National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Ian Patterson
Student ID: 18124917

School of Computing
National College of Ireland

Supervisor: Dr. Catherine Mulwa

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: …Ian Patterson……

Student ID: …18124917……..……

Programme: …MSC Data Analytics…… **Year:** ……2021…….

Module: …MSc Research Project……..……

Lecturer: ……Dr. Catherine Mulwa……..……

Submission Due Date: ……16/08/2021……..……

Project Title: Novel Genetic Algorithms for Optimization of House Price Prediction: USA

Word Count: ……4144…… **Page Count:** ……27……..……..……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: ……

Date: ……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Ian Patterson
Student ID: 18124917

1 Introduction

This Configuration Manual details the steps required to set-up the environment and reproduce the results of the investigation titled "Novel Genetic Algorithms for Optimization of House Price Prediction: USA". This includes a step-by-step guide in signing up for and configuring the Google Colaboratory environment, and explains at a function level, how each analysis was performed and how each result and visualisation was produced.

2 Hardware and Software Requirements

2.1 Hardware Description

Research was performed on a desktop computer with the following technological specifications

- **GPU**: Nvidia GeForce GTX 1080
- **CPU**: Intel i7-7700 CPU, 3.6 Ghz
- **RAM**: 16GB
- **System** Type: 64-bit Operating System, x64-based processor

However, it should be noted that almost all processing was performed through the Google Colaboratory system, which utilises GPUs and CPUs located in Google's data centres (Google Colab, 2020).

2.2 Software Description

- **Latex**: Nvidia GeForce GTX 1080
- **Google Chrome**: Intel i7-7700 CPU, 3.6 Ghz
- Google Colaboratory Environment
    - **GPU**: Unknown (depended on runtime)
    - Disk Space: 110 GB
    - **RAM**: 13GB

3 Environment Configuration

This section describes the steps required to activate and configure the Google Colab environment, including the collection of packages required for the analysis.

3.1 Google Account Creation

A Google account is required to use the Google Colab environment. If no Google account is available, a profile may be created using the following link: https://accounts.google.com/signup/v2/webcreateaccount?hl=en&flowName=GlifWebSignIn&flowEntry=SignUp
Once an account is setup and the user is logged in, the Colab site can be accessed at https://colab.research.google.com/

To create a new profile file, click "New Notebook" (Figure 1).

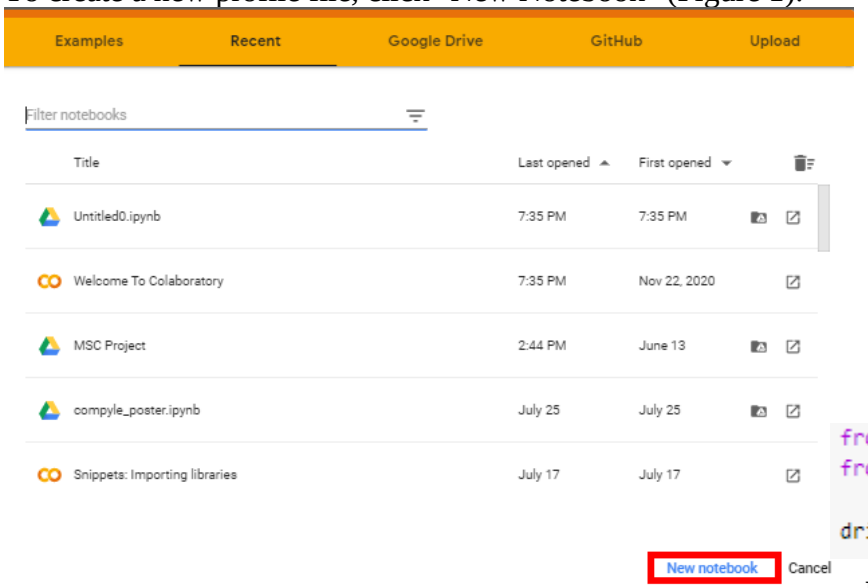

Figure 1 Click "New Notebook" to create a new project file.

```
from google.colab import files
from google.colab import drive

drive.mount('/content/gdrive/')
```

Figure 2. Code to mount Google Drive

In this new project, you may mount the Google Drive that comes with your account, using the snippet in Figure 2.
This will prompt you to authenticate your rights to use the Google Drive in the Colab environment. Visit the link given by this prompt, select your Google account and click "Sign-in". Copy the authentication token, return to Colab, and paste this token into the prompt (Figure 3).

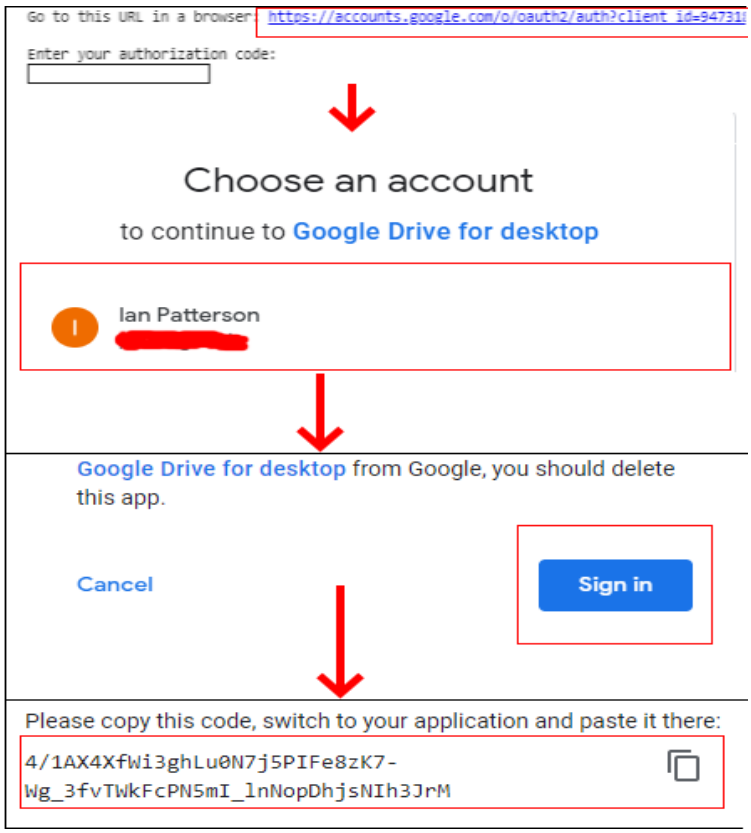

Figure 3. *Step-by-step guide to authentication and mounting of Google Drive in Colab environment*

Once mounted, you will have access to all files and folders in the Google Drive.

3.2 Import and Install Packages

Installing

Because the Google Colab environment comes with most Python packages pre-installed, only one additional package need be installed: 'scikit_posthocs' (Terpilowski, 2019). The pip install command given in Figure 4 installs this package.

```
[ ]  pip install scikit_posthocs
```

Figure 4. *Installing the scikit_posthocs package*

Importing

Once all packages are installed, the required packages are imported. These are described in Figure 5. Some packages are imported "as" abbreviations, and in some cases, only specific modules from packages are imported; these choices are for convenience's sake.

```
[ ]  import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
     import matplotlib
     import seaborn as sns
     import random
     import sklearn
     import time
     import xgboost as xgb
     import networkx as nx
     import sys
     import scipy
     import scikit_posthocs

     import matplotlib.patches as patches

     from datetime import datetime
     from sklearn import manifold

     from datetime import timedelta

     from sklearn.ensemble import RandomForestRegressor
     from sklearn.ensemble import GradientBoostingRegressor
     from sklearn.linear_model import LinearRegression
     from sklearn.tree import DecisionTreeRegressor
     from sklearn.cluster import KMeans

     from sklearn.model_selection import KFold
     from sklearn.metrics import mean_absolute_error
```

Figure 5. *List of packages to be imported for the analysis*

4 Data Acquisition and Ingestion

The data used in this investigation is sourced from Zillow's competition, hosted on Kaggle. The files described in Table 1 were downloaded from https://www.kaggle.com/c/zillow-prize-1/data (Kaggle, 2018). Full descriptions are available in Appendix 1.

Table 1. Data Files Acquired for Analysis

Filename	Description
Properties_2016.csv	CSV containing descriptions of properties on market in 2016
Properties_2017.csv	CSV containing descriptions of properties on market in 2017
Train_2016.csv	Csv containing list of transactions for the year 2016
Train_2017.csv	CSV containing list of transactions for the year 2017
Zillow_data_dictionary.xlsx	Data dictionary describing each variable, and categories within categorical variables.

Open Google Drive in the browser and add a new folder titled "MSC Files". Each of the CSVs above are to be uploaded to this folder, allowing for their ingestion by Colab.
Once uploaded, these files are read into Colab using the code in Figure 6.

```
[ ]  #Ingesting data

     properties_16 = pd.read_csv("/content/gdrive/MyDrive/MSC Files/properties_2016.csv")
     properties_17 = pd.read_csv("/content/gdrive/MyDrive/MSC Files/properties_2017.csv")
     transactions_16 = pd.read_csv("/content/gdrive/MyDrive/MSC Files/train_2016_v2.csv")
     transactions_17 = pd.read_csv("/content/gdrive/MyDrive/MSC Files/train_2017.csv")
```

Figure 6. *Reading in dataset files from Google Drive*

5 Pre-Processing and Transformation

5.1 Pre-Processing

The property datasets from 2016 and 2017 were first compared to ensure the same column names were present in both (Figure 7).

Each properties dataset was pre-processed separately due to limitations on disk space in Colab. The columns were renamed with the help of the Zillow data dictionary (Figure 8).

```
[ ]  #checking number of entries in each dataset
     print("number of properties in 2016 =",str(len(properties_16)))
     print("number of properties in 2017 =",str(len(properties_17)))

     #confirming whether columns are exactly the same across datasets
     print(properties_16.columns == properties_17.columns)
```

Figure 7. *Checking to ensure the 2016 and 2017 files have the same column names*

```
[ ]  #Renaming the columns of each data source
     properties_16.columns = ["parcel_id", "aircon_sys","arch_style","area_base","baths",
                              "beds","build_class","build_qual","calc_baths","decktype",
                              "area_living1","area_living_tot","area_living_fin","perim_living",
                              "area_tot", "area_living1_2","area_fin_unfin","fips_code","fires",
                              "full_baths","garages","area_garage","has_tub_spa","heating_sys",
                              "lat","long","area_lot","pools","area_pool","pooltype_ts",
                               "pooltype_and_ts","pooltype_no_ts","zoning_county","land_zone",
                               "zones","census_raw","city","county","hood","zipcode","rooms",
                              "storytype","three_qtr_baths","constructtype","units",
                              "area_patio","sheds","built","stories","has_fire","struct_val",
                              "tax_value","assessed_year","land_val","tax","tax_del",
                               "tax_del_year","census"]

     transactions_16.columns = ["parcel_id","logerror","trans_date"]
```

Figure 8. *Renaming the colours for ease of use*

```
properties_16.has_tub_spa.fillna(0,inplace = True)

#changing float64 to float32 as 64 not required, and prevented merging (memor
cols= properties_16.select_dtypes(include=[np.float64]).columns
properties_16[cols] = properties_16[cols].astype(np.float32)

convert_dict = {
    "parcel_id": str,
    "aircon_sys": str,
    "arch_style": str,
    "build_class": str,
    "decktype": str,
    "fips_code": str,
    "has_tub_spa": int,
    "heating_sys": str,
    "pooltype_ts": str,
    "pooltype_and_ts": str,
    "pooltype_no_ts": str,
    "land_zone": str,
    "zones": str,
    "census": str,
    "census_raw": str,
    "city": str,
    "county": str,
    "hood": str,
    "zipcode": str,
    "storytype": str,
    "constructtype": str,
}

properties_16 = properties_16.astype(convert_dict)
```

Figure 9. *Changing data types to match the appropriate form for each variable*

Data types were then changed to the appropriate form, and float64 and int64 were converted to float 32 and int 32 to reduce impact on disk space (Figure 9).

The transactions 2016 dataset was similarly processed and then merged with the properties 2016 dataset, through an inner join, using parcel_id as the key (Figure 10).

```
[ ]  #changing the datatypes in the transactions dataset
     transactions_16.parcel_id = transactions_16.parcel_id.astype(str)

     #merging of the descriptive data with the training/target data
     analysis_16 = transactions_16.merge(properties_16, how = "inner", on = "parcel_id")
     print(len(analysis_16))
     print(analysis_16.columns)

     #deleting the precursors due to disk space limits
     del properties_16
     del transactions_16
```

Figure 10. *Merging the transaction and properties datasets*

To investigate missing values, the dataframe was looped over such that the sum of the "NA" values for each column in the feature set was printed. The seaborn package was also used to create a heatmap representation of the missing values across variables (Figure 11).

```
#Initial View of Missing Values
for j in range(len(analysis.columns)):
    print("column {} is missing {}".format(analysis.columns[j],
                                           analysis.iloc[:,j].isnull().sum()))

sns.heatmap(analysis.isnull(),cbar = False)
```

Figure 11. *Checking and visualising the missing data across the dataset*

5.2 Feature Engineering

To create a version of the transaction date that could be incorporated into machine learning and matrix multiplication-based models, the date was converted to a "Days since 0" field, named "date_dif" (Figure 12). Values in the binary categorical "tax_del" field were also changed such that "Y" was replaced with 1 and "N" or missing was replaced with 0 Figure 13). Redundant columns were dropped from the dataset (Figure 14).

```
[ ] #creating a date difference column to allow for treatment as a numeric variable
    analysis["trans_date"] = analysis["trans_date"].apply(lambda x: datetime.strptime(x, "%Y-%m-%d"))
    analysis["date_dif"] = analysis["trans_date"].apply(lambda x: x- analysis["trans_date"][0]).dt.days
```

Figure 12. *Creating a date field appropriate for data mining*

```
[ ] #cursory replacement of values
    analysis["tax_del"].replace("Y",1,inplace= True)
    analysis["tax_del"].fillna(0,inplace = True)
    analysis["tax_del"]= analysis["tax_del"].astype(int)
```

Figure 13. *Editing the codes used for the "tax_del" variable*

```
[ ] #dropping useless or repeated columns
    analysis.drop(columns = ["parcel_id", "census", "census_raw",
                             "zipcode", "zones","hood","city"], inplace = True)
```

Figure 14. *Dropping redundant columns*

Missing values were then replaced either through median imputation or replaced with 0s. Other binary categorical variables were converted such that codes were either 0 or 1, rather than the inconsistent codes used across variables (Figure 15).

```
#filling missing values
analysis["tax_del_year"].fillna(np.median(analysis["tax_del_year"]),inplace = True)
analysis["stories"].fillna(np.median(analysis["stories"]),inplace = True)
analysis["built"].fillna(np.median(analysis["built"][analysis["built"] != 0]),inplace = True)
analysis.loc[analysis["built"] == 0, "built"] = np.median(analysis.loc[analysis["built"] != 0, "built"])
analysis["lat"].fillna(np.median(analysis.loc[~(analysis["lat"].isna()),"lat"]),inplace = True)
analysis["long"].fillna(np.median(analysis.loc[~(analysis["long"].isna()),"long"]),inplace = True)


for i in analysis.columns:
  analysis[i].fillna(0, inplace = True)

analysis["build_class"].replace(to_replace = {"nan" :"0",
                                              "4.0": "1"}, inplace = True)
analysis["build_class"] = analysis["build_class"].astype(np.float32)

analysis["decktype"].replace(to_replace = {"nan" :"0",
                                           "66.0": "1"}, inplace = True)
analysis["decktype"] = analysis["decktype"].astype(np.float32)

analysis["pooltype_ts"].replace(to_replace = {"nan" :"0",
                                              "7.0": "1"}, inplace = True)
analysis["pooltype_ts"] = analysis["pooltype_ts"].astype(np.float32)

analysis["pooltype_and_ts"].replace(to_replace = {"nan" :"0",
                                              "7": "1"}, inplace = True)
analysis["pooltype_and_ts"] = analysis["pooltype_and_ts"].astype(np.float32)

analysis["pooltype_no_ts"].replace(to_replace = {"nan" :"0",
                                              "7": "1"}, inplace = True)
analysis["pooltype_no_ts"] = analysis["pooltype_no_ts"].astype(np.float32)

analysis["storytype"].replace(to_replace = {"nan" :"0",
                                              "7": "1"}, inplace = True)
analysis["storytype"] = analysis["storytype"].astype(np.float32)

analysis["has_fire"].replace(to_replace = {"0" :"0",
                                              "True": "1"}, inplace = True)
analysis["has_fire"] = analysis["storytype"].astype(np.float32)
```

Figure 15. *Imputing values and otherwise harmonizing the codes used for binary categorical variables*

6 Exploratory Analysis

6.1 Descriptive Statistics and Histograms

Matplotlib and SciPy were used to create descriptive statistics, perform normality tests, and generate histograms describing the target variable ("logerror"). The first two histograms were built off the total target variable, whereas the second two histograms zoomed in on the values within the interquartile range (Figure 16). Both histograms are available in the technical report (TR Fig. 3).

```
#Kolmogorov Smirnov test used to test the normality of the target variable
#Kolmogorov Smirnov compares the distribution of the given variable vs
#another distribution. It is a normality test when the comparative dist is norm
print(scipy.stats.kstest(rvs = analysis.logerror, cdf = "norm"))
#deemed not normal, histograms below show this is likely due to extreme
#kurtosis of errors. Tightly packed around 0/median
#therefore, later statistical tests will require non-parametric alternatives

#Histogram of log error
fig, axes = plt.subplots(nrows = 2, ncols = 2,figsize = (22,16))
fig.tight_layout(pad = 6)
plt.subplot(2,2,1)
bins = np.round(np.arange(-5.5, 5.5, .1),3)
plt.hist(analysis.logerror, bins = bins)
plt.xticks(np.arange(-5.5,5.5,0.5),
           rotation = 60, size =16)
plt.yticks(size = 16)
plt.ylabel("Count of LogError", size =16)
plt.title("Total Distribution of Errors", size =16)

plt.subplot(2,2,2)
bins = np.round(np.arange(0,5.6,0.1),3)
plt.hist(np.abs(analysis.logerror), bins = bins)
plt.xticks(np.arange(0,5.6,0.5),
           rotation = 60, size =16)
plt.yticks(size = 16)
plt.ylabel("Count of Absolute LogError", size =16)
plt.title("Total Distribution of Absolute Errors", size =16)

#if we zoom in to the interquartile range
iqr_analysis = analysis[(analysis["logerror"] >=-0.254) &
                        (analysis["logerror"] <= 0.393)]

#fig, axes = plt.subplots(2,2,figsize = (12,4))
#fig.tight_layout(pad = 5)
plt.subplot(2,2,3)
bins = np.round(np.arange(-.25, .54, .01),3)
plt.hist(iqr_analysis.logerror, bins = bins)
plt.yticks(size = 16)
plt.xticks(np.arange(-.25,.54,0.05),
           rotation = 60, size =16)
plt.ylabel("Count of LogError", size =16)
plt.title("Distribution of Errors within the Interquartile Range", size =16)

plt.subplot(2,2,4)
bins = np.round(np.arange(0,.41, .01),3)
plt.hist(np.abs(iqr_analysis.logerror), bins = bins)
plt.yticks(size = 16)
plt.xticks(np.arange(0,.54,0.05),
           rotation = 60, size =16)
plt.ylabel("Count of Absolute LogError", size =16)
plt.title("Distribution of Absolute Errors within the Interquartile Range", size =16)
plt.show()

#descriptive statistics
print(analysis.logerror.describe())
```

Figure 16. *Code for descriptive statistics, normality test and histogram generation*

6.2 Correlation Analysis

The code in Figures 17-19 comprise a single for loop. This looped through all variables and performed tests appropriate to the data type. Numeric and binary categorical variables underwent Spearman's Rho correlation produced scatter plots (Figure 17). Multi-class categorical variables underwent Kruskal Wallis tests and were plotted on boxplots (Figure 18). Where significant Kruskal Wallis results were found ($p < 0.05$), post-hoc Dunn tests with Bonferroni correction were performed to identify differences within the variable. Results of these post-hoc Dunn tests added as annotations on relevant graphs (Figure 19). Plots for the tests that yielded significant results are available in the technical report (TR Fig. 4)

```
#correlation of numeric values with log error
numeric_types = ["float32", "float64", "int64", "int32"]
categorical_types = ["object"]

for i in analysis.columns:

  if analysis[i].dtype in numeric_types:
    fig, ax = plt.subplots()
    plt.title("Plot of LogError against " + str(i), size =16)
    plt.ylabel(str(i), size =16)
    plt.yticks(size = 16)
    plt.xlabel("LogError", size =16)
    plt.scatter(analysis.logerror, analysis[i])
    plt.xticks(size =16)

    #Spearman correlation used because logerror failed normality test earlier
    corr_s = scipy.stats.spearmanr(analysis.logerror,
                                   analysis[i])
    corr_s = np.round(corr_s, 4)

    ax.text(x = 6, y = label_loc(analysis[i]),
            s = "Spearmann Rho: "+str(corr_s[0])+"\nP-Value: "+str(corr_s[1]),
            bbox=dict(facecolor='tab:orange', alpha=1), fontsize = 16)

    plt.show()
```

Figure 17. *Spearman's Rho correlation, used for numerical and binary categorical variables*

```
else:
  data = []
  for j in range(len(labels)):
    vector = analysis.loc[analysis[i]==labels[j], "logerror"]
    data.append(vector)

  fig, ax = plt.subplots()
  plt.title("Boxplot of LogError by Category of "+str(i), size =16)
  plt.ylabel("LogError", size =16)
  plt.xlabel(str(i), size =16)
  plt.xticks(size =16)
  plt.yticks(size = 16)
  plt.boxplot(x = data, labels = labels)

  krusk = scipy.stats.kruskal(*data)
  krusk = np.round(krusk, 3)

  ax.text(x = len(labels)+0.6, y = 4,
        s = "Kruskal Wallis: "+str(krusk[0])+"\nP-Value: "+str(krusk[1]),
        bbox=dict(facecolor='tab:orange', alpha=1), fontsize = 16)
```

Figure 18. *Kruskal Wallis tests, used for multi-class categorical variables*

```
if krusk[1] <= 0.05:
  post_hoc = scikit_posthocs.posthoc_dunn(data, p_adjust = "bonferroni")
  count = 0
  tot_string = str()
  for row in range(len(post_hoc-1)):
    for col in range(row,len(post_hoc)-1):
      comparison = str(i)+" "+str(labels[row])+" - "+str(i)+" "+labels[col]
      value = np.round(post_hoc.iloc[row,col],6)

      if value <= 0.05:
        if count ==0:
          substring = "Comparison: "+comparison+"\nP-Value: "+str(value)
        else:
          substring = "\n\nComparison: "+comparison+"\nP-Value: "+str(value)
        count += 1
        tot_string = tot_string+substring

  ax.text(x = len(labels)+0.6, y = -4,s = tot_string,
          bbox=dict(facecolor='tab:orange', alpha=1), fontsize = 16)

plt.show()
```

Figure 19. *Post-hoc Dunn tests triggered by significant p-values in the Kruskal Wallis tests*

The label_loc function (Figure 20) standardised the location of the label on each of the plots generated by the code above.

```
[ ]  def label_loc(array):
       max = np.max(array)
       min = np.min(array)
       dif = max-min
       adjustment = 0.90*dif
       new_y = min+adjustment

       return new_y
```

Figure 20. *Function that helps position chart annotations consistently*

6.3 Mapping of Transactions

The latitude and longitude were divided by 1 million to convert the raw values into the standard format used in mapping (Figure 21).

```
[ ]  #Investigating the Distribution of prediction errors across Los Angeles
     analysis["lat"] = analysis["lat"]/1e6
     analysis["long"] = analysis["long"]/1e6
```

Figure 21. *Dividing latitude and longitude values to match mapping standards*

The borders of the map were then extracted from the dataset based on the minimum and maximum of the latitudes and longitudes (Figure 22).

```
[ ]  left = np.min(analysis.long)
     right = np.max(analysis.long)
     bottom = np.min(analysis.lat)
     top = np.max(analysis.lat)
```

```
[ ]  print(left, right, bottom, top)
```

Figure 22. *Boundaries of the location calculated from latitudes and longitudes*

These coordinates were then input into http://Openstreetmap.org (OpenStreetMap, 2021.) to generate a map for the corresponding area. A screenshot of the map was downloaded and then uploaded to the MSC files folder on Google Drive (Figure 23).

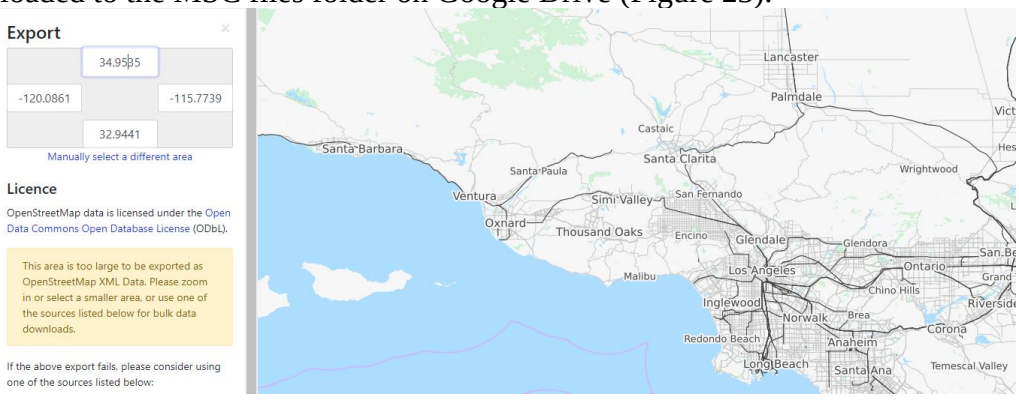

Figure 23. *Openstreet website where map image file is sourced*

Once saved, it was read into the Colab environment using the snippet in Figure 24.

```
[ ]  map_file = plt.imread("/content/gdrive/MyDrive/MSC Files/test_image.PNG")
```

Figure 24. *Reading in the map image file*

Matplotlib and a distribution-based colour map from seaborn were used to highlight the distribution of large errors in price prediction across the analysis are (Los Angeles). The matplotlib package "patches" was used to add a red circle around an area of dense errors (Figure 25). The plot generated by this code is available in the technical report (TR Fig. 6).

```
[30] #Show the distributions of the transactions across geographies, can potentially
     #highlight on a map of the USA depending on how well-distrubted they are.

     fig, ax = plt.subplots()
     fig.set_figwidth(16)
     fig.set_figheight(12)
     plt.xlabel("Latitude", size = 16)
     plt.ylabel("Longitude", size = 16)
     plt.title("Geographic Distribution of Price Prediction Error", size = 16)
     plt.yticks(size = 16)
     plt.xticks(size = 16)

     norm = matplotlib.colors.Normalize(-1,1)
     colors = [[norm(-1), "white"],
               [norm(-0.99), "lightblue"],
               [norm( 0), "blue"],
               [norm( 1.0), "darkblue"]]
     cmap = matplotlib.colors.LinearSegmentedColormap.from_list("", colors)

     points = plt.scatter(analysis.long, analysis.lat, s = 10, c=np.abs(analysis.logerror),
                          cmap=cmap, label = "Don't Fit Criteria", alpha = 1,zorder = 1)
     fig.colorbar(points)

     ax.legend(["Fit Criteria"])

     leg = ax.get_legend()
     leg.legendHandles[0].set_color("green")

     rect = patches.Circle(xy = (-117.89,33.77), radius = 0.2,
                           facecolor = "none", edgecolor = "r")

     ax.text(-118.15, 33.54, "Laguna \n&\nHuntington", bbox=dict(facecolor='tab:orange', alpha=1),
             fontsize = 12, va = "center", ha = "center")
     ax.add_patch(rect)
     plt.imshow(map_file,zorder = 0,
                extent = [np.min(analysis.long),
                          np.max(analysis.long),
                          np.min(analysis.lat),
                          np.max(analysis.lat)])

     plt.show()
```

Figure 25. *Code that produces a map showing the density and magnitude of prediction errors across the map*

6.4 Timeseries Analysis of Transactions

For the time series analysis of transactions, pandas was used to summarise the price errors and count of transactions per month, and matplotlib was used to plot these on a dual-axis line graph (Figure 26). The plot generated by this code is available in the technical report (TR Fig. 5).

```
#error of Zestimate vs time (month/date etc.)

#Create
min_date = min(analysis.trans_date)
max_date = max(analysis.trans_date)

f, ax = plt.subplots(1, figsize = (16,4))

plt.title("Median Error and Count of Properties Sold Over Time", size = 16)
ax.plot(analysis.groupby(analysis["trans_date"].dt.strftime("%Y-%m"))["logerror"].median(),
       color = "tab:blue", linewidth = 4, label = "Median Error")

ax2 = ax.twinx()
ax2.plot(analysis.groupby(analysis["trans_date"].dt.strftime("%Y-%m"))["logerror"].count(),
        color = "tab:red", ls = "--", linewidth = 1, label = "Houses Sold")
ax.set_ylim(ymin = 0)
ax.legend(loc = "upper left", fontsize = "x-large")
ax2.legend(loc = "lower right", fontsize = "x-large")

dates= pd.date_range(min_date, max_date, freq='MS').strftime("%Y-%m").tolist()
date_ticks = [dates[i] for i in range(0,len(dates)) if i%2 == 0]
ax.xaxis.set(ticks = date_ticks,
             ticklabels = date_ticks)
ax.tick_params(axis = 'x', labelsize = 16, rotation = 45)
ax.tick_params(axis = "y", labelsize = 16)
ax2.tick_params(axis = "y", labelsize = 16)

plt.show()
```

Figure 26. *Code used to plot time series analysis of transactions*

Following creation of this graph, the "trans_date" field was deleted, as it was only needed for this graph. The "date_dif" field generated in the feature engineering section replaced it for the machine learning component (Figure 27).

```
[ ] #dropping the date column, as it's no longer needed (replaced by date dif)
    analysis.drop(columns = "trans_date", inplace = True)
```

Figure 27. *Dropping of the old transaction date field*

7 Initial Data Mining

Two functions were created for the modelling phase: make_dummies and predictor.
Make_dummies was essentially a slightly more customised version of the standard pandas.get_dummies function. (Figure 28)

```
def make_dummies(dataframe):

    nums = dataframe._get_numeric_data().columns
    numerics = dataframe.loc[:,nums]
    numerics.fillna(0, inplace = True)
    cats =  dataframe.columns[~dataframe.columns.isin(nums)]

    dummies = pd.get_dummies(dataframe.loc[:,cats], drop_first = True)
    dummies.fillna("na",inplace = True)

    dataframe = pd.concat([numerics, dummies], axis = 1)

    return dataframe
```

Figure 28. *Function for creating dummies when applicable*

The predictor function receives the dataframe, the number of cross-validation stages (K), and the model. This splits the data into train and test dataframes. For each train-test split, the models were fit to the training data and tested against the test. The evaluation metric was mean absolute error (MAE). Following the K iterations, the MAE for the model was calculated by taking the average MAE across the iterations. The XGBoost model required some additional parameters and processing steps such as use of DMatrix rather than dataframes (Figure 29).

```
def predictor(model, dataframe, K):

    start_time = time.time()

    model_df = make_dummies(dataframe)
    train_df = model_df.drop("logerror", axis = 1)
    test_df = model_df["logerror"].values

    maes = list()
    random.seed(42)
    folded = KFold(n_splits = 5, shuffle = True)
    for train_index, test_index in folded.split(model_df):

        x_train = train_df.loc[train_index,:]
        x_test = train_df.loc[test_index,:]

        y_train = test_df[train_index]
        y_test = test_df[test_index]

        if model == xgb:
          xg_train = xgb.DMatrix(x_train, label = y_train)
          xg_test = xgb.DMatrix(x_test, label = y_test)

          param = {}
          param["objective"] = "reg:squarederror"
          param["eval_metric"] = "mae"
          param["booster"] = "gbtree"
          # scale weight of positive examples
          param['nthread'] = 4
          param['gpu_id'] = 0
          param['tree_method'] = 'gpu_hist'
          param["num_feature"] = 52

          # watchlist allows us to monitor the evaluation result on all data in the list

          xgb_r = xgb.train(params = param, dtrain = xg_train,
                            num_boost_round = 10)
          preds = xgb_r.predict(xg_test)
          mae = mean_absolute_error(y_test, preds)
          maes.append(mae)


        else:
          model.fit(x_train, y_train)
          preds = model.predict(x_test)
          mae = mean_absolute_error(y_test, preds)
          maes.append(mae)

    mae_avg = np.mean(maes)
    time_req = time.time() - start_time

    return mae_avg, time_req
```

Figure 29. *Function for applying machine learning model to dataset and evaluating performance.*

A loop through a list of models was used to generate and tabulate the results from the unoptimized models (Figure 30).

```
[ ]  models = [xgb, RandomForestRegressor(), LinearRegression(),
               DecisionTreeRegressor()]

     names = ["XGBoost","Random Forest","Linear Regression","Decision Tree"]

     results_df = pd.DataFrame()
     for i in range(len(models)):


       mae, time_req = predictor(models[i], analysis, 5)
       results_df = results_df.append(pd.DataFrame({
           "model":[names[i]],
           "mae":[mae],
           "time":[time_req]
       }), ignore_index = True)

       print(results_df)
```

Figure 30. *Code for looping through list of models to be applied*

The results of this modelling were then visualised using matplotlib. A bar chart was generated to compare the MAE obtained by each model, a bar chart was created to compare the time required by each model, and a third bar chart was used to compare the time required by each model with Random Forest excluded (to allow for easier comparison of the comparable models) (Figure 31). The plot generated by this code is available in the technical report (TR Fig. 7).

```
[ ]  #Visualise comparison of mae for each
     #bar graph of all 4
     fig, ax = plt.subplots(1,3, figsize = (24,6))
     fig.tight_layout(pad = 5)

     plt.subplot(1,3,1)
     plt.title("MAE by Model", size = 16)
     plt.ylabel("MAE", size = 16)
     plt.xticks(size = 16, rotation = 20)
     plt.yticks(size = 16)
     plt.xlabel("Model", size = 16)
     plt.bar(x = results_df["model"],
             height = results_df["mae"],
             color = ["red", "green", "blue", "orange"])

     #second bar graph for time required for each
     plt.subplot(1,3,2)
     plt.title("Time Required by Model", size = 16)
     plt.ylabel("Seconds", size = 16)
     plt.xticks(size = 16, rotation = 20)
     plt.yticks(size = 16)
     plt.xlabel("Model", size = 16)
     plt.bar(x = results_df["model"],
             height = results_df["time"],
             color = ["red", "green", "blue", "orange"])

     plt.subplot(1,3,3)
     plt.title("Time Required by Model Excluding Random Forest", size = 16)
     plt.ylabel("Seconds", size = 16)
     plt.yticks(size = 16)
     plt.xticks(size = 16, rotation = 20)
     plt.xlabel("Model", size = 16)
     plt.bar(x = results_df["model"][results_df["model"] != "Random Forest"],
             height = results_df["time"][results_df["model"] != "Random Forest"],
             color = ["red", "blue", "orange"])

     plt.show()
```

Figure 31. *Visualisations for unoptimized modelling*

8 Genetic Algorithm Coding

As there were four types of genetic algorithms used for this analysis, there were multiple operations shared among algorithms. Each operation (function) is described below.

Create Chromosome

In the feature selection algorithm, chromosomes were vectors of 1's and 0's, representing inclusion or exclusion of a feature, respectively. Chromosomes for the Travelling Salesman GA were vectors of integers from 0 to the number of features. Therefore, the create_chromosome needed to be able to switch between these modes; the "mode" argument was used (Figure 32).

```
[ ]  def create_chromosome(mode = "standard", col_num = None):

         if mode == "standard":
           chromosome = np.random.choice([0,1], size = 52)

         elif mode == "TSP":
           chromosome = random.sample(range(0, col_num), col_num)

         return chromosome
```

Figure 32. *Function that creates a chromosome*

Change Chromosome

This function was a simple method of changing the feature selection chromosomes from 1's and 0's to Trues and False, to allow for filtering of the dataframe for the variables represented by 1s in the vector (Figure 33).

```
[ ]  def change_chromo(chromosome, to = bool):

         new_chromo = chromosome.astype(to)

         return new_chromo
```

Figure 33. *Function that converts an integer vector to a bool vector*

Initialise_pop

This function was required at the start of each genetic algorithm, to generate the initial population of chromosomes to undergo evolution (in all types of GA). This function allows for creation of populations of any desired size. The mode argument determines which type of chromosome creation is used (Figure 34).

```
[ ]  def initialise_pop(pop_size, mode = "standard", col_num = None):

         population = list()
         for i in range(pop_size):
             chromo = create_chromosome(mode = mode, col_num = col_num)
             population.append(chromo)

         return population
```

Figure 34. *Function that initialises a population of chromosomes*

Model_chromos

This function allowed for the application of the chromosome to the machine learning problem. It applies the chromosomes to machine learning model and produces results that allow for comparison of fitness (Figure 35).

```
def model_chromos(model, dataframe, K, population):

    chromo_results = list()

    for i in range(len(population)):

        booled = change_chromo(population[i])
        chromo_cols = dataframe.columns[dataframe.columns!="logerror"][booled]
        tot_cols = np.append(chromo_cols, "logerror")

        model_df = dataframe.loc[:, tot_cols]

        #run the train/test split and produce MAE values
        avg_mae, time_req = predictor(model, model_df, 5)

        #create other metrics for chromo scoring/fitness
        rounded_mae = np.round(avg_mae, 3)
        #number of features is number of Trues in the chromosome
        n_features = booled.sum()
        #reverted chromosome
        chromo = change_chromo(booled, int)

        chromo_array = [chromo, avg_mae, rounded_mae, n_features, time_req]

        chromo_results.append(chromo_array)

    #sorting by x2 and x3 places low mae and low feature chromosomes
    #at the top of the population.
    return sorted(chromo_results, key = lambda x: (x[2],x[3],x[1]))
```

Figure 35. *Function that applies the chromosome to modelling problem*

Hamming

The hamming distance function is a method of comparing two vectors and calculating the difference between each (how many changes would need to be made to one vector such that it matched the other) (Macleod, 1993). This was used to determine the overall diversity of the population (Figure 36).

```
[ ]  def hamming(chromo1, chromo2):
         assert len(chromo1) == len(chromo2)
         return sum(b1 != b2 for b1, b2 in zip(str(chromo1), str(chromo2)))
```

Figure 36. *Function that calculates the hamming distance between two chromosomes (measures diversity)*

Select_mates

This function was used to select the chromosomes from the population who would be used to reproduction. The selection probability was based on rank selection. The probability of each chromosome being selected was determined by its ranked position in the population (best = 1, worst = 30) (Figure 37).

```
[ ]  def select_mates(chromo_results, pairs):

        #calculates the selection probabilities for each chromosome
        just_chromos = [i[0] for i in chromo_results]
        score_list = [iter[1] for iter in chromo_results]

        #select_probs = [score/np.sum(score_list) for score in score_list]

        #rank selection
        select_probs = []
        nums = range(len(just_chromos))
        for i in nums:
          select_probs.append((len(just_chromos)-1-i)/sum(nums))
        mating_index = np.random.choice(nums, size = 2*pairs, p =select_probs, replace = False)

        mating_chromos = np.array(just_chromos)[mating_index]


        return mating_chromos, just_chromos
```

Figure 37. *Function that selects parents for mating, from the population of chromosomes*

Pm_crossover

The PM_crossover function is one of two crossover methods used for the genetic algorithms. Specifically, it is used only for the TSGA, whereas the crossover function is used for the feature selection ga.

This function takes in two parent chromosomes from the select_mates function, and two positions (explained in the crossover function). It works by crossing the two chromosomes, and where any repetitions of numbers appears, swaps them with the corresponding position in the other chromosome. An explanation is available here (Figure 38).

```
def pm_crossover(chromo1, chromo2, pos1, pos2):
  child = [None]*len(chromo1)

  child[pos1:pos2] = chromo1[pos1:pos2]

  for i, x in enumerate(chromo2[pos1:pos2]):
    i += pos1
    if x not in child:
      while child[i] != None:
        i = chromo2.index(chromo1[i])
      child[i] = x

  for i, x in enumerate(child):
    if x == None:
      child[i] = chromo2[i]

  return child
```

Figure 38. *Function that crosses over chromosomes in the Travelling Salesman-type Genetic Algorithm, using the partial-map operator*

Crossover

The crossover function include pm_crossover, and the regular crossover. The mode is determined by the "mode" argument. Pos 1 (position 1) and Pos 2 are selected at random, as being the points at which the chromosomes will crossover. With two crossover points, there are three chunks of chromosome. Each chunk is exchanged and stuck together to form the children (c1 and c2). The Partial Map Operator was used, as it has been found to be the most effective operator for Travelling Salesman Genetic Algorithms (Hussain et al., 2017), each child is specified by the pm_crossover function described above (Figure 39).

```
[ ]  def crossover(chromo1, chromo2, mode = "Standard"):

       assert len(chromo1) == len(chromo2)
       pos1=random.randint(1, len(chromo1)-2)
       pos2=random.randint(pos1+1, len(chromo2)-1)

       if mode == "Standard":
         c1 = np.append(np.append(chromo1[0:pos1],chromo2[pos1:pos2]), chromo1[pos2:])
         c2 = np.append(np.append(chromo1[0:pos1],chromo1[pos1:pos2]),chromo2[pos2:])

       elif mode == "pmx":
         c1 = pm_crossover(chromo1, chromo2, pos1, pos2)
         c2 = pm_crossover(chromo2, chromo1, pos1, pos2)


       return c1, c2
```

Figure 39. *Function for crossing over of chromosomes*

Split_chromos

The split_chromos function is uniquely applied in the Multi-Chromosomal Genetic Algorithm rubric. For MCGA, each chromosome is further split into sub chromosomes based on the gene clusters determined by k means clustering (as will be seen later). This function essentially creates mini chromosomes, which each behave the same as large chromosomes (Figure 40).

```
[ ]  def split_chromos(chromo, cluster_df):
       cluster_num = len(cluster_df.cluster.unique())
       subc = []
       for i in range(cluster_num):

         subc.append([chromo[j] for j in cluster_df[cluster_df["cluster"]==i].order])

       return subc
```

Figure 40. *Function for splitting chromosomes into sub-chromosomes, used for the Multi-Chromosomal Genetic Algorithm*

Flip

The flip function is a component of the mutation function. It flips a 1 to a 0 when activated (Figure 41).

```
[ ]  #flips a 1 to 0 vice versa. only applied if mutation triggered
     def flip(x):
         return 1 if x==0 else 0
```

Figure 41. *Function for flipping a gene to the opposite value, in the feature selection rubric*

Mutate

The mutate function again comes in two forms. For the feature selection algorithm, it flips 1's to 0's and 0's to 1 based on a mutation rate (each gene has a x chance to flip, where x = the mutation rate). For the TSP GA, instead of flipping values from 1 to 0, the gene takes on a new random value within the rage (defined by number of features). Because this is likely to cause a single value to be included at two positions in the route (not allowed), the second occurrence is changed to the previous value. That is, in the vector [1,2,3,4], if the "2" changed to 4, the original 4 changes to 2: [1,2,3,4] → [1,4,3,4] → [1,4,3,2] (Figure 42).

```
[ ]  def mutate(chromo, mut_rate, mode = "standard"):

         chromo = list(chromo).copy()

         if mode == "standard":

           for i in range(len(chromo)):
               x = random.random()

               if x < mut_rate:
                   chromo[i] = flip(chromo[i])

         elif mode == "TSP":
           for i in range(len(chromo)):
             x = random.random()
             if x < mut_rate:
               i1 = chromo.index(chromo[i])
               new_val = random.randrange(0, len(chromo))
               i2 = chromo.index(new_val)

               chromo[i1], chromo[i2] = chromo[i2], chromo[i1]
         return chromo
```

Figure 42. *Function for mutating the chromosome, based on a mutation rate and the mode*

Cross_mute

The cross_mute function essentially combines the crossover and mutation steps. As these functions allow for multiple parent pairs, a for loop cycles through each pair of parents. The Multi-Chromo mode requires an additional for loop because the split chromos function turns each parental chromosome into an array of sub chromosomes. Each sub chromosome is then crossover and mutated (Figure 43).

In both methods, the new chromosomes are added to an array called "children"

```
[ ]  def cross_mute(mating_chromos, pairs, cluster_df,  mut_rate, mode = "Standard"):

            children = list()


            if mode == "Multi_Chromo":

              mating_subcs = [split_chromos(chromo,cluster_df) for chromo in mating_chromos]

              for j in range(pairs+1):
                if j%2 == 0:
                  c1 = mating_subcs[j]
                  c2 = mating_subcs[j+1]


                  newc1 = []
                  newc2 = []
                  for x in range(len(c1)):
                    subc1 = c1[x]
                    subc2 = c2[x]

                    newsubc1,newsubc2 = crossover(subc1,subc2)
                    newsubc1 = mutate(newsubc1, mut_rate)
                    newsubc2 = mutate(newsubc2, mut_rate)

                    newc1.append(newsubc1)
                    newc2.append(newsubc2)

                  newc1 = [gene for subchromo in newc1 for gene in subchromo]
                  newc2 = [gene for subchromo in newc2 for gene in subchromo]

                  children.append(newc1)
                  children.append(newc2)

            else:

              for j in range(pairs+1):
                  if j%2 == 0:
                      newc1,newc2 = crossover(mating_chromos[j],mating_chromos[j+1])

                      newc1 = mutate(newc1, mut_rate)
                      newc2 = mutate(newc2, mut_rate)

                      children.append(newc1)
                      children.append(newc2)

            return np.array(children)
```

Figure 43. *Combination of the crossover and mutation processes*

Eval_chromo_results

Before being added back to the population of chromosomes, the performance of each child is assessed. Following that modelling however, the eval_chromo_results function takes those results and puts them in a results vector, to allow for cleaner and easier sorting.

To assess whether the new children are better or worse than their parents or the other chromosomes in the population, they are applied to the modelling (Figure 44).

```
[ ]  def eval_chromo_results(chromo_results):
         avg_mae = np.mean([result[1] for result in chromo_results])
         var_mae = np.var([result[1] for result in chromo_results])

         hammings = [hamming(chromo_results[i], chromo_results[i+1])
                                 for i in range(len(chromo_results)- 1)]
         diversity = np.mean(hammings)

         return avg_mae, var_mae, diversity
```

Figure 44. *Function for evaluating the performance of new chromosomes when applied to the modelling problem*

Route distance

The Route distance function is unique to the Travelling Salesman problem. As will be seen later, the TSP involves the optimisation of a route through a network. Therefore, this function takes in a distance matrix, and calculates the sum total distance required to traverse a specific route through the matrix network. The route is determined by the chromosome. This function

essentially determines the fitness of the chromosome, with shorter route distances corresponding to more fit chromosomes (Figure 45).

```
[ ]  def route_distance(chromosome, dist_mat):

       route_dist = 0
       for i in range(len(chromosome)-1):
         i1 = chromosome[i]
         i2 = chromosome[i+1]
         section_dist = dist_mat.iloc[i1,i2]
         route_dist += section_dist

       return route_dist
```

Figure 45. *Function for calculating the total distance of a route through the network*

FS_GA

The FS_GA function sticks the above functions together to create a single algorithm for the feature selection variant of the genetic algorithm. It is used for the Standard, Co-Location and Multi-Chromosomal Genetic Algorithm rubrics (Figure 46).

```
def FS_GA(model, dataframe, K, pop_size, gens, pairs, mut_rate, cross_rate,
          mode = "standard", cluster_df = None):

    #initialise population of chromosomes
    start_population = initialise_pop(pop_size)
    gen_pop = start_population.copy()

    #model the initial chromosomes and create generational results dataframe
    chromo_results = model_chromos(model, dataframe, K, gen_pop)
    avg_mae, var_mae, diversity = eval_chromo_results(chromo_results)

    gen_df = pd.DataFrame({"gen": 0,
                           "top_chromo": [chromo_results[0][0]],
                           "top_mae": [chromo_results[0][2]],
                           "avg_mae": [avg_mae],
                           "var_mae": [var_mae],
                           "top_features": [chromo_results[0][3]],
                           "diversity": [diversity]},)

    total_results = chromo_results.copy()
    uncut_results = chromo_results.copy()
    #looping for addition and evaluation of children
    for i in range(gens):
        this_gen = i+1
        print(this_gen)

        mating_chromos, next_pop = select_mates(total_results, pairs)
        children = list()
        children = cross_mute(mating_chromos, pairs, cluster_df, mut_rate, mode)
        children_results = model_chromos(model, dataframe, K, children)
        avg_mae, var_mae, diversity = eval_chromo_results(chromo_results)

        #adding the children to the population
        total_results = np.vstack((total_results,children_results))

        #sorting the population by the evaluation metrics
        total_results = sorted(total_results, key = lambda x: (x[2],x[3],x[1]))

        #culling the lowest ranking chromosomes by restricting pop size
        uncut_results = np.vstack((uncut_results, children_results))
        total_results = total_results[:pop_size]

        new_gen_entry = pd.DataFrame({"gen": [this_gen],
                                      "top_chromo": [total_results[0][0]],
                                      "top_mae": [total_results[0][2]],
                                      "avg_mae": [avg_mae],
                                      "var_mae": [var_mae],
                                      "top_features": [total_results[0][3]],
                                      "diversity": [diversity]})

        gen_df = gen_df.append(new_gen_entry, ignore_index = True)

    #get the end population after all generations

    gen_df["mut_rate"] = mut_rate
    gen_df["cross_rate"] = cross_rate

    return gen_df.sort_values(by = ["top_mae","top_features"]),total_results, uncut_results
```

Figure 46. *Function that amalgamates all functions involved in the feature select genetic algorithm process into a single function.*

TSP_GA

The TSP GA is almost the same as the FS GA, and it may have been possible to reduce some redundancy between the functions. The process is the same in that a population is initialised, then for each chromosome fitness is determined and a generational results dataframe is created. For each generation, parents are selected, crossed over and mutated to create children. The fitness of these children is evaluated, and the population is restricted such that the worst performers are removed (Figure 47).

```
[ ]  def TSP_GA(dist_mat, pop_size, gens, pairs, mut_rate, cross_rate):
       nodes = len(dist_mat.columns)
       start_pop = initialise_pop(pop_size, mode = "TSP", col_num = nodes)

       chromo_results = list()
       for chromosome in start_pop:
         chromo_route_dist = route_distance(chromosome, dist_mat)
         chromo_result = [chromosome, chromo_route_dist]
         chromo_results.append(chromo_result)

       chromo_results = sorted(chromo_results, key = lambda x: x[1])

       gen_df = pd.DataFrame({"gen": 0,
                              "top_chromo": [chromo_results[0][0]],
                              "top_dist": [chromo_results[0][1]],})

       total_results = chromo_results.copy()
       uncut_results = chromo_results.copy()

       for i in range(gens):
         this_gen = i+1
         if this_gen%1000 == 0:
           print(this_gen)

         mating_chromos, next_pop = select_mates(total_results, pairs)

         children = list()
         for j in range(pairs+1):
           if j%2 == 0:
             newc1, newc2 = crossover(list(mating_chromos[j]),list(mating_chromos[j+1]), mode = "pmx")
             newc1 = mutate(newc1, mut_rate, mode = "TSP")
             newc2 = mutate(newc2, mut_rate, mode = "TSP")
             children.append(newc1)
             children.append(newc2)
         children = np.array(children)

         children_results = list()
         for child in children:
           child_route_dist = route_distance(child, dist_mat)
           child_result = [child, child_route_dist]
           children_results.append(child_result)

         children_results = sorted([[i[0],i[1]] for i in children_results], key = lambda x: x[1])
         total_results = np.vstack((total_results,children_results))
         total_results = sorted(total_results, key = lambda x: x[1])

         #culling the lowest ranking chromosomes by restricting pop size
         uncut_results = np.vstack((uncut_results, children_results))
         total_results = total_results[:pop_size]

         new_gen_entry = pd.DataFrame({"gen": i,
                                       "top_chromo": [total_results[0][0]],
                                       "top_dist": [total_results[0][1]],})
         gen_df = gen_df.append(new_gen_entry, ignore_index = True)

       gen_df["mut_rate"] = mut_rate
       gen_df["cross_rate"] = cross_rate
       gen_df.sort_values(by = ["top_dist", "gen"], inplace = True)
       return gen_df, total_results, uncut_results
```

Figure 47. *Function that amalgamates all functions involved in the Traveling Salesman genetic algorithm process into a single function*

9 Co-Location Genetic Algorithm Pre-Processing

The construction of the Co-Location Genetic Algorithm methodology required pre-processing such that the optimal order of genes could be determined. This involved the creation of a distance matrix and network graph. The optimal route through this network was determined by the TSP GA, and a new dataframe was created by reindexing the analysis dataframe with the new order

Creation of distance matrix/correlation network

Dummy columns were created for each of the categorical variables with more than 2 levels. The dataframe was used to create a correlation matrix. Because this included correlations between dummy columns, the correlations of each dummy column were aggregated to the root level using a for-loop and regex (Figure 48).

```
#categorical variables with more than 2 options/levels
dummy_colnames = ["aircon_sys","arch_style","fips_code","heating_sys","zoning_county", "land_zone","county","constructtype"]

dummy_df = pd.get_dummies(analysis[analysis.columns[~analysis.columns.isin(["logerror"])]],
                          drop_first = False)

dummy_corr = dummy_df.corr()


for root in dummy_colnames:
  #allows for grouping of dummy columns by the root column
  root_regex = "(?<!_)"+str(root)+"_*"

  #calculate the correlaitons of each dummy col to the other columns
  root_df = dummy_corr.filter(regex = root_regex)
  root_cols = root_df.columns
  root_df = root_df[~root_df.index.isin(root_cols)]
  dummy_corr.drop(columns = root_cols, inplace = True)

  dummy_corr = dummy_corr[~dummy_corr.index.isin(root_cols)]

  #average the sub correlations
  avg_corrs = np.mean(root_df, axis = 1)

  #root's correlation added
  avg_corrs_row = np.append(avg_corrs,1)
  avg_corrs_row = pd.Series(avg_corrs_row, index = np.append(dummy_corr.index, root))

  dummy_corr.insert(len(dummy_corr.columns), root, avg_corrs)
  avg_corrs_row = avg_corrs_row.rename(root)
  dummy_corr = dummy_corr.append(pd.Series(avg_corrs_row))
```

Figure 48. *Code required to create the correlation matrix on which the distance matrix and network graphs are based*

Plotting of matrix

The correlation matrix was then turned into a distance matrix by taking the absolute value of each correlation. This was then processed into a dataframe with three columns. A “From”, a “To”, and a “Distance” column (represented in this analysis as “Var 1”, “Var 2” and “value”) (Figure 49). These links were then stacked and used to create the network graph seen in the Technical Report (TR Fig. 10).

```
#converting correlation to a distance matrix
distance_mat = np.absolute(dummy_corr)

#plotting of distance matrix onto a network
links = distance_mat.stack().reset_index()
links.columns = ["var1","var2","value"]
links_filtered=links.loc[ links['var1'] != links['var2'] ]
G=nx.from_pandas_edgelist(links_filtered, 'var1', 'var2')
plt.figure(3, figsize=(12,12))
nx.draw(G, with_labels=True, node_color='orange', node_size=1000,
        edge_color='black', linewidths=0, width=0.1, font_size=15)
```

Figure 49. *Code for translating correlation matrix into a distance matrix, and plotting as a network graph*

Running and plotting results of TSP GA

As the composition of the TSP has been described in earlier sections, it will not be described here. Figure 50 described the parameters used to run the travelling salesman GA. Due to the reduced computational cost of this optimisation problem as compared to machine learning (seen in section 11), a larger population of chromosomes was used, evolution was allowed run for 100,000 generations, and instead of 1 pair of parents being selected each generation, 6 pairs of parents were selected. This allowed for more reproduction per generation.

```
tsp_results_df, total_results, uncut_results = TSP_GA(dist_mat=distance_mat,
                                                      pop_size=50,
                                                      gens=100000,
                                                      pairs=6,
                                                      mut_rate=0.2,
                                                      cross_rate=1)
```

Figure 50. *Code used to run the TSP ga*

The results of this analysis were plotted (Figure 51), and the results are available in the technical report (TR Fig. 11). A relatively simple snippet of matplotlib code was required to generate this graph. The results were also saved and downloaded.

```
#Visualisation of how number of generations improves the distance/calc
#per generation

#figure out why there are weird jaggedness to it
fig, axes = plt.subplots(1, 1, figsize = (8,4))
plt.title("Progression of total distance by generation", size = 16)
tsp_results_df = tsp_results_df.sort_values(by = "gen")
plt.plot(tsp_results_df.gen, tsp_results_df.top_dist)
plt.ylabel("Total Route Length", size = 16)
plt.xlabel("Number of Generations", size = 16)
plt.xticks(np.arange(0,100001, 10000), size = 16,
           rotation = 60)
plt.show()

#downloading in case ever need to redo or check etc.
tsp_results_df.to_csv("tsp_results_100kgens.csv")
files.download("tsp_results_100kgens.csv")
```

Figure 51. *Code for plotting of the TSP results*

Reindexing dataframe such that order matched the optimal column order

The best chromosome from the tsp results corresponded to the optimal route through the network. This chromosome was used to select the column names in the correct order, from the distance matrix. This ordered list of columns was then used to reindex the analysis dataframe and generate the colocation analysis dataframe (coloc_df) (Figure 52).

The opt_order_df was also generated in this step, to be used later in generating the k means clustering df for the MCGA rubric.

```
[ ] #have to add in 1 because analysis has logerror at position 0 of the dataframe
    opt_order = tsp_results_df.top_chromo[0]
    opt_order_cols = distance_mat.columns[opt_order]
    coloc_df = analysis.reindex(columns = np.append("logerror",opt_order_cols))

    #used later with kmeans for MCGA rubric
    opt_order_df = pd.DataFrame({"Var": opt_order_cols,
                                 "order": opt_order})
```

Figure 52. *Code for applying the optimal order to reindex the analysis dataframe*

10 Multi-Chromosomal Genetic Algorithm Pre-Processing

Translation of network onto 2D plane

Translating the distances from the distance matrix onto a 2D plane involved the use of the manifold package, and specifically the MDS package (Figure 53). This uses the relative distance between each node on the network to map each node to coordinates on a 2D plane.

```
[ ]  #list of distances from each variable to next
     dists = []
     vars = []
     for i in range(len(distance_mat)):
         vars.append(distance_mat.columns[i])
         dists.append(distance_mat.iloc[:,i].values)

     #calculating proportional distances between vars
     adist = np.array(dists)
     amax = np.amax(adist)
     adist /= amax

     #translation of distances onto 2d plane
     mds = manifold.MDS(n_components=2, dissimilarity="precomputed", random_state=6)
     results = mds.fit(adist)

     #getting coordinates for each var
     coords = results.embedding_

     #plotting coordinates
     plt.subplots_adjust(bottom = 0.1)
     plt.scatter(
         coords[:, 0], coords[:, 1], marker = 'o'
         )

     plt.show()
```

Figure 53. *Code for mapping and plotting of the distance matrix onto a 2D plane*

K means clustering on 2D plane

Once the variables were mapped to a 2D plane, k means clustering was used to cluster each variable into groups of related variables. A for loop was used to vary K and record the total sum squared distance for each K, such that it was possible to evaluate the best value for K. this step also created a kmeans_df, which would be later used to split chromosomes into sub chromosomes in the genetic algorithm rubric (as mentioned earlier, when describing the split_chromos function) (Figure 54). A second snippet then created a colour version of the 2D plane such that the sub chromosome for each variable was visible (Figure 55).

```
[ ]  kmeans_df = pd.DataFrame({"Var": distance_mat.columns,
                               "X": [i[0] for i in coords],
                               "Y": [i[1] for i in coords]})
     x = coords
     tot_ss = list()
     val_range = range(2,11)
     for i in val_range:
       kmeans = KMeans(n_clusters = i, random_state = 0).fit(coords)
       tot_ss.append(kmeans.inertia_)

     kmeans = KMeans(n_clusters = 5, random_state = 0).fit(coords)
     kmeans_df["cluster"] = kmeans.labels_

     plt.figure(figsize=(8,6))
     plt.plot(val_range, tot_ss, color="blue", linestyle= "--", marker="o",
              markerfacecolor="red", markersize=5)
     plt.title("Total Sum Squared Distance vs K")
     plt.xlabel("K")
     plt.ylabel("Total Sum Squared Distance")
     plt.show()
```

Figure 54. *Code for looping through different values for K and assessing optimal value.*

```
[ ]  plt.scatter(kmeans_df["X"], kmeans_df["Y"], c = kmeans_df["cluster"])
     plt.show()
```

Figure 55. *Code for plotting the 2D plane but with variables coloured based on the cluster to which they belong*

Plots created by the codes above (Figures 53,54 and 55) are each represented in the technical report (TR Fig. 14).

11 Modelling and Results Visualisation

Modelling of Genetic Algorithms

The sections above have explained in depth how each genetic algorithm methodology was conceived and implemented. However, the actual computation section is almost the same for each model, and for each genetic algorithm. Therefore, a single representative snippet is included in Figure 56. The points at which minor differences are present are highlighted and commented upon.

For each genetic algorithm and for each machine learning model, the snippet below was run, with some slight variations. These variations are given in the colour comment boxes to the right.

```
[ ]  #xgb ga
     #Do first with XGB
     MCGA_xgb_df, MCGA_xgb_results, uncut_MCGA_xgb_results = FS_GA(model = xgb,
                                                                   dataframe = coloc_df,
                                                                   K = 5,
                                                                   pop_size = 30,
                                                                   gens = 250,
                                                                   pairs = 1,
                                                                   mut_rate = 0.3,
                                                                   cross_rate = 0.2,
                                                                   mode = "Multi Chromosomal")

     MCGA_xgb_df["model"] = "XGBoost"
     MCGA_xgb_df["Genetic Algorithm"] = "Multi-Chromosomal"
     MCGA_xgb_df= MCGA_xgb_df.sort_values(by = "gen")
     MCGA_xgb_df.to_csv("MCGA_xgb_Results.csv")
     files.download("MCGA_xgb_Results.csv")
```

The MCGA abbreviation was used for the Multi-Chromosomal Algorithm only. SGA and CLGA were used for the Standard GA and Co-Location GA respectively.

xgb denoted the XGBoost model. LR and DT were used for the Linear Regression and Decision Tree models respectively

The "coloc_df" was used for the MCGA and CLGA rubrics. For the Standard GA, the default "analysis" dataframe was used.

K = 5, pop_size = 30, gens = 250, pairs = 1, mut_rate = 0.3 was used for all GA modelling steps.

Figure 56. *Representative code used for running of each model for each Genetic Algorithm. Colour-coded comments describe the changes to be made to the code for each model and Genetic Algorithm*

Visualisation of results (per genetic algorithm)

Visualisations of the results within each genetic algorithm section was mediated through the intra_ga_vis function (Figure 57). This function took in the results dataframes from each model and produced the results visualisations for that genetic algorithm. For example, the snippet below produced the results for the MCGA algorithm, seen in the technical report (TR Fig. 8, 9, 12, 13, 15, 16).

```
[ ]  #Visualisation of results
     intra_ga_viz(MCGA_xgb_df, MCGA_LR_df, MCGA_DT_df)
```

Figure 57. *Code that calls a function that visualises the results of each GA modelling*

The function itself is a composite of matplotlib plots. Each plot is relatively simple and self-explanatory (Figure 58).

```
[ ]  def intra_ga_viz(results_df1, results_df2, results_df3):
       #Visualisation of results

       #BEST MAPE achieved by each model
       #bar graph
       fig, ax = plt.subplots(1,2, figsize = (12,6))
       fig.tight_layout(pad = 5)
       plt.subplot(1,2,1)

       plt.title("Best MAE achieved", size = 16)
       plt.ylabel("MAE", size = 16)
       plt.xticks(size = 16, rotation = 20)
       plt.yticks(size = 16)
       plt.bar(x = ["XGBoost", "Linear Regression", "Decision Tree"],
               height = [results_df1.at[250,"top_mae"],
                         results_df2.at[250,"top_mae"],
                         results_df3.at[250,"top_mae"]],
               color = ["red", "blue", "orange"])
       #Number of features required for best MAPE for each model
       #bar graph

       plt.subplot(1,2,2)
       plt.title("Number of Features Required for Best Error", size = 16)
       plt.ylabel("Features Used", size = 16)
       plt.xticks(size = 16, rotation = 20)
       plt.yticks(size = 16)
       plt.bar(x = ["XGBoost", "Linear Regression", "Decision Tree"],
               height = [results_df1.at[250,"top_features"],
                         results_df2.at[250,"top_features"],
                         results_df3.at[250,"top_features"]],
               color = ["red", "blue", "orange"])

       plt.show()
       #Show how MAPE changed over generations for each model, on same graph
       #cumulative graph
       fig, ax = plt.subplots(1,1, figsize = (12,6))
       plt.title("Progression of MAE per Model per Generation", size = 16)
       plt.ylabel("MAE of Best Chromosome", size = 16)
       plt.xlabel("Generation", size = 16)
       plt.yticks(size = 16)
       plt.xticks(np.arange(0, np.max(results_df2.gen)+1, 25), size = 16, rotation = 60)
       plt.plot(results_df2.gen, results_df2.top_mae, color = "blue")
       plt.plot(results_df1.gen, results_df1.top_mae, color = "red")
       plt.plot(results_df3.gen, results_df3.top_mae, color = "orange")
       plt.show()

       fig, ax = plt.subplots(1,1, figsize = (12,6))
       plt.title("Progression of Minimum Feature Requirement per Model per Generation", size = 16)
       plt.ylabel("Minimum Features", size = 16)
       plt.yticks(size = 16)
       plt.xlabel("Generation", size = 16)
       plt.xticks(np.arange(0, np.max(results_df2.gen)+1, 25), size = 16, rotation = 60)
       plt.plot(results_df2.gen, results_df2.top_features, color = "blue")
       plt.plot(results_df1.gen, results_df1.top_features, color = "red")
       plt.plot(results_df3.gen, results_df3.top_features, color = "orange")
       plt.show()
```

Figure 58. *Function that visualises the results of each model's performance for a given genetic algorithm*

Visualisation of genetic algorithm benchmarking

Following modelling of each machine learning method and each Genetic Algorithm methodology, visualisations were used to compare the evolutionary performance of each genetic algorithm. Again, this involved a composite of simple matplotlib graphs. Two bar charts compared the best MAE or best Features achieved by each model and genetic algorithm combination, and multiple line plots compared the course of evolution per model and per genetic algorithm. A representative snippet of code is given for the bar charts (Figure

59), and for the line plots (Figure 60). These snippets generated the visuals in the technical report (TR Fig. 17, 18).

```
#---------------------- Top MAE by model and by ga
labels = ["XGBoost", "Linear Regression", "Decision Tree"]

unoptimized_heights = results_df["mae"][results_df["model"]!= "Random Forest"].values
SGA_heights = Total_results.loc[(Total_results["Genetic Algorithm"] == "Standard")&(Total_results["gen"] == 250), "top_mae"]
CLGA_heights = Total_results.loc[(Total_results["Genetic Algorithm"] == "Co-Location")&(Total_results["gen"] == 250), "top_mae"]
MCGA_heights = Total_results.loc[(Total_results["Genetic Algorithm"] == "Multi-Chromosomal")&(Total_results["gen"] == 250), "top_mae"]

x = np.arange(len(labels))  # the label locations
width = 0.2  # the width of the bars

fig, ax = plt.subplots(figsize = (7,6))
rects1 = ax.bar(x - 1.5*width, unoptimized_heights, width, label='Unoptimized', color = "black")
rects2 = ax.bar(x - width/2, SGA_heights, width, label='Standard GA', color = "brown")
rects3 = ax.bar(x + width/2, CLGA_heights, width, label='Co-Location GA', color = "purple")
rects4 = ax.bar(x + 1.5*width, MCGA_heights, width, label='Multi-Chromosomal GA', color = "green")

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('MAE', size = 16)
ax.set_yticklabels(np.round(np.arange(0,1.2, .2),2), fontdict={"size": 16})
ax.set_title('MAE by Model and Optimization Method', size = 16)
ax.set_xticks(x)
ax.set_xticklabels(labels, size = 16, rotation = 20)
ax.legend(prop = {"size": 16})

fig.tight_layout()

plt.show()
```

Figure 59. *Representative code for a bar chart used for comparing genetic algorithm with one another*

```
#---------------------- Progress of Feature per generation, per genetic algorithm

fig, ax = plt.subplots(1,1, figsize = (12,6))
plt.title("Progression of XGBoost Features per Genetic Algorithm per Generation", size = 16)
plt.ylabel("MAE of Best Chromosome", size = 16)
plt.yticks(size = 16)
plt.xlabel("Generation", size = 16)
plt.xticks(np.arange(0, np.max(Total_results.gen)+1, 25), size = 16, rotation = 60)
plt.plot(range(0,251), Total_results["top_features"][(Total_results["model"] == "XGBoost")&(Total_results["Genetic Algorithm"]=="Standard")], color = "brown", label = "Standard")
plt.plot(range(0,251), Total_results["top_features"][(Total_results["model"] == "XGBoost")&(Total_results["Genetic Algorithm"] == "Co-Location")], color = "purple", label = "Co-Location")
plt.plot(range(0,251), Total_results["top_features"][(Total_results["model"] == "XGBoost")&(Total_results["Genetic Algorithm"] == "Multi-Chromosomal")], color = "green", label = "Multi-Chromosomal")
plt.legend(loc = 1, prop = {"size": 16})
plt.show()
```

Figure 60. *Representative code for a line chart used for comparing genetic algorithms with one another*

12 Appendix 1

Table 2 – Description of Properties Data Tables

Properties 2016 & 2017	**Rows: 5,970,434 Columns: 58**
Variable	**Description**
'airconditioningtypeid'	Type of cooling system present in the home (if any)
'architecturalstyletypeid'	Architectural style of the home (i.e. ranch, colonial, split-level, etc…)
'basementsqft'	Finished living area below or partially below ground level
'bathroomcnt'	Number of bathrooms in home including fractional bathrooms
'bedroomcnt'	Number of bedrooms in home
'buildingqualitytypeid'	Overall assessment of condition of the building from best (lowest) to worst (highest)
'buildingclasstypeid'	The building framing type (steel frame, wood frame, concrete/brick)
'calculatedbathnbr'	Number of bathrooms in home including fractional bathroom
'decktypeid'	Type of deck (if any) present on parcel
'threequarterbathnbr'	Number of 3/4 bathrooms in house (shower + sink + toilet)
'finishedfloor1squarefeet'	Size of the finished living area on the first (entry) floor of the home
'calculatedfinishedsquarefeet'	Calculated total finished living area of the home
'finishedsquarefeet6'	Base unfinished and finished area
'finishedsquarefeet12'	Finished living area

'finishedsquarefeet13'	Perimeter living area
'finishedsquarefeet15'	Total area
'finishedsquarefeet50'	Size of the finished living area on the first (entry) floor of the home
'fips'	Federal Information Processing Standard code - see https://en.wikipedia.org/wiki/FIPS_county_code for more details
'fireplacecnt'	Number of fireplaces in a home (if any)
'fireplaceflag'	Is a fireplace present in this home
'fullbathcnt'	Number of full bathrooms (sink, shower + bathtub, and toilet) present in home
'garagecarcnt'	Total number of garages on the lot including an attached garage
'garagetotalsqft'	Total number of square feet of all garages on lot including an attached garage
'hashottuborspa'	Does the home have a hot tub or spa
'heatingorsystemtypeid'	Type of home heating system
'latitude'	Latitude of the middle of the parcel multiplied by 10e6
'longitude'	Longitude of the middle of the parcel multiplied by 10e6
'lotsizesquarefeet'	Area of the lot in square feet
'numberofstories'	Number of stories or levels the home has
'parcelid'	Unique identifier for parcels (lots)
'poolcnt'	Number of pools on the lot (if any)
'poolsizesum'	Total square footage of all pools on property
'pooltypeid10'	Spa or Hot Tub
'pooltypeid2'	Pool with Spa/Hot Tub
'pooltypeid7'	Pool without hot tub
'propertycountylandusecode'	County land use code i.e. it's zoning at the county level
'propertylandusetypeid'	Type of land use the property is zoned for
'propertyzoningdesc'	Description of the allowed land uses (zoning) for that property
'rawcensustractandblock'	Census tract and block ID combined - also contains blockgroup assignment by extension
'censustractandblock'	Census tract and block ID combined - also contains blockgroup assignment by extension
'regionidcounty'	County in which the property is located
'regionidcity'	City in which the property is located (if any)
'regionidzip'	Zip code in which the property is located
'regionidneighborhood'	Neighborhood in which the property is located
'roomcnt'	Total number of rooms in the principal residence
'storytypeid'	Type of floors in a multi-story house (i.e. basement and main level, split-level, attic, etc.). See tab for details.
'typeconstructiontypeid'	What type of construction material was used to construct the home
'unitcnt'	Number of units the structure is built into (i.e. 2 = duplex, 3 = triplex, etc...)
'yardbuildingsqft17'	Patio in yard
'yardbuildingsqft26'	Storage shed/building in yard
'yearbuilt'	The Year the principal residence was built
'taxvaluedollarcnt'	The total tax assessed value of the parcel
'structuretaxvaluedollarcnt'	The assessed value of the built structure on the parcel
'landtaxvaluedollarcnt'	The assessed value of the land area of the parcel
'taxamount'	The total property tax assessed for that assessment year
'assessmentyear'	The year of the property tax assessment
'taxdelinquencyflag'	Property taxes for this parcel are past due as of 2015
'taxdelinquencyyear'	Year for which the unpaid propert taxes were due

Table 3 – Description of Transaction Data Tables

Transactions 2016-2017	**Rows: 167,838 Columns: 3**
Variable	**Description**

Transaction_date	Date the sale of the property occurred
Parcel_id	Id of the property sold
logerror	Error achieved by the Zillow Zestimate model

References

Google Colab. (2020). *Welcome to Colaboratory - Colaboratory*. Getting Started - Introduction. https://colab.research.google.com/notebooks/intro.ipynb

Hussain, A., Muhammad, Y. S., Nauman Sajid, M., Hussain, I., Mohamd Shoukry, A., & Gani, S. (2017). Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator. *Computational Intelligence and Neuroscience*, *2017*. https://doi.org/10.1155/2017/7430125

Kaggle. (2018). *Zillow Prize: Zillow's Home Value Prediction*. https://www.kaggle.com/c/zillow-prize-1/data?select=properties_2017.csv

Macleod, M. D. (1993). *Hamming Distance - an overview ScienceDirect Topics*. https://www.sciencedirect.com/topics/engineering/hamming-distance

OpenStreetMap. (n.d.). *OpenStreetMap*. Retrieved August 2, 2021, from https://www.openstreetmap.org/#map=9/33.9548/-117.9300&layers=T

Terpilowski, M. (2019). scikit-posthocs: Pairwise multiple comparison tests in Python. *Journal of Open Source Software*, *4*(36), 1169. https://doi.org/10.21105/JOSS.01169