Data Analytics (COMP47350): Homework 2

Introduction

The purpose of this assignment is, broadly speaking, the training and evaluation of certain prediction models for a specific dataset. The dataset we will be working with is a randomly selected dataset of approximately 10,000 entries from the Irish Residential Property Price Register, hereafter known as the **RPPR** (link to register can be found here). The dataset has been cleaned and additional features have been added to it as part of Homework 1 (which can be provided on request). The cleaned dataset is saved down in this directory with the title **homework1.csv**.

The specific predictive models we will be using are Linear Regression, Decision Tree and Random Models. These models will be built to capture the relationship between descriptive features and the target feature "Price".

The various steps, preparatory or otherwise, are specifically set out in more detail throughout this assignment as sub-headings. The code is largely explained in detail in the markdown cells, however, additional detail is provided where necessary with through commentsin the code itself.

Please note that all references to theory are, unless otherwise stated, are obtained from Dr Georgiana Ifrim's lectures in module COMP47350. The code contained in the sample solution and the labs uploaded to Brightspace were also referenced when preparing this assignment.

It should also be flagged that most of the code is explained in the markdown cells in this notebook. In the event that something in the code is not particularly clear, comments will be included directly in such code cells. If some elements of the code are repeated, either from earlier parts of the assignment, or from Assignment 1, then comments will not be provided as this would lead to redundancy. Unless otherwise specified, the graphs generated by this notebook were not exported as .png files as this was not requested in the assignment.

Finally, this notebook is split up in the same sections as the questions in the assignment sheet. Some additional headings were added were this was thought appropriate. Each heading has the corresponding question from the assignment sheet listed in *italics* underneath it for ease of reference.

Contents of this folder

This folder contains the following files:

- **Vlad_Rakhmanin_21200122_COMP47350.ipynb**: The Jupyter Notebook version of this assignment.
- **Vlad_Rakhmanin_21200122_COMP47350.pdf**: The PDF version of this assignment. Please note, however, that due to the size of some of the tables involved in our analysis, some of these may be cut of in the PDF version, and as such, the notebook version should be used for the purposes of the review where possible.
- **homework1.csv**: The .csv file containing the cleaned version of the data assigned as part of Homework 1
- **22032022-PPR-Price-recent.csv**: The .csv file that is needed for Question 5 of this assignment.

- **inflation.csv**: The inflation csv created as part of Assignment 1, which will be needed to create the inflation feature on the test dataset in Question 5.
- **tree_graphs**: A directory containing saved down graphs of the decision trees, which are a little easier to see than the ones embedded in this notebook.
- **requirements.txt**: A list of dependencies required to run this notebook. This was obtained as part of a "pip freeze" of a conda environment I was using for this assignment, so it contains all the dependencies required for the libraries listed within.
- **install_requirements.sh**: A small shell script that installs the dependencies listed in requirements.txt in a single click - this has been included for convenience.

Question 1: Data Understanding and Preparation

To begin, we will import the various Python libraries required for the purposes of this assignment. The following libraries are expected to be used:

- **pandas**: to simplify working with, and navigation of, the dataset;
- **matplotlib**: to create various plots and charts so that the data can be visualised;
- **numpy**: to perform certain calculations;
- **seaborn**: to perform some more advanced data visualisation that would have taken longer to do in matplotlib;
- **sklearn**: to build and evaluate prediction models; and
- **math**: to perform certain mathematical operations.

In [314…
```
# Importing requirements

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib
from sklearn import metrics, tree, ensemble
from sklearn.model_selection import train_test_split, cross_val_score, KFold
from sklearn.linear_model import LinearRegression
from sklearn.dummy import DummyRegressor
from sklearn.preprocessing import PolynomialFeatures
import math

# The below allows plots to run directly in the notebook
%matplotlib inline
```

Next, we will read in the csv file containing the dataset, and examine the head, tail, shape and feature types to ensure that the resulting dataframe aligns with our expectations.

Please note that I have assumed that the CSV file is already UTF-8 encoded. I have based this assumption on the fact that there were no format issues with working with this dataset. If this was not the case, the file would need to be converted to UTF-8 and re-saved as a new file so that Pandas would be able to read it more clearly.

In [315…
```
df = pd.read_csv("homework1.csv")
```

In [316…
```
df.head()
```

Out[316]:

	Date of Sale	Address	Postal Code (Dublin)	County	Price (€)	Full Market Price	VAT Exclusive	Second-Hand	Year	Quarter	Multiple Property Purchase
0	2020-06-26	1 GANDON PLACE, GANDON PARK, Lucan	NaN	Dublin	370044.05	True	True	False	2020	2	False
1	2014-12-19	72 ST ASSAMS PARK, RAHENY, DUBLIN 5	Dublin 5	Dublin	480000.00	True	False	True	2014	4	False
2	2010-02-11	37 Ivy Hill, Gort Road, Ennis	NaN	Clare	194000.00	True	False	True	2010	1	False
3	2018-08-16	14 TYRCONNELL PLACE, TYRCONNELL RD, INCHICORE ...	Dublin 8	Dublin	275000.00	True	False	True	2018	3	False
4	2019-02-27	7 THE NEST, TUBBERCURRY, SLIGO	NaN	Sligo	75000.00	True	False	True	2019	1	False

In [317…
```
df.tail()
```

Out[317]:

	Date of Sale	Address	Postal Code (Dublin)	County	Price (€)	Full Market Price	VAT Exclusive	Second-Hand	Year	Quarter	Mult Prop Purch
10104	2017-05-17	35 2 4 GEORGES QUAY, DUBLIN 2, DUBLIN	Dublin 2	Dublin	77666.666667	True	False	True	2017	2	-
10105	2015-06-23	40 40A 40B & 40C Carrowmore Drive, Knock	NaN	Mayo	31750.000000	True	True	False	2015	2	-
10106	2015-06-23	40 40A 40B & 40C Carrowmore Drive, Knock	NaN	Mayo	31750.000000	True	True	False	2015	2	-
10107	2015-06-23	40 40A 40B & 40C Carrowmore Drive, Knock	NaN	Mayo	31750.000000	True	True	False	2015	2	-
10108	2015-06-23	40 40A 40B & 40C Carrowmore Drive, Knock	NaN	Mayo	31750.000000	True	True	False	2015	2	-

In [318…
```
df.shape
```

Out[318]:
```
(10109, 13)
```

In [319…
```
df.dtypes
```

Out[319]:
```
Date of Sale                  object
Address                       object
Postal Code (Dublin)          object
County                        object
Price (€)                    float64
Full Market Price               bool
VAT Exclusive                   bool
Second-Hand                     bool
Year                           int64
Quarter                        int64
Multiple Property Purchase      bool
Inflation                    float64
Province                      object
dtype: object
```

The above results match our expectation based on the outcome of Homework 1.

1.0 Additional Data Cleaning

Despites best efforts to clean the data in Assignment 1, there are still some outstanding issues that yet remain, that will need to be fixed before we proceed with splitting out the dataset in appropriate proportions. The reason why we are making these edits here, at the start of the notebook, as opposed to *after* having split the dataset, is because these issues relate specifically **data cleaning**, which means that really these issues should have been resolved at the data cleaning stage in Assignment 1.

We will now go through each of these issues in turn.

NaN Values

There still seem to be NaN values inside our dataframe, which will negatively impact the models that we will be creating. We should check which of the columns have NaN values and replace them with the apporopriate values.

In [320…
```
for column in df.columns:
    print(f"The column {column} has {df[column].isnull().values.sum()} null values.")
```

```
The column Date of Sale has 0 null values.
The column Address has 0 null values.
The column Postal Code (Dublin) has 8217 null values.
The column County has 0 null values.
The column Price (€) has 0 null values.
The column Full Market Price has 0 null values.
The column VAT Exclusive has 0 null values.
The column Second-Hand has 0 null values.
The column Year has 0 null values.
The column Quarter has 0 null values.
The column Multiple Property Purchase has 0 null values.
The column Inflation has 1079 null values.
The column Province has 7279 null values.
```

As we can see, the Postal Code (Dublin), Inflation and Province columns still contain NaN values. We will deal with each of these in the following ways:

- **Postal Code (Dublin)**: We know that the NaN values in this column represent properties not based in Dublin. For this reason, we can replace all the NaN values with a new category called "Not in Dublin".
- **Inflation**: Because this is a numeric value, we can try and replace NaN values with the mean.

- **Province**: There are too many NaN values here for the column to provide us with any kind of useful information. For this reason, we we will drop it.

While we are making these edits, we will also rename the "Price (€)" column to just "Price" so that it's easier to refer to.

In [321…
```
# Renaming price columns
df.rename(columns = {"Price (€)": "Price"}, inplace = True)

# Filling NaN values
df[["Postal Code (Dublin)"]] = df[["Postal Code (Dublin)"]].fillna("Not in Dublin")
df[["Inflation"]] = df[["Inflation"]].fillna(df["Inflation"].mean())

# Dropping province column
df = df.drop(["Province"], axis=1)
```

Dropping Columns that are Not Useful

Having had another look at our features, and having reviewed our data quality plan from Assignment 1 again, some of them are just not feasible to use in our model reliably, and as such we will drop them.

- **Date of Sale**: The feedback for Assignment 1 that I received noted that "Date of Sale" should **not** be used as a continuous feature. It should be noted that we derived several other categorical features from "Date of Sale", such as "Year" and "Quarter". Because those derived features are largely giving us the same information, there is no use in keeping "Date of Sale", and as such we can drop it.

- **Address**: As noted in Assignment 1, it was not possible to use GeoPy on this feature to lower its cardinality, and as such there is not real way that we can make use of this feature. In any event, we have the "County" and "Postal Code (Dublin)" features that can represent the geographical location of the properties, even though the detail may not be as granular. For this reason, we will drop "Address".

- **Multiple Property Purchase**: Having reviewed the way we derived this feature in Assignment 1, it was decided that the methodology used was perhaps not as accurate as we would want it to be when preparing a feature for use in a predictive model. There is no way for certain to know whether or not a property was part of a joint purchase, or what the individual payments for each property were in the overall sale. We were simply making an educated guess, and it was decided that this was not sufficiently accurate for use with the predictive model. For this reason, we will drop this feature.

In [322…
```
# Dropping unnecessary columns
df = df.drop(["Date of Sale", "Address", "Multiple Property Purchase"], axis=1)
```

Reverting Descriptors

When we exported the data into a csv file as part of Homework 1, some of the datatype descriptors were lost. As such, we should run the same code used in Homework 1 just to be sure that all categories are marked with an appropriate descriptor.

In [323…
```
# Selecting categorical columns
categorical_columns = df[["Postal Code (Dublin)", "County", "Quarter", "Year"]].columns
boolean_columns = df[["Full Market Price", "VAT Exclusive", "Second-Hand"]].columns

# Looping through columns and casting as category
```

```
for column in categorical_columns:
    df[column] = df[column].astype('category')

# Looping through columns and casting as bool
for column in boolean_columns:
    df[column] = df[column].astype(bool)
```

In [324…
```
df.dtypes
```

Out[324]:
```
Postal Code (Dublin)    category
County                  category
Price                    float64
Full Market Price           bool
VAT Exclusive               bool
Second-Hand                 bool
Year                    category
Quarter                 category
Inflation                float64
dtype: object
```

The above is now in order and we are ready to begin on the specific requirements listed in the assignment.

1.1 Split the dataset into training and test datasets

Randomly shuffle the rows of your dataset and split the dataset into two datasets: 70% training and 30% test. Keep the test set aside.

In order to split our dataset, we can use the "train_test_split" function contained in Pandas. This will randomly shuffle the rows in our dataframe and will output two dataframes. We can pass in a "test_size" parameter which will set the size of the test dataset, so we will set this to 0.3 in order to match the 70/30 split requested in the assignment.

It should be noted that the reason that we are splitting the dataset *now* (aside from the fact that this is requested in the assignment) is so that we can replicate a real-life scenario where we may only initially have access to the train dataset, and we may not have access to test data. For this reason, we cannot make any assumptions about the test dataset, and in the initial part of this assignment we will be working with the train dataset *only*. We should, however, keep it in the back of our minds that any transformations that we eventually make to our train data we will have to make to our test data - this will be discussed in more detail later on.

Please note also that we have set the random seed to the value "1" in our `train_test_split` function. This is to ensure that every time that the notebook is run, we achieve the same result, meaning that if the marker wishes to rerun the code to test that all is functioning correctly, they can do so, and the analysis will still be applicable to the new outcome.

In [325…
```
df_train, df_test = train_test_split(df, test_size=0.3, random_state=1)
```

In [326…
```
print(f"The training dataset contains {df_train.shape[0] / df.shape[0] * 100}% of the entire
print(f"The testing dataset contains {df_test.shape[0] / df.shape[0] * 100}% of the entire da
```

```
The training dataset contains 69.9970323474132% of the entire dataset.
The testing dataset contains 30.002967652586804% of the entire dataset.
```

As we can see from the above, we now have two dataframes which contain roughly the appropriate

amount of data as requested.

In [327…
```
df_train.head()
```

Out[327]:

	Postal Code (Dublin)	County	Price	Full Market Price	VAT Exclusive	Second-Hand	Year	Quarter	Inflation
3018	Not in Dublin	Louth	145000.0	True	False	True	2014	2	0.182542
5794	Not in Dublin	Kerry	229074.0	True	True	False	2011	2	2.557189
1627	Not in Dublin	Kildare	220000.0	True	False	True	2021	3	0.374889
3722	Not in Dublin	Dublin	208500.0	True	False	True	2017	4	0.340532
5769	Not in Dublin	Louth	500000.0	True	False	True	2019	2	0.939044

In [328…
```
df_test.head()
```

Out[328]:

	Postal Code (Dublin)	County	Price	Full Market Price	VAT Exclusive	Second-Hand	Year	Quarter	Inflation
7765	Dublin 3	Dublin	425000.0	True	False	True	2019	3	0.939044
5227	Not in Dublin	Wicklow	400000.0	True	True	False	2021	3	0.374889
858	Not in Dublin	Kerry	210000.0	True	False	True	2010	1	-0.922096
2702	Not in Dublin	Sligo	35500.0	False	False	True	2017	1	0.340532
7331	Not in Dublin	Carlow	177500.0	True	False	True	2015	3	-0.289879

1.2 Review promising features for use with the predictive models

On the training set:

- Plot the correlations between all the continuous features (if any). Discuss what you observe in these plots.

- For each continuous feature, plot its interaction with the target feature (a plot for each pair of continuous feature and target feature). Discuss what you observe from these plots, e.g., which continuous features seem to be better at predicting the target feature? Choose a subset of continuous features you find promising (if any). Justify your choices.

- For each categorical feature, plot its pairwise interaction with the target feature. Discuss what knowledge you gain from these plots, e.g., which categorical features seem to be better at predicting the target feature? Choose a subset of categorical features you find promising (if any). Justify your choices.

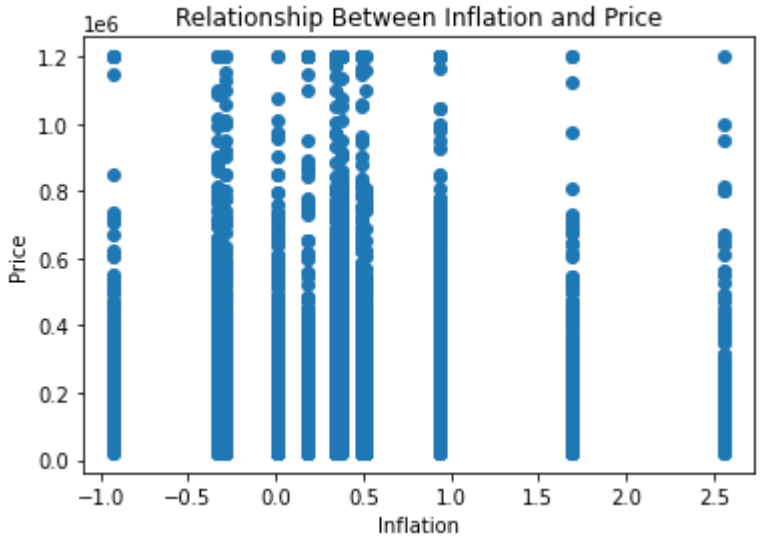
As mentioned above, we are working under the assumption that we *only* have access to the train set. For this reason, our review will consist of a review of correlation between our target feature (Price) and the various continuous and categorical features.

(a) Correlation between continuous features

We only have one continuous feature in our dataset, which is Inflation.

Inflation

In [329…
```
plt.scatter(df_train["Inflation"], df_train["Price"])
plt.xlabel("Inflation")
plt.ylabel("Price")
plt.title("Relationship Between Inflation and Price")
plt.show()
```



Discussion

The above suggests that there are very few distinct values contained in the inflation index, and as such it seems to be being treated almost like a categorical feature. It does not seem like there is very strong correlation between Inflation and Price, and as such it would probably not be a particularly good fit for the predictive model.

One possible way of working with this feature is by turning it into a categorical feature instead. We will discuss this later in the assignment, but for the time being this is not a likely candidate as a feature for our predictive model.

(b) Correlation between categorical features

The majority of the features in our model are categorical. They are as follows:

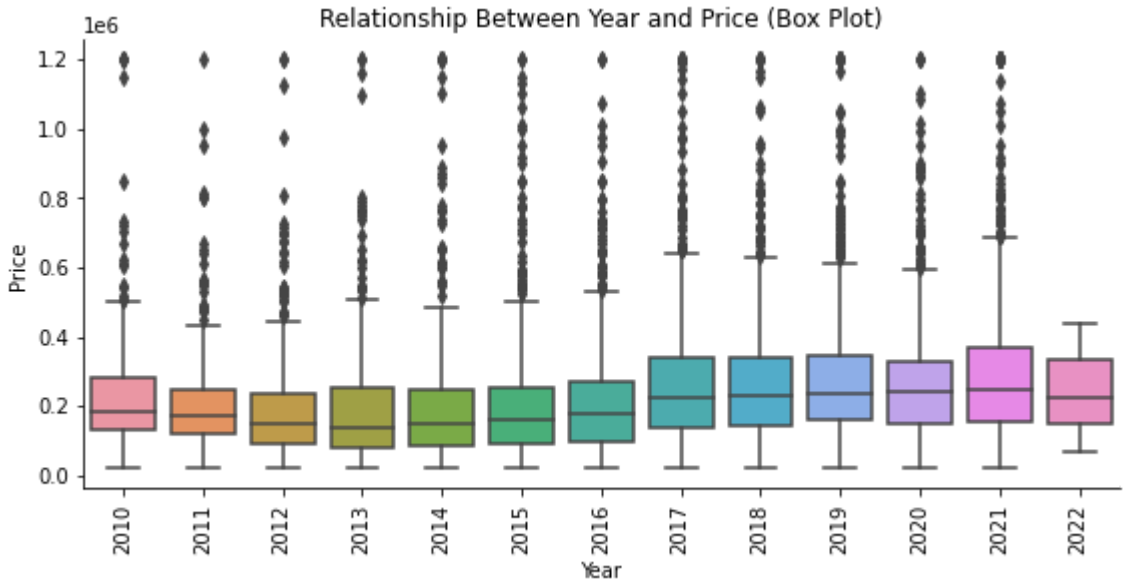
- Year;
- Postal Code (Dublin);
- County;
- Full Market Price (bool);
- VAT Exclusive (bool);
- Second-Hand (bool); and
- Quarter.

The features that are of boolean datatype are clearly marked in the list above. As per Homework 1, the categorical features will be plotted against Price using box plots, whereas the boolean features will be plotted against Price using bar plots. The analysis as to why these types of graphs were chosen is identical to that contained in Homework 1 and as such will not be replicated here.

Year

In [330…
```
sns.catplot(x="Year", y="Price", kind="box", data=df_train, height=4, aspect=2)
plt.xticks(rotation=90)
```

```
plt.title("Relationship Between Year and Price (Box Plot)")
plt.show()
```



Discussion

There is a clear correlation between year and price that can be seen here. It seems that on average, property prices grew steadily over the past decade, with a minor dip in 2012/2013. While the increase in price is not particularly dramatic, it is very clear that property prices are growing nonetheless, which makes sense as the country continues its recovery from the economic crisis.

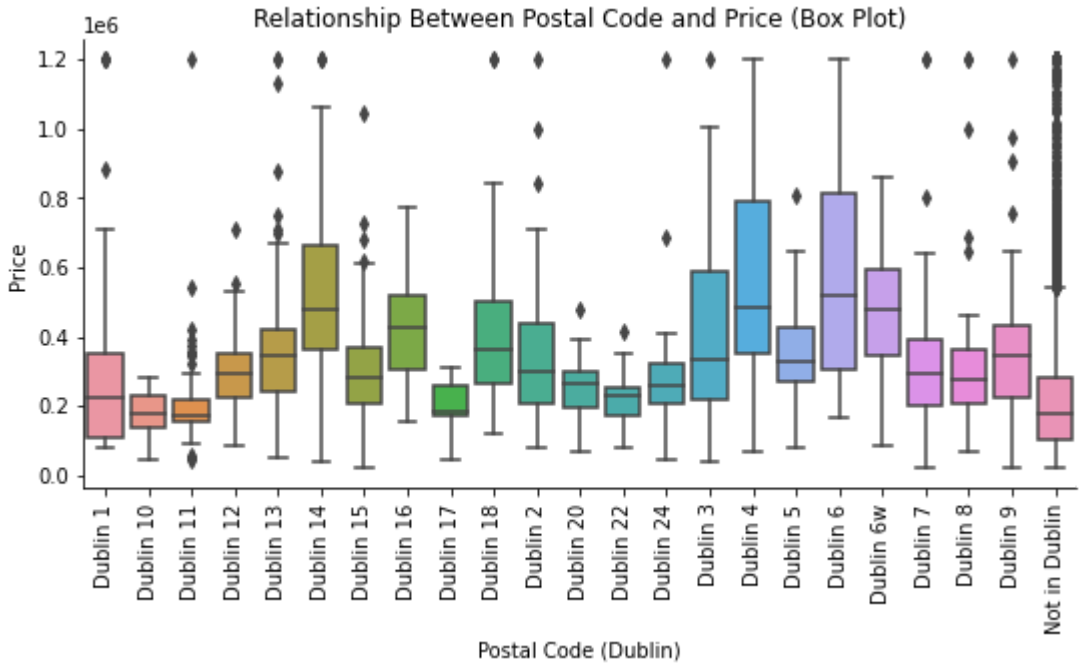
One interesting point is that it looks like the correlation is not precisely linear and is more of a curve (due to the aforementioned dip in 2012/2013). This is something to keep in mind when we are choosing an appropriate model type later in this assignment.

Overall, this should be a useful feature to include for the purposes of the predictive model.

Postal Code

In [331…

```
sns.catplot(x="Postal Code (Dublin)", y="Price", kind="box", data=df_train, height=4, aspect=
plt.xticks(rotation=90)
plt.title("Relationship Between Postal Code and Price (Box Plot)")
plt.show()
```



Discussion

The above graphs suggest a correlation between geographic location and price.

The two graphs show us that overall, the most expensive properties tend to be located in Dublin 4, Dublin 6, Dublin 6W and Dublin 14. The cheapest properties were sold in Dublin 11, Dublin 17, and Dublin 10. This is inline with expectation, as the cheapest properties are located in postal codes that are further out from the city center, and the most expensive properties are ones that are located in postal codes that are closer to city center but not in the inner city.

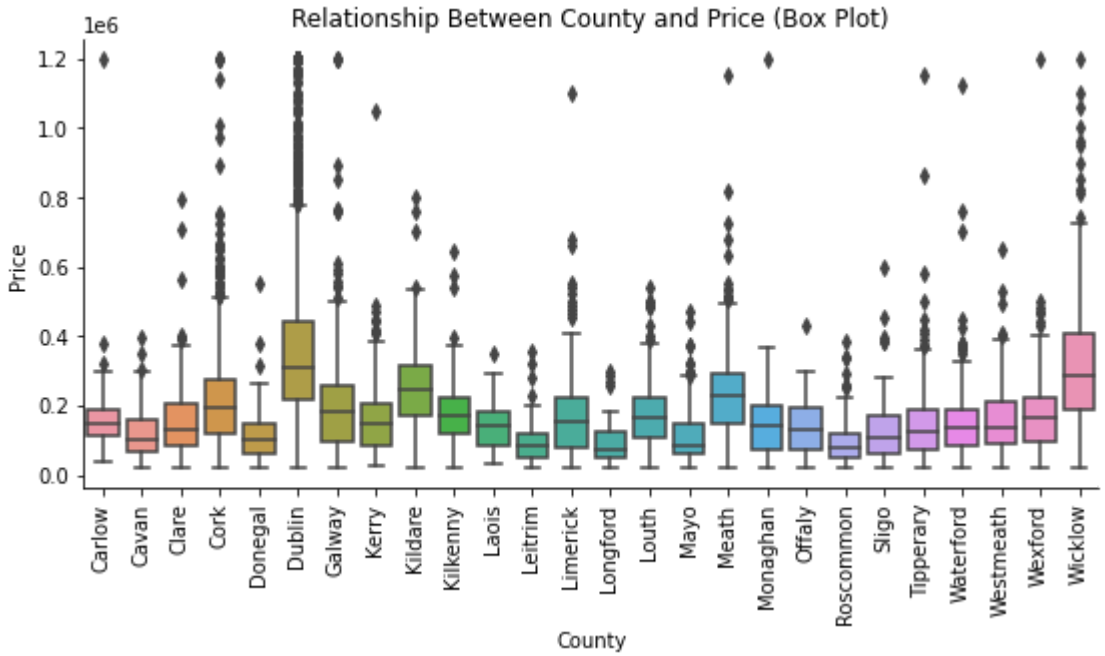
There also seems to be a larger spread of property prices overall in Dublin 6 and Dublin 4, which would seem somewhat unusual given that these locations are, on average, where the most expensive properties are located.

This should be a useful feature to include for the purposes of the predictive model.

County

In [332…

```
sns.catplot(x="County", y="Price", kind="box", data=df_train, height=4, aspect=2)
plt.xticks(rotation=90)
plt.title("Relationship Between County and Price (Box Plot)")
plt.show()
```



Discussion

The above graphs suggest a correlation between geographic location and price.

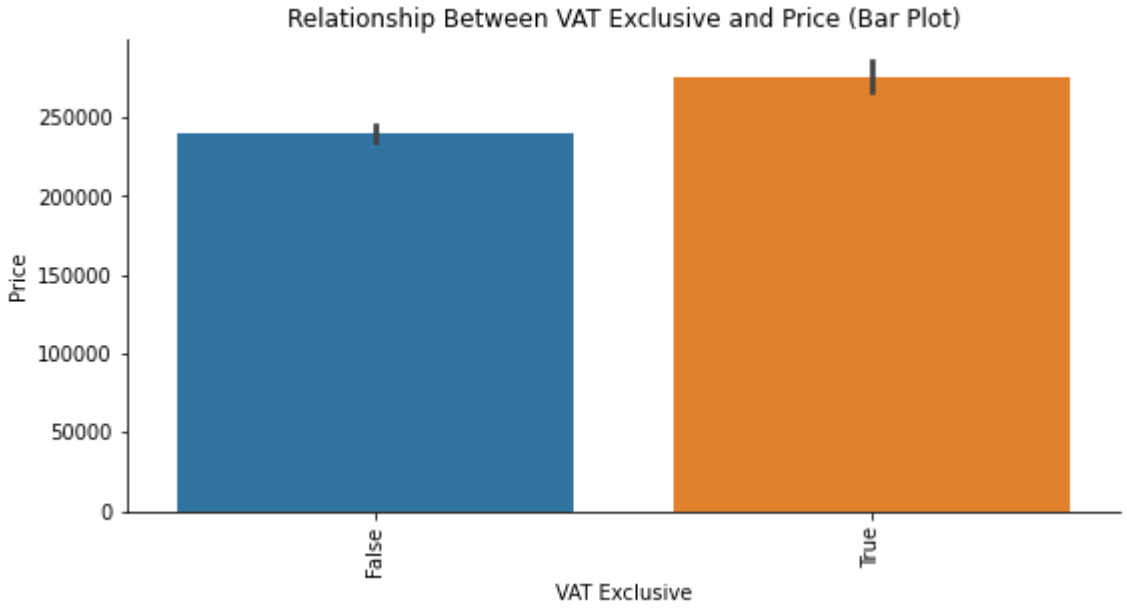
From the highest priced properties, we can see that Dublin and Wicklow have the highest prices, followed by Kildare and Meath, with the lowest prices being located in Leitrim, Longford and Donegal. It is curious that counties containing large cities such as Cork and Galway seem to have lower prices that counties like Meath or Kildare - perhaps this is something that should be discussed with a domain expert to see if there is a reason for this.

This should be a useful feature to include for the purposes of the predictive model.

VAT Exclusive

In [333…

```
sns.catplot(x="VAT Exclusive", y="Price", kind="bar", data=df_train, height=4, aspect=2)
plt.xticks(rotation=90)
plt.title("Relationship Between VAT Exclusive and Price (Bar Plot)")
plt.show()
```



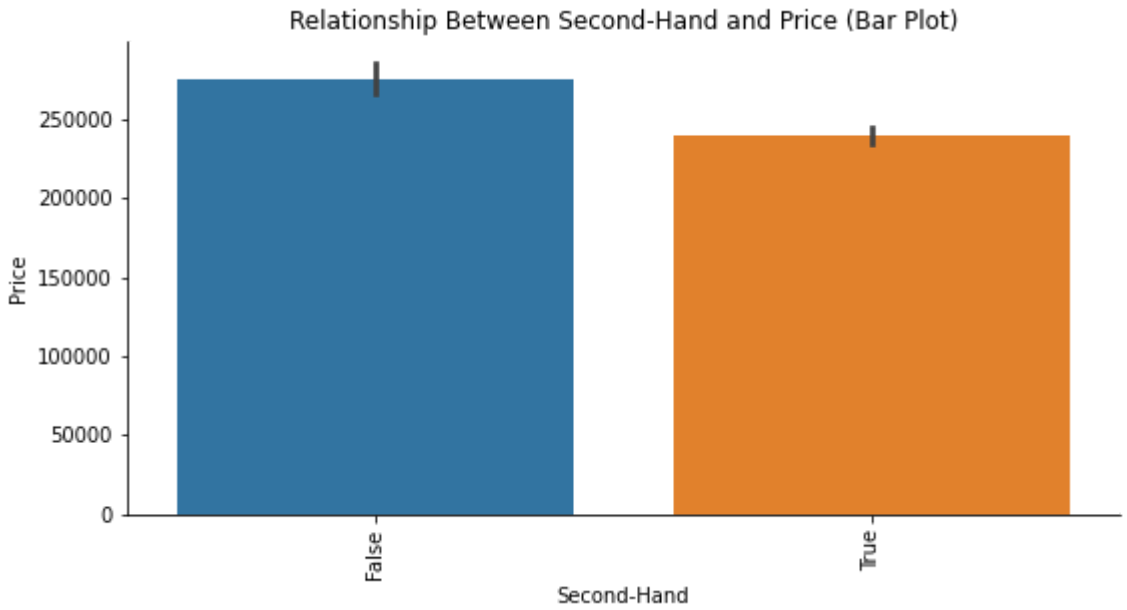
Discussion

As expected, and it seems that homes where VAT is payable are slightly more expensive than those that are not, suggesting that there is a slight correlation between the two features. This makes sense, as homes where VAT is payable are likely to be new, which would drive the price up.

The correlation seems smaller here when compared to the other features, and as such we will most likely exclude this from the predictive model.

Second-Hand

In [334…
```
sns.catplot(x="Second-Hand", y="Price", kind="bar", data=df_train, height=4, aspect=2)
plt.xticks(rotation=90)
plt.title("Relationship Between Second-Hand and Price (Bar Plot)")
plt.show()
```



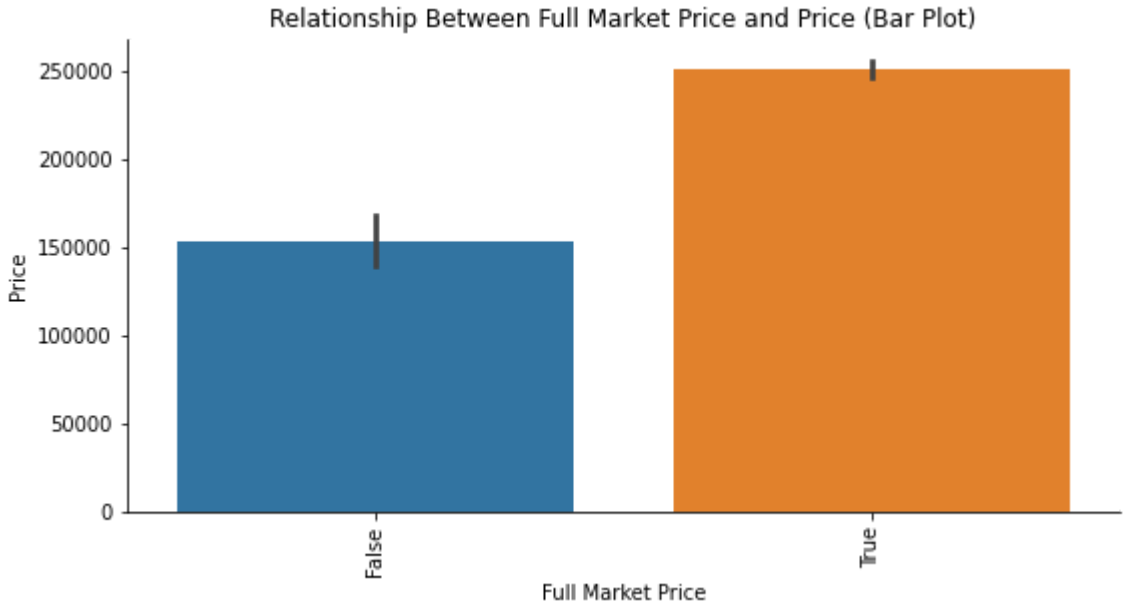
Discussion

From the above, we can see that while the difference between second hand and new homes is not large, on average new homes do seem to be a little more expensive than second-hand ones. This is in line with expectations, as one would expect that a second-hand home would be sold at a slight discount as opposed to a newly-built one. Therefore, there seems to be a weak correlation between second-hand and price. The spread between the two features seems largely the same.

The correlation seems smaller here when compared to the other features, and as such we will most likely exclude this from the predictive model.

Full Market Price

In [335…
```
sns.catplot(x="Full Market Price", y="Price", kind="bar", data=df_train, height=4, aspect=2)
plt.xticks(rotation=90)
plt.title("Relationship Between Full Market Price and Price (Bar Plot)")
plt.show()
```



Discussion

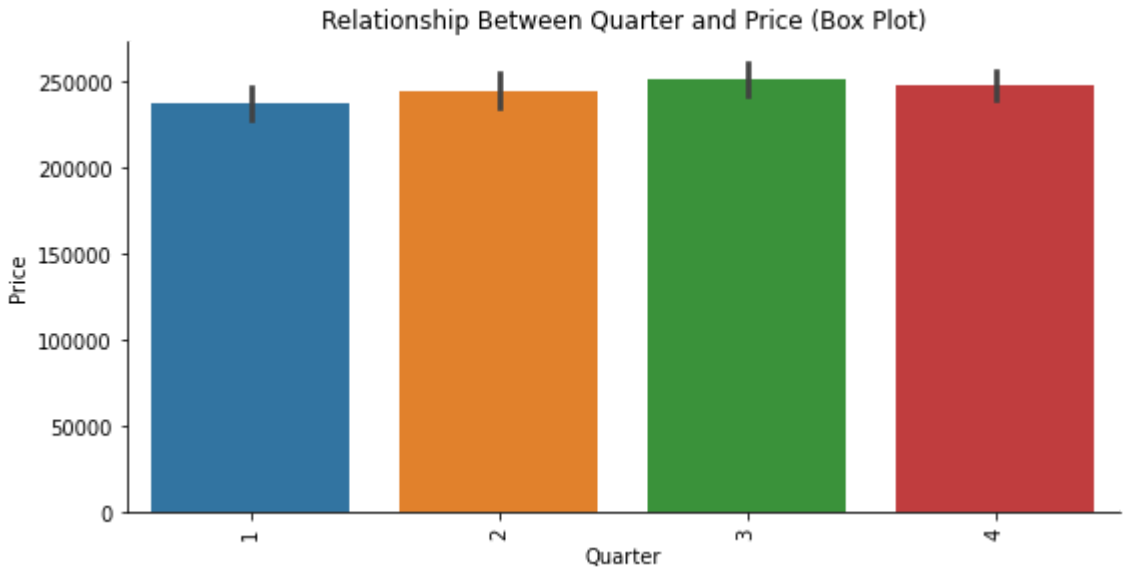
The outcome here is very unsurprising - properties marked as being sold at "Full Market Price" were sold for a higher price than those that are not. In fact, we would be surprised if the opposite was true.

Because there does seem to be a correlation here, we will include this feature in the predictive model.

Quarter

In [336…

```
sns.catplot(x="Quarter", y="Price", kind="bar", data=df_train, height=4, aspect=2)
plt.xticks(rotation=90)
plt.title("Relationship Between Quarter and Price (Box Plot)")
plt.show()
```



Discussion

The above suggests that most properties tend to be bought in Q4, and the least number of properties tend to be bought in Q1. The spread for each quarter seems to be roughly the same.

The above also suggests that there is a very weak correlation between price and quarter. We do, however, see that the most expensive properties tend to be bought in Q3, whereas the least expensive properties tend to be bought in Q1.

Because the correlation is quite weak, we will not be including this feature in the predictive model.

Conclusion

Based on the above analysis, our final list for descriptive features that will be included in the predictive model are set out in the summary table below.

Descriptive Feature	Inclusion
Inflation	No
Postal Code (Dublin)	Yes
Full Market Price	Yes
VAT Exclusive	No
Second-Hand	No
County	Yes
Year	Yes
Quarter	No

In total, we now have four descriptive features - Postal Code (Dublin), Full Market Price, Year and County - that we will use for the purposes of our predictive model.

It is interesting to note that we have only categorical features in our model. However, given the fact that we only had one continuous feature, which had poor correlation, this was, in our view, justifiable.

We may attempt to select a different sub-set of features (such as using the entire feature set) later on when we attempt to optimise our model.

Feature Conversion

Before training the model, we will need to convert our features in a way that the predictive model will understand. All of our features are categorical in nature, and are either boolean or non-boolean.

For a predictive model to understand a categorical feature, we will need to either split it out, or convert it, to a dummy variable that can be represented by a 1 or a 0 (denoting a value or the absence of one). For boolean features, this is quite simple - we can just map the value "True" to "1" and "False" to "0".

With non-boolean features, however, we will need to split such feature up. Each entry will contain new columns containing all possible values of such feature - all of these columns will be set to 0, with the exception of the category under which such entry falls.

It is easier to explain this with an example - let us take the County feature. For each row, we will have a new dummy feature representing each individual county, i.e. "Dublin", "Cork", etc. If the relevant property was located in Dublin, then all the dummy columns will be set to 0 with the exception of the "Dublin" column, which will be set to 1.

Note on Refactoring

A crucial point here is that we know that we will eventually need to perform the same work on

- The test partition;

- The full dataset (for K-Fold analysis); and
- The test dataset included in the assignment sheet.

When the steps are completed, we should convert the code into a single function that can be applied on any new datasets.

With an assignment such as this (and with all large coding projects) I find this sort of refactoring to be absolutely imperative so that:

- We do not get lost in our own code;
- We avoid unnecessary duplication of code; and
- We ensure that each time we apply some sort of transformation to our dataset, what we are doing to it is crystal clear, so that we don't accidentally create some sort of side effect that will skew the data.

Therefore, the reader will note such refactoring occur on certain complex operations that I believe we will be repeated on the dataset throughout the assignment. The reason for such refactoring will be clearly explained.

Because we will at some stage want to test our model on **all** the features that we have to see if it performs better, we should transform **all** of our values, in the first instance, but for our descriptive features select only the ones that we have chosen in the section above.

We will begin by transforming the boolean columns to "1" and "0" values.

In [337…
```
# Mapping boolean values to 0 and 1
df_train["Full Market Price"] = df_train["Full Market Price"].map({False: 0, True: 1})
df_train["VAT Exclusive"] = df_train["VAT Exclusive"].map({False: 0, True: 1})
df_train["Second-Hand"] = df_train["Second-Hand"].map({False: 0, True: 1})
```

Next, we will prepare dummy features for the remaining variables. Please note that instead of manually adding the variables, we can just use the Pandas "get_dummies" function, which will generate such dummies only.

In [338…
```
# Getting dummy variables. Note that we are treating inflation as a categorical feature for r
postal_code_dummies = pd.get_dummies(df_train["Postal Code (Dublin)"])
year_dummies = pd.get_dummies(df_train["Year"])
quarter_dummies = pd.get_dummies(df_train["Quarter"])
inflation_dummies = pd.get_dummies(df_train["Inflation"])
county_dummies = pd.get_dummies(df_train["County"])
```

We will now add these dummy features to our existing dataframe. Please note that even though we created dummies for all of our features, we are adding **only** the variables that we selected in the section above. Note also that we are dropping the original features that we converted into dummies - for example, "Year" was converted into a certain number of individual dummy variables, and as such the original "Year" variable is no longer needed, and can thus be dropped.

In [339…
```
# Concatenating existing dataframe with dummy variables and setting column names to string (a
df_train = pd.concat([df_train, postal_code_dummies, year_dummies, county_dummies], axis=1)
df_train.columns = df_train.columns.astype(str)

df_train = df_train.drop(["Postal Code (Dublin)", "VAT Exclusive", "Second-Hand", "Year", "Qu
```

We will print out all of our columns here to ensure that everything is working as intended.

In [340…

```
for column in df_train.columns:
    print(column)
```

```
Price
Full Market Price
Dublin 1
Dublin 10
Dublin 11
Dublin 12
Dublin 13
Dublin 14
Dublin 15
Dublin 16
Dublin 17
Dublin 18
Dublin 2
Dublin 20
Dublin 22
Dublin 24
Dublin 3
Dublin 4
Dublin 5
Dublin 6
Dublin 6w
Dublin 7
Dublin 8
Dublin 9
Not in Dublin
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
Carlow
Cavan
Clare
Cork
Donegal
Dublin
Galway
Kerry
Kildare
Kilkenny
Laois
Leitrim
Limerick
Longford
Louth
Mayo
Meath
Monaghan
Offaly
Roscommon
Sligo
Tipperary
Waterford
Westmeath
Wexford
Wicklow
```

We should separate our transformed `df_train` dataframe into X and y dataframes, where `X_train`

will represent the descriptive features, and `y_train` will represent the target feature. The reason we are doing this is because sklearn, the module we will use to generate the predictive models, will require two different dataframes to represent the target and descriptive features.

In [341…
```
# Assign the price column only to the y dataframe
y_train = df_train["Price"]

# Assign everything but the price column to the X dataframe
X_train = df_train.drop("Price", axis=1)
```

We will now print some features from each dataset to ensure that everything is functioning as expected.

In [342…
```
print(f"Sample of descriptive features in train dataset:\n{X_train.head(5)}")
print()
print(f"Sample of target features in train dataset:\n{y_train.head(5)}")
```

```
Sample of descriptive features in train dataset:
      Full Market Price  Dublin 1  Dublin 10  Dublin 11  Dublin 12  Dublin 13  \
3018                  1         0          0          0          0          0
5794                  1         0          0          0          0          0
1627                  1         0          0          0          0          0
3722                  1         0          0          0          0          0
5769                  1         0          0          0          0          0

      Dublin 14  Dublin 15  Dublin 16  Dublin 17  ...  Meath  Monaghan  \
3018          0          0          0          0  ...      0         0
5794          0          0          0          0  ...      0         0
1627          0          0          0          0  ...      0         0
3722          0          0          0          0  ...      0         0
5769          0          0          0          0  ...      0         0

      Offaly  Roscommon  Sligo  Tipperary  Waterford  Westmeath  Wexford  \
3018       0          0      0          0          0          0        0
5794       0          0      0          0          0          0        0
1627       0          0      0          0          0          0        0
3722       0          0      0          0          0          0        0
5769       0          0      0          0          0          0        0

      Wicklow
3018        0
5794        0
1627        0
3722        0
5769        0

[5 rows x 63 columns]

Sample of target features in train dataset:
3018    145000.0
5794    229074.0
1627    220000.0
3722    208500.0
5769    500000.0
Name: Price, dtype: float64
```

The above confirms that the training dataset has now been implemented correctly. However, we can see that the indices have been rearranged due to the shuffling. We should reset all the indices before proceeding, otherwise the dataframes will be much more difficult to concatenate later on. We will print some of the entries in each dataframe to ensure that everything is working as expected.

In [343…
```
# Resetting the index
X_train.reset_index(drop=True, inplace=True)
y_train.reset_index(drop=True, inplace=True)
```

```
X_train.head()
```

Out[343]:

	Full Market Price	Dublin 1	Dublin 10	Dublin 11	Dublin 12	Dublin 13	Dublin 14	Dublin 15	Dublin 16	Dublin 17	...	Meath	Monaghan
0	1	0	0	0	0	0	0	0	0	0	...	0	0
1	1	0	0	0	0	0	0	0	0	0	...	0	0
2	1	0	0	0	0	0	0	0	0	0	...	0	0
3	1	0	0	0	0	0	0	0	0	0	...	0	0
4	1	0	0	0	0	0	0	0	0	0	...	0	0

5 rows × 63 columns

In [344…
```
y_train.head()
```

Out[344]:
```
0    145000.0
1    229074.0
2    220000.0
3    208500.0
4    500000.0
Name: Price, dtype: float64
```

Now that we have decided on an approach when it comes to creating our X and y dataframes, as mentioned above, we should convert it into a function, because we will need to carry out the same steps on (a) the test partition and (b) the test dataset.

Please note that we have included a default argument named "all" which is set to false. If the user sets all to true, then the X and y values that will be provided will include **all** the descriptive features that we had in our original dataframe. This will be useful in the last part of the question where we will try to measure a different subset of features.

In [345…
```
def get_X_and_y(df, all=False):
    """Function that produced X and y dataframes based on an inputted dataframe"""
    # Map boolean values
    df["Full Market Price"] = df["Full Market Price"].map({False: 0, True: 1})
    df["VAT Exclusive"] = df["VAT Exclusive"].map({False: 0, True: 1})
    df["Second-Hand"] = df["Second-Hand"].map({False: 0, True: 1})

    # Get all dummies
    postal_code_dummies = pd.get_dummies(df["Postal Code (Dublin)"])
    year_dummies = pd.get_dummies(df["Year"])
    quarter_dummies = pd.get_dummies(df["Quarter"])
    inflation_dummies = pd.get_dummies(df["Inflation"])
    county_dummies = pd.get_dummies(df["County"])

    # If all flag is on, add all dummies. Otherwise, just add high correlation features
    if all:
        df = pd.concat([df, postal_code_dummies, year_dummies, county_dummies, quarter_dummie
    else:
        df = pd.concat([df, postal_code_dummies, year_dummies, county_dummies], axis = 1)
    df.columns = df.columns.astype(str)

    # If all flag is off, drop all features not related to high correlation
    if all:
        df = df.drop(["Postal Code (Dublin)", "County", "Year", "Quarter", "Inflation"], axis
    else:
        df = df.drop(["Postal Code (Dublin)", "County", "VAT Exclusive", "Second-Hand", "Year
```

```
    # Create X and y dataframes
    y = df["Price"]
    X = df.drop("Price", axis=1)

    # Reset the index
    X.reset_index(drop=True, inplace=True)
    y.reset_index(drop=True, inplace=True)

    # Provide output
    return X, y
```

Question 2: Predictive Modelling and Evaluation (Linear Regression)

The first model that we will be using is linear regression. Linear regression is represented by the following formula:

$$target\_value = w_0 + w_1 * feature\_1 + w_2 * feature\_2 + \ldots + w_n * feature\_n$$

In this formula, the $feature\_1$ to $feature\_n$ represent the descriptive features (i.e. the features that we have stored in the "X" variable), the $target\_value$ represents the value of the target feature (in this case, Price), and w_0 to w_n represent the "weights" or coefficients assigned to each feature. The coefficients will be further explained below.

Linear regression works by plotting the values of the descriptive features against the values of the target feature, and attempts to draw a straight line through these in order to represent a linear relationship between all of these features. The resulting line is represented by the formula in the form noted above, with the coefficients affixed to each respective feature representing the slope of the resulting line.

The value w_0 represents the "intercept", i.e. the place where our line intersects the y-axis.

- **Advantages**: The model is relatively simple to implement and works very well when there is a linear relationship between the data.
- **Disadvantages**: The main issue with linear regression is that if there is no linear relationship between the data, then the model will be inaccurate. It is entirely possible for there to be a relationship between descriptive features and a target feature, but the resulting model looks more like a curve than a line. In such a situation, something like polynomial regression would probably be a better fit for the data. Linear regression is also very sensitive to outliers, as these may shift the slope of the line in a significant way.

2.1 Training the model

On the training set, train a linear regression model to predict the target feature, using only the descriptive features selected in exercise (1) above.

We have already prepared our training set in the section above. In order to train a linear regression model, we will use the `LinearRegression()` object of sklearn, which will prepare the various elements, such as coefficients and the intercept, required for later prediction, and will add these elements as properties of the object for ease of use.

In order to train the model, we will create a LinearRegression model and will use its built-in `fit` method, using our X_train and y_train dataframes as input. This process will be very similar for all the models that we will train over the course of this assignment.

We will train the model and will print the various properties out for ease of review.

In [346…

```
linear_regression = LinearRegression().fit(X_train, y_train)

print("LINEAR REGRESSION MODEL SUMMARY")
print("***")
print(f"Descriptive Features:\n{X_train.columns}")
print("***")
print(f"Target Feature:\nPrice")
print("***")
print(f"Coefficients:\n{linear_regression.coef_}")
print("***")
print(f"Intercept:\n{linear_regression.intercept_}")
```

```
LINEAR REGRESSION MODEL SUMMARY
***
Descriptive Features:
Index(['Full Market Price', 'Dublin 1', 'Dublin 10', 'Dublin 11', 'Dublin 12',
       'Dublin 13', 'Dublin 14', 'Dublin 15', 'Dublin 16', 'Dublin 17',
       'Dublin 18', 'Dublin 2', 'Dublin 20', 'Dublin 22', 'Dublin 24',
       'Dublin 3', 'Dublin 4', 'Dublin 5', 'Dublin 6', 'Dublin 6w', 'Dublin 7',
       'Dublin 8', 'Dublin 9', 'Not in Dublin', '2010', '2011', '2012', '2013',
       '2014', '2015', '2016', '2017', '2018', '2019', '2020', '2021', '2022',
       'Carlow', 'Cavan', 'Clare', 'Cork', 'Donegal', 'Dublin', 'Galway',
       'Kerry', 'Kildare', 'Kilkenny', 'Laois', 'Leitrim', 'Limerick',
       'Longford', 'Louth', 'Mayo', 'Meath', 'Monaghan', 'Offaly', 'Roscommon',
       'Sligo', 'Tipperary', 'Waterford', 'Westmeath', 'Wexford', 'Wicklow'],
      dtype='object')
***
Target Feature:
Price
***
Coefficients:
[ 8.14455417e+04 -9.75071578e+17 -9.75071578e+17 -9.75071578e+17
 -9.75071578e+17 -9.75071578e+17 -9.75071578e+17 -9.75071578e+17
 -9.75071578e+17 -9.75071578e+17 -9.75071578e+17 -9.75071578e+17
 -9.75071578e+17 -9.75071578e+17 -9.75071578e+17 -9.75071578e+17
 -9.75071578e+17 -9.75071578e+17 -9.75071578e+17 -9.75071578e+17
 -1.25568634e+17 -1.25568634e+17 -1.25568634e+17 -1.25568634e+17
 -1.25568634e+17 -1.25568634e+17 -1.25568634e+17 -1.25568634e+17
 -1.25568634e+17 -1.25568634e+17 -1.25568634e+17 -1.25568634e+17
 -1.25568634e+17 -1.77154838e+17 -1.77154838e+17 -1.77154838e+17
 -1.77154838e+17 -1.77154838e+17 -1.77154838e+17 -1.77154838e+17
 -1.77154838e+17 -1.77154838e+17 -1.77154838e+17 -1.77154838e+17
 -1.77154838e+17 -1.77154838e+17 -1.77154838e+17 -1.77154838e+17
 -1.77154838e+17 -1.77154838e+17 -1.77154838e+17 -1.77154838e+17
 -1.77154838e+17 -1.77154838e+17 -1.77154838e+17 -1.77154838e+17]
***
Intercept:
1.277795050083349e+18
```

Please note that while the above was useful for getting a general sense of what the features are and what the coefficients / intercept are, the descriptive features and the target feature will remain the same throughout this project. For this reason, going forward we will just be providing the coefficients / feature importance with the features for the remaining models.

2.2 Interpreting the model

Can you interpret the linear regression model? Discuss any knowledge you can gain in regard of the working of this model.

Before generating any kind of metrics for the model, it would be useful to see which of the features have a bigger impact on the outcome of the prediction. The features that have a bigger impact are the ones with the bigger coefficient, as they will affect the slope of the line.

We have already shown the coefficients that the model has produced above, but it would be useful to pair them with the appropriate feature.

We will print out each feature and its coefficient, and sort them by the size of such coefficient.

Note: It would have been useful to visualise the linear regression model by plotting it as a graph. However, due to the number of features this is unfortunately not possible.

In [347…

```
feature_importance = pd.DataFrame({'feature': X_train.columns, 'importance':linear_regression

# Print the entire dataframe
with pd.option_context("display.max_rows", None, "display.max_columns", None):
    print(feature_importance.sort_values(by='importance', ascending=False, key=abs))
```

```
              feature    importance
2           Dublin 10 -9.750716e+17
3           Dublin 11 -9.750716e+17
9           Dublin 17 -9.750716e+17
13          Dublin 22 -9.750716e+17
12          Dublin 20 -9.750716e+17
7           Dublin 15 -9.750716e+17
14          Dublin 24 -9.750716e+17
4           Dublin 12 -9.750716e+17
21           Dublin 8 -9.750716e+17
1            Dublin 1 -9.750716e+17
20           Dublin 7 -9.750716e+17
22           Dublin 9 -9.750716e+17
23      Not in Dublin -9.750716e+17
17           Dublin 5 -9.750716e+17
15           Dublin 3 -9.750716e+17
11           Dublin 2 -9.750716e+17
10          Dublin 18 -9.750716e+17
5           Dublin 13 -9.750716e+17
8           Dublin 16 -9.750716e+17
19          Dublin 6w -9.750716e+17
6           Dublin 14 -9.750716e+17
18           Dublin 6 -9.750716e+17
16           Dublin 4 -9.750716e+17
50           Longford -1.771548e+17
48            Leitrim -1.771548e+17
56          Roscommon -1.771548e+17
41            Donegal -1.771548e+17
38              Cavan -1.771548e+17
52               Mayo -1.771548e+17
55             Offaly -1.771548e+17
47              Laois -1.771548e+17
57              Sligo -1.771548e+17
58          Tipperary -1.771548e+17
59          Waterford -1.771548e+17
54           Monaghan -1.771548e+17
39              Clare -1.771548e+17
60          Westmeath -1.771548e+17
44              Kerry -1.771548e+17
37             Carlow -1.771548e+17
49           Limerick -1.771548e+17
61            Wexford -1.771548e+17
51              Louth -1.771548e+17
46           Kilkenny -1.771548e+17
43             Galway -1.771548e+17
40               Cork -1.771548e+17
53              Meath -1.771548e+17
45            Kildare -1.771548e+17
62            Wicklow -1.771548e+17
42             Dublin -1.771548e+17
27               2013 -1.255686e+17
28               2014 -1.255686e+17
26               2012 -1.255686e+17
29               2015 -1.255686e+17
25               2011 -1.255686e+17
30               2016 -1.255686e+17
24               2010 -1.255686e+17
31               2017 -1.255686e+17
36               2022 -1.255686e+17
32               2018 -1.255686e+17
34               2020 -1.255686e+17
33               2019 -1.255686e+17
35               2021 -1.255686e+17
0   Full Market Price  8.144554e+04
```

Initial thoughts on the model are that it does not seem to be particularly accurate. As we can see from

section 2.1 above, the intercept is equal to 1.277795050083349e+18, which represents the value of the target feature when all other descriptive features are set to 0. However, this is such a large number, that it is impossible that any housing price would ever be that high.

Furthermore, as we can see from the values of the coefficients, their size also implies that they will have drastic effects on the slope of the line for outside the range of what we would consider normal for the value of the Irish property.

One interesting point to note is the order of feature importance. Note that whether a coefficient is positive or negative implies whether or not a feature is has positive or negative correlation with the target price, and as such we have printed these by their abs value (as a high negative correlation is as important to the model as a high positive correlation).

We can roughly see that features relating to the postcode of a property based in Dublin have the biggest effect on the model, followed by the county, the year, and then finally whether or not the property was sold for full market price. The last of these has a considerable smaller coefficient when compared to the rest of the properties, and it is the only feature that has a positive correlation with the target feature.

This is roughly in line with the correlations that we saw in the prior section of this assignment, which shows that our analysis was on the right track.

2.3 Initial model metrics

Print the predicted target feature value for the first 10 training examples. Print a few regression evaluation measures computed on the full training set and discuss your findings so far.

As stated in the assignment sheet, we should first print the predicted target feature value for the first 10 training examples.

In [348…
```
linear_regression_train_predictions = linear_regression.predict(X_train)

# Concatenate the column containin actual price with the predictions
pd.concat([y_train, pd.DataFrame(linear_regression_train_predictions, columns = ["Prediction"
```

Out[348]:

	Price	Prediction
0	145000.00	134400.0
1	229074.00	140032.0
2	220000.00	310784.0
3	208500.00	396544.0
4	500000.00	220416.0
5	99000.00	112128.0
6	810572.68	396544.0
7	295000.00	186880.0
8	243000.00	223744.0
9	670000.00	364288.0

Initial review indicates that the price does seem quite a bit off. However, before making any definitive statements about the quality of the model, we should print some metrics to see the performance. I have chosen the following metrics to evaluate the model:

- Root Mean Squared Error (RMSE)
- Mean Absolute Error (MAE)
- R2

We will use the built-in "metrics" feature of sklearn in order to calcalute these. However, before doing so, it would be useful to briefly explain what each of these metrics represent and how to interpret their results.

Root Mean Squared Error (RMSE)

RMSE works by taking the difference between the actual and predicted values, obtaining the squared value of that difference, taking the mean, and then taking the square root.

Because the difference between actual and predicted values are raised to the power of two, particularly larger errors will be given extra weight in the final score. This means that RMSE is particularly useful to see if there are any particularly large errors in our model, which can be useful if low variance between actual value and prediction is important.

Mean Absolute Error (MAE)

MAE just takes the absolute mean of the difference between actual and predicted values. It is similar to RMSE in this way, however, it does not give any particular weight to large errors, and as such is better for getting a broader overview of the performance of the model.

R2

R2 is calculated by subtracting from 1 the sum of squared errors, i.e. the squared sum of the predictions of our model, divided by the total sum of squares, i.e. the predictions of the average model.

Effectively, this tells us how much better our model is when compared to just taking the average of the target feature for every guess, with a score of 1 being good and a score of 0 being bad.

Because we know that we will need to get metrics for every model in this assignment, we will make create a function to assist us with this. We will do this by providing an input of actual and predicted values, and then calculate the various metrics by calling the relevant sklearn metrics method. We will then save these in a dictionary for ease of access.

In [349…

```
def get_metrics(actual_values, prediction_values):
    """Function that calculates metrics and outputs dictionary containing such metrics"""
    mse = metrics.mean_squared_error(actual_values, prediction_values)
    rmse = math.sqrt(mse)
    mae = metrics.mean_absolute_error(actual_values, prediction_values)
    r2 = metrics.r2_score(actual_values, prediction_values)
    return {"RMSE": rmse, "MAE": mae, "R2": r2}

# Get metrics for linear regression train
linear_regression_train_results = get_metrics(y_train, linear_regression_train_predictions)
```

We will also need some way to cleanly display these metrics. For this, we will also create a function,

which will read in the title of the model, as well as the dictionary outputted from the `get_metrics` method above.

In [350…
```
def print_metric_results(title, metric_results):
    """Function that provides a neat output for the metrics"""
    print(f"{title} METRICS")
    print("***")
    print(f"RMSE: {metric_results['RMSE']}")
    print(f"MAE: {metric_results['MAE']}")
    print(f"R2: {metric_results['R2']}")
    print("***")

print_metric_results("LINEAR REGRESSION TRAIN", linear_regression_train_results)
```

```
LINEAR REGRESSION TRAIN METRICS
***
RMSE: 160315.0117722458
MAE: 105072.40856644438
R2: 0.3301420396261916
***
```

As we can see from the above, the RMSE and MAE are fairly high, meaning that on average, our prediction will be between 105k and 160k off. While there is great variation in the prices of homes in our database, which is something that we noted in Assignment 1, but such errors are still not ideal.

The fact that the RMSE is quite a bit larger than MAE also suggests that there are a lot of larger errors for some of the predictions in our databass (even though some may be a bit more accurate).

The R2 error is also somewhat low, suggesting that our predictions are only roughly 33% better than the "average" model.

2.4 Evaluation using test set and cross-validation

Evaluate the model using regression evaluation measures on the hold-out (30% examples) test set. Compare these results with the evaluation results obtained on the training (70% examples) dataset. Also compare these results with a cross-validated model (i.e., a new model trained and evaluated using cross-validation on the full dataset). You can use classic k-fold cross-validation or k repeated random train/test (70/30) splits. Compare the cross-validation metrics to those obtained on the single train/test split and discuss your findings.

Having trained the model using the test set, we can now check how it performs on data that we have not seen before, i.e. the 30% test set. We will perform the same steps as listed in 2.2 and 2.3, initially just printing the first 10 predictions, and then performing full metrics analysis.

It should be noted that this is the first time that we are working with the test set, meaning that it is not yet transformed for use with predictive models. We should confirm this by printing a few entries from the test set.

The reason why we need to test our model on the test set is because predictive models tend to be prone to "over-fitting", which means that they become very good at predicting data that they have already seen, similar to memorising the answers to a test without actually understanding their meaning. If we provide a sample input of something that the model has not seen before, then we should get a better understanding at how good the model actually is at predicting data given a set of input variables.

In [351…]
```
df_test.head()
```

Out[351]:

	Postal Code (Dublin)	County	Price	Full Market Price	VAT Exclusive	Second-Hand	Year	Quarter	Inflation
7765	Dublin 3	Dublin	425000.0	True	False	True	2019	3	0.939044
5227	Not in Dublin	Wicklow	400000.0	True	True	False	2021	3	0.374889
858	Not in Dublin	Kerry	210000.0	True	False	True	2010	1	-0.922096
2702	Not in Dublin	Sligo	35500.0	False	False	True	2017	1	0.340532
7331	Not in Dublin	Carlow	177500.0	True	False	True	2015	3	-0.289879

Thankfully, we have prepared a function that will easily provide us with X and y values for the test dataset. Because we just want to use the descriptive features that we selected in the sections above, we will keep the default "all" argument as "False".

In [352…]
```
X_test, y_test = get_X_and_y(df_test)

X_test.head()
```

Out[352]:

	Full Market Price	Dublin 1	Dublin 10	Dublin 11	Dublin 12	Dublin 13	Dublin 14	Dublin 15	Dublin 16	Dublin 17	...	Meath	Monaghan
0	1	0	0	0	0	0	0	0	0	0	...	0	0
1	1	0	0	0	0	0	0	0	0	0	...	0	0
2	1	0	0	0	0	0	0	0	0	0	...	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0
4	1	0	0	0	0	0	0	0	0	0	...	0	0

5 rows × 63 columns

In [353…]
```
y_test.head()
```

Out[353]:
```
0    425000.0
1    400000.0
2    210000.0
3     35500.0
4    177500.0
Name: Price, dtype: float64
```

We know that we will need to perform K-Fold cross-validation in this part of the question also. K-Fold cross-validation needs to be performed on the **entire** dataset, for reasons that will be explained in more detail below. For this reason, we can prepare a fully transformed version of the full dataset, too.

It should be noted that we will make a copy of the dataframe before transforming it just so that we do not make any changes to hte original dataset, just in case.

In [354…]
```
df_full = df.copy()

X_full, y_full = get_X_and_y(df_full)

X_full.head()
```

Out[354]:

	Full Market Price	Dublin 1	Dublin 10	Dublin 11	Dublin 12	Dublin 13	Dublin 14	Dublin 15	Dublin 16	Dublin 17	...	Meath	Monaghan
0	1	0	0	0	0	0	0	0	0	0	...	0	0
1	1	0	0	0	0	0	0	0	0	0	...	0	0
2	1	0	0	0	0	0	0	0	0	0	...	0	0
3	1	0	0	0	0	0	0	0	0	0	...	0	0
4	1	0	0	0	0	0	0	0	0	0	...	0	0

5 rows × 63 columns

Test Data

We will now make predictions based on our newly created test partition. Now that we **do not** use `fit` on the model again. The whole purpose of this exercise is that we use the model trained using the "train" dataset, but use the "test" dataset for the input, to avoid the overfitting issue described above.

In [355…

```
linear_regression_test_predictions = linear_regression.predict(X_test)

pd.concat([y_test, pd.DataFrame(linear_regression_test_predictions, columns = ["Prediction"])
```

Out[355]:

	Price	Prediction
0	425000.0	448256.0
1	400000.0	381184.0
2	210000.0	152064.0
3	35500.0	75520.0
4	177500.0	143104.0
5	120000.0	201216.0
6	171000.0	161792.0
7	122000.0	165632.0
8	445000.0	332032.0
9	311400.0	310784.0

In [356…

```
linear_regression_test_results = get_metrics(y_test, linear_regression_test_predictions)
print_metric_results("LINEAR REGRESSION TEST", linear_regression_test_results)
```

```
LINEAR REGRESSION TEST METRICS
***
RMSE: 159630.91629566884
MAE: 103548.9535818792
R2: 0.3323223335222596
***
```

The results that we get are a little surprising. Typically, we would expect test data to perform a little bit worse than train data, because the model will be working with data it had not seen yet. Here, however, if we compare the metrics against those that we received in the "train" dataset, we can see that for the test set, then we can see that there is a 0.43% improvement in the RMSE metric, and a 0.66% improvement in the R2 metric. In the MAE metric, however, the test dataset performed 1.45% worse.

So while we can see that the model is generalising, and is able to predict information based on data it has not seen before roughly with the same level of accuracy, it is a little unusual that it seems to be performing *better* when compared to the initial calculations.

Due to the very small variations in the metrics it is possible that this is just coincidental due to the way in which the data was split. We will get a better understanding of how well or poorly the model performs during the next step, which is cross-validation.

Note: All percentages calculated above have been rounded to two decimal places.

Cross-Validation

We will now attempt to evaluate what we have achieved so far with a cross-validated model. Note that the assignment states that we can use either k-fold cross validation or k-repeated random train / test split.

We have decided to use k-fold cross validation, as there is a very helpful sklearn function that allows us to calculate this quickly and efficiently. What this entails is repeatedly testing the model on different "folds" (we have chosen five folds) to ensure that we are in effect getting a different "train / test" split each time.

The purpose of this is to avoid over-fitting, which, as noted above, is a phenomenon that occurs when a model is very familiar with the dataset that it is trained on, and is not particularly generalised. By testing on random folds of the dataset, we can get a better sense of how the model will perform when it encounters data that it has not seen before.

Note that because we will need to prepare this process for all models that we create, we will create a function that will output lists of values for RMSE, MAE and R2 for each fold, stored in a dictionary as per the previous metric function that we created. We will use five folds total, and again will set `random_state` on to ensure that we get the same values each time this notebook is run.

In [357…
```
def get_k_fold_metrics(model, X_data, y_data):
    """Function that provides cross-evaluation metrics for given X and y dataframes"""
    # Get K-Folds
    k_folds = KFold(n_splits = 5, shuffle = True, random_state = 1)

    # Get the MSE score
    mse_scores = abs(cross_val_score(model, X_data, y_data, scoring = "neg_mean_squared_error

    # Get the square root of MSE to get the RMSE score
    rmse_scores = np.array([math.sqrt(x) for x in mse_scores])

    # Get the MAE score
    mae_scores = abs(cross_val_score(model, X_data, y_data, scoring = "neg_mean_absolute_erro

    # Get the R2 score
    r2_scores = cross_val_score(model, X_data, y_data, scoring = "r2", cv = k_folds)

    # Output
    return {"RMSE": rmse_scores, "MAE": mae_scores, "R2": r2_scores}
```

In [358…
```
linear_regression_scores = get_k_fold_metrics(linear_regression, X_full, y_full)
linear_regression_scores
```

Out[358]:
```
{'RMSE': array([161159.01851517, 161125.65436924, 167159.80239002, 155687.72751709,
        158565.42431849]),
 'MAE': array([102745.34680802, 107501.83313035, 107082.99659591, 105944.32944614,
        102692.69041939]),
 'R2': array([0.3279919 , 0.35719752, 0.30498628, 0.31583828, 0.31732542])}
```

The above has provided us with the list of RMSE, MAE and R2 values as expected. However, as it stands, it is a little bit difficult to visualise how these all compare to our train and test metrics. For ease of review, we will take the average of these metrics before continuing our analysis.

In [359…
```
linear_regression_scores_average = {"RMSE": linear_regression_scores["RMSE"].mean(), "MAE": l
linear_regression_scores_average
```

Out[359]:
```
{'RMSE': 160739.52542200344,
 'MAE': 105193.43927996237,
 'R2': 0.32466787892465676}
```

The above scores are much more in line with what we would expect. As we can see there is a clear decrease in performance in all three categories of metric (by roughly 1k-2k in MAE and RMSE, and 0.01 in R2). However, the decrease in performance is largely minor, meaning that the model has generalised well.

Because we know that we are going to have to repeat the process of getting the metrics for every model that we train, we should find a way to consolidate all of this data in a single function.

Upon further consideration, it was decided that a dataframe will be created, where each column will contain the metric of a particular dataset, and each row will represent a given model. This way, we will be able to easily compare all of our results against each other.

It should be noted that we have decided to include as parameters X and y dataframes for each type of dataset, to separate the training process from the metric calculation process.

In [360…
```
global_results = pd.DataFrame()

def get_full_metrics(model, X_train, y_train, X_test, y_test, X_full, y_full):
    """Get  full metrics of a dataset and return dataframe containing the results"""
    df_results = pd.DataFrame()

    # Prediction based on train dataset
    train_predictions = model.predict(X_train)

    # Call get_metrics function and loop through the results, adding them to the output dataf
    train_results = get_metrics(y_train, train_predictions)
    for metric in train_results.keys():
        df_results[f"Train ({metric})"] = pd.Series(train_results[metric])

    # Predition based on test dataset
    test_predictions = model.predict(X_test)

    # Call get_metrics function and loop through the results, adding them to the output dataf
    test_results = get_metrics(y_test, test_predictions)
    for metric in test_results.keys():
        df_results[f"Test ({metric})"] = pd.Series(test_results[metric])

    # Call get_k_fold_metrics function and add average scores to the output dataframe
    model_scores = get_k_fold_metrics(model, X_full, y_full)
    model_scores_average = {"RMSE": model_scores["RMSE"].mean(),
                            "MAE": model_scores["MAE"].mean(),
                            "R2": model_scores["R2"].mean()}

    for metric in model_scores_average.keys():
```

```
        df_results[f"K-Fold ({metric})"] = pd.Series(model_scores_average[metric])

    # Rearrange the output columns to make them easier to read
    df_results = df_results[["Train (RMSE)", "Test (RMSE)", "K-Fold (RMSE)",
                             "Train (MAE)", "Test (MAE)", "K-Fold (MAE)",
                             "Train (R2)", "Test (R2)", "K-Fold (R2)"]]

    # Output results
    return df_results

# Add linear regression results to the global dataframe
df_linear_regression = get_full_metrics(linear_regression, X_train, y_train, X_test, y_test,
global_results = pd.concat([global_results, df_linear_regression], ignore_index=True, axis=0)
global_results = global_results.rename(index={0: "Linear Regression"})
global_results
```

Out[360]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Trai (R2
Linear Regression	160315.011772	159630.916296	160739.525422	105072.408566	103548.953582	105193.43928	0.33014

We can see from the above that the row has been successfully added to the global results dataframe, and we can now proceed with training and evalulating more models and adding them to this dataframe.

Question 3: Predictive Modelling and Evaluation (Decision Tree)

The second model that we will be using is the decision tree. The decision tree is an important model in machine learning because it is not only useful on its own, but is also used as part of consensus models such as random forest (which we will look at later in the assignment).

We will begin with a short explanation of how the decision tree model works, and then will proceed with creating and training such a model on our dataset.

It is first important to note what kind of decision tree we will be using in this assignment. Depending on the target variable, a decision tree can be either categorical and regressive. Because our target variable, Price, is continuous, we will be using a regressive decision tree.

A decision tree works by splitting the entire data set along certain categories (being the decision nodes) in such a way that the impurity of the data set is reduced with each split (by using a metric like the Gini index). The goal is that, having received a particular set of category values, we should be able to follow the tree down to the bottom by moving through the decisions and receive a prediction.

It is a useful model, but unfortunately it tends to be prone to overfitting, which is something that we will have to watch out for when reviewing the metrics.

The model is a little more intuitive to explain by using a diagram, so we will now proceed by creating decision trees with different depths, and drawing diagrams for each one.

3.1 Training the model

On the training set, train a decision tree model to predict the target feature, using the descriptive features

selected in exercise (1) above.

We will begin by importing the tree object from sklearn and creating two trees, one having a depth of 4 and the other having a depth of 10. We will do this so that we can examine how the predictions increase or decrease the more levels are added. We will ensure to set a random seed on these so that we can reproduce the exact values whenever the code in this notebook is re-run.

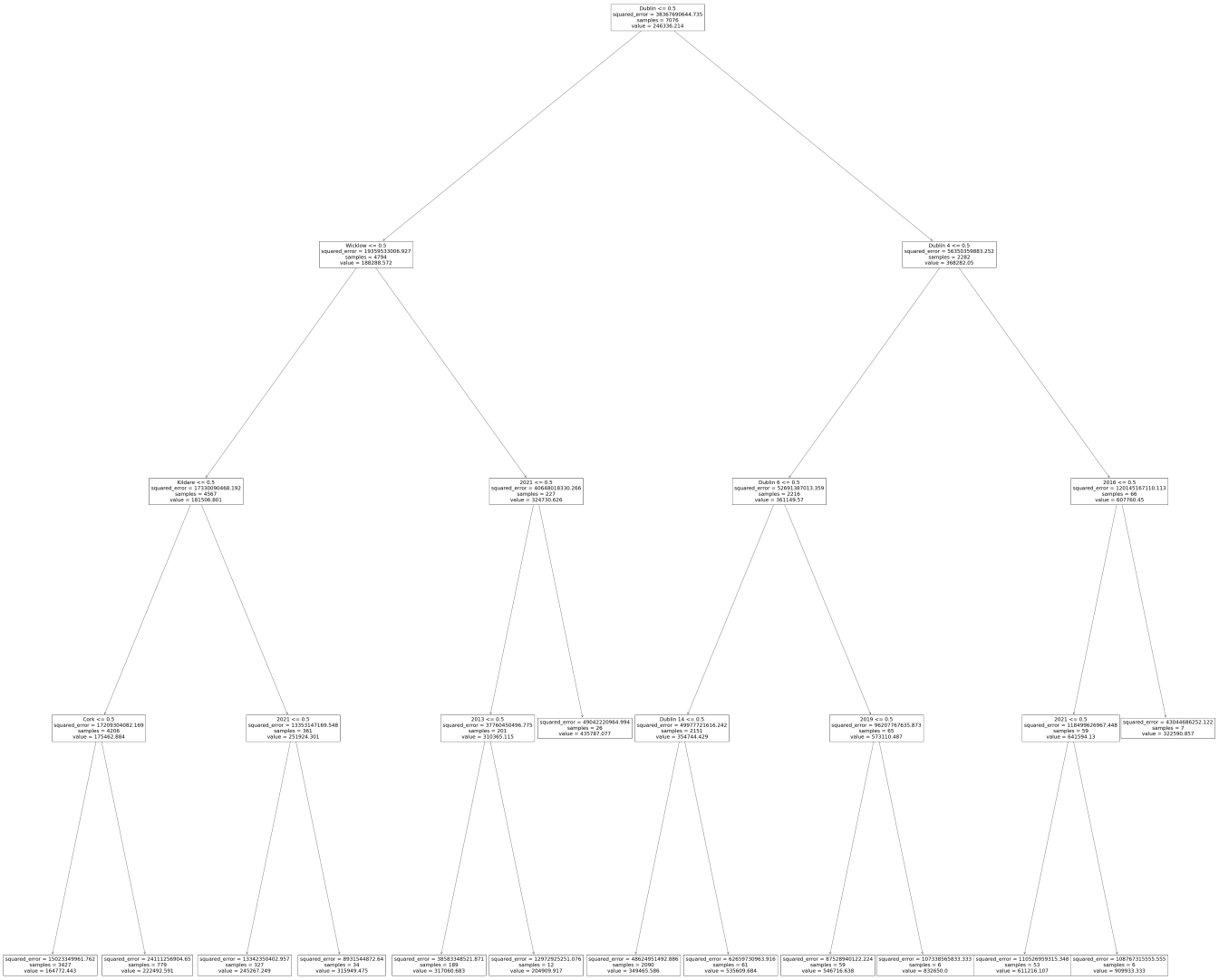
We will then proceed by training both of the models on the dataset, and visualising both trees using the "plot_tree" sklearn method.

In [361…

```
from sklearn import tree

tree_depth_4 = tree.DecisionTreeRegressor(max_depth = 4, random_state = 1)
tree_depth_10 = tree.DecisionTreeRegressor(max_depth = 10, random_state = 10)

tree_4 = tree_depth_4.fit(X_train, y_train)
tree_10 = tree_depth_10.fit(X_train, y_train)
```

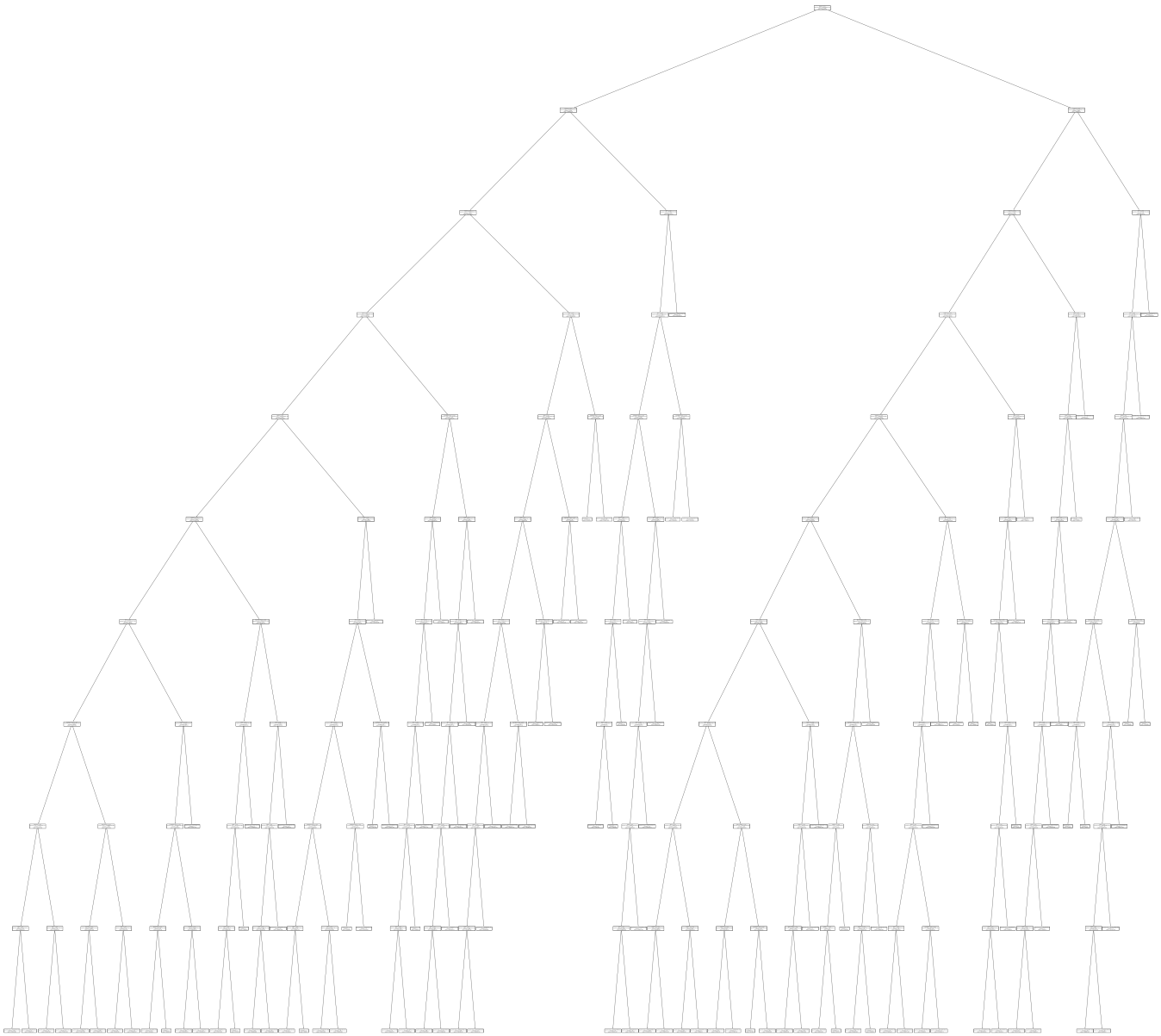
In [362…

```
fig = plt.figure(figsize = (100, 100))

_ = tree.plot_tree(tree_4, feature_names =  X_train.columns, class_names = "Price")

plt.savefig("./tree_graphs/tree_4.png")
```



In [363…

```
fig = plt.figure(figsize = (100, 100))

_ = tree.plot_tree(tree_10, feature_names =  X_train.columns, class_names = "Price")

plt.savefig("./tree_graphs/tree_10.png")
```



The above representations of the decision tree models are a good visualisation of how this model is structured. We can see the specific features that are being used for the decision nodes at every step of the way, as well as the relevant value at the leaf nodes that will constitute our prediction.

As we can see from the above, the visualised trees are quite large, and as such may be difficult to see within the notebook. For reason, we would recommend viewing the specific .png files saved in this directory.

We will use the tree with a depth of 10 for this question. However, we should later come back to this point and test more specifically how much the model will improve when the depth is increased. We will review this in Question 5, where we will be looking at ways of improving our model.

3.2 Interpreting the model

Can you interpret the decision tree model? Discuss any knowledge you can gain in regard of the working of this model.

In a similar way to how we reviewed our linear regression model, we will need to make a comment on the model itself before moving on to the training stage. While we do not have coefficients here like we did in the linear regression model, we do have the `feature_importances` column, which is the tree's assumption as to which features will have the biggest impact on the prediction.

We will print them below and discuss in detail.

In [364…

```
importance_decision_tree = pd.DataFrame({'feature': X_train.columns, 'importance':tree_10.fea

with pd.option_context("display.max_rows", None, "display.max_columns", None):
    print(importance_decision_tree.sort_values('importance', ascending=False))
```

	feature	importance
42	Dublin	5.600649e-01
62	Wicklow	4.960076e-02
16	Dublin 4	4.358344e-02
18	Dublin 6	3.363979e-02
35	2021	2.780333e-02
40	Cork	2.364483e-02
6	Dublin 14	2.296324e-02
26	2012	2.253326e-02
27	2013	2.198718e-02
45	Kildare	2.173347e-02
0	Full Market Price	2.120164e-02
53	Meath	1.913929e-02
3	Dublin 11	1.738230e-02
29	2015	1.489840e-02
28	2014	1.427597e-02
33	2019	1.172874e-02
43	Galway	1.172412e-02
14	Dublin 24	9.780160e-03
32	2018	7.368106e-03
23	Not in Dublin	7.266684e-03
30	2016	7.120252e-03
31	2017	6.831697e-03
34	2020	4.449389e-03
41	Donegal	3.760702e-03
56	Roscommon	2.921141e-03
24	2010	2.336526e-03
5	Dublin 13	2.311257e-03
22	Dublin 9	2.238969e-03
19	Dublin 6w	1.659183e-03
11	Dublin 2	1.457615e-03
50	Longford	9.670297e-04
25	2011	8.510149e-04
61	Wexford	3.548194e-04
60	Westmeath	1.753231e-04
15	Dublin 3	1.297515e-04
21	Dublin 8	1.149109e-04
20	Dublin 7	7.917623e-07
7	Dublin 15	0.000000e+00
54	Monaghan	0.000000e+00
51	Louth	0.000000e+00
52	Mayo	0.000000e+00
57	Sligo	0.000000e+00
55	Offaly	0.000000e+00
13	Dublin 22	0.000000e+00
48	Leitrim	0.000000e+00
58	Tipperary	0.000000e+00
59	Waterford	0.000000e+00
49	Limerick	0.000000e+00
46	Kilkenny	0.000000e+00
47	Laois	0.000000e+00
1	Dublin 1	0.000000e+00
2	Dublin 10	0.000000e+00
44	Kerry	0.000000e+00
10	Dublin 18	0.000000e+00
17	Dublin 5	0.000000e+00
9	Dublin 17	0.000000e+00
4	Dublin 12	0.000000e+00
39	Clare	0.000000e+00
38	Cavan	0.000000e+00
12	Dublin 20	0.000000e+00
36	2022	0.000000e+00
8	Dublin 16	0.000000e+00
37	Carlow	0.000000e+00

Rather curiously, this paints a slightly different picture to the size of the coefficients of the linear

regression model. It is interesting to see that largely half of the features seem to have no importance at all to the decision tree, although that could be due to the depth of the tree that we created.

We can also very clearly see that the more important features are at the top of the tree diagram that we have seen drawn above, which makes sense, as the decision tree will begin by splitting the nodes at the mode important features.

Also of note is the fact that the dummy features are significantly more spread out here. For example, it is not surprising to see a large number of Dublin postcodes having a high importance, but it is perhaps more surprising to see a large number of counties being labelled as having no importance at all, with some of the year dummy variables ranking higher than them (which was not the case in our linear regression model).

3.3 Initial model metrics

Print the predicted target feature value for the first 10 training examples. Print a few regression evaluation measures computed on the full training set and discuss your findings so far.

We will start by performing the same process that we carried out with linear regression, namely printing a few predictions based on the training set, and getting the metrics for those predictions.

Because we have set up functions to do all the metric calculations for us, the amount of code that we have to write is significantly shorter when compared to the linear regression section.

In [365…
```
decision_tree_train_predictions = tree_10.predict(X_train)
pd.concat([y_train, pd.DataFrame(decision_tree_train_predictions, columns = ["Prediction"])],
```

Out[365]:

	Price	Prediction
0	145000.00	153513.348673
1	229074.00	153513.348673
2	220000.00	311887.337576
3	208500.00	391580.363193
4	500000.00	153513.348673
5	99000.00	153513.348673
6	810572.68	391580.363193
7	295000.00	206406.808596
8	243000.00	234580.567037
9	670000.00	391580.363193

In [366…
```
decision_tree_train_results = get_metrics(y_train, decision_tree_train_predictions)
print_metric_results("DECISION TREE TRAIN", decision_tree_train_results)
```

```
DECISION TREE TRAIN METRICS
***
RMSE: 160401.60973505685
MAE: 106271.71323477366
R2: 0.32941816478265873
***
```

A full evaluation comparing the models in detail will be carried out in the appropriate segment in the assignment, namely Question 5. In the meanwhile, what we can say about the metrics now is that they seem to once again be very high, with a much higher RMSE than MAE. We will see how this fares in comparison with the test set in the next section of the question.

3.4 Evaluation using test set and cross-validation

Evaluate the model using regression evaluation measures on the hold-out (30% examples) test set. Compare these results with the evaluation results obtained on the training (70% examples) dataset. Also compare these results with a cross-validated model (i.e., a new model trained and evaluated using cross-validation on the full dataset). You can use classic k-fold cross-validation or k repeated random train/test (70/30) splits. Compare the cross-validation metrics to those obtained on the single train/test split and discuss your findings.

Test Data

We will begin by making some predictions based on the test set, and printing some examples. Once again we will be using the functions that we have set up in advance in order to minimise the amount of code that we have to write.

In [367…
```
decision_tree_test_predictions = tree_10.predict(X_test)
```

In [368…
```
linear_regression_test_predictions = linear_regression.predict(X_test)

pd.concat([y_test, pd.DataFrame(linear_regression_test_predictions, columns = ["Prediction"])
```

Out[368]:

	Price	Prediction
0	425000.0	448256.0
1	400000.0	381184.0
2	210000.0	152064.0
3	35500.0	75520.0
4	177500.0	143104.0
5	120000.0	201216.0
6	171000.0	161792.0
7	122000.0	165632.0
8	445000.0	332032.0
9	311400.0	310784.0

In [369…
```
decision_tree_test_results = get_metrics(y_test, decision_tree_test_predictions)
print_metric_results("DECISION TREE TEST", decision_tree_test_results)
```

```
DECISION TREE TEST METRICS
***
RMSE: 166079.48481048775
MAE: 109010.2558135802
R2: 0.27728875134263886
***
```

Based on the above, we can see that the test set is performing worse than the the train set, with a 3.42% increase in RMSE, 2.5% increase in MAE, and 15% decrease in R2. In the grand scheme of things, apart from the R2 value, these changes are not particularly large, meaning that the model seems to be generalising pretty well, but the decrease between linear regression and decision tree seems to be more significant, which is probably due to the overfitting issue discussed before. As noted above, a decrease in performance on the test set is something that is to be expected.

Cross-Validation

As above, we will now also prepare cross-validation scores and take the average for ease of review.

In [370…
```
decision_tree_scores = get_k_fold_metrics(tree_10, X_full, y_full)
decision_tree_scores
```

Out[370]:
```
{'RMSE': array([169558.35195733, 164587.95509281, 173305.6843056 , 162330.24637289,
        161776.64162763]),
 'MAE': array([109120.96425905, 110313.24918149, 110944.08618889, 110435.24888176,
        106135.1189734 ]),
 'R2': array([0.25611868, 0.32927537, 0.25294032, 0.25621244, 0.28939481])}
```

In [371…
```
decision_tree_scores_average = {"RMSE": decision_tree_scores["RMSE"].mean(), "MAE": decision_
decision_tree_scores_average
```

Out[371]:
```
{'RMSE': 166311.77587125008,
 'MAE': 109389.73349691587,
 'R2': 0.2767883247379528}
```

We can see that the cross-validation scores are slightly worse than those of the train dataset, which is again something that is to be expected, given that we are taking random slices of data that the model would largely not have seen before. The differences are, however, almost negligible.

Before moving on, we will add the metrics related to the model into our global results dataframe, so that we can review them all later.

In [372…
```
df_decision_tree = get_full_metrics(tree_10, X_train, y_train, X_test, y_test, X_full, y_full
global_results = pd.concat([global_results, df_decision_tree], ignore_index=True, axis=0)
global_results = global_results.rename(index={0: "Linear Regression"})
global_results = global_results.rename(index={1: "Decision Tree"})
global_results
```

Out[372]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Tra (R
Linear Regression	160315.011772	159630.916296	160739.525422	105072.408566	103548.953582	105193.439280	0.3301
Decision Tree	160401.609735	166079.484810	166311.775871	106271.713235	109010.255814	109389.733497	0.3294

Question 4: Predictive Modelling and Evaluation (Random Forest)

The final model that we have been asked to implement is the Random Forest model. Random Forest works by generating several Decision Trees in the exact same way as described in the Decision Tree model listed above. However, this time, the model introduces a level of randomness when generating such trees.

While Decision Tree would always choose the most important feature to split on for the decision nodes, Random Forest will choose a different feature in each tree within a certain subset of features. Once complete, the model will merge the trees and generate a consensus, which hopefully will provide us a better answer to just using one single Decision Tree.

4.1 Training the model

On the training set, train a random forest model to predict the target feature, using the descriptive features selected in exercise (1) above.

We will begin by training the model using the `RandomForestRegressor` object. As usual, we will switch on the random_state to ensure that we can replicate the results. As we can see, `sklearn` allows us to select the number of trees that we have in our forest with the `n_estimators` default argument.

We will choose a somewhat large number for this such as 100. As with the decision tree depth, we can consider what the best number to use for this will be later on in the assignment (in Question 5).

This model is useful because of the consensus mechanism being implemented. If many different iterations of decision trees are used, then the likelihood is that we will get a more generalised prediction that ultimately be more accurate. However, it is possible that the likelihood of overfitting will also increase, given that the decision trees ultimately form the basis of the model.

One thing I thought was worthwhile looking into was what the maximum depth of the tree in our model will be. Having had a look at the documentation (link here), it looks like if no maximum depth is specfied, then the trees will continue to be expanded until either the leaf nodes are "pure" or until we have the less than the minimum number of samples reqauired to split a node (which by default is set to two). This seems like a reasonable approach, and as such it was decided to leave the `max_depth` variable to its default state.

Because of the number of decision trees in our model, we will not make an attempt to visualise all of these. We did not consider this necessary given that we have already visualised individual decision trees, and as such are aware that a random forest model will consist of many of those types of decision trees.

In [373…
```
random_forest = ensemble.RandomForestRegressor(n_estimators=100, max_features='auto', oob_sco
random_forest.fit(X_train, y_train)
```

Out[373]:
```
RandomForestRegressor(oob_score=True, random_state=1)
```

4.2 Interpreting the model

Can you interpret the random forest model? Discuss any knowledge you can gain in regard of the working of this model.

Similarly to the decision tree mode, we can print out the `feature_importances` attribute of our random forest model. We will do so below.

In [374…
```
importance_random_forest = pd.DataFrame({'feature': X_train.columns, 'importance':random_fore
```

```
with pd.option_context("display.max_rows", None, "display.max_columns", None):
    print(importance_random_forest.sort_values('importance', ascending=False))
```

	feature	importance
42	Dublin	0.424012
62	Wicklow	0.038526
16	Dublin 4	0.034515
35	2021	0.032822
0	Full Market Price	0.031175
18	Dublin 6	0.024027
26	2012	0.023110
33	2019	0.021524
6	Dublin 14	0.019038
27	2013	0.018425
32	2018	0.017987
31	2017	0.017950
28	2014	0.017910
45	Kildare	0.017539
40	Cork	0.016829
30	2016	0.014979
29	2015	0.014679
34	2020	0.014647
23	Not in Dublin	0.014588
53	Meath	0.014557
24	2010	0.013441
3	Dublin 11	0.012561
25	2011	0.010827
43	Galway	0.009234
10	Dublin 18	0.007794
15	Dublin 3	0.007581
21	Dublin 8	0.006586
19	Dublin 6w	0.006011
5	Dublin 13	0.005969
14	Dublin 24	0.005537
11	Dublin 2	0.005381
13	Dublin 22	0.005373
1	Dublin 1	0.005357
20	Dublin 7	0.005131
8	Dublin 16	0.004932
7	Dublin 15	0.004400
22	Dublin 9	0.004313
59	Waterford	0.003041
41	Donegal	0.002685
46	Kilkenny	0.002635
52	Mayo	0.002625
58	Tipperary	0.002587
56	Roscommon	0.002515
2	Dublin 10	0.002488
61	Wexford	0.002482
49	Limerick	0.002194
50	Longford	0.002187
54	Monaghan	0.002155
17	Dublin 5	0.002029
37	Carlow	0.002023
4	Dublin 12	0.001965
48	Leitrim	0.001922
51	Louth	0.001827
44	Kerry	0.001808
9	Dublin 17	0.001750
60	Westmeath	0.001602
39	Clare	0.001551
57	Sligo	0.001465
38	Cavan	0.001353
12	Dublin 20	0.001254
55	Offaly	0.000960
47	Laois	0.000905
36	2022	0.000726

From the above, we can see that the positioning of the features is roughly in line with what we saw for

our decision tree model. While there are some changes in order (probably due to the consensus algorithm) the features order is very similar. The exception here, however, is that **all** of our features now have an importance score larger than 0, which is probably due to the way the default `max_depth` argument works for random forests in sklearn (as discussed above).

Our initial predictions, then, are that random forest seems like a slightly more fine-tuned version of decision tree, and as such, should perform with similar metric values, although maybe a little bit improved in comparison.

4.3 Initial model metrics

Print the predicted target feature value for the first 10 training examples. Print a few regression evaluation measures computed on the full training set and discuss your findings so far.

We will start by performing the same process that we carried out with linear regression and decision tree, namely printing a few predictions based on the training set, and getting the metrics for those predictions.

In [375…
```
random_forest_train_predictions = random_forest.predict(X_train)
pd.concat([y_train, pd.DataFrame(random_forest_train_predictions, columns = ["Prediction"])],
```

Out[375]:

	Price	Prediction
0	145000.00	121862.645147
1	229074.00	218013.466300
2	220000.00	309552.821789
3	208500.00	425550.176199
4	500000.00	225020.683985
5	99000.00	161297.798043
6	810572.68	425550.176199
7	295000.00	206093.575802
8	243000.00	252095.516054
9	670000.00	334121.028227

In [376…
```
random_forest_train_results = get_metrics(y_train, random_forest_train_predictions)
print_metric_results("RANDOM FOREST TRAIN", random_forest_train_results)
```

```
RANDOM FOREST TRAIN METRICS
***
RMSE: 152388.2094271595
MAE: 99190.38931631202
R2: 0.3947468304141314
***
```

A full evaluation comparing the models in detail will be carried out in the appropriate segment in the assignment, namely Question 5. In the meanwhile, what we can say about the metrics now is that they seem to once again be very high, with a much higher RMSE than MAE, although this is the first time that we have seen a model with an MAE lower than 100k. It is important not to make any assumptions here,

however, given the fact that the model could perform worse when it comes to the test set and also during cross-validation, due to the overfitting issue.

4.4 Evaluation using test set and cross-validation

Evaluate the model using regression evaluation measures on the hold-out (30% examples) test set. Compare these results with the evaluation results obtained on the training (70% examples) dataset. Also compare these results with a cross-validated model (i.e., a new model trained and evaluated using cross-validation on the full dataset). You can use classic k-fold cross-validation or k repeated random train/test (70/30) splits. Compare the cross-validation metrics to those obtained on the single train/test split and to the random forest out-of-sample error and discuss your findings.

Test Data

We will begin by making some predictions based on the test set, and printing some examples.

In [377…
```
random_forest_test_predictions = random_forest.predict(X_test)
pd.concat([y_test, pd.DataFrame(random_forest_test_predictions, columns = ["Prediction"])], a
```

Out[377]:

	Price	Prediction
0	425000.0	519054.182540
1	400000.0	434271.845135
2	210000.0	190799.724475
3	35500.0	99642.830600
4	177500.0	110692.843382
5	120000.0	201801.585364
6	171000.0	181402.912338
7	122000.0	161402.564069
8	445000.0	295888.903054
9	311400.0	309552.821789

In [378…
```
random_forest_test_results = get_metrics(y_test, random_forest_test_predictions)
print_metric_results("RANDOM FOREST TEST", random_forest_test_results)
```

```
RANDOM FOREST TEST METRICS
***
RMSE: 164287.33747704528
MAE: 107299.85921714776
R2: 0.2928020078260821
***
```

As expected, the model performs worse when compared to the train set. In fact, it would seem that the increase in error is larger to that of the decision tree, with the RMSE and MAE increasing by approximately 10k, and the R2 decreasing by 0.1. This is inline with our initial prediction that the overfitting will be more of an issue with random forest.

We should also now consider the cross-validation score.

In [379…
```
random_forest_scores = get_k_fold_metrics(random_forest, X_full, y_full)
```

```
random_forest_scores
```

Out[379]:
```
{'RMSE': array([167218.17544001, 162859.39970119, 169249.36032791, 160472.38898174,
        160236.20473607]),
 'MAE': array([107517.82767731, 108390.02195742, 107806.88727621, 109154.69268646,
        104404.01541018]),
 'R2': array([0.27651048, 0.34328972, 0.28750183, 0.2731402 , 0.30286314])}
```

In [380…
```
random_forest_scores_average = {"RMSE": random_forest_scores["RMSE"].mean(), "MAE": random_fo
random_forest_scores_average
```

Out[380]:
```
{'RMSE': 164007.1058373848,
 'MAE': 107454.68900151702,
 'R2': 0.2966610760199666}
```

We can see that the cross-validation scores are slightly worse than those of the train dataset, which is again something that is to be expected, given that we are taking random slices of data that the model would largely not have seen before. The decrease in performance between the test set and the cross-validation scores is in a similar range to that of the decision tree model.

Before moving on, we will add the metrics related to the model into our global results dataframe, so that we can review them all later.

In [381…
```
df_random_forest = get_full_metrics(random_forest, X_train, y_train, X_test, y_test, X_full,
global_results = pd.concat([global_results, df_random_forest], ignore_index=True, axis=0)
global_results = global_results.rename(index={0: "Linear Regression"})
global_results = global_results.rename(index={1: "Decision Tree"})
global_results = global_results.rename(index={2: "Random Forest"})
global_results
```

Out[381]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Tra (F
Linear Regression	160315.011772	159630.916296	160739.525422	105072.408566	103548.953582	105193.439280	0.3301
Decision Tree	160401.609735	166079.484810	166311.775871	106271.713235	109010.255814	109389.733497	0.3294
Random Forest	152388.209427	164287.337477	164007.105837	99190.389316	107299.859217	107454.689002	0.3947

Before proceeding on to the next step, it is also useful to test the "Out of the Bag" score for this random forest model, or the "out of sample" score. This is a score that is calculated on data points in the model that are not a part of the specific sample, making them a very strong validation tool.

In [382…
```
random_forest.oob_score_
```

Out[382]:
```
0.2847986071738722
```

As we can see from the above, the "out of the bag" score is just slightly worse than the K-Fold equivalent, which is roughly in line with what we would have expected.

Question 5: Improving Predictive Models

In this section, we will compare the models that we have created so far, and also attempt to improve on them, before testing on brand new data provided in the assignment.

5.1 Best model and average price

Which model of the ones trained above performs better at predicting the target feature? Is it more accurate than a simple model that always predicts the average price computed from the training set, for the same year as the test example? Justify your answers.

Comparing the models

In this part of the question, we are first asked to compare the models that we have so far. We will begin by printing our global results dataframe, so that we can get a better look at all the results obtained to date.

In [383…
```
global_results
```

Out[383]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Tr (R
Linear Regression	160315.011772	159630.916296	160739.525422	105072.408566	103548.953582	105193.439280	0.3301
Decision Tree	160401.609735	166079.484810	166311.775871	106271.713235	109010.255814	109389.733497	0.3294
Random Forest	152388.209427	164287.337477	164007.105837	99190.389316	107299.859217	107454.689002	0.3947

After reviewing our global results, some interesting patterns begin to emerge. As we can see, the performance of each model is roughly the same, with some minor variation of 2-3%, with the decision tree model largely performing worse than linear regression by a little bit. The variance in error is not particularly dramatic.

The errors that we are getting are quite large, however, most likely due to the fact that the variance of prices in the housing market is very large, which could lead to issues with the models.

Of interest is the fact that random forest seems to be performing better than both linear regression and decision tree when it comes to the train set. On the test set and during cross-evaluation, it performs worse than linear regression, however, although still better than decision tree. This is probably due to the fact that overfitting is more of an issue with the random forest model when compared to the other two models.

As discussed above, we can also very clearly see overfitting happening in each of the models, with decision tree and random forest seeming particularly susceptible to this issue.

A high cross-evaluation score is more important to us than high metrics on the train set, as it shows how the model adapts to new data. For this reason, we can say that as it stands, linear regression seems to perform best out of the three models, followed by random forest, and finally ending with decision tree. We should be careful when saying that it is outright the best, however, given the fact that valid outliers could be affecting the performance of this model.

It should be stressed once again, however, that the difference in performance between all these models is quite small.

Average price

We have been asked to check if our models perform better than a simple model that just takes the mean of the price given a particular year. This type of model will, in effect, constitute a single horizontal line. It represents a baseline prediction where we simply just take the average of all the prices in order to make our prediction (which in theory should be pretty inaccurate). It is useful to check our models against this just to ensure that our performance is at the very least not worse than baseline. This is not dissimilar to the way the R2 is calculated.

Thankfully, `sklearn` provides us with a very useful `DummyRegressor` object that achieves just that. We will create some dummy variables representing the years of the property sales, and set them in our `X` descriptive features dataframe. The `y` dataframe will once again consist of the target features.

We will then train our DummyRegressor with the "strategy" argument set to "mean" (which is what we have been asked to do in the assignment sheet) and print the metrics that arise as a result.

Finally, we will check if the resulting metrics are better or worse than any of the values in our global_results dataframe. If they are worse, then we know that our models perform better than baseline.

While it does not really matter all that much given that our predictions should be the same across the board, we will be using the "full" dataset for the purposes of this exercise.

In [384…
```
years = []

# Loop through years and them to list
for year in range(2010, 2023):
    # Ensure that we are parsing years to string as the column names in our dataframe are str
    years.append(str(year))

# Create X dataframe by selecting just the years
X_full_dummy = X_full[years]

dummy_regressor = DummyRegressor(strategy="mean")
dummy_regressor.fit(X_full_dummy, y_full)
```

Out[384]:
```
DummyRegressor()
```

Before moving on to generating the metrics, let's have a look at the comparison between the price and prediction in the same way that we did for the other models, just to make sure that that predictions that we are getting are in line with expectations.

In [385…
```
dummy_regressor_predictions = dummy_regressor.predict(X_full_dummy)

pd.concat([y_full, pd.DataFrame(dummy_regressor_predictions, columns = ["Prediction"])], axis
```

Out[385]:

	Price	Prediction
0	370044.05	246228.781768
1	480000.00	246228.781768
2	194000.00	246228.781768
3	275000.00	246228.781768
4	75000.00	246228.781768
5	132000.00	246228.781768
6	105000.00	246228.781768
7	242290.75	246228.781768
8	295000.00	246228.781768
9	54000.00	246228.781768

As expected from the above, every prediction that we are making is giving us the same value, which is the mean of the target feature. Let us know proceed with generating the metrics for this "model".

In [386…

```
dummy_regressor_results = get_metrics(y_full, dummy_regressor_predictions)
print_metric_results("DUMMY REGRESSION TEST", dummy_regressor_results)
```

```
DUMMY REGRESSION TEST METRICS
***
RMSE: 195721.62470064824
MAE: 135389.63420693038
R2: 0.0
***
```

As we can see, the errors here are quite large. Of particular interest is the R2 value, which is set to 0 - this means that there is no point in our dataset that passes through the regression line. We would expect this kind of low accuracy from our baseline performance.

We will now print out our global results dataframe again just so that we can have the data in front of us to compare to the dummy regressor.

In [387…

```
global_results
```

Out[387]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Tra (R
Linear Regression	160315.011772	159630.916296	160739.525422	105072.408566	103548.953582	105193.439280	0.3301
Decision Tree	160401.609735	166079.484810	166311.775871	106271.713235	109010.255814	109389.733497	0.3294
Random Forest	152388.209427	164287.337477	164007.105837	99190.389316	107299.859217	107454.689002	0.3947

Looking through this dataframe, we can see that there are no metrics that we can see that are performing worse than the baseline metrics. It is perhaps a little bit worrying that the RMSE and MAE are a little closer to the baseline performance than we would like, but nonetheless it would be fair to say that all of our models are performing better than baseline.

Before moving on, it was not entirely clear to me whether or not it was expected that we would use the

dummy regressor, or just use the year of the test dataset provided to us (which is 2022). Just in case, I took the mean of all the prices of 2022 and ran the metrics on that value.

In [388…
```
predictions_2022 = df_full.loc[df_full["Year"]==2022, "Price"].mean()
predictions_2022_series = pd.Series(predictions_2022).repeat(10109)

predictions_2022_results = get_metrics(y_full, predictions_2022_series)
predictions_2022_results
```

Out[388]:
```
{'RMSE': 199190.21305678796,
 'MAE': 147325.44543476115,
 'R2': -0.035758170402203415}
```

As we can see from the above, this performs even worse than the dummy regressor, so the performance of our model is better than baseline.

5.2 Improving the models

Summarise your understanding of the problem and of your predictive modeling results so far. Can you think of any new ideas to improve the best model so far (e.g., by using further data prep such as: feature selection, feature re-scaling, creating new features, combining models, or using other knowledge)? Please show how your ideas actually work in practice, by training and evaluating your proposed models. Summarise your findings so far.

To summarise the results so far, we have three models, each of which is performing better than baseline, but that are perhaps performing worse than we would like, given the massive size of the errors. It is entirely possible that this is unavoidable due to the sheer variation of prices in the target feature, but we should definitely consider how we can improve on the results before moving on to testing the new "test" dataset.

One point to note is that given that these are housing prices, which have a wide range of values anyway (i.e. depending on location a property can cost anywhere from 200k to several million) we are going to have fluctuations in price that will be hard to predict. Therefore, given the size of the values, an error of, for example, 100k may not be as extreme as it might initially seem.

The assignment states that we should think of new ideas "to improve the best model so far", implying that we should try and improve just *one* model. However, due to the very minor variation in performance between all of these models, it is entirely possible, for example, that an improved version of random forest, for example, may perform better than an improved version of linear regression, or vice versa.

For this reason, I will attempt to improve all three models, and check the new results against the unoptimised models.

We should first consider some of the suggestions made in the assignment itself.

- **Feature selection**: We should try this.
- **Feature re-scaling**: We should try this.
- **Creating new features**: None of the features that we created as a result of Assignment 1 performed particularly well, and some had to be dropped altogether. For this reason, we will not be attempting to create any new features.

- **Combining models**: While it probably does not make sense to combine our existing models, we can try and slightly modify our existing models (such as convert linear regression to polynomial regression).
- **Using other knowledge**: We should try this.

Having done some research and a careful review of our models so far, it was decided that there are two categories of prediction improvement:

- **Data Improvement**: This category relates to improvements to the data that is used to train the models. This category applies to **all models**. Proposed improvements are: using all features, data normalisation.
- **Model Improvement**: This category relates to improvements to the way the models are trained. This category applies to **specific models**. Proposed improvements are: applying polynomial features for linear regression, using a different depth for decision tree, and using a different number of estimators for random forest.

We will explain our reasoning for each of these improvements in the sections below, and will attempt to train new models using the proposed improvements so that we have concrete evidence as to whether or not these strategies helped with predicting the target outcome.

(a) Data Improvement

(i) Using Subset of Features

In our initial run of training various models, we used just the features that we had considered to have high correlation. It would be helpful, however, to test our models with different subsets of features, to see if this has an effect on performance (even if the correlation of those features is not, for example, particularly strong). It is possible that this will improve our accuracy.

In order to do this, we will make another copy of df, and make some train / test splits, but this time, when we generate our X and y dataframes for train and test, we will switch on the "all" default argument, which will ensure that all the features in our dataframe will be included.

In [389…
```
df_all_features = df.copy()

df_all_features_train, df_all_features_test = train_test_split(df_all_features, test_size=0.3
X_all_features_train, y_all_features_train = get_X_and_y(df_all_features_train, all)
X_all_features_test, y_all_features_test = get_X_and_y(df_all_features_test, all)
X_all_features_full, y_all_features_full = get_X_and_y(df_all_features, all)
```

We will proceed by training our models on the new X and y parameters, and then acquire the various metrics required to measure the performance. Please note that these steps is largely identical to the ones taken earlier in this assignment, and as such we will not go into detail on the explanations of how this is done in order to avoid repetition.

In [390…
```
linear_regression_all_features = LinearRegression().fit(X_all_features_train, y_all_features_

tree_all_features_depth = tree.DecisionTreeRegressor(max_depth = 10, random_state = 10)
tree_all_features = tree_all_features_depth.fit(X_all_features_train, y_all_features_train)

random_forest_all_features = ensemble.RandomForestRegressor(n_estimators=100, max_features='a
random_forest_all_features.fit(X_all_features_train, y_all_features_train)
```

Out[390]:
```
RandomForestRegressor(oob_score=True, random_state=1)
```

In [391…
```
df_linear_regression_all_features = get_full_metrics(linear_regression_all_features, X_all_fe
df_decision_tree_all_features = get_full_metrics(tree_all_features, X_all_features_train, y_a
df_random_forest_all_features = get_full_metrics(random_forest_all_features, X_all_features_t
```

We can add these to a new dataframe for ease of comparison.

In [392…
```
global_results_all_features = pd.concat([df_linear_regression_all_features, df_decision_tree_
global_results_all_features = global_results_all_features.rename(index={0: "Linear Regression
global_results_all_features = global_results_all_features.rename(index={1: "Decision Tree"})
global_results_all_features = global_results_all_features.rename(index={2: "Random Forest"})
global_results_all_features
```

Out[392]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Tra (R
Linear Regression	160235.186389	159663.464526	161009.730392	105091.305209	103506.166902	105365.969333	0.3308
Decision Tree	156996.608407	167955.585980	168789.306278	103257.685133	109091.588056	110621.745199	0.3575
Random Forest	134373.928744	172276.080839	172018.452316	84115.238382	112615.096665	112380.528883	0.5293

We will now print out our original results dataframe so that we can compare our two dataframes more easily.

In [393…
```
global_results
```

Out[393]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Tra (R
Linear Regression	160315.011772	159630.916296	160739.525422	105072.408566	103548.953582	105193.439280	0.3301
Decision Tree	160401.609735	166079.484810	166311.775871	106271.713235	109010.255814	109389.733497	0.3294
Random Forest	152388.209427	164287.337477	164007.105837	99190.389316	107299.859217	107454.689002	0.3947

From the above, we can see that while the models that use all features perform slightly better when it comes to the train dataset, they end up actually performing worse when it comes to the test and cross-evaluation metrics. The reason why the performance is better on the train set is most likely due to overfitting, as the model internalises the additional data and is able to predict the values with much more success.

However, due to the fact that we are are largely looking to use this model on unseen data, it makes more sense to judge this approach by looking at test and k-fold, where the performance is worse, which means that we should not take this subset of features for our models.

Another approach we can take is by taking a single feature. We saw in the first question of this assignment that the Year seemed to have a somewhat linear relationship with the target feature - so

perhaps using just the Year as our descriptive feature will yield a more accurate result. We will now repeat this entire process using the Year as the descriptive feature **only**.

In [394…]
```
df_year = df.copy()

df_year_train, df_year_test = train_test_split(df_year, test_size=0.3, random_state=1)
X_year_train, y_year_train = get_X_and_y(df_year, all)
X_year_test, y_year_test = get_X_and_y(df_year_test, all)
X_year_full, y_year_full = get_X_and_y(df_year, all)
```

In [395…]
```
years = []

for year in range(2010, 2023):
    years.append(str(year))

X_year_train = X_year_train[years]
X_year_test = X_year_test[years]
X_year_full = X_year_full[years]
```

In [396…]
```
linear_regression_year = LinearRegression().fit(X_year_train, y_year_train)
tree_year_depth = tree.DecisionTreeRegressor(max_depth = 10, random_state = 10)
tree_year = tree_year_depth.fit(X_year_train, y_year_train)
random_forest_year = ensemble.RandomForestRegressor(n_estimators=100, max_features='auto', oo
random_forest_year.fit(X_year_train, y_year_train)
```

Out[396]:
```
RandomForestRegressor(oob_score=True, random_state=1)
```

In [397…]
```
df_linear_regression_year = get_full_metrics(linear_regression_year, X_year_train, y_year_tra
df_decision_tree_year = get_full_metrics(tree_year, X_year_train, y_year_train, X_year_test,
df_random_forest_year = get_full_metrics(random_forest_year, X_year_train, y_year_train, X_ye
```

In [398…]
```
global_results_year = pd.concat([df_linear_regression_year, df_decision_tree_year, df_random_
global_results_year = global_results_year.rename(index={0: "Linear Regression"})
global_results_year = global_results_year.rename(index={1: "Decision Tree"})
global_results_year = global_results_year.rename(index={2: "Random Forest"})
global_results_year
```

Out[398]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Tra (F
Linear Regression	192360.528760	192244.350486	192531.012260	131162.249867	130762.378885	131313.828720	0.0340
Decision Tree	192370.970181	192275.058599	192540.109444	131198.172876	130826.286674	131310.919700	0.0339
Random Forest	192361.316348	192238.161394	192524.842618	131189.366843	130784.685497	131259.831986	0.0340

As we can see, while all the models now perform very similarly, they are also performing markedly worse than both previous subsets of features. Of particular concern is the R2 value which is very close to zero, which is worrying. It looks like using just the high correlation features produces the optimal result, and for this reason we will continue to use that subset for the remainder of this assignment.

Note that we did not choose to use other subsets of features due to the fact that they had lower correlation than our optimal subset - for this reason it was likely that they would have performed worse. If there was additional time left, we could have tested this just in case, but given time constraints and the fact that it was unlikely that we would receive any improvement, it was decided to focus on other types of improvements, as more particularly seen below.

(ii) Feature normalisation

In our dataset, as determined as part of Assignment 1, we have established that there is a very strong variation between our prices. Predictive models generally do not deal well with large values that have strong variation between them. For this reason, an approach is often taken called "data normalisation", where each target feature is converted into a value between a small set of values, such as 0 and 1, or -1 and 1.

We will try doing something similar here to see if it helps improve our predictions.

It should be noted that while sklearn has a built-in normalisation option, reversing the normalisation does not work where we try to reverse values that have not been explicitly scaled by the scaler. What I mean by this is that while we will be able to normalise the target feature, train the model using the new values and make predictions, we will not be able to convert the predictions back into the regular values, because the conversion was applied to the target feature only, not to the predictions.

For this reason, I created a standalone scaler class which will use the mean and standard deviation to compute the scaled value, and will also be able to undo this scaling.

In [399…
```
class Scaler():
    """Class responsible for creating a scaler"""

    def __init__(self, y):
        """Constructor"""
        # Note that we are saving the mean and standard deviation for later transformation
        self.y = y
        self.y_mean = y.mean()
        self.y_std = y.std()

    def scale(self):
        """Function that scales the values"""
        return (self.y - self.y_mean) / self.y_std

    def unscale(self, y_new):
        """Function that unscales the values"""
        return (y_new * self.y_std) + self.y_mean
```

We will start by creating a scaler object and using it to scale our y values. We then proceed by training all of our models as usual, using the steps described over the course of this assignment.

In [400…
```
# Creating the scaler object
linear_regression_scaler = Scaler(y_train)

# Scaling the y train values
y_train_normalised = linear_regression_scaler.scale()
linear_regression_norm = LinearRegression().fit(X_train, y_train_normalised)
```

In [401…
```
tree_depth_10 = tree.DecisionTreeRegressor(max_depth = 10, random_state = 10)
tree_norm = tree_depth_10.fit(X_train, y_train_normalised)
```

In [402…
```
random_forest_norm = ensemble.RandomForestRegressor(n_estimators=100, max_features='auto', oo
random_forest_norm.fit(X_train, y_train_normalised)
```

Out[402]:
```
RandomForestRegressor(oob_score=True, random_state=1)
```

Unfortunately, due to the fact that we have to scale and unscale our values now, we will not be able to use the same `get_metrics` function as before. We will, however, just re-work the function slightly so

that we are scaling and unscaling our values as appropriate.

Please note that to save time we will initially just get the metrics for the train and test data sets. If these perform better than the initial metrics, then we can do the cross-evaluation scores as well. If they perform worse, then there is no need to proceed as cross-evaluation scores perform the worst out of all of our metrics.

In [403…]
```
def get_full_metrics_normalised(model, scaler):
    """Creating metrics for normalised data"""
    df_results = pd.DataFrame()

    # Predicting the values based on the inputted model.
    # Note that we only expect to use this function on the regular dataset,
    # and as such we can hardcode the X_train etc. values. We will unscale
    # our predictions after we make them (because the model will have been
    # trained on normalised data)
    train_predictions = scaler.unscale(model.predict(X_train))
    train_results = get_metrics(y_train, train_predictions)
    for metric in train_results.keys():
        df_results[f"Train ({metric})"] = pd.Series(train_results[metric])

    test_predictions = scaler.unscale(model.predict(X_test))
    test_results = get_metrics(y_test, test_predictions)
    for metric in test_results.keys():
        df_results[f"Test ({metric})"] = pd.Series(test_results[metric])

    return df_results

# Get the regression values
df_linear_regression = get_full_metrics_normalised(linear_regression_norm, linear_regression_
df_tree_norm = get_full_metrics_normalised(tree_norm, linear_regression_scaler)
df_forest_norm = get_full_metrics_normalised(random_forest_norm, linear_regression_scaler)
```

In [404…]
```
df_linear_regression
```

Out[404]:

	Train (RMSE)	Train (MAE)	Train (R2)	Test (RMSE)	Test (MAE)	Test (R2)
0	160322.189405	105006.79121	0.330082	159656.939051	103481.192162	0.332105

In [405…]
```
df_tree_norm
```

Out[405]:

	Train (RMSE)	Train (MAE)	Train (R2)	Test (RMSE)	Test (MAE)	Test (R2)
0	160401.609735	106271.713235	0.329418	166079.48481	109010.255814	0.277289

In [406…]
```
df_forest_norm
```

Out[406]:

	Train (RMSE)	Train (MAE)	Train (R2)	Test (RMSE)	Test (MAE)	Test (R2)
0	152386.717697	99190.447727	0.394759	164305.691676	107303.616659	0.292644

We will reprint our global results dataframe so we can compare it against the above results.

In [407…]
```
global_results
```

Out[407]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Tra (F
Linear Regression	160315.011772	159630.916296	160739.525422	105072.408566	103548.953582	105193.439280	0.3301
Decision Tree	160401.609735	166079.484810	166311.775871	106271.713235	109010.255814	109389.733497	0.3294
Random Forest	152388.209427	164287.337477	164007.105837	99190.389316	107299.859217	107454.689002	0.3947

As we can see, the performance is largely identical, and is in fact even slightly worse for some of the models. The only model that performs a little better is random forest, but the performance improvement is so marginal (i.e. a 0.01% improvement in the "Test" dataset) that it does not make sense to put in extra effort in using normalisation given that we are unlikely to get that much benefit from it.

(b) Model Improvement

(i) Polynomial regression

Polynomial regression is a type of model very similar to linear regression, in the sense that a line is plotted to show the relationship between the X and y values. However, the difference is that polynomial regression specifically plots a **curve** as opposed to a line, meaning that some of the elements of the formula for the curve will be squared or cubed, etc.

The reason for why we considered this for our model was because we noted that the "Year" subset feature of our dataset showed a kind of curve, with a dip in the 2012-2013 region. Therefore, it is possible that the relationship between the features may be better represented as a curve.

In order to create a polynomial regression model, we must create polynomial features out of our "X" features using the `PolynomialFeatures` object. We will then proceed to train the model using the polynomial features and attempt to get the metrics as usual.

In [408…
```
from sklearn.preprocessing import PolynomialFeatures
```

In [409…
```
# Create polynomial features
polynomial_features_train = PolynomialFeatures()

# Fit the X_train to the polynomial features
X_train_poly = polynomial_features_train.fit_transform(X_train)
X_train_poly
```

Out[409]:
```
array([[1., 1., 0., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 0., 0.],
       ...,
       [1., 1., 0., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 0., 0.]])
```

In [410…
```
# Here we will train our model using the polynomial features
linear_regression_poly = LinearRegression()
polynomial_regression_train = linear_regression_poly.fit(X_train_poly, y_train)
poly_train_predictions = polynomial_regression_train.predict(X_train_poly)
poly_train_results = get_metrics(y_train, poly_train_predictions)
print_metric_results("POLYNOMIAL REGRESSION TRAIN", poly_train_results)
```

```
POLYNOMIAL REGRESSION TRAIN METRICS
***
RMSE: 154114.0579624201
MAE: 101725.77542625219
R2: 0.3809598007456946
***
```

In [411…
```
polynomial_features_test = PolynomialFeatures()
X_test_poly = polynomial_features_test.fit_transform(X_test)
X_test_poly
```

Out[411]:
```
array([[1., 1., 0., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 0., 1.],
       [1., 1., 0., ..., 0., 0., 0.],
       ...,
       [1., 1., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 0., 0.]])
```

In [412…
```
# Here we will produce the
polynomial_regression_test = linear_regression_poly.fit(X_test_poly, y_test)
poly_test_predictions = polynomial_regression_test.predict(X_test_poly)
poly_test_results = get_metrics(y_test, poly_test_predictions)
print_metric_results("POLYNOMIAL REGRESSION TEST", poly_test_results)
```

```
POLYNOMIAL REGRESSION TEST METRICS
***
RMSE: 145376.41804296777
MAE: 93271.02422920309
R2: 0.4462410424203854
***
```

In [413…
```
polynomial_features_full = PolynomialFeatures()
X_full_poly = polynomial_features_full.fit_transform(X_full)
X_full_poly
```

Out[413]:
```
array([[1., 1., 0., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 0., 0.],
       ...,
       [1., 1., 0., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 0., 0.]])
```

In [414…
```
polynomial_regression_scores = get_k_fold_metrics(linear_regression_poly, X_full_poly, y_full
polynomial_regression_scores
```

Out[414]:
```
{'RMSE': array([2.63811961e+16, 1.61301772e+16, 1.51953788e+16, 5.51419848e+15,
        2.96118277e+16]),
 'MAE': array([1.38357038e+15, 8.78050675e+14, 6.04378642e+14, 2.70788293e+14,
        1.38907985e+15]),
 'R2': array([-1.80075341e+22, -6.44208471e+21, -5.74318414e+21, -8.58251954e+20,
        -2.38082334e+22])}
```

In [415…
```
polynomial_regression_scores_average = {"RMSE": polynomial_regression_scores["RMSE"].mean(),
polynomial_regression_scores_average
```

Out[415]:
```
{'RMSE': 1.856655565293619e+16,
 'MAE': 905173567075419.2,
 'R2': -1.0971857666585892e+22}
```

As we can see from the above, while it looks like the train and test datasets are performing very well, unfortunately it seems that the cross-evaluation scores that we have received are absolutely catastrophic, with enormous errors. Of particular note is the fact that our R2 metric is negative, suggesting that the model is performing worse than baseline.

It should be noted, however, that the polynomial relationship that we spotted in our initial analysis was related specifically to the "Year" feature. For this reason, we will attempt to create a polynomial regression model using *just* the Year as the descriptive feature, to see if this gives us better performance.

In [416…
```
polynomial_features_train_year = PolynomialFeatures()
X_train_poly_year = polynomial_features_train_year.fit_transform(X_year_train)
linear_regression_poly_year = LinearRegression()
polynomial_regression_train_year = linear_regression_poly_year.fit(X_train_poly_year, y_year_
poly_train_predictions_year = polynomial_regression_train_year.predict(X_train_poly_year)
poly_train_results_year = get_metrics(y_year_train, poly_train_predictions_year)
print_metric_results("POLYNOMIAL REGRESSION TRAIN (YEAR))", poly_train_results_year)
```

```
POLYNOMIAL REGRESSION TRAIN (YEAR)) METRICS
***
RMSE: 192404.44120154303
MAE: 131136.00779602333
R2: 0.03360970357403559
***
```

In [417…
```
polynomial_features_test_year = PolynomialFeatures()
X_test_poly_year = polynomial_features_test_year.fit_transform(X_year_test)
polynomial_regression_test_year = linear_regression_poly_year.fit(X_test_poly_year, y_year_te
poly_test_predictions_year = polynomial_regression_test_year.predict(X_test_poly_year)
poly_test_results_year = get_metrics(y_year_test, poly_test_predictions_year)
print_metric_results("POLYNOMIAL REGRESSION TEST (YEAR))", poly_test_results_year)
```

```
POLYNOMIAL REGRESSION TEST (YEAR)) METRICS
***
RMSE: 192267.01290478406
MAE: 130581.227490158
R2: 0.03140461693711882
***
```

In [418…
```
polynomial_features_full_year = PolynomialFeatures()
X_full_poly_year = polynomial_features_full_year.fit_transform(X_year_full)
X_full_poly_year
```

Out[418]:
```
array([[1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 0., 0.],
       ...,
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.]])
```

In [419…
```
polynomial_regression_scores_year = get_k_fold_metrics(linear_regression_poly_year, X_full_po
polynomial_regression_scores_year
```

Out[419]:
```
{'RMSE': array([194603.44000297, 196377.81253488, 198101.81901508, 185476.44581784,
        188677.43760039]),
 'MAE': array([129822.84400645, 135119.90858182, 133659.33536784, 129067.05665038,
        129740.80525269]),
 'R2': array([0.02013486, 0.04515475, 0.02387236, 0.02898141, 0.0334225 ])}
```

In [420…
```
polynomial_regression_scores_average_year = {"RMSE": polynomial_regression_scores_year["RMSE"
polynomial_regression_scores_average_year
```

Out[420]:
```
{'RMSE': 192647.39099423107,
 'MAE': 131481.9899718351,
 'R2': 0.03031317732927887}
```

Sadly, we have no luck with this approach either, as using just a single feature does not help the model perform any better. Much like with the feature subset attempts above, the R2 score is worryingly close to zero, with the rest of the metrics also seeing some alarming jumps. For this reason, we will not be using the polynomial regression model in this assignment.

(ii) Optimal tree depth

We have already discussed what the depth of a decision tree means. We also briefly mentioned that it may be possible to increase or decrease this depth in order to check how this affects the metrics. Generally, if a tree's depth is particularly high, then we will need to be cautious of over-fitting - for this reason, we will ensure to review all the metrics produced carefully.

We will cycle through depths from 5 to 50 and get metrics for each of these, adding them to a results dataframe as we go. Afterwords, we will plot a graph of each metric to see if we can spot any interesting patterns.

In [421…

```
tree_depth_metrics = pd.DataFrame()

# Loop through depths between 5 and 50
for i in range(5, 50):

    # Create tree with specific depth and traing the model
    current_tree_depth = tree.DecisionTreeRegressor(max_depth=i, random_state=1)
    current_tree = current_tree_depth.fit(X_train, y_train)

    # Get metrics and add to dataframe
    df_current_tree = get_full_metrics(current_tree, X_train, y_train, X_test, y_test, X_full
    tree_depth_metrics = pd.concat([tree_depth_metrics, df_current_tree], ignore_index=True,

# Because we are starting with depth of five, increase the index so that the index represents
tree_depth_metrics.index += 5
tree_depth_metrics
```

Out[421]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Train (R2)	
5	166885.305442	168408.121260	169356.284507	112139.976184	112035.125456	113262.847573	0.274110	0.25
6	165251.872873	167460.227193	168327.051999	110745.320298	111023.130853	112334.595409	0.288251	0.26
7	163801.234720	167182.304282	167680.535197	109394.902751	109995.791249	111428.808923	0.300692	0.26
8	162533.715840	166733.311975	167145.760109	108276.065716	109669.504812	110762.705633	0.311473	0.27
9	161397.854073	166612.179754	166709.094600	107203.076728	109654.026728	110005.770730	0.321062	0.27
10	160401.609735	166148.675406	166312.331784	106271.713235	109040.095441	109391.292754	0.329418	0.27
11	159501.598665	165276.124941	165930.132703	105442.945260	108103.476165	108967.032932	0.336922	0.28
12	158716.479758	165115.267227	165569.799219	104629.112322	108022.582316	108635.711223	0.343434	0.28
13	158066.907725	164709.089374	165059.974692	103811.153634	107745.476577	108210.436310	0.348797	0.28
14	157438.783298	163729.843362	164664.288153	103188.814329	106809.909640	108057.773892	0.353962	0.29
15	156866.983780	163993.238846	164597.712337	102608.448937	106868.529765	107794.279786	0.358647	0.29
16	156334.585229	163458.019775	164404.850719	102015.551126	106547.382478	107538.532434	0.362993	0.29
17	155798.422582	163495.534417	164312.201406	101563.922659	106586.283681	107355.065615	0.367354	0.29
18	155436.116971	163490.950180	164104.819903	101028.537051	106606.934748	107289.248537	0.370293	0.29
19	155156.842780	163128.749788	164424.242300	100651.862844	106242.270642	107376.303429	0.372554	0.30
20	154921.265973	163319.702438	164366.298276	100383.868104	106434.598846	107432.306161	0.374458	0.30
21	154334.067163	164036.260403	164839.541495	100003.482052	106791.940423	107580.581063	0.379191	0.29
22	154126.473745	163927.721156	164808.982965	99827.563750	106695.284001	107665.320977	0.380860	0.29
23	153879.512910	163948.122901	164889.202421	99535.391927	106669.186288	107731.920558	0.382843	0.29
24	153687.878101	163916.777402	164908.405814	99388.330111	106643.381924	107823.410993	0.384379	0.29
25	153565.602651	163831.215430	164977.465538	99277.312182	106580.753066	107919.439345	0.385358	0.29
26	153295.425606	164094.693354	164944.411820	99175.313098	106804.494057	107848.230605	0.387519	0.29
27	153106.696284	164167.354957	164953.463562	99077.733302	106752.716799	107866.841050	0.389026	0.29
28	153016.408536	164227.430890	164982.414812	98967.221254	106780.391910	107867.303847	0.389746	0.29
29	152872.133731	164288.162280	165188.232870	98851.086407	106760.837940	107965.873572	0.390897	0.29
30	152754.190871	164555.307032	165165.049927	98765.385560	106853.793684	107929.791311	0.391836	0.29
31	152500.355404	164477.959863	165206.659804	98588.381370	107043.358012	107982.555353	0.393856	0.29
32	152405.116040	164666.982202	165051.121887	98489.673534	107105.257765	107977.624350	0.394613	0.28
33	152318.411155	164907.568144	165266.863451	98471.875327	107265.802228	108043.618231	0.395301	0.28
34	152135.155222	165172.572284	165282.596582	98359.831493	107383.800003	108063.578744	0.396755	0.28
35	152126.371615	165191.992117	165278.448905	98348.379380	107447.079004	108067.228888	0.396825	0.28
36	152119.695894	165282.811504	165289.000600	98341.478320	107473.866946	108078.985928	0.396878	0.28
37	152117.264572	165207.369417	165310.308347	98337.409811	107432.232665	108100.888762	0.396897	0.28
38	152116.982872	165207.258444	165312.889809	98332.854029	107432.675009	108102.706877	0.396899	0.28
39	152116.982872	165207.258444	165309.426429	98332.854029	107432.675009	108097.276902	0.396899	0.28
40	152116.982872	165207.258444	165309.426429	98332.854029	107432.675009	108097.276902	0.396899	0.28
41	152116.982872	165207.258444	165309.426429	98332.854029	107432.675009	108097.276902	0.396899	0.28

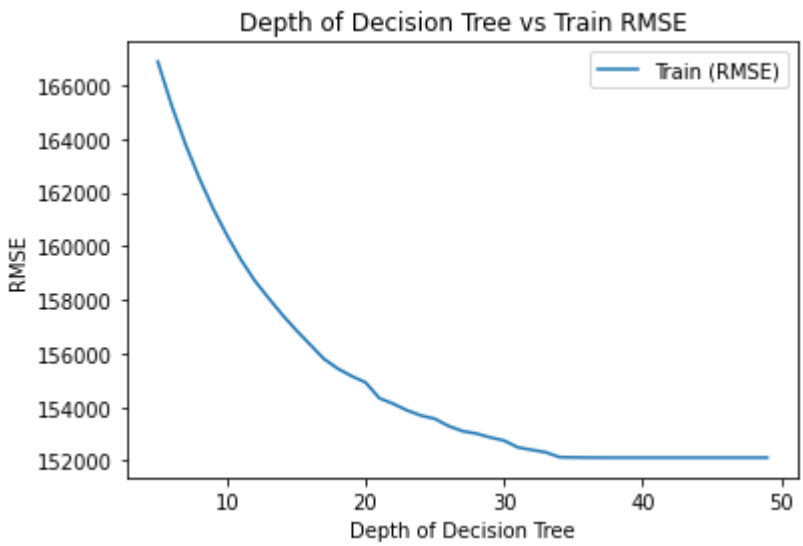
	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Train (R2)	
42	152116.982872	165207.258444	165309.426429	98332.854029	107432.675009	108097.276902	0.396899	0.28
43	152116.982872	165207.258444	165309.426429	98332.854029	107432.675009	108097.276902	0.396899	0.28
44	152116.982872	165207.258444	165309.426429	98332.854029	107432.675009	108097.276902	0.396899	0.28
45	152116.982872	165207.258444	165309.426429	98332.854029	107432.675009	108097.276902	0.396899	0.28
46	152116.982872	165207.258444	165309.426429	98332.854029	107432.675009	108097.276902	0.396899	0.28
47	152116.982872	165207.258444	165309.426429	98332.854029	107432.675009	108097.276902	0.396899	0.28
48	152116.982872	165207.258444	165309.426429	98332.854029	107432.675009	108097.276902	0.396899	0.28
49	152116.982872	165207.258444	165309.426429	98332.854029	107432.675009	108097.276902	0.396899	0.28

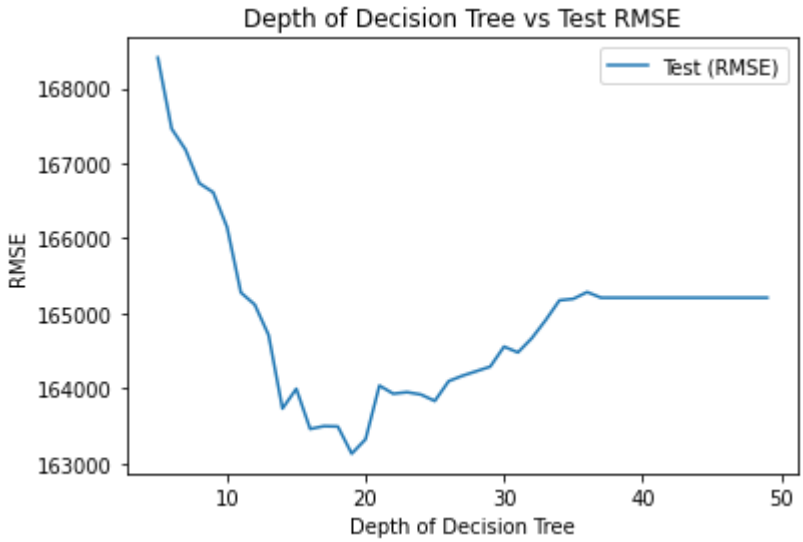
In [422…

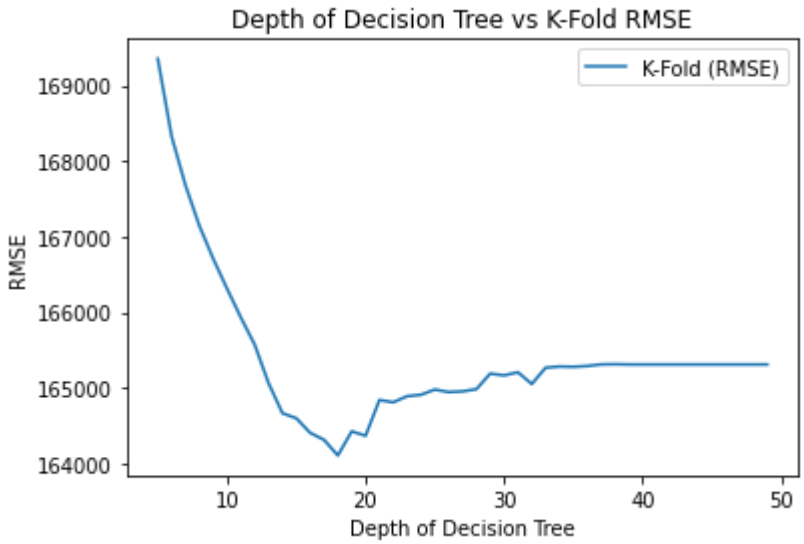
```
tree_plot_rmse = tree_depth_metrics.plot(y="Train (RMSE)", use_index=True)
tree_plot_rmse.set_title("Depth of Decision Tree vs Train RMSE")
tree_plot_rmse.set_xlabel("Depth of Decision Tree")
tree_plot_rmse.set_ylabel("RMSE")

tree_plot_rmse = tree_depth_metrics.plot(y="Test (RMSE)", use_index=True)
tree_plot_rmse.set_title("Depth of Decision Tree vs Test RMSE")
tree_plot_rmse.set_xlabel("Depth of Decision Tree")
tree_plot_rmse.set_ylabel("RMSE")


tree_plot_rmse = tree_depth_metrics.plot(y="K-Fold (RMSE)", use_index=True)
tree_plot_rmse.set_title("Depth of Decision Tree vs K-Fold RMSE")
tree_plot_rmse.set_xlabel("Depth of Decision Tree")
tree_plot_rmse.set_ylabel("RMSE")
```

Out[422]:

```
Text(0, 0.5, 'RMSE')
```





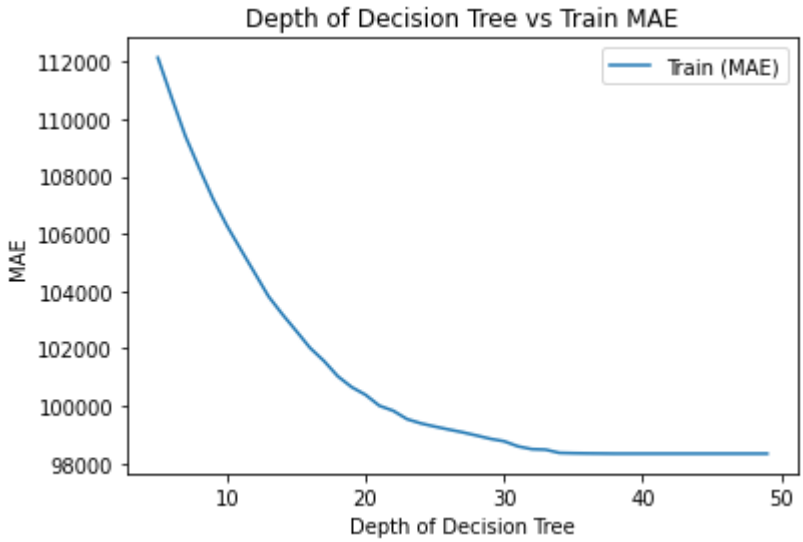


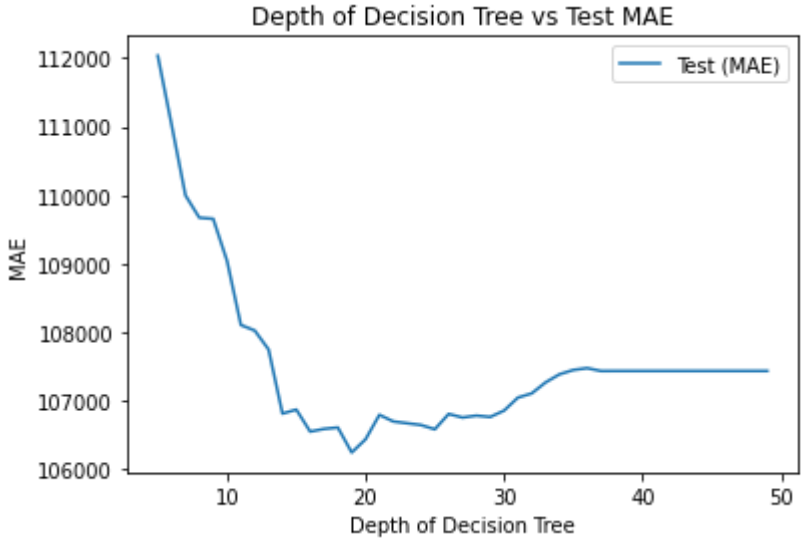
In [423…

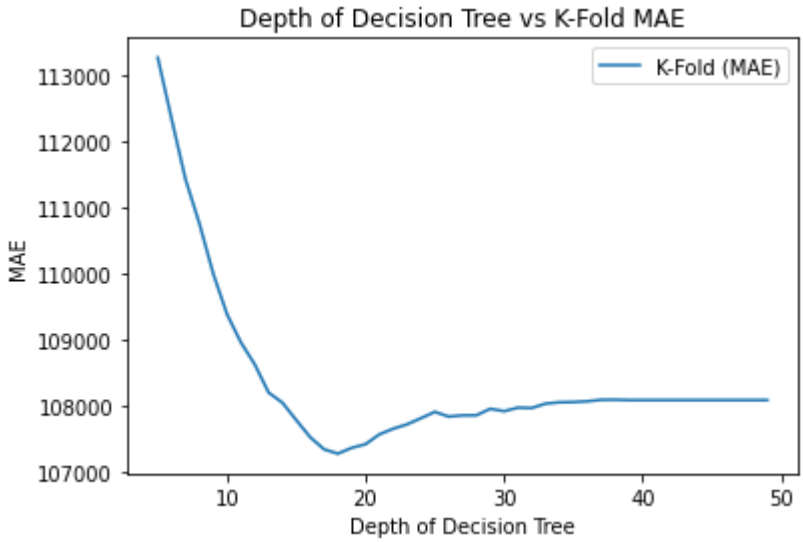
```
tree_plot_mae = tree_depth_metrics.plot(y="Train (MAE)", use_index=True)
tree_plot_mae.set_title("Depth of Decision Tree vs Train MAE")
tree_plot_mae.set_xlabel("Depth of Decision Tree")
tree_plot_mae.set_ylabel("MAE")

tree_plot_mae = tree_depth_metrics.plot(y="Test (MAE)", use_index=True)
tree_plot_mae.set_title("Depth of Decision Tree vs Test MAE")
tree_plot_mae.set_xlabel("Depth of Decision Tree")
tree_plot_mae.set_ylabel("MAE")

tree_plot_mae = tree_depth_metrics.plot(y="K-Fold (MAE)", use_index=True)
tree_plot_mae.set_title("Depth of Decision Tree vs K-Fold MAE")
tree_plot_mae.set_xlabel("Depth of Decision Tree")
tree_plot_mae.set_ylabel("MAE")
```

Out[423]:

```
Text(0, 0.5, 'MAE')
```





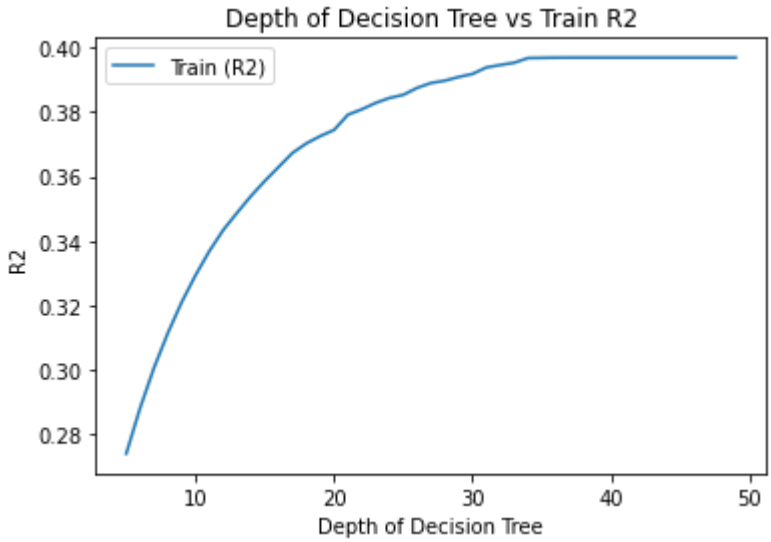


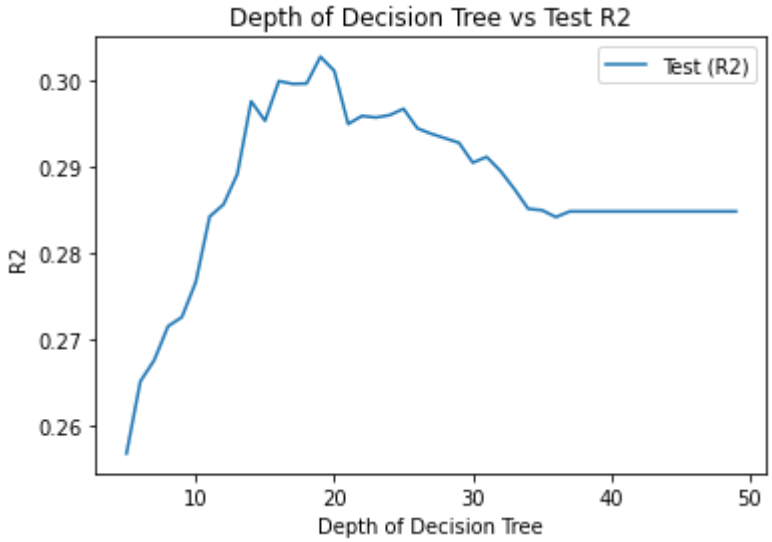
In [424…

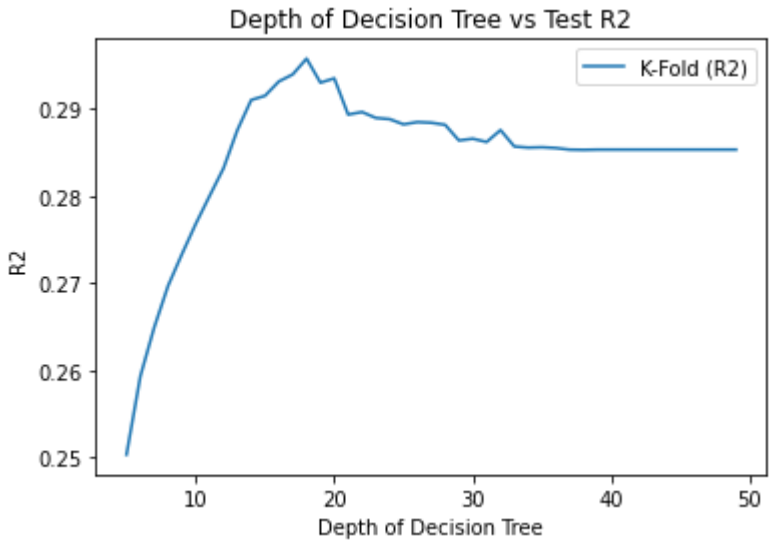
```
tree_plot_r2 = tree_depth_metrics.plot(y="Train (R2)", use_index=True)
tree_plot_r2.set_title("Depth of Decision Tree vs Train R2")
tree_plot_r2.set_xlabel("Depth of Decision Tree")
tree_plot_r2.set_ylabel("R2")

tree_plot_r2 = tree_depth_metrics.plot(y="Test (R2)", use_index=True)
tree_plot_r2.set_title("Depth of Decision Tree vs Test R2")
tree_plot_r2.set_xlabel("Depth of Decision Tree")
tree_plot_r2.set_ylabel("R2")

tree_plot_r2 = tree_depth_metrics.plot(y="K-Fold (R2)", use_index=True)
tree_plot_r2.set_title("Depth of Decision Tree vs Test R2")
```

```
tree_plot_r2.set_xlabel("Depth of Decision Tree")
tree_plot_r2.set_ylabel("R2")
```

Out[424]:
```
Text(0, 0.5, 'R2')
```







What we can see from the above graphs is that for the train dataset, the curve continues to grow, up until the point where it caps out at around a depth of around 33 or 34, after which the model does not improve at all.

With test dataset and the cross-evaluation score, however, we can see that that the optimum performance peaks at a depth of around 18 or 19, after which the model decreases in performance again and levels out at around 33 or 34. The reasoning for this is probably that there is a sweet spot at

the 18/19 depth range whereby the model has not seen enough data to be prone to overfitting, but has seen enough to make reasonable accurate predictions.

For this reason, we will set our maximum depth at 18 to get the optimal scores for our decision tree model. Let's calculate the metrics for this and compare them against the existing models.

In [425…
```
tree_depth_optimal = tree.DecisionTreeRegressor(max_depth = 18, random_state = 1)

tree_optimal = tree_depth_optimal.fit(X_train, y_train)
```

In [426…
```
df_tree_optimal = get_full_metrics(tree_optimal, X_train, y_train, X_test, y_test, X_full, y_
global_results = pd.concat([global_results, df_tree_optimal], ignore_index=True, axis=0)
global_results = global_results.rename(index={0: "Linear Regression"})
global_results = global_results.rename(index={1: "Decision Tree"})
global_results = global_results.rename(index={2: "Random Forest"})
global_results = global_results.rename(index={3: "Decision Tree (Optimal)"})
global_results
```

Out[426]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Tr (R
Linear Regression	160315.011772	159630.916296	160739.525422	105072.408566	103548.953582	105193.439280	0.3301
Decision Tree	160401.609735	166079.484810	166311.775871	106271.713235	109010.255814	109389.733497	0.3294
Random Forest	152388.209427	164287.337477	164007.105837	99190.389316	107299.859217	107454.689002	0.3947
Decision Tree (Optimal)	155436.116971	163490.950180	164104.819903	101028.537051	106606.934748	107289.248537	0.3702

When comparing the metrics, we can see that our optimum depth model is performing better than the optimised version of the decision tree. When compared to the other models, however, it is still performing a little worse than linear regression, and roughly the same as the unoptimised version of random forest. The difference in metrics is not too large, but it is still a model modification worth considering.

(iii) Optimal random forest estimators

We had previously discussed that a random forest consists of many different decision trees. We should therefore try and conduct an exercise similar to the one relating to depth for decision tree to test what the optimum number of estimators is for our model. The methodology here is the same - we will loop through a range of estimators (noting that random forests take longer to train, so we should select a smaller sample, such as between 100 and 125) and then plot our results.

In [427…
```
# Warning: This cell takes a significant amount of time to run (approx. five minutes)

forest_estimator_metrics = pd.DataFrame()

# Loop through estimators in range 100 to 125
for i in range(100, 125):
    # Include explanatory message so that we can see what stage of the process we are at
    print(f"Reviewing with {i} estimators")

    # Create forest with specified number of estimators and train the model
```

```
    current_forest = ensemble.RandomForestRegressor(n_estimators=i, max_features='auto', oob_
    current_forest.fit(X_train, y_train)

    # Add results to dataframe
    df_current_forest = get_full_metrics(current_forest, X_train, y_train, X_test, y_test, X_
    forest_estimator_metrics = pd.concat([forest_estimator_metrics, df_current_forest], ignor

# Increase index by 100 so that the index represents the number of estimators
forest_estimator_metrics.index += 100
forest_estimator_metrics
```

```
Reviewing with 100 estimators
Reviewing with 101 estimators
Reviewing with 102 estimators
Reviewing with 103 estimators
Reviewing with 104 estimators
Reviewing with 105 estimators
Reviewing with 106 estimators
Reviewing with 107 estimators
Reviewing with 108 estimators
Reviewing with 109 estimators
Reviewing with 110 estimators
Reviewing with 111 estimators
Reviewing with 112 estimators
Reviewing with 113 estimators
Reviewing with 114 estimators
Reviewing with 115 estimators
Reviewing with 116 estimators
Reviewing with 117 estimators
Reviewing with 118 estimators
Reviewing with 119 estimators
Reviewing with 120 estimators
Reviewing with 121 estimators
Reviewing with 122 estimators
Reviewing with 123 estimators
Reviewing with 124 estimators
```

Out[427]:

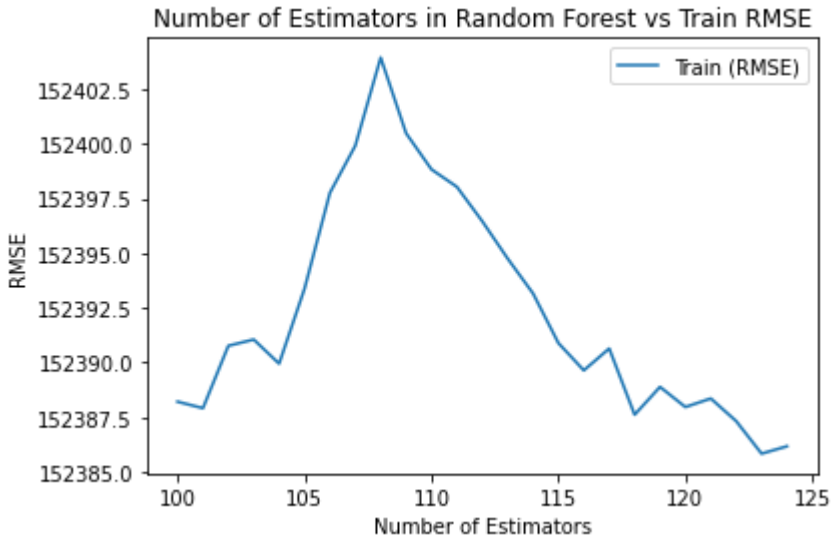
	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Train (R2)	
100	152388.209427	164287.337477	164007.105837	99190.389316	107299.859217	107454.689002	0.394747	0.29
101	152387.912645	164269.156157	164005.342021	99188.230647	107287.095871	107452.596178	0.394749	0.29
102	152390.763569	164260.968507	163998.497083	99193.810280	107292.288455	107448.221132	0.394727	0.29
103	152391.050379	164270.634069	163998.823752	99193.588624	107296.334217	107448.814909	0.394724	0.29
104	152389.948099	164250.496527	163984.672336	99200.154578	107291.597154	107439.945559	0.394733	0.29
105	152393.375023	164240.836607	163979.915938	99202.198764	107293.936249	107438.208474	0.394706	0.29
106	152397.755914	164219.539991	163989.170858	99205.059086	107282.083673	107445.949732	0.394671	0.29
107	152399.912851	164203.448417	163988.784949	99204.216767	107266.235065	107445.289663	0.394654	0.29
108	152403.938029	164197.801051	163989.761358	99205.194343	107266.748707	107447.867285	0.394622	0.29
109	152400.480880	164200.069354	163990.443293	99197.204197	107259.818238	107452.065618	0.394649	0.29
110	152398.827802	164191.500974	163987.663018	99197.065264	107257.565470	107454.375984	0.394662	0.29
111	152398.032072	164188.825561	163994.967951	99198.793118	107252.867557	107455.778636	0.394669	0.29
112	152396.458622	164186.465370	163993.327820	99197.075378	107244.374183	107462.923789	0.394681	0.29
113	152394.752591	164193.570421	163986.030300	99181.586853	107231.488590	107461.498713	0.394695	0.29
114	152393.156670	164193.727975	163987.831855	99177.914125	107221.183126	107457.352777	0.394708	0.29
115	152390.886555	164187.762107	163987.966821	99167.751116	107209.719282	107457.428927	0.394726	0.29
116	152389.640740	164179.583450	163996.981773	99169.923008	107201.237839	107459.341799	0.394735	0.29
117	152390.639941	164168.597430	163979.520248	99173.299464	107198.077474	107447.283996	0.394728	0.29
118	152387.613973	164171.277094	163985.198068	99167.282662	107193.560218	107448.559550	0.394752	0.29
119	152388.884741	164160.549225	163994.801688	99169.439898	107185.525818	107453.299907	0.394741	0.29
120	152387.965482	164170.689081	163986.968147	99168.454053	107194.259628	107448.772540	0.394749	0.29
121	152388.354971	164161.014719	163981.485579	99164.817855	107187.717107	107445.429314	0.394746	0.29
122	152387.320515	164157.049694	163998.555364	99156.892383	107174.708902	107454.933341	0.394754	0.29
123	152385.834168	164159.540813	164011.696045	99153.616671	107171.001490	107458.759262	0.394766	0.29
124	152386.170423	164146.427372	163997.471289	99153.647042	107166.432928	107449.497922	0.394763	0.29

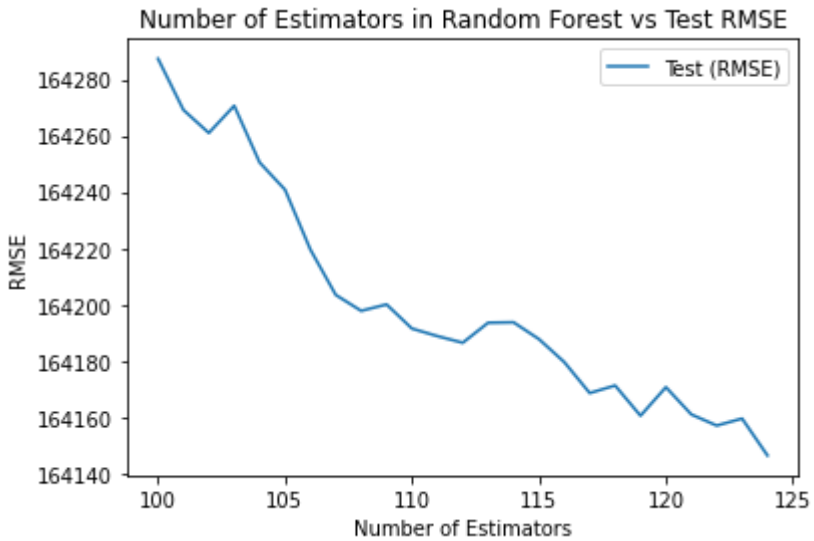
In [428…

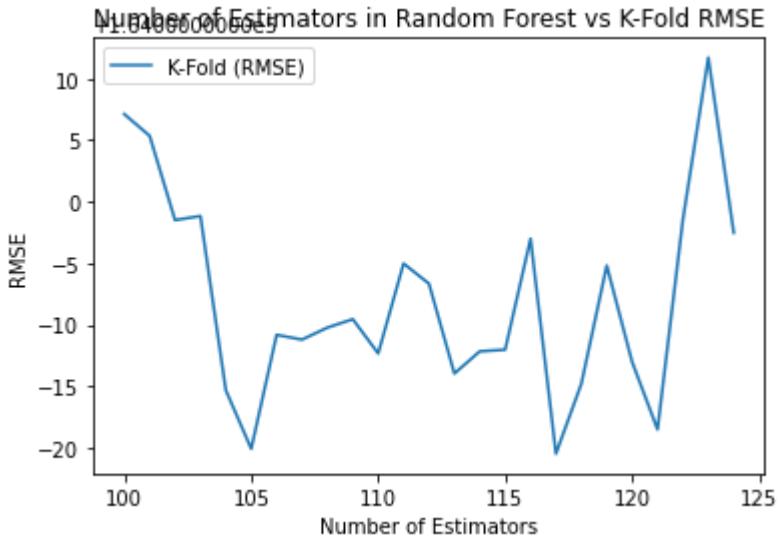
```
forest_plot_rmse = forest_estimator_metrics.plot(y="Train (RMSE)", use_index=True)
forest_plot_rmse.set_title("Number of Estimators in Random Forest vs Train RMSE")
forest_plot_rmse.set_xlabel("Number of Estimators")
forest_plot_rmse.set_ylabel("RMSE")

forest_plot_rmse = forest_estimator_metrics.plot(y="Test (RMSE)", use_index=True)
forest_plot_rmse.set_title("Number of Estimators in Random Forest vs Test RMSE")
forest_plot_rmse.set_xlabel("Number of Estimators")
forest_plot_rmse.set_ylabel("RMSE")

forest_plot_rmse = forest_estimator_metrics.plot(y="K-Fold (RMSE)", use_index=True)
forest_plot_rmse.set_title("Number of Estimators in Random Forest vs K-Fold RMSE")
forest_plot_rmse.set_xlabel("Number of Estimators")
forest_plot_rmse.set_ylabel("RMSE")
```

Out[428]:

```
Text(0, 0.5, 'RMSE')
```







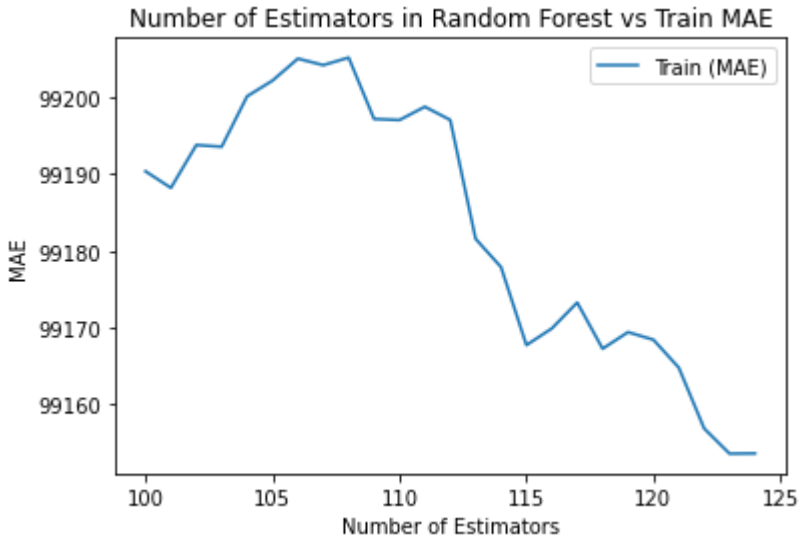
In [429…

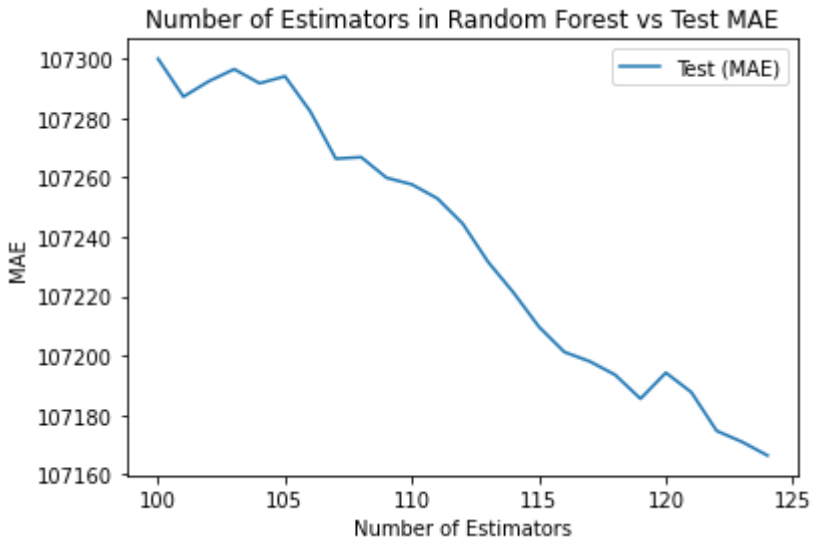
```
forest_plot_mae = forest_estimator_metrics.plot(y="Train (MAE)", use_index=True)
forest_plot_mae.set_title("Number of Estimators in Random Forest vs Train MAE")
forest_plot_mae.set_xlabel("Number of Estimators")
forest_plot_mae.set_ylabel("MAE")

forest_plot_mae = forest_estimator_metrics.plot(y="Test (MAE)", use_index=True)
forest_plot_mae.set_title("Number of Estimators in Random Forest vs Test MAE")
forest_plot_mae.set_xlabel("Number of Estimators")
forest_plot_mae.set_ylabel("MAE")

forest_plot_mae = forest_estimator_metrics.plot(y="K-Fold (MAE)", use_index=True)
forest_plot_mae.set_title("Number of Estimators in Random Forest vs K-Fold MAE")
```

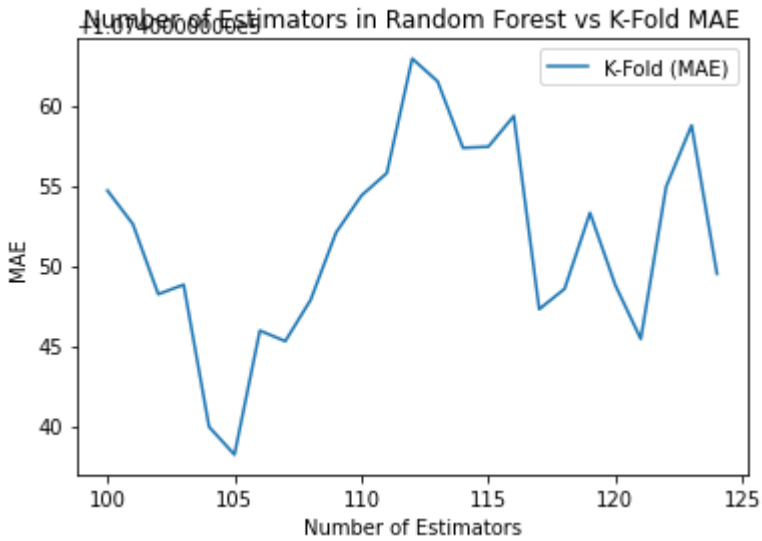
```
forest_plot_mae.set_xlabel("Number of Estimators")
forest_plot_mae.set_ylabel("MAE")
```

Out[429]:

```
Text(0, 0.5, 'MAE')
```





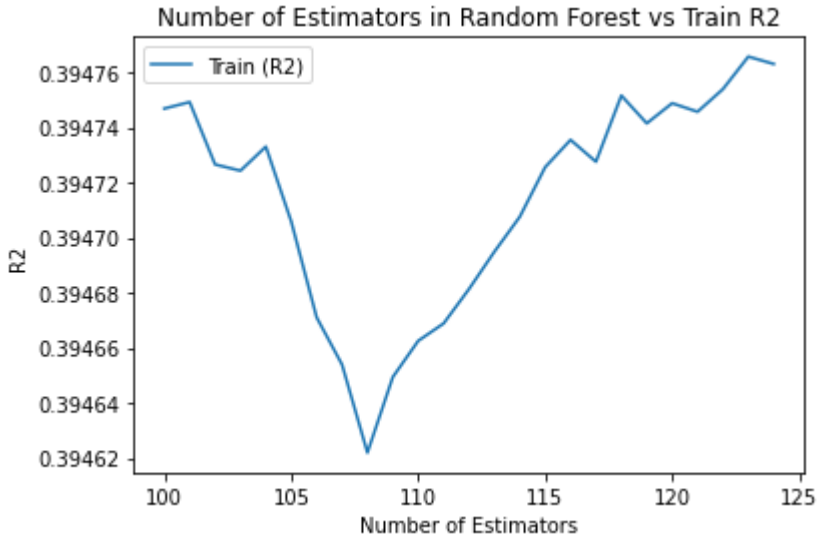


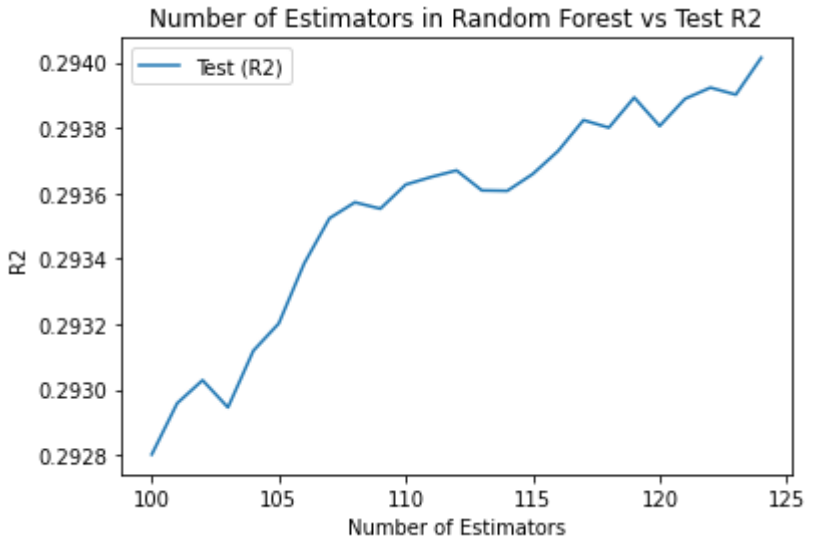
In [430…

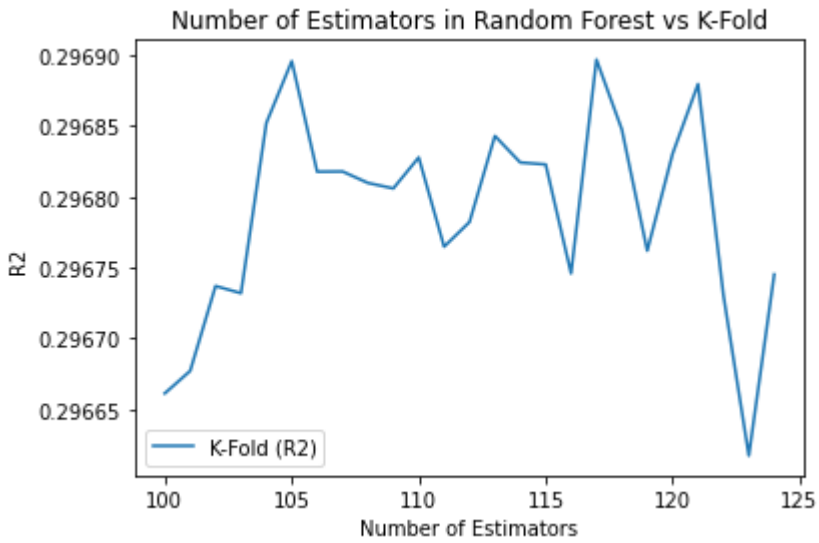
```
forest_plot_r2 = forest_estimator_metrics.plot(y="Train (R2)", use_index=True)
forest_plot_r2.set_title("Number of Estimators in Random Forest vs Train R2")
forest_plot_r2.set_xlabel("Number of Estimators")
forest_plot_r2.set_ylabel("R2")

forest_plot_r2 = forest_estimator_metrics.plot(y="Test (R2)", use_index=True)
forest_plot_r2.set_title("Number of Estimators in Random Forest vs Test R2")
forest_plot_r2.set_xlabel("Number of Estimators")
forest_plot_r2.set_ylabel("R2")
```

```
forest_plot_r2 = forest_estimator_metrics.plot(y="K-Fold (R2)", use_index=True)
forest_plot_r2.set_title("Number of Estimators in Random Forest vs K-Fold")
forest_plot_r2.set_xlabel("Number of Estimators")
forest_plot_r2.set_ylabel("R2")
```

Out[430]:
```
Text(0, 0.5, 'R2')
```







The results here seem to be a little more erratic, without a very clear pattern that we can distinguish. Perhaps this because each time we are running the random forest model, the decision trees that will be calculated will be slightly different, causing some fluctuation in the results.

In any event, it would seem that taken as a whole, the best overall number of estimators that provides us with the best average of metrics is around 122. For this reason we will treat this as the optimal random forest model and will add this to our dataframe.

In [431…
```
forest_optimal = ensemble.RandomForestRegressor(n_estimators=122, max_features='auto', oob_sc
forest_optimal.fit(X_train, y_train)
```

Out[431]:
```
RandomForestRegressor(n_estimators=122, oob_score=True, random_state=1)
```

In [432…
```
df_forest_optimal = get_full_metrics(forest_optimal, X_train, y_train, X_test, y_test, X_full
global_results = pd.concat([global_results, df_forest_optimal], ignore_index=True, axis=0)
global_results = global_results.rename(index={0: "Linear Regression"})
global_results = global_results.rename(index={1: "Decision Tree"})
global_results = global_results.rename(index={2: "Random Forest"})
global_results = global_results.rename(index={3: "Decision Tree (Optimal)"})
global_results = global_results.rename(index={4: "Random Forest (Optimal)"})
global_results
```

Out[432]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Tra (R
Linear Regression	160315.011772	159630.916296	160739.525422	105072.408566	103548.953582	105193.439280	0.3301
Decision Tree	160401.609735	166079.484810	166311.775871	106271.713235	109010.255814	109389.733497	0.3294
Random Forest	152388.209427	164287.337477	164007.105837	99190.389316	107299.859217	107454.689002	0.3947
Decision Tree (Optimal)	155436.116971	163490.950180	164104.819903	101028.537051	106606.934748	107289.248537	0.3702
Random Forest (Optimal)	152387.320515	164157.049694	163998.555364	99156.892383	107174.708902	107454.933341	0.3947

As we can see, the optimised random forest model does perform better than its unoptimised counterpart, but marginally so - the improvement gain is not particularly significant, with a very, very, minor increase in performance.

It looks like overall, plain linear regression on the same subset of features that we had initially chosen is the model that gives us the best performance. It seems that even our best attempt at optimising these models could not outperform just the basic linear regression model, which just goes to show that sometimes a simple approach to predictive modelling is the best one.

Therefore, this is the model that we will ultimately take for the last part of the question.

We should note, however, that due to valid outliers, it is possible that the model will not perform well on brand new dataset. It might make sense for us to test the new dataset

---

(c) New test set

Take your best model trained and selected based on past data (i.e. your cleaned Homework1 dataset), and evaluate it on the new test dataset provided with this homework (in file '22032022-PPR-Price-recent.csv').

Note that the new test data has to be transformed using the same steps as the past training data, otherwise the trained model cannot be used for prediction on the new data. Discuss your findings.

We will now read in the new dataset provided to us with the assignment sheet and use our best model - being linear regression - and check the metrics of how well it predicts the target feature.

To start, we will need to prepare the new dataset for use with our model. Because we technically do not know what this dataset contains, we should first read in the CSV file and print out some values from it to see how the data is set out.

Note that it is possible that linear regression may perform poorly due to valid outliers. For this reason, just to be sure, we will also test our best performing decision tree and random forest (with the appropriate optimised depth and estimators respectively) on the new dataset.

In [433…
```
df_new = pd.read_csv("22032022-PPR-Price-recent.csv")
```

In [434…
```
df_new.head()
```

Out[434]:

	DateofSale(dd/mm/yyyy)	Address	PostalCode	County	Price(€)	NotFullMarketPrice	VATExclusive
0	2022-01-15	24 FOREST WALK, SWORDS, DUBLIN	NaN	Dublin	154000.00	No	No
1	2022-01-15	5 CRAGAUN, FATHER RUSSELL ROAD, DOORADOYLE	NaN	Limerick	370000.00	No	No
2	2022-01-16	19 The Lawn, Mullen Park, Maynooth	NaN	Kildare	409691.63	No	Yes
3	2022-01-16	MONTEVIDEO, HOSPITAL HILL, BUNCLODY	NaN	Wexford	100000.00	No	No
4	2022-01-17	1 CILL BEG MANOR, STRADBALLY, LAOIS	NaN	Laois	225000.00	No	No

As expected, we can see that the dataframe is very similar to the original dataframe that we had at the start of Assignment 1, before the data has been cleaned. Already we can spot a few issues with the data that will need to be fixed.

- There are no spaces in the feature names that comprise of multiple words. These will need to be renamed to exactly the same names of the features that we trained our models with.
- We do not have any of the derived features, such as "Year" or "Quarter". We will need to derive these by first converting Date of Sale to a timestamp and then accessing its year and quarter properties.
- Certain columns that were dropped as part of our initial data quality plan (such as Property Size Description) will need to be dropped here, too.

- The various steps listed at the start of this assignment, such as removing NaN variables, etc., will also need to be carried out.

Because these are all steps that have already been carried out in explained both as part of Assignment 1 and also at the start of this assignment, we will not go into any great detail on the reasoning behind these here, as data cleaning is not the focus of this assignment. Where any steps require additional clarification, however, this will be provided.

In [435…
```
# Convert datetime string to datetime object
df_new["DateofSale(dd/mm/yyyy)"] = pd.to_datetime(df_new["DateofSale(dd/mm/yyyy)"], infer_dat

# Rename datetime column
df_new.rename(columns={"DateofSale(dd/mm/yyyy)": "Date of Sale"}, inplace=True)
```

In [436…
```
df_new.rename(columns={'PostalCode': 'Postal Code (Dublin)'}, inplace=True)
df_new.head()
```

Out[436]:

	Date of Sale	Address	Postal Code (Dublin)	County	Price(€)	NotFullMarketPrice	VATExclusive	DescriptionofPropert
0	2022-01-15	24 FOREST WALK, SWORDS, DUBLIN	NaN	Dublin	154000.00	No	No	Second-Hand Dwellin house /Apartmen
1	2022-01-15	5 CRAGAUN, FATHER RUSSELL ROAD, DOORADOYLE	NaN	Limerick	370000.00	No	No	Second-Hand Dwellin house /Apartmen
2	2022-01-16	19 The Lawn, Mullen Park, Maynooth	NaN	Kildare	409691.63	No	Yes	New Dwelling hous /Apartmen
3	2022-01-16	MONTEVIDEO, HOSPITAL HILL, BUNCLODY	NaN	Wexford	100000.00	No	No	Second-Hand Dwellin house /Apartmen
4	2022-01-17	1 CILL BEG MANOR, STRADBALLY, LAOIS	NaN	Laois	225000.00	No	No	Second-Hand Dwellin house /Apartmen

In [437…
```
df_new.rename(columns={'DescriptionofProperty': 'Second-Hand'}, inplace=True)

df_new["Second-Hand"] = df_new["Second-Hand"].map({"Second-Hand Dwelling house /Apartment": T
```

In [438…
```
df_new.drop(columns=["PropertySizeDescription"], inplace = True)
```

In [439…
```
df_new.rename(columns={'NotFullMarketPrice': 'Full Market Price'}, inplace=True)

df_new["Full Market Price"] = df_new["Full Market Price"].map({"No": True, "Yes": False})
df_new.head()
```

Out[439]:

	Date of Sale	Address	Postal Code (Dublin)	County	Price(€)	Full Market Price	VATExclusive	Second-Hand
0	2022-01-15	24 FOREST WALK, SWORDS, DUBLIN	NaN	Dublin	154000.00	True	No	True
1	2022-01-15	5 CRAGAUN, FATHER RUSSELL ROAD, DOORADOYLE	NaN	Limerick	370000.00	True	No	True
2	2022-01-16	19 The Lawn, Mullen Park, Maynooth	NaN	Kildare	409691.63	True	Yes	False
3	2022-01-16	MONTEVIDEO, HOSPITAL HILL, BUNCLODY	NaN	Wexford	100000.00	True	No	True
4	2022-01-17	1 CILL BEG MANOR, STRADBALLY, LAOIS	NaN	Laois	225000.00	True	No	True

In [440…

```
df_new.rename(columns={'VATExclusive': 'VAT Exclusive'}, inplace=True)
df_new["VAT Exclusive"] = df_new["VAT Exclusive"].map({"Yes":True, "No":False})
df_new.head()
```

Out[440]:

	Date of Sale	Address	Postal Code (Dublin)	County	Price(€)	Full Market Price	VAT Exclusive	Second-Hand
0	2022-01-15	24 FOREST WALK, SWORDS, DUBLIN	NaN	Dublin	154000.00	True	False	True
1	2022-01-15	5 CRAGAUN, FATHER RUSSELL ROAD, DOORADOYLE	NaN	Limerick	370000.00	True	False	True
2	2022-01-16	19 The Lawn, Mullen Park, Maynooth	NaN	Kildare	409691.63	True	True	False
3	2022-01-16	MONTEVIDEO, HOSPITAL HILL, BUNCLODY	NaN	Wexford	100000.00	True	False	True
4	2022-01-17	1 CILL BEG MANOR, STRADBALLY, LAOIS	NaN	Laois	225000.00	True	False	True

In [441…

```
df_new['Year'] = pd.DatetimeIndex(df_new['Date of Sale']).year
df_new["Quarter"] = df_new["Date of Sale"].dt.quarter
```

In [442…

```
df_new.head()
```

Out[442]:

	Date of Sale	Address	Postal Code (Dublin)	County	Price(€)	Full Market Price	VAT Exclusive	Second-Hand	Year	Quarter
0	2022-01-15	24 FOREST WALK, SWORDS, DUBLIN	NaN	Dublin	154000.00	True	False	True	2022	1
1	2022-01-15	5 CRAGAUN, FATHER RUSSELL ROAD, DOORADOYLE	NaN	Limerick	370000.00	True	False	True	2022	1
2	2022-01-16	19 The Lawn, Mullen Park, Maynooth	NaN	Kildare	409691.63	True	True	False	2022	1
3	2022-01-16	MONTEVIDEO, HOSPITAL HILL, BUNCLODY	NaN	Wexford	100000.00	True	False	True	2022	1
4	2022-01-17	1 CILL BEG MANOR, STRADBALLY, LAOIS	NaN	Laois	225000.00	True	False	True	2022	1

We should also include the Inflation feature that we derived from the World Bank CSV as part of Assignment 1. While this was not part of our high correlation feature subset, nonetheless we should include it just in case at some later stage we find a model that scales well with it.

In [443…

```
inflation_df = pd.read_csv("inflation.csv")
irish_inflation_df = inflation_df.loc[inflation_df["Country Name"] == "Ireland"]
irish_columns = irish_inflation_df[["2010", "2011", "2012", "2013", "2014", "2015", "2016", "
irish_inflation_df = irish_inflation_df[irish_columns]
```

In [444…

```
# Create new inflation column and set to 0
df_new["Inflation"] = 0

# Loop through rows
for index, row in df_new.iterrows():

    # Get current year, and do nothing if the year is 2021 or 2022 (as we do not have the val
    current_year = str(row["Year"])
    if current_year == "2021" or current_year == "2022":
        value_to_insert = None

    # For all other years insert the value of the inflation index
    else:
        value_to_insert = float(irish_inflation_df[current_year])
    df_new.at[index, "Inflation"] = value_to_insert
```

In [445…

```
df_new.head()
```

Out[445]:

	Date of Sale	Address	Postal Code (Dublin)	County	Price(€)	Full Market Price	VAT Exclusive	Second-Hand	Year	Quarter	Inflation
0	2022-01-15	24 FOREST WALK, SWORDS, DUBLIN	NaN	Dublin	154000.00	True	False	True	2022	1	NaN
1	2022-01-15	5 CRAGAUN, FATHER RUSSELL ROAD, DOORADOYLE	NaN	Limerick	370000.00	True	False	True	2022	1	NaN
2	2022-01-16	19 The Lawn, Mullen Park, Maynooth	NaN	Kildare	409691.63	True	True	False	2022	1	NaN
3	2022-01-16	MONTEVIDEO, HOSPITAL HILL, BUNCLODY	NaN	Wexford	100000.00	True	False	True	2022	1	NaN
4	2022-01-17	1 CILL BEG MANOR, STRADBALLY, LAOIS	NaN	Laois	225000.00	True	False	True	2022	1	NaN

In [446…
```
df_new.rename(columns = {"Price(€)": "Price"}, inplace = True)
df_new[["Postal Code (Dublin)"]] = df_new[["Postal Code (Dublin)"]].fillna("Not in Dublin")
df_new[["Inflation"]] = df_new[["Inflation"]].fillna(df_new["Inflation"].mean())
```

In [447…
```
df_new = df_new.drop(["Date of Sale", "Address"], axis=1)
df_new.head()
```

Out[447]:

	Postal Code (Dublin)	County	Price	Full Market Price	VAT Exclusive	Second-Hand	Year	Quarter	Inflation
0	Not in Dublin	Dublin	154000.00	True	False	True	2022	1	NaN
1	Not in Dublin	Limerick	370000.00	True	False	True	2022	1	NaN
2	Not in Dublin	Kildare	409691.63	True	True	False	2022	1	NaN
3	Not in Dublin	Wexford	100000.00	True	False	True	2022	1	NaN
4	Not in Dublin	Laois	225000.00	True	False	True	2022	1	NaN

We have now successfully created our cleaned dataframe. We should at this point create X and y dataframes and check the descriptive features that we have in them.

In [448…
```
X_new, y_new = get_X_and_y(df_new, True)

for column in X_new:
    print(column)
```

```
Full Market Price
VAT Exclusive
Second-Hand
Dublin 1
Dublin 10
Dublin 11
Dublin 12
Dublin 13
Dublin 14
Dublin 15
Dublin 16
Dublin 17
Dublin 18
Dublin 2
Dublin 20
Dublin 22
Dublin 24
Dublin 3
Dublin 4
Dublin 5
Dublin 6
Dublin 7
Dublin 8
Dublin 9
Not in Dublin
2022
Carlow
Cavan
Clare
Cork
Donegal
Dublin
Galway
Kerry
Kildare
Kilkenny
Laois
Leitrim
Limerick
Longford
Louth
Mayo
Meath
Monaghan
Offaly
Roscommon
Sligo
Tipperary
Waterford
Westmeath
Wexford
Wicklow
1
```

As we can see from the above, these properties seem to all have been sold in 2022. This is an issue for us because this means that we only have one dummy variable for the year, which is 2022. Our model, however, will have dummy variables for all years of the past decade. Our best option here, then, is to loop through the columns in the dataset that we used to train our model, and check if that column is in our new test dataframe. If it is not, we can add it and set it to 0 (as if the column does not exist, then this means that there are no entries in the test dataset that will be part of this category).

In [449…

```
# Loop through columns in main dataset
for column in X_full.columns:
```

```
    # If column is not in new dataset, print explanatory message and create the column filled
    if column not in X_new.columns:
        print(f"Column {column} is missing. Adding to dataframe.")
        X_new[column] = 0
```

```
Column Dublin 6w is missing. Adding to dataframe.
Column 2010 is missing. Adding to dataframe.
Column 2011 is missing. Adding to dataframe.
Column 2012 is missing. Adding to dataframe.
Column 2013 is missing. Adding to dataframe.
Column 2014 is missing. Adding to dataframe.
Column 2015 is missing. Adding to dataframe.
Column 2016 is missing. Adding to dataframe.
Column 2017 is missing. Adding to dataframe.
Column 2018 is missing. Adding to dataframe.
Column 2019 is missing. Adding to dataframe.
Column 2020 is missing. Adding to dataframe.
Column 2021 is missing. Adding to dataframe.
```

The above means that we have added all of the missing dummy variable. The last step that we have to take is to ensure that the columns in our test dataframe are in the same order as in the original dataset. We need to do this because sklearn sometimes has issues if the features are in the wrong order.

In [450…
```
X_new = X_new[X_train.columns]
```

In [451…
```
# Check to see if the new columns are identical to those of the original dataset
X_new.columns == X_train.columns
```

Out[451]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True])
```

In [452…
```
X_new.head()
```

Out[452]:

	Full Market Price	Dublin 1	Dublin 10	Dublin 11	Dublin 12	Dublin 13	Dublin 14	Dublin 15	Dublin 16	Dublin 17	...	Meath	Monaghan
0	1	0	0	0	0	0	0	0	0	0	...	0	0
1	1	0	0	0	0	0	0	0	0	0	...	0	0
2	1	0	0	0	0	0	0	0	0	0	...	0	0
3	1	0	0	0	0	0	0	0	0	0	...	0	0
4	1	0	0	0	0	0	0	0	0	0	...	0	0

5 rows × 63 columns

We can now see that our new dataframe is fully set up and has features identical and in the same order as the original dataframe. We can now proceed with making predictions using the models that we have already generated during the initial part of the assignment.

In [453…
```
linear_regression_new_predictions = linear_regression.predict(X_new)
pd.concat([y_new, pd.DataFrame(linear_regression_new_predictions, columns = ["Prediction"])],
```

Out[453]:

	Price	Prediction
0	154000.00	404736.0
1	370000.00	200704.0
2	409691.63	278528.0
3	100000.00	200704.0
4	225000.00	163840.0
5	475000.00	249344.0
6	130000.00	200704.0
7	305000.00	227328.0
8	216000.00	121856.0
9	325992.00	262912.0

In [454…

```
linear_regression_new_results = get_metrics(y_new, linear_regression_new_predictions)
print_metric_results("LINEAR REGRESSION NEW", linear_regression_new_results)
```

```
LINEAR REGRESSION NEW METRICS
***
RMSE: 549137.9072045721
MAE: 139702.86956836536
R2: 0.05198961998068463
***
```

In [455…

```
decision_tree_new_predictions = tree_optimal.predict(X_new)
pd.concat([y_new, pd.DataFrame(decision_tree_new_predictions, columns = ["Prediction"])], axi
```

Out[455]:

	Price	Prediction
0	154000.00	430449.148481
1	370000.00	169625.282322
2	409691.63	152000.000000
3	100000.00	169625.282322
4	225000.00	169625.282322
5	475000.00	199351.863457
6	130000.00	169625.282322
7	305000.00	87500.000000
8	216000.00	65081.833684
9	325992.00	280408.887500

In [456…

```
decision_tree_new_results = get_metrics(y_new, decision_tree_new_predictions)
print_metric_results("DECISION TREE NEW", decision_tree_new_results)
```

```
DECISION TREE NEW METRICS
***
RMSE: 555174.5221867256
MAE: 156157.51098682595
R2: 0.03103230302688409
***
```

In [457…

```
random_forest_new_predictions = forest_optimal.predict(X_new)
pd.concat([y_new, pd.DataFrame(random_forest_new_predictions, columns = ["Prediction"])], axi
```

Out[457]:

	Price	Prediction
0	154000.00	394806.675554
1	370000.00	212268.505273
2	409691.63	182894.332917
3	100000.00	182001.820364
4	225000.00	151806.092167
5	475000.00	228187.296881
6	130000.00	212268.505273
7	305000.00	180823.685929
8	216000.00	91886.409757
9	325992.00	235613.041826

In [458…

```
random_forest_new_results = get_metrics(y_new, random_forest_new_predictions)
print_metric_results("RANDOM FOREST NEW", random_forest_new_results)
```

```
RANDOM FOREST NEW METRICS
***
RMSE: 553151.8015427535
MAE: 148677.09870521218
R2: 0.03808010672484197
***
```

Now that we have received our output for the new test set, we should reprint our global results dataframe to compare the scores.

In [459…

```
global_results
```

Out[459]:

	Train (RMSE)	Test (RMSE)	K-Fold (RMSE)	Train (MAE)	Test (MAE)	K-Fold (MAE)	Tra (F
Linear Regression	160315.011772	159630.916296	160739.525422	105072.408566	103548.953582	105193.439280	0.3301
Decision Tree	160401.609735	166079.484810	166311.775871	106271.713235	109010.255814	109389.733497	0.3294
Random Forest	152388.209427	164287.337477	164007.105837	99190.389316	107299.859217	107454.689002	0.3947
Decision Tree (Optimal)	155436.116971	163490.950180	164104.819903	101028.537051	106606.934748	107289.248537	0.3702
Random Forest (Optimal)	152387.320515	164157.049694	163998.555364	99156.892383	107174.708902	107454.933341	0.3947

As expected, given that this is brand new data, the model has performed significantly worse than our initial trials. We can see that the RMSE is roughly 70% worse in most cases, the MAE is roughly 32% worse, and the R2 value is roughly 84% worse.

While this is not a great result, it should be noted that this is somewhat expected given that the model is seeing this rather large chunk of data for the very first time. It is concerning, however, that the R2 scores

in the new data are so low, as this would suggest that we are only seeing a marginal improvement gain from baseline performance when making these predictions.

Question 6: Conclusion

Overall, it is fair to say that while all the models that we trained performed within a similar range, with plain linear regression performing the best out of the three.

It was of interest to see that the margin of error was still very large, regardless of any optimisations that were attempted, and also that the model ultimately performed quite poorly on the test dataset.

It is entirely plausible that the size of errors are merely due to the massive range in prices in the housing market in Ireland. Another reason for why this could have occured is that there could simply have been too many flaws in the data (given that the RPPR does not make any checks as to the quality of the data). It would also have been helpful to perhaps train the models with more data to see if this could have improved results.

One point of note that was interesting was the fact that a low number of features, coupled with a simple linear regression model, ended up performing better than other, more complex models, showing that sometimes a simpler approach is more useful.

If given more time, it would have been interesting to try some other types of models, such as logistic regression (by converting our target feature into a classifier). We could then have used some other metrics, too, such as Receiver Operating Characterstics and Area Under ROC Curve.

Finally, it would also be interesting to perhaps try a feature selection algorithm, such as Recursive Feature Elimination. While we were largely satisfied with the feature selection in our model, it would have been interesting to see what features an algorithm would output as the solution.

END