:", np.round(rf2_pred_RMSE,␣
 ↪3), '\n')
```

```
RMSE for model predictions of the test data: 1.517
```



Base Case Model Comparison

While figures obtained above can be used to compare each regression model against each other, it can also be useful to calculate the root mean squared error for a base model which simply returns the mean value of the given training data for all predictions.

[73]:
```
test_RMSE_World = np.sqrt(np.mean((np.mean(y_train_val) - y_test)**2))
print("RMSE for base model predictions:", np.round(test_RMSE_World,3))
```

```
RMSE for base model predictions: 1.823
```

5 Results

The results obtained during the exploratory data analysis and statistical analysis sections are summarised below. Again, the results relating to the Irish counties data and the countries data will be reported separately.

5.1 Irish Counties

5.1.1 Exploratory Data Analysis

Looking at the second pairs plot for the Irish counties data in section 4.1.2 above, the bottom row shows the scatter plots that plot each of the continuous variables considered for predictor variables in our statistical analysis against the proportion of reported COVID-19 cases per 10,000 population, the dependent variable used in out statistical analysis.

From observing the data points and the regression lines on each of these plots, it appears that there is a posistive but relatively weak linear relationship evident between the log of the population and the proportion of COVID-19 cases per 10,000 population, a posistive and moderate linear relationship evident between the log of the population density and the proportion of COVID-19 cases per 10,000 population, no clear evidence of a linear relationship evident between the average percentage of unemployment and the proportion of COVID-19 cases per 10,000 population, and a negative but weak linear relationship evident between the average number of deaths of children under 10 per 10,000 population and the proportion of COVID-19 cases per 10,000 population.

The correlation coefficients between each of the predictor variables and dependent variables are shown in the table below. Based on these figures, the log of the population density and the number of COVID-19 cases per 10,000 population appear to have the strongest linear relationship between a predictor variable and the dependent variable for the counties data.

```
[74]: display(Markdown(corr_IE.iloc[0:-1,-1].to_markdown()))
```

	PopProportionCovidCases
LogPopulation	0.255565
LogPopulationDensity	0.401346
AvgDeathsU10PerPop	-0.0822182
AvgUnemployedPerPop	-0.175225

Box plots of the number of COVID-19 cases per 10,000 population split by province indicated a significant relationship between the two variables. In particular, the mean number of COVID-19 cases per 10,000 population in Ulster counties in the data set is significantly higher than the remaining provinces, with Leinster counties also having a visibly larger mean value than both Munster and Connacht counties.

5.1.2 Statistical Analysis

The table below contains the model parameter coefficients for the mulitiple linear regression, lasso regression and ridge regression models.

For all three models, the numeric predictor variable with the most influence on the model prediction was the log of population density. With Connacht considered the base category in relation to the province data, both Munster and Ulster also had significant influence on each model.

```
[75]: # merge coefficient data frames from all county models into single data frame
df_coefsIE = pd.merge(mlr_coefs, lasso_coefs, how='inner', on='Parameter')
df_coefsIE = pd.merge(df_coefsIE, ridge_coefs, how='inner', on='Parameter')
df_coefsIE.columns = ['Predictor', 'MLR Coeffs', 'Lasso Coeffs', 'Ridge Coeffs']
# round values to 2 decimal places
df_coefsIE[['MLR Coeffs', 'Lasso Coeffs', 'Ridge Coeffs']] = df_coefsIE[['MLR␣
 ↪Coeffs', 'Lasso Coeffs', 'Ridge Coeffs']].round(2)

display(Markdown(df_coefsIE.to_markdown()))
```

	Predictor	MLR Coeffs	Lasso Coeffs	Ridge Coeffs
0	Intercept	76.7	84.81	84.27
1	LogPopulation	-6.93	0	0.21
2	LogPopulationDensity	19.4	9.51	12.71
3	AvgUnemployedPerPop	-9.57	-0	-4.89
4	Province_Leinster	5.84	0	1.74
5	Province_Munster	-18.37	-7.18	-21.53
6	Province_Ulster	84.39	8.03	35.95

The R-squared, adjusted R-squared and root mean squared error (RMSE) values for each of the four tested regression models are summarised in the table below.

Random forest regression appeared to be the best model of the given county data, with a relatively high adjusted R-squared value of 0.69. It also performed the best at predicting the values from the test data, returning the lowest RMSE value of the four models, 23.75.

```
[76]: # create data frame of R2, adjusted R2 and RMSE from the testing data
df_EvaluationIE = pd.DataFrame(np.array([['Multiple Linear', mlrR2, mlrR2adj,␣
 ↪mlr_pred_RMSE, mlr_pred_NRMSE], ['Lasso', lassoR2, lassoR2adj,␣
 ↪lasso_pred_RMSE, lasso_pred_NRMSE], ['Ridge', ridgeR2, ridgeR2adj,␣
 ↪ridge_pred_RMSE, ridge_pred_NRMSE], ['Random Forest', rfR2, rfR2adj,␣
 ↪rf_pred_RMSE, rf_pred_NRMSE], ['Base Case Model', 0, 0, test_RMSE_IE, 0]]),
                    columns=['Model Type', 'R2', 'Adj R2', 'RMSE Test', 'NRMSE␣
 ↪Test'])
# round values to 2 decimal places
df_EvaluationIE[['R2', 'Adj R2', 'RMSE Test', 'NRMSE Test']] =␣
 ↪df_EvaluationIE[['R2', 'Adj R2', 'RMSE Test', 'NRMSE Test']].astype(float).
 ↪round(2)
# replace base model R2 and adjR2 values with 'N/A'
```

```
df_EvaluationIE.iloc[4,[1,2,4]]='N/A'

display(Markdown(df_EvaluationIE.to_markdown()))
```

	Model Type	R2	Adj R2	RMSE Test	NRMSE Test
0	Multiple Linear	0.69	0.55	26.63	0.63
1	Lasso	0.28	-0.03	36.82	0.87
2	Ridge	0.57	0.38	31.43	0.74
3	Random Forest	0.78	0.69	23.75	0.56
4	Base Case Model	N/A	N/A	38.04	N/A

5.2 Countries

5.2.1 Exploratory Data Analysis

Looking at the second pairs plot for the countries data in section 4.2.2 above, the bottom row shows the scatter plots that plot each of the continuous variables considered for predictor variables in our statistical analysis against the log of the number of reported COVID-19 cases per 10,000 population, the dependent variable used in out statistical analysis.

From observing the data points and the regression lines on each of these plots, it appears that both the number of days since the first reported case cubed and the unemployment percentage have a weak to moderate posistive linear relationship with the log number of COVID-19 cases per 10,000 population, a negative and moderate linear relationship is evident between the log of the number of deaths of chilren under 10 per 10,000 population and the log of the number of COVID-19 cases per 10,000 population, and no clear evidence of a linear relationship with the dependent variable for either the log of the population or the log of the population density.

The correlation coefficients between each of the predictor variables and dependent variables are shown in the table below. Based on these figures, the log of the deaths of children under 10 per 10,000 population (*LogDeathsU10PerPop*) and the log of the number of COVID-19 cases per 10,000 population (*LogtotalCasesPerPop*) appear to have the strongest linear relationship between a predictor variable and the dependent variable for the countries data.

```
[77]: display(Markdown(corr_World.iloc[0:-1,-1].to_markdown()))
```

	LogtotalCasesPerPop
LogPopulation	-0.0870123
DaysSinceFirstCase_Cubed	0.233941
UnemployedPercent	0.258038
LogDeathsU10PerPop	-0.511074
LogPopDensity	-0.071248

5.2.2 Statistical Analysis

The table below contains the model parameter coefficients for the mulitiple linear regression, lasso regression and ridge regression models.

For all three models, the numeric predictor variable with the most influence on the model prediction was the log deaths under 10 per 10,000 population, with percentage of employment also contributing signifcatly in each case. With high income and Africa considered the base categories in relation to the income group and continent variables respectively, the lower middle income group had the most influence on the models with repect to income groups, while America was the continent that had most influence on the models.

[78]:
```
# merge coefficient data frames from all country models into single data frame
df_coefsWorld = pd.merge(mlr2_coefs, lasso2_coefs, how='inner', on='Parameter')
df_coefsWorld = pd.merge(df_coefsWorld, ridge2_coefs, how='inner',␣
 ↪on='Parameter')
df_coefsWorld.columns = ['Predictor', 'MLR Coeffs', 'Lasso Coeffs', 'Ridge␣
 ↪Coeffs']
# round values to 2 decimal places
df_coefsWorld[['MLR Coeffs', 'Lasso Coeffs', 'Ridge Coeffs']] =␣
 ↪df_coefsWorld[['MLR Coeffs', 'Lasso Coeffs', 'Ridge Coeffs']].round(2)

display(Markdown(df_coefsWorld.to_markdown()))
```

	Predictor	MLR Coeffs	Lasso Coeffs	Ridge Coeffs
0	Intercept	3.64	2.99	3.19
1	LogPopulation	0.15	0	0.02
2	DaysSinceFirstCase_Cubed	-0.12	0	0.06
3	UnemployedPercent	0.5	0.37	0.46
4	LogDeathsU10PerPop	-1.06	-0.87	-0.85
5	LogPopDensity	-0.12	-0	-0.07
6	IncomeGroup_Low income	-0.53	-0	-0.1
7	IncomeGroup_Lower middle income	-0.79	-0	-0.31
8	IncomeGroup_Upper middle income	-0.3	-0	-0.16
9	Continent_America	-0.91	-0	-0.36
10	Continent_Asia	-0.67	-0	-0.28
11	Continent_Europe	-0.22	0	0.13
12	Continent_Oceania	-0.62	-0	-0.07

The R-squared, adjusted R-squared and root mean squared error (RMSE) values for each of the four tested regression models are summarised in the table below.

As with the counties data, random forest regression appeared to give the best model of the given county data, with a very high adjusted R-squared value of 0.9. However when tested on unknown sample of test data, each model performed similarly at predicting the test values, with the ridge regression model slightly outperforming the other models with the lowest resulting RMSE value of 1.5.

```
[79]: # create data frame of R2, adjusted R2 and RMSE from the testing data
df_EvaluationWorld = pd.DataFrame(np.array([['Multiple Linear', mlr2R2,␣
 ↪mlr2R2adj, mlr2_pred_RMSE, mlr2_pred_NRMSE], ['Lasso', lasso2R2,␣
 ↪lasso2R2adj, lasso2_pred_RMSE, lasso2_pred_NRMSE], ['Ridge', ridge2R2,␣
 ↪ridge2R2adj, ridge2_pred_RMSE, ridge2_pred_NRMSE], ['Random Forest', rf2R2,␣
 ↪rf2R2adj, rf2_pred_RMSE, rf2_pred_NRMSE], ['Base Case Model', 0, 0,␣
 ↪test_RMSE_World, 0]]),
                    columns=['Model Type', 'R2', 'Adj R2', 'RMSE Test', 'NRMSE␣
 ↪Test'])
# round values to 2 decimal places
df_EvaluationWorld[['R2', 'Adj R2', 'RMSE Test', 'NRMSE Test']] =␣
 ↪df_EvaluationWorld[['R2', 'Adj R2', 'RMSE Test', 'NRMSE Test']].
 ↪astype(float).round(2)
# replace base model R2, adjR2 and NRMSE values with 'N/A'
df_EvaluationWorld.iloc[4,[1,2,4]]='N/A';

display(Markdown(df_EvaluationWorld.to_markdown()))
```

	Model Type	R2	Adj R2	RMSE Test	NRMSE Test
0	Multiple Linear	0.39	0.33	1.52	0.82
1	Lasso	0.32	0.26	1.51	0.81
2	Ridge	0.37	0.31	1.5	0.81
3	Random Forest	0.91	0.9	1.52	0.82
4	Base Case Model	N/A	N/A	1.82	N/A

6 Conclusions

Regression Modelling

The primary aim for this project was to determine if suitable regression models could be created to predict the total number of reported COVID-19 cases per 10,000 population for a given county in Ireland, and for a given country, using data from the provided data files. This was successfully achieved for both Irish counties and countries, as regression models in both cases were more accurate at predicting values from unknown test data than baseline models which returned the mean value of the training data for each prediction.

For Irish counties, the random forest regression model performed the best at predicting the test data, returning the lowest root mean squared error (RMSE) value of 23.75. The baseline model in comparison had a significantly higher RMSE value of 38.04 for the predictions. For the countries data, the ridge regression model performed the best at predicting the test data, returning the lowest RMSE value of 1.5. The baseline model in comparison had a higher RMSE value of 1.82 for the predictions.

Considering the normalised root mean squared error (NRMSE) figures returned for each model, it appeared that the Irish county models were in general better at predicting the total number of reported COVID-19 cases in a given county than the country models were at predicting the log of the number of reported COVID-19 cases in a given country, exluding the lasso regression model which performed poorly on the counties data. This wasn't surprising in most cases, as the muliple linear regression, ridge regression and random forest regression models for the county data had larger adjusted R-squared values than each of the muliple linear regression, lasso regression and ridge regression models for the countries data. However, it was surprising that the random forest regression model for the countries data didn't perform better, considering it had an adjusted R-squared value of 0.9 which was relatively large compared to all other models.

It was also interesting that the performance of the models for the counties data varied far more between each regression type in comparison to the models for the countries data, which all produced a NRMSE value of approximately 0.81. For example, the random forest regression model for the counties had had a NRMSE of 0.56, in comparison to the lasso regression model which had a relatively high NRMSE value of 0.87 and also a slightly negative adjusted R-squared value indicating that the model was worse at predicting the training data than the baseline model. This was likely due to the relatively small amount of data available for training the county models, an unavoidable consequence of there only being 26 counties in total that could be considered for such models. The limitation in county data resulted in only five entries being utilised for testing, so I believe that each model for the county data would be susceptible to large changes in their predicition performance depending on the randomly selected training and test data sets, making the results for these models more difficult to interpret with confidence. In contrast, the relatively small number of countries in the available data was still large in comparison to the counties, with 172 entries in total available. This likely resulted in less variation in the prediction performance of each model.

Overall, there does appear to be some value in utilising regression techniques in order to estimate the total number of reported COVID-19 cases per 10,000 population for both counties in Ireland and for countries. However, using the indicator variables chosen for this project, it was not possible to create a model for either data set that predicted the total number to a reasonably high level of accuracy. Therefore, there are likely a number of other important indicators that weren't considered here that had a significant effect on the total number of reported COVID-19 cases per 10,000 population.

Promising Model Idicator Variables

Perhaps the most interesting findings from this project were from the relationships observed between some of the indicator variables and the dependent variables for the regression models.

Firstly, the log of population and the log of population density both had significant positive linear relationships with the total number of confirmed COVID-19 cases per 10,000 population for the counties data, with correaltion coefficients of 0.26 and 0.4 respectively. In comparison, the log of population and the log of population density for the countries data both had very weak, negative linear relationships with the total number of confirmed COVID-19 cases per 10,000 population, with correaltion coefficients of -0.29 and -0.07 respectively. This is possibly due to geographical reasons, with far more variation in the area and population of the countries in the data set than for Irish counties.

Another interesting finding from this project was the nature of the relationship between the number of deaths of children under the age of 10 per 10,000 population and the total number of reported cases per 10,000 population. While the relationship was insignificant for the counties data, a significant negative linear relationship was observed between between the number of deaths of children under the age of 10 per 10,000 population and the log of the total number of reported cases per 10,000 population for the countries data. Intuitively, I would have expected any relationship between these variables to be positive, as a larger number of child deaths would typically correspond to a less developed country with poorer health services, which I believe would increase the spread of disease.

A similar surprising relationship was seen when looking at the relationship between income group and the log of the number of cases per 10,000 population for the countries data. Again, before analysing the data I expected that higher income countries would be better equipped to manage the spread of COVID-19 than lower income countries due to better health services and general hygeine standards. However, the oppoiste trend was observed, with low income countries having a significantly lower mean value for the log of total number of recorded cases per population than high income countries.

I have two possible theories that could explain these seemingly unlikely results relating to the child deaths and income groups data. Firstly, the larger number of reported cases observed in developed countries could partly be explained by the greater volume of travel in and out of those countries, both for business and tourism purposes. Logically, the larger influx of people into these countries would increase the likelihood of the virus being introduced by infected travellers, thus accelerating the spread of the disease.

The second theory I have which could explain these results is related to the fact that the quantity of interest relates to the reported numbers of cases, rather than the actual number of cases in each country. More developed countries have performed and continue to perform much higher numbers of tests to detect COVID-19 then less developed countries, thus leading to higher reported case numbers. Therefore, I believe that COVID-19 if far more widespread in many underdeveloped and low income countries than the available data would suggest.

In summary, there is a significant difference between the variables that are most strongly related with the number of reported COVID-19 cases for counties in Ireland and for countries.

Future Areas of Investigation

While it is difficult at present to assess the true effects that the COVID-19 pandemic has had in every country, and in particular the effects it has had on less developed countries where numbers of reported cases are likely far lower than the true figures, one interesting area of future investigation would be to analyse the total number of recorded deaths in each country in comparison to figures prior to the pandemic. A significant number of excess deaths could potentially be related to the virus, particularly in underdeveloped countries with poorer healthcare facilities. Similar exploratory data analysis and regression modelling to what was performed in this project could be performed again but with excess number of deaths considered as a dependent variable, to see if any trends emerge.

In relation to the current project, further iterations of analysis could be carried out to determine which regression model is best suited for both the counties data and the countries data, using different combinations of randominsed training and test data sets. This would be particularly beneficial for the counties data, as due to the relatively small number of available data entries and small training and test sets as a result, the fitted regression models are more susceptible to bias, which could have significantly affected the results obtained as part of this project.

Finally, for other pandemics that will occur in the future, the most influential indicator variables related to the total number of reported COVID-19 cases per 10,000 population that were identified in this project could also be used to assist with prediciting the spread of the disease.