Computation of quality life index for world countries

by
Simone Finelli - sba22524
Faraj Alheajneh -sba22317
Thomas Kelly - sba22259
Akash Jyothis Parappurathu - sba22316

Higher Diploma in Science in Artificial Intelligence Applications

Strategic Thinking

James Garza

CCT College

Dublin, Ireland

Abstract

When we consider relocating to a new country, we often focus solely on the salary offered by a potential employer. However, it is important to also consider other factors that can impact our overall quality of life. There are many financial and nonfinancial reasons when considering a move but also there are a few that influence directly or to some degree of effect where the freedom of movement in different countries is not an issue. It allows us to move to different locations, but we need to assess the quality of life of our potential new home before making a decision.

The use of a shared currency in some regions, ie:euro in our case, can sometimes make it seem like a higher salary automatically translates to a higher standard of living, but this is not always the case. Therefore, in this study, we aim to determine the quality of life in a country based on the cost of different commodities, some semi-financial factors, and non-financial factors.

We acknowledge that the concept of "quality of life" can be controversial and subject to generalisation, so we aim to provide a more objective method for predicting and evaluating it by associating multiple dimensions, with the use of magic that machine learning provides to reach a high accuracy prediction.

To carry out this research, we will use the Cross-Industry Standard Process for Data Mining (CRISP-DM) methodology. This methodology consists of five stages: Business Understanding, Data Understanding, Data Preparation, Modelling, Evaluation, and Deployment. In the study, we will delve into each of these stages and use techniques such as Price Level Indices, Purchasing Power Parities, and Random Forest models to analyse the data.

In this study we aim to provide a more comprehensive and objective approach to evaluating the quality of life in different countries, considering factors beyond just salary. By doing so, we hope to help individuals make more informed decisions when considering relocation to a new country.

Keywords: CRISP-DM, Price Level Indices, Purchasing Power Parities, Random Forest, XGBoost, Neural Network

Table of Contents

Table of Figures

Business Understanding

The first stage of the CRISP-DM methodology is the Business Understanding stage, which is essential in defining the goals and objectives of the project (Hotz, 2022). The objective of this study is to evaluate the quality of life index for a country based on its commodity prices and quality of life indicators (Health, economics and political)

When evaluating the quality of life for a country, it is important to consider various factors such as the cost of living and how it may change over time due to inflation and macroeconomic conditions. Therefore, our hypothesis is that if we relate a job salary offer to the quality of life index of a country, we can make better financial-oriented decisions.

To achieve the objective, the project will focus on determining the quality of life index for a country based on a subset of its commodity prices.

The project will use data from Eurostat and OECD to build the model, covering the period from 1995 to 2021 and 1997 to 2021, respectively. However, the result may be slightly biased since we considered only a subset of commodities available from the Eurostat website.

We also analysed other factors that could contribute to the index from different dimensions:

- Education level. This could be a significant factor in determining the quality of life index, as countries with higher levels of education tend to have better job prospects and higher incomes.

- Health indicators. Health indicators such as life expectancy, access to healthcare, and disease prevalence could be relevant in predicting the quality of life index.

- Economic indicators.
    1. Labour transitions by employment status
    2. Arrears (mortgage or rent, utility bills or hire purchase)

- Political stability. Political stability and the level of corruption could also impact the quality of life index.

There are a few reasons why political stability may not be an appropriate feature. It is a subjective and very complex concept that may be difficult to accurately measure and quantify. It can be influenced by a wide range of factors such as government policies, social unrest, and international relations. Therefore, it may not be a reliable predictor of quality of life index.

The selected commodities we decided to work from Eurostat are:

- Food and non-alcoholic beverages
- Alcoholic beverages, tobacco and narcotics
- Clothing and footwear
- Housing, water, electricity, gas and other fuels
- Household furnishings, equipment and maintenance
- Health
- Transport
- Communication
- Recreation and culture
- Education
- Restaurants and hotels
- Miscellaneous goods and services

The success criteria and the project goal consist of predicting the country's quality of life index with at least a 90% without overfitting.

Python libraries that will be used:

- Pandas
- NumPy
- Matplotlib
- Seaborn
- Scikit-learn
- Tensorflow

Modelling: Random Forest, XGBoost and Neural Network to solve our regression problem.

Data Understanding

Data understanding is the second phase of the CRISP-DM methodology for data mining. It drives the focus to identify, collect, and analyse the data sets that can help you accomplish the project goals. (Nick Hotz, 2022)

To reach our goal, we decided to use two different datasets: the Price Level Indices (PLI) dataset from the Organisation for Economic Co-operation and Development (OECD) as a quality of life index, and the Purchasing Power Parities (PPP) dataset from Eurostat as commodity costs.

The PLI dataset “provides a measure of the differences in the general price levels of different countries” (OECD data, 2022), while the PPP dataset shows “how many currency units a given quantity of goods and services costs in different countries” (Eurostat, 2022).

Together, these datasets will help us understand the relationship between prices and purchasing power in different countries.

The PLI dataset contains data on the Price Level Indices of different countries from 1997 to 2021. Before importing the data, we filtered out the Price Level Indices from aggregated countries such as the Euro area (18 countries) to only have values for single countries.

After importing the data, we used the **.head()** function to get a first look at the dataset.

```
In [3]: # Head function is used to have a quick look of our dataset, specifically its first 5 rows

        pli.head()
```

Out[3]:

	LOCATION	INDICATOR	SUBJECT	MEASURE	FREQUENCY	TIME	Value	Flag Codes
0	AUS	PLI	TOT	OECDIDX	A	1997	97	NaN
1	AUS	PLI	TOT	OECDIDX	A	1998	84	NaN
2	AUS	PLI	TOT	OECDIDX	A	1999	86	NaN
3	AUS	PLI	TOT	OECDIDX	A	2000	82	NaN
4	AUS	PLI	TOT	OECDIDX	A	2001	77	NaN

Figure 1. PLI head

We then used the **.shape** and **.info()** functions to learn more about the dataset's structure, and found that it contains 1175 rows and 8 columns, with data types of **object**, **int64**, and **float64**.

```
In [63]: # Shape function is used to return the dataset dimension

         pli.shape

Out[63]: (1175, 8)

In [64]: # Info function is used to get an understanding of the data types we are working with

         pli.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1175 entries, 0 to 1174
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   LOCATION    1175 non-null   object
 1   INDICATOR   1175 non-null   object
 2   SUBJECT     1175 non-null   object
 3   MEASURE     1175 non-null   object
 4   FREQUENCY   1175 non-null   object
 5   TIME        1175 non-null   int64
 6   Value       1175 non-null   int64
 7   Flag Codes  0 non-null      float64
dtypes: float64(1), int64(2), object(5)
memory usage: 73.6+ KB
```

Figure 2. PLI shape and info

Looking at the **.describe()** result, we determined that we would likely only need the *"LOCATION"*, *"TIME"*, and *"Value"* columns, as the other columns seemed to contain repetitive information. This has been proven using the following code:

```
In [61]: # Let's loop over all columns and, for each column, we want to print all unique values available in the dataset

         for i in pli.columns:
             print(i + ": ")
             print(pli[i].unique())

LOCATION:
['AUS' 'AUT' 'BEL' 'CAN' 'CZE' 'DNK' 'FIN' 'FRA' 'DEU' 'GRC' 'HUN' 'ISL'
 'IRL' 'ITA' 'JPN' 'KOR' 'LUX' 'MEX' 'NLD' 'NZL' 'NOR' 'POL' 'PRT' 'SVK'
 'ESP' 'SWE' 'CHE' 'TUR' 'GBR' 'USA' 'BRA' 'CHL' 'CHN' 'EST' 'IND' 'IDN'
 'ISR' 'RUS' 'SVN' 'ZAF' 'OECD' 'EA18' 'COL' 'LTU' 'LVA' 'EU27' 'CRI']
INDICATOR:
['PLI']
SUBJECT:
['TOT']
MEASURE:
['OECDIDX']
FREQUENCY:
['A']
TIME:
[1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010
 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021]
Value:
[ 97  84  86  82  77  81  91  99 104 106 115 117 111 136 149 158 143 135
 126 125 131 120 124 105 101  90  93  95 108 119 116 110 100 102 107  89
  92 103 109  87  88  94 123 121  41  45  42  40  49  52  55  60  64  68
  71  70  72  67  63  62  65  69 132 133 128 118 140 141 145 147 142 129
 134 122 114  98 127 113 112  96  78  80  85  75  76  73  74  47  51  56
  59  57  58  54 150 162 138 130 137 139 146 154  79  66  61  53 148 155
 157 152  46  50  48  83  37  36  39  38  44  33  34  43  35  24  23  25
  27  31  32  30  21  11  15  18  17  19  20  28]
Flag Codes:
[nan]
```

Figure 3. PLI unique values of columns

Finally, we used the **.describe()** function to get a better statistical analysis of the *"TIME"* and *"Value"* columns.

```
In [66]: # Describe function is used to generate descriptive statistics

         pli.describe()
```

Out[66]:

	TIME	Value	Flag Codes
count	1175.000000	1175.000000	0.0
mean	2009.000000	85.252766	NaN
std	7.214173	30.885704	NaN
min	1997.000000	11.000000	NaN
25%	2003.000000	60.000000	NaN
50%	2009.000000	88.000000	NaN
75%	2015.000000	109.000000	NaN
max	2021.000000	162.000000	NaN

Figure 4. PLI description

We confirmed that the *"TIME"* variable ranges from 1997 to 2021, and that the *"Value"* column, which will be our target in the final dataset, ranges from 11 to 162 with a standard deviation above the mean.

We also tried to have a quick look at how the Price Level Indices evolve over time for each country using the pyplot library.

```
In [8]: # let's plot how the pli evolves over time for each country

plt.figure(figsize=(30, 20))
for i in np.unique(pli["LOCATION"]):
    x = pli.loc[pli["LOCATION"]==i]["TIME"]
    y = pli.loc[pli["LOCATION"]==i]["Value"]
    plt.plot(x, y, label = i)

plt.xlabel("Year")
plt.ylabel("PLI")
plt.legend()
plt.show()
```

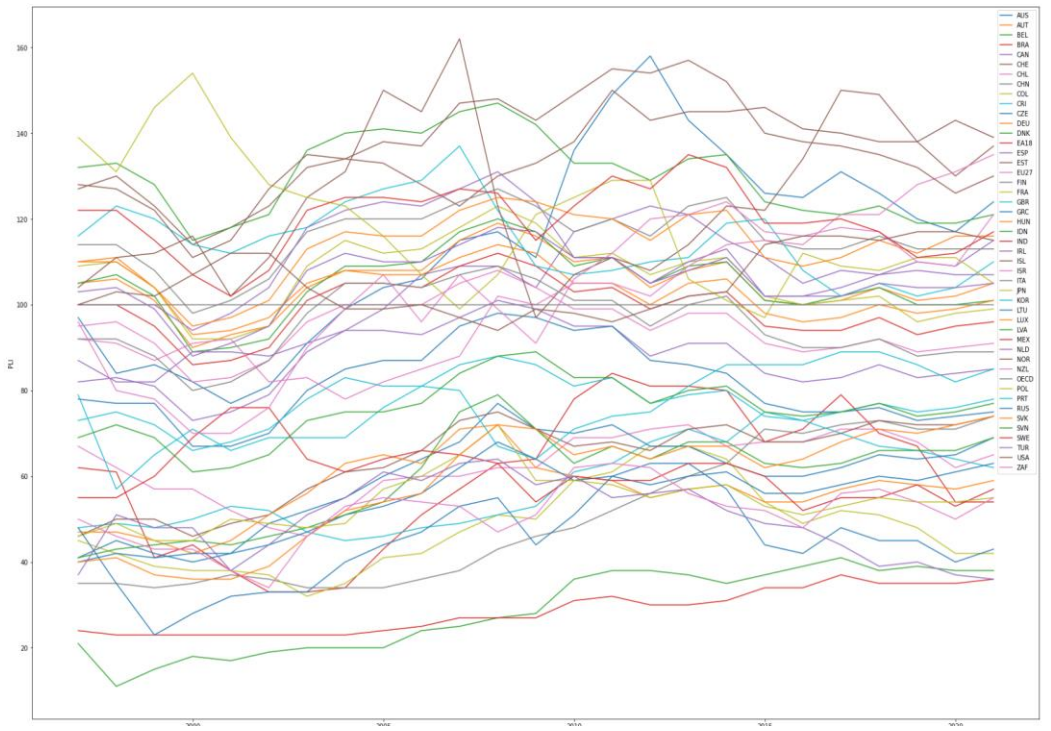

Figure 5. PLI dataset Price Level Indices over time for countries

The PPP dataset contains the data that will be used as features in the final dataset. It includes purchasing power parities indices for specific categories in different countries from 1995 to 2021. Before importing the data, we filtered out a list of purchasing power parities indices for a subset of categories to avoid redundancy and exclude macroeconomic information.

The PPP categories are represented by an encoded string, the mapping between encoded name and actual name is as follows:

- A0101 -> Food and non-alcoholic beverages
- A0102 -> Alcoholic beverages, tobacco and narcotics
- A0103 -> Clothing and footwear
- A0104 -> Housing, water, electricity, gas and other fuels
- A0105 -> Household furnishings, equipment and maintenance
- A0106 -> Health
- A0107 -> Transport
- A0108 -> Communication
- A0109 -> Recreation and culture
- A0110 -> Education
- A0111 -> Restaurants and hotels
- A0112 -> Miscellaneous goods and services

After importing the data, we used the **.head()** function to get a first look at the dataset.

```
In [10]: ppp.head()
```

Out[10]:

	DATAFLOW	LAST UPDATE	freq	na_item	ppp_cat	geo	TIME_PERIOD	OBS_VALUE	OBS_FLAG
0	ESTAT:PRC_PPP_IND(1.0)	24/06/22 23:00:00	A	EXP_EUR	A0101	AL	2005	1945.0	NaN
1	ESTAT:PRC_PPP_IND(1.0)	24/06/22 23:00:00	A	EXP_EUR	A0101	AL	2006	2094.0	NaN
2	ESTAT:PRC_PPP_IND(1.0)	24/06/22 23:00:00	A	EXP_EUR	A0101	AL	2007	2318.0	NaN
3	ESTAT:PRC_PPP_IND(1.0)	24/06/22 23:00:00	A	EXP_EUR	A0101	AL	2008	2606.0	NaN
4	ESTAT:PRC_PPP_IND(1.0)	24/06/22 23:00:00	A	EXP_EUR	A0101	AL	2009	2599.0	NaN

Figure 6. PPP head

We then used the **.pivot_table()** function to translate the dataset so that the ppp categories were distributed as columns across countries.

```
In [74]: ppp_new = ppp.pivot_table("OBS_VALUE", ["geo","TIME_PERIOD"], "ppp_cat").reset_index()

In [78]: ppp_new.head()
```

Out[78]:

ppp_cat	geo	TIME_PERIOD	A0101	A0102	A0103	A0104	A0105	A0106	A0107	A0108	A0109	A0110	A0111	A0112
0	AL	1997	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	AL	1998	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	AL	1999	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	AL	2000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	AL	2001	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Figure 7. PPP pivot and head

We used the **.shape** and **.info()** functions to learn more about the dataset's structure, and found that it contains 898 rows and 14 columns, with data types of **object**, **int64**, and **float64**.

```
In [24]: ppp_new.shape
Out[24]: (898, 14)

In [25]: ppp_new.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 898 entries, 0 to 897
Data columns (total 14 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   geo          898 non-null    object
 1   TIME_PERIOD  898 non-null    int64
 2   A0101        898 non-null    float64
 3   A0102        898 non-null    float64
 4   A0103        898 non-null    float64
 5   A0104        890 non-null    float64
 6   A0105        898 non-null    float64
 7   A0106        891 non-null    float64
 8   A0107        898 non-null    float64
 9   A0108        898 non-null    float64
 10  A0109        890 non-null    float64
 11  A0110        890 non-null    float64
 12  A0111        898 non-null    float64
 13  A0112        890 non-null    float64
dtypes: float64(12), int64(1), object(1)
memory usage: 98.3+ KB
```

Figure 8. PPP shape and info

Next, we used the **.describe()** function to get a statistical analysis of the *"TIME_PERIOD"* and ppp categories columns. We got confirmation that "*TIME_PERIOD*" of our ppp category columns ranges from 1995 to 2021 and most of ppp category columns have a high standard deviation from the mean, indicating a very spread-out dataset.

```
In [27]: ppp_new.describe()
```

Out[27]:

ppp_cat	TIME_PERIOD	A0101	A0102	A0103	A0104	A0105	A0106	A0107	A0108	A0109	A0110	A0111	A0112
count	898.000000	898.000000	898.000000	898.000000	890.000000	898.000000	891.000000	898.000000	898.000000	890.000000	890.000000	898.000000	890.000000
mean	2009.275056	14419.447169	4828.378723	4562.535961	22509.690920	5503.594658	13642.676543	12334.078496	2799.519145	9847.631004	8975.012482	6994.521060	15403.622121
std	7.308084	28415.500683	11514.768658	6781.450073	37053.980297	9383.733141	23014.186847	21141.075158	5318.099122	16104.152371	15066.060920	11742.628755	25541.110900
min	1995.000000	338.208456	68.497599	58.898867	384.935123	48.974213	55.000000	78.459423	46.379340	42.693864	240.644930	49.834739	95.419214
25%	2003.000000	1932.635753	861.746997	510.734587	3344.487019	651.620309	1482.740230	1565.912358	380.508765	985.663724	1503.097278	899.711473	1595.493525
50%	2010.000000	5801.883658	1793.528901	2030.330321	9841.688372	2520.716020	5957.808662	5154.618524	1083.404955	4333.823023	3862.321435	3164.767186	5771.505664
75%	2016.000000	18402.651911	5570.653798	4805.732264	27883.648900	5532.009798	15844.098976	12925.592607	3246.508933	11359.838127	10911.069628	6494.936173	17497.014898
max	2021.000000	548967.812889	235567.474333	92154.153111	668950.970667	160406.541111	399839.145111	331984.744778	88191.572556	252281.954778	262894.190778	185152.048444	416551.389778

Figure 9. PPP description

Finally, some of the ppp category columns present null values.

```
In [26]: # Let's see how many null values we have in our dataset

         ppp_new.isnull().sum()
Out[26]: ppp_cat
         geo            0
         TIME_PERIOD    0
         A0101          0
         A0102          0
         A0103          0
         A0104          8
         A0105          0
         A0106          7
         A0107          0
         A0108          0
         A0109          8
         A0110          8
         A0111          0
         A0112          8
         dtype: int64
```

Figure 10. PPP dataset null values

To create our dataset, an outer merge on time and location will be performed during the data preparation phase.

Data Preparation

This phase, which is often referred to as “data munging”, prepares the final data set(s) for modeling. (Nick Hotz, 2022)

As previously anticipated, we are working with two separate datasets that need to be merged to be able to build our model. The two columns in common between the datasets are [“LOCATION”, “TIME”] for PLI and [“geo”, “TIME_PERIOD”] for PPP. The main problem we had is that countries would follow a different naming in the datasets (i.e., Australia would be AUS in PLI and AL in PPP).

To have a homogeneous naming for countries, those were renamed in both datasets to convert the country naming convention used in the dataset to the normal name of the country.

After this pre-processing, [“geo”, “TIME_PERIOD”] columns from PPP were renamed to [“LOCATION”, “TIME”]:

```
In [35]: # column renaming for PPP

         ppp.rename(columns = {"geo":"LOCATION", "TIME_PERIOD":"TIME"}, inplace = True)
```

Figure 11. PPP columns renaming

Finally, an outer merge could be made on the two datasets:

```
In [36]: # PPP and PLI outer merge on LOCATION and TIME columns

         df = pd.merge(ppp, pli, how="outer", on=["LOCATION", "TIME"])
```

Figure 12. df dataset creation from PPP and PLI outer merge

Given that different locations were not in common to the two datasets, (i.e., we didn’t have commodity prices association to price level index and the other way around for different countries), we had to drop null values as following:

```
In [37]: # dropping null values on the merget dataset

         df = df.dropna()
```

Figure 13. df drop of null values

The final dataset we got was made of 610 rows and 15 columns.

The next step we had planned was to remove the string ‘LOCATION’ values and replace them with integer values. This was so the column could be used as a feature in our model, making sure that the output labels could be mapped back to

their origin country. We used the pandas **applymap()** function to map the integers to the 25-string locations.

```
# Mapping countries in dataset to integer values
mymap = {'Austria':1, 'Belgium':2, 'Denmark':3, 'Estonia':4, 'Finland':5, 'France':6, 'Germany
```

```
# Appling map to df, making counties integer values
df = df.applymap(lambda k: mymap.get(k) if k in mymap else k)
```

Figure 14. Mapping location integers

After getting the 'LOCATION' column's data type resolved, we established this stage to be the best opportunity to scale the data.

Using sklearn **preprocessing**, we scaled the dataset and converted it from a NumPy array back into a DataFrame. Then, we replaced the columns which would be integer values with their original DataFrame values. These columns being: 'LOCATION', 'TIME' and 'Value'.

```
# Scaling Data
from sklearn import preprocessing
df_scaled = preprocessing.scale(df)

# Turning df_scaled from NumPy array to DataFrame
df_scaled = pd.DataFrame(df_scaled, columns=df.columns)

# Putting values that shouldn't be scaled back to their original
df_scaled['LOCATION'] = df['LOCATION']
df_scaled['TIME'] = df['TIME']
df_scaled['Value'] = df['Value']
df = df_scaled

df
```

	LOCATION	TIME	A0101	A0102	A0103	A0104	A0105	A0106	A0107	A0108	A0109	A0110	A0111	A0112	Value
0	1.0	1997.0	-0.485698	-0.421678	-0.436349	-0.564463	-0.449353	-0.539756	-0.518745	-0.509364	-0.508559	-0.530179	-0.408135	-0.593299	105.0
1	1.0	1998.0	-0.484662	-0.413318	-0.432181	-0.557421	-0.448269	-0.522640	-0.513350	-0.497781	-0.494183	-0.520727	-0.378950	-0.581765	106.0
2	1.0	1999.0	-0.509506	-0.427995	-0.492928	-0.578422	-0.484317	-0.549639	-0.537603	-0.507677	-0.524675	-0.563426	-0.417749	-0.610965	101.0
3	1.0	2000.0	-0.503286	-0.424030	-0.485925	-0.567652	-0.468000	-0.540231	-0.523238	-0.445969	-0.509548	-0.555257	-0.396850	-0.596321	90.0
4	1.0	2001.0	-0.497269	-0.421694	-0.478835	-0.559531	-0.472091	-0.532061	-0.523924	-0.446004	-0.502162	-0.548446	-0.400207	-0.594599	93.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
604	25.0	2015.0	0.517664	0.353436	2.227137	1.354393	1.045582	1.191934	1.413789	0.530926	2.501343	1.180048	1.627268	1.462004	120.0
605	25.0	2016.0	0.549566	0.360169	2.088087	1.335296	1.142752	1.191813	1.418779	0.553214	2.561104	1.207803	1.745132	1.460125	108.0
606	25.0	2017.0	0.598644	0.437093	2.142212	1.397006	1.304192	1.218495	1.571320	0.874844	2.536890	1.242746	1.807173	1.644649	102.0
607	25.0	2018.0	0.652671	0.447273	2.260266	1.447971	1.462020	1.366394	1.650928	0.834703	2.694389	1.393560	1.839955	1.690805	105.0
608	25.0	2019.0	0.692107	0.563930	1.690638	1.426210	1.519691	1.587461	0.795869	0.645044	2.340413	1.396766	0.396652	1.429184	102.0

Figure 15. Scaling the dataset df

Following scaling the data, we created our 'X' and 'y' labels. This would be useful going forward with visualisations we needed to identify outliers. Our features for the 'X' label, were the first fourteen columns of the dataset, with the fifteenth column ('Values') being our target variable ('y').

```
# Creating our features label and target variable
X = df.iloc[:,0:14]
y = df.iloc[:,-1]
```

```
X.head()
```

	LOCATION	TIME	A0101	A0102	A0103	A0104	A0105	A0106	A0107	A0108	A0109	A0110	A0111	A0112
0	1	1997	-0.394902	-0.334167	-0.406500	-0.471679	-0.390710	-0.460854	-0.462510	-0.445156	-0.453544	-0.453619	-0.368401	-0.517210
1	1	1998	-0.394116	-0.328095	-0.402795	-0.466103	-0.389824	-0.446995	-0.457922	-0.435548	-0.441328	-0.445955	-0.343586	-0.507590
2	1	1999	-0.412964	-0.338755	-0.456795	-0.482730	-0.419292	-0.468856	-0.478547	-0.443756	-0.467238	-0.480575	-0.376576	-0.531943
3	1	2000	-0.408245	-0.335875	-0.450570	-0.474204	-0.405953	-0.461238	-0.466331	-0.392574	-0.454384	-0.473952	-0.358806	-0.519730
4	1	2001	-0.403680	-0.334179	-0.444267	-0.467774	-0.409297	-0.454624	-0.466914	-0.392603	-0.448108	-0.468429	-0.361660	-0.518293

```
y.head()
```

```
0    105.0
1    106.0
2    101.0
3     90.0
4     93.0
```

Figure 16. Creating our 'X' and 'y' labels

After creating the 'X' and 'y' labels, we could create graphs to detect outliers in our data. To do this, we created multiple scatter plot graphs using seaborn (**sns**). We plotted our different feature columns over 'TIME'.

Then, using the key, we identified which colour (country) was an outlier. As seen in 'Figure 17' and 'Figure 18', location nine has a value substantially higher than the others. This occurs through each feature column.

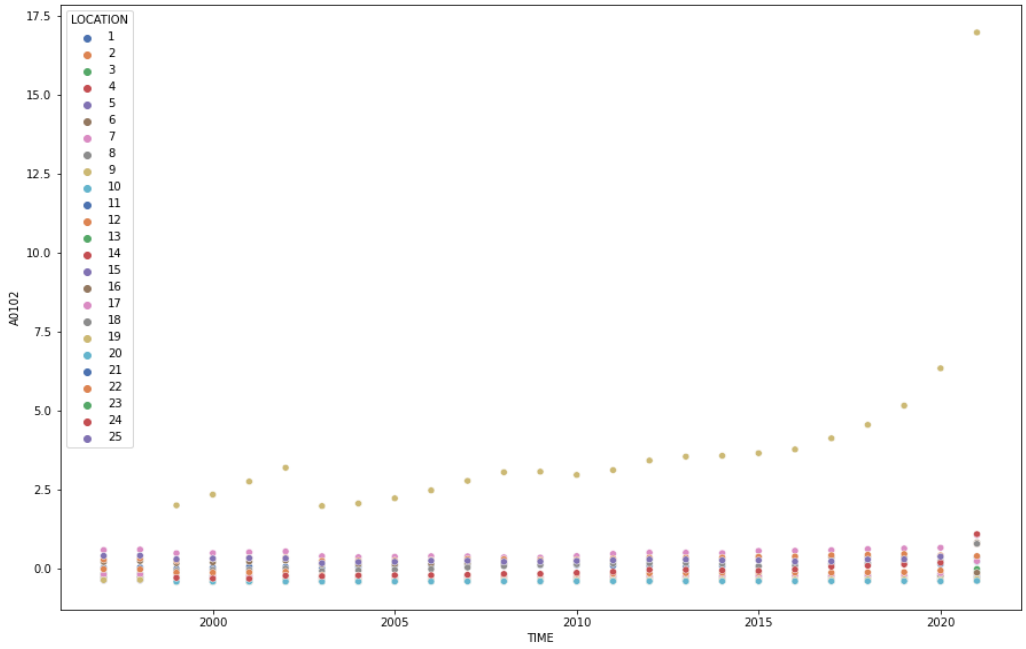

Figure 17. Food Value over time. See the location nine value in the top right

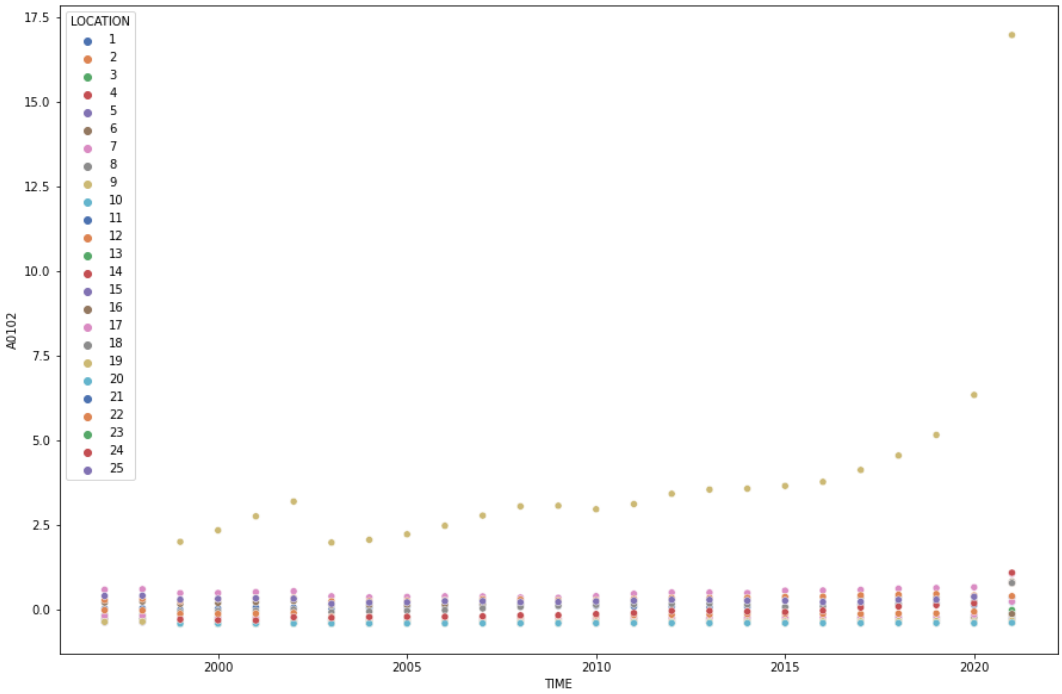

Figure 18. Alcohol Value over time. See the location nine value in the top right

We traced the outlying value back to row 270 in the data set, with the country being Hungary in 2021. We then dropped the row from the dataset (see Figure 19).

As can be seen from the figures above; Hungary's values are higher than the others, this however does not mean they are all outliers. Hungary's PPP has simply increased rapidly in previous years, the 2021 data, however, was evidently inaccurate.

```
df = df.drop(labels=270, axis=0)
```

Figure 19. Dropping line 270 from df

Subsequently, after dropping line 270, the dataset features are more closely related. This can be seen in 'Figure 20'.

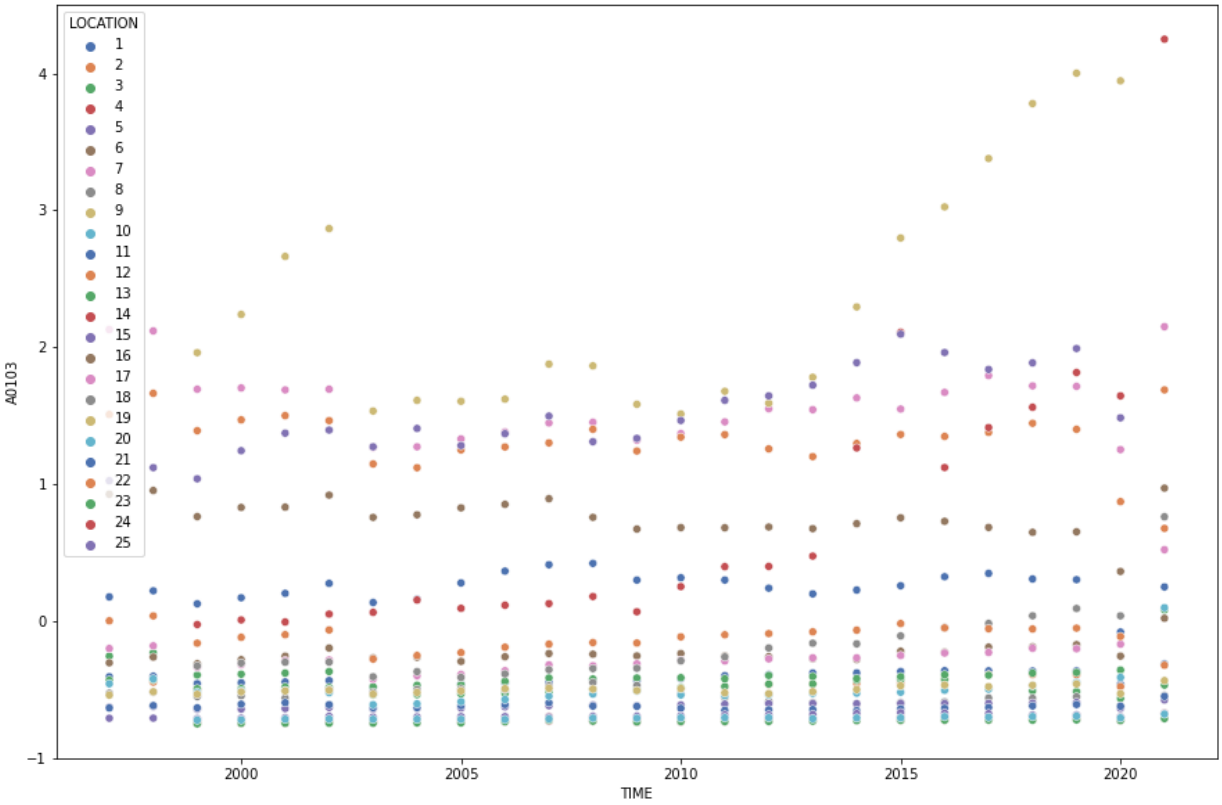

Figure 20. Feature 'A0103' after removing the outlier

To ensure the values in our 'y' label contained no outliers, we plotted their own visualisations. They all adequately followed a normal distribution, so we were happy they contained no outliers.

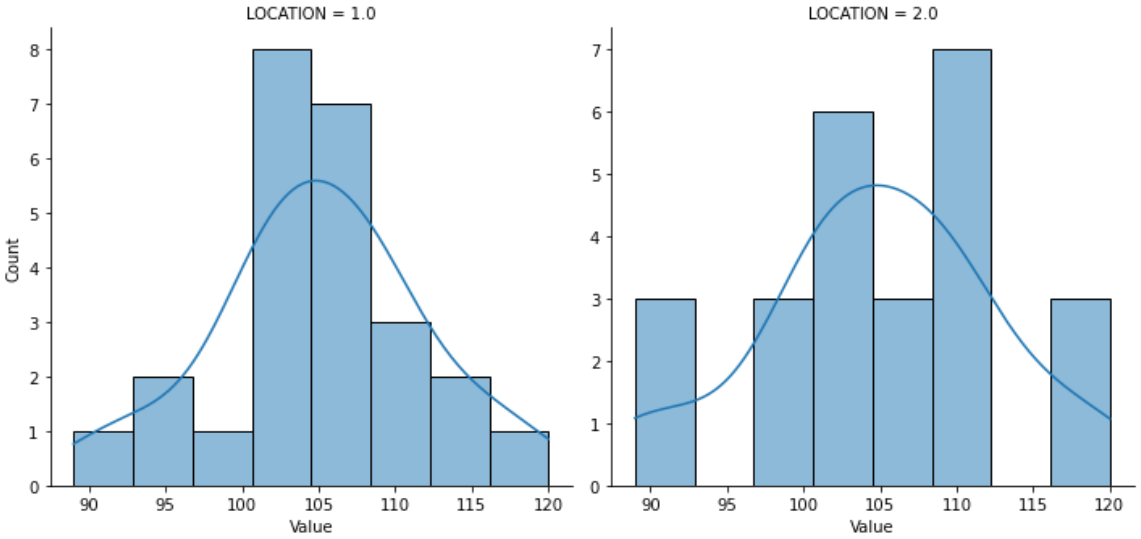

Figure 21. 'y' label visualisations for Austria and Belgium

Figure 22. 'y' label visualisations of the value for each country over time

Modeling

In this phase of the CRISP-DM methodology, we would ideally build and assess various models based on several different modeling techniques. (Nick Hotz, 2022)

The machine learning model which we opted for this dataset are listed below:

- Random Forest
- XGBoost
- Neural Network

The goal is to create different models which will be trained, tested, and evaluated based on various scoring techniques. Hyperparameters tuning will be implemented thanks to cross-validation technique to extract the most performing model that could eventually be deployed in production.

For this specific regression problem, R-squared (R^2) and Mean Squared Error (MSE) are the metrics used to evaluate the model performance:

- Mean squared error (MSE) measures the average squared difference between the predicted and actual values of the target variable. It is a measure of the model's accuracy, with lower values indicating better performance.
- R-squared (R^2) measures the proportion of variance in the target variable that is explained by the model. It ranges from 0 to 1, with higher values indicating better performance.

Random Forest – Training with no optimizations

Random Forest Regression is a supervised learning algorithm that uses ensemble learning method for regression. Ensemble learning method is a technique that combines predictions from multiple machine learning algorithms to make a more accurate prediction than a single model. (Chaya Bakshi, 2020)

In the following pages, we will proceed to describe the implementation of the Random Forest Regression model and improvements made from the last submitted report.

In the image below, the **train_test_split** function is imported from the sklearn library. Following this, the dataset is split into the training and testing sets; the **test_size** indicates what proportion of the data will be used for testing: in this case it is 30%, with the remaining 70% being used for training. The **random_state** number 42 is used for the reproducibility of the model:

```
In [55]: # Import train_test_split function
         from sklearn.model_selection import train_test_split

         # Split dataset into training set and test set
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42)
```

Figure 23. Train Test Split

A general regressor object is then created without specifying the number of n_estimators, meaning that the default value of 100 is taken. The newly created regressor is then fitted with the X_train and y_train data previously created:

```
# Create a RandomForestRegressor model and train it on the training data
rfr = RandomForestRegressor()
rfr.fit(X_train, y_train)
```

Figure 24. Simple Random Forest training

Once the regressor is fitted, R^2 and MSE scores are computed for both test and train:

```
# Make predictions on your training and test data
y_train_pred = rfr.predict(X_train)
y_test_pred = rfr.predict(X_test)

# Calculate the R2 scores
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Calculate the MSE scores
train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

print(f"R2 Training score: {train_r2:.3f}")
print(f"R2 Testing score: {test_r2:.3f}")
print()
print(f"MSE Training score: {train_mse:.3f}")
print(f"MSE Testing score: {test_mse:.3f}")
```

```
R2 Training score: 0.986
R2 Testing score: 0.906

MSE Training score: 10.665
MSE Testing score: 64.002
```

Figure 25. Simple Random Forest scoring

And the following code is used to build a graph showcasing the distribution of actual and predicted values for both train and test splits:

```
In [173]:
# Setting the size of the figure
fig, axs = plt.subplots(ncols=2, figsize=(15, 7))

# Plotting the distribution of the actual values in blue and the predicted values in orange for training data
sns.distplot(y_train, hist=False, label="Actual values training", ax=axs[0])
sns.distplot(y_train_pred, hist=False, label="Predicted values training", ax=axs[0])
axs[0].legend()

# Setting the title of the plot for training data
axs[0].set_title("Actual vs Fitted values for Training Data")

# Plotting the distribution of the actual values in blue and the predicted values in orange for test data
sns.distplot(y_test, hist=False, label="Actual values", ax=axs[1])
sns.distplot(y_test_pred, hist=False, label="Predicted values", ax=axs[1])
axs[1].legend()

# Setting the title of the plot for test data
axs[1].set_title("Actual vs Fitted values for Test Data")

# Displaying the plot
plt.show()
```

Figure 26. Simple Random Forest visualization code

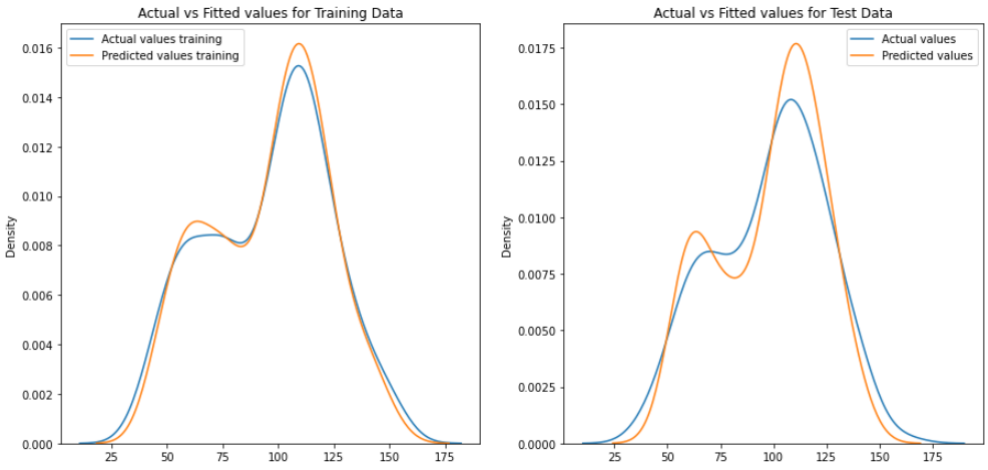

Figure 27. Simple Random Forest score visualization

Without any parameter tuning the model seems to return not too bad results if looking just at the R^2 score.

Those are the model scores:

- R2 Training score 0.986
- R2 Testing score 0.906
- MSE Training score 10.665
- MSE Testing score 64.002

In this case, the model has a high R^2 training score, which suggests that it is able to accurately predict the target variable on the training data. However, the lower testing score indicates that the model may be overfitting to the training data, and may not perform as well on new, unseen data (as we can see from the plotted graph).

For this reason, we further evaluate the model and consider ways to reduce overfitting, testing normalization and cross-validation techniques.

To showcase improvements from our latest report submission, I recall previous results we got using a Random Forest model with 1000 estimators:

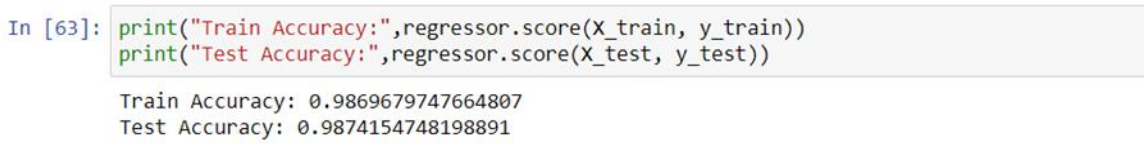

Figure 28. Previous Random Forest score results

The difference between results gotten from previous report are mainly because there was a conceptual error in how the model was trained.

Even though a train and test split was performed, the model was still trained on the whole dataset – resulting high accuracy when performing a test accuracy.

Random Forest – Training with dataset scaling

Before trying to optimize our results using cross validation techniques, we try to scale our dataset.

Feature scaling is a technique used to standardize the range of features or variables in a dataset. The goal of feature scaling is to ensure that each feature contributes equally to the analysis, as features with larger ranges can dominate the analysis.

To do so, the following code is implemented:

In [96]:
```
# Creating our features label and target variable
X = df.iloc[:,0:14]
y = df.iloc[:,-1]
y = y.values.reshape(-1,1)

PredictorScaler=StandardScaler()
TargetVarScaler=StandardScaler()

# Storing the fit object for later reference
PredictorScalerFit=PredictorScaler.fit(X)
TargetVarScalerFit=TargetVarScaler.fit(y)

# Generating the standardized values of X and y
X_transform=PredictorScalerFit.transform(X)
y_transform=TargetVarScalerFit.transform(y)

# Split the data into training and testing set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_transform, y_transform, test_size=0.3, random_state=42)

# Quick sanity check with the shapes of Training and testing datasets
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(426, 14)
(426, 1)
(183, 14)
(183, 1)
```

Figure 29. Random Forest training with scaling

And a general regressor is created and fitted:

In [175]:
```
# Create a RandomForestRegressor model and train it on the training data
rfr = RandomForestRegressor()
rfr.fit(X_train, y_train)

# Make predictions on your training and test data
y_train_pred = rfr.predict(X_train)
y_test_pred = rfr.predict(X_test)

# Calculate the R2 scores
train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Calculate the MSE scores
train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

print(f"R2 Training score: {train_r2:.3f}")
print(f"R2 Testing score: {test_r2:.3f}")
print()
print(f"MSE Training score: {train_mse:.3f}")
print(f"MSE Testing score: {test_mse:.3f}")
```

```
R2 Training score: 0.984
R2 Testing score: 0.905

MSE Training score: 0.016
MSE Testing score: 0.086
```

Figure 30. Random Forest scoring with scaling

As we can see from the screenshot the following scores are obtained:

- R2 Training score 0.984
- R2 Testing score 0.905
- MSE Training score 0.016
- MSE Testing score 0.086

Also, we can visualize how the model is behaving when plotting the actual and predicted values for both test and train:

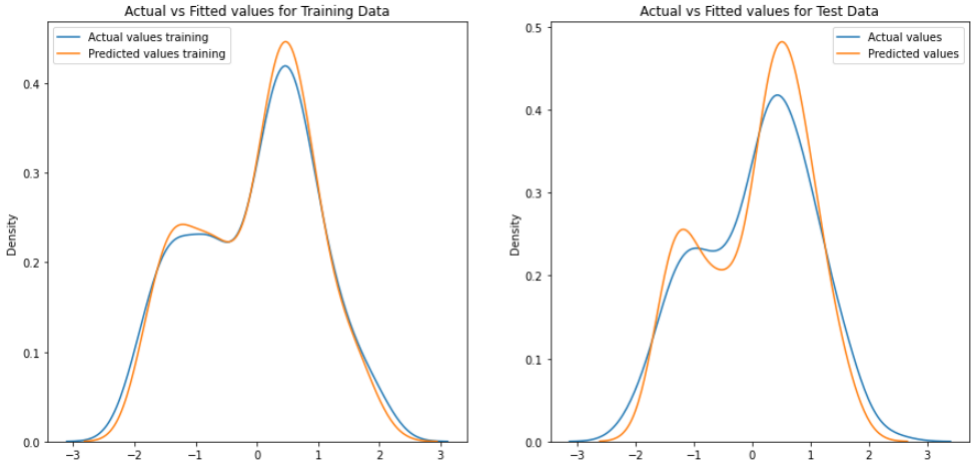

Figure 31. Scaled Random Forest score visualization

If we compare to previous run without scaling, R^2 results are not improving that much as we can still observe overfitting.

On the other hand, scaling makes the MSE scoring below 0. This is due to the fact that we are now computing the squared root on the average difference of values that are ranging from 0 to 1.

Random Forest – GridSearchCV

We now try to improve our model, hoping to increase test score while aligning it closer to the training score. To do so, we will use GridSearchCV.

Cross-validation is a technique used in machine learning to assess the performance of a model and to avoid overfitting. GridSearchCV uses cross-validation to evaluate the performance of the model for each combination of hyperparameters and to select the best hyperparameters that result in the highest performance.

Cross-validation works by splitting the data into multiple subsets or folds. We will use 5-fold cross-validation, in this way the data is split into 5 subsets of equal size. Then, the model is trained on 4 of the subsets and evaluated on the remaining subset. This process is repeated 5 times, with each subset being used as the evaluation set once. The performance of the model is then averaged over the 5 folds to get an estimate of the model's performance on new, unseen data.

The hyperparameters we are trying to tune are:

- n_estimators, the number of trees in the forest
- max_depth, the maximum depth of each tree
- min_samples_split, the minimum number of samples required to split an internal node
- min_samples_leaf, the minimum number of samples required to be at a leaf node.

```
In [108]:
# Define the parameter grid to search over
parameters = {
    'n_estimators': [50, 100, 200, 300],
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
```

Figure 32. Random Forest parameters to tune

Once the model is fitted, the following best parameters are identified:

```
In [110]: model.best_params_

Out[110]: {'max_depth': 10,
           'min_samples_leaf': 1,
           'min_samples_split': 2,
           'n_estimators': 200}
```

Figure 33. Random Forest best parameters

Produced scores are:

- R2 Training score 0.986
- R2 Testing score 0.907
- MSE Training score 0.015
- MSE Testing score 0.084

And the following plot is produced as before:

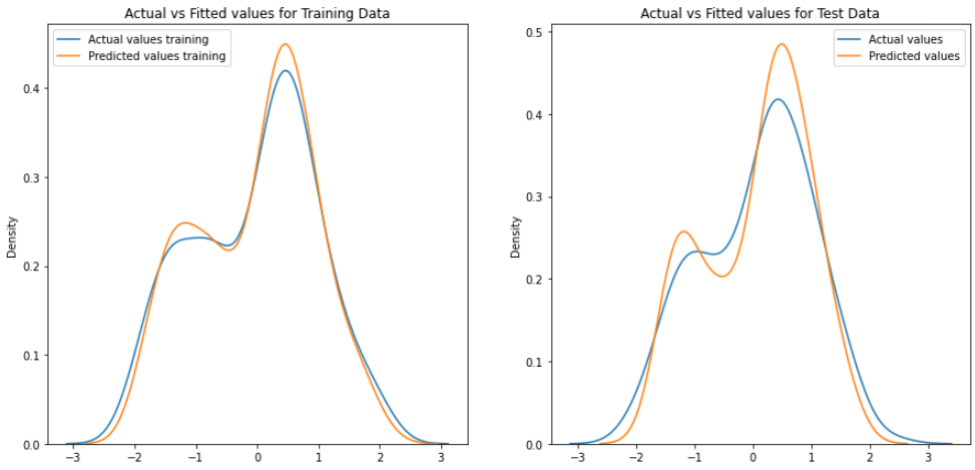

Figure 34. Best Random Forest score visualization

Overall, in below table we can summarize scores we got for all models tested:

	RFR - No optimization		RFR - Dataset scaling		RFR - GridSearchCV	
	Training	*Test*	*Training*	*Test*	*Training*	*Test*
R2 Score	0.986	0.906	0.984	0.905	0.986	0.907
MSE Score	10.665	64.002	0.016	0.086	0.015	0.084

Figure 35. Random Forest score comparison

XGBoost – Training with no optimizations

To use XGBoost, firstly, we installed all necessary dependencies used to train and evaluate the model.

```
In [1]: #pip install xgboost

In [2]: import pandas as pd
        import numpy as np
        import xgboost as xgb
        from sklearn.metrics import r2_score, mean_squared_error
        from sklearn.model_selection import train_test_split, GridSearchCV
        import seaborn as sns
        import matplotlib.pyplot as plt
        from sklearn.preprocessing import LabelEncoder
        from sklearn.preprocessing import MinMaxScaler
```

Figure 36. XGBoost libraries

When initially creating our train, test split, we used a test split size of 20%, this was altered later during our GridSearchCV optimisation.

```
In [5]: # Creating train/test split
        ## Test size is 20%
        ## Random State is 42
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 37. XGBoost train test split

We defined our model and fit the training values. As seen below the model had no hyperparameter tuning. All parameters were set to their default values.

```
In [7]: # Defining XGB Regression Model
        model = xgb.XGBRegressor()

        # Fitting training splits
        model.fit(X_train.values, y_train)

Out[7]: XGBRegressor(base_score=None, booster=None, callbacks=None,
                     colsample_bylevel=None, colsample_bynode=None,
                     colsample_bytree=None, early_stopping_rounds=None,
                     enable_categorical=False, eval_metric=None, feature_types=None,
                     gamma=None, gpu_id=None, grow_policy=None, importance_type=None,
                     interaction_constraints=None, learning_rate=None, max_bin=None,
                     max_cat_threshold=None, max_cat_to_onehot=None,
                     max_delta_step=None, max_depth=None, max_leaves=None,
                     min_child_weight=None, missing=nan, monotone_constraints=None,
                     n_estimators=100, n_jobs=None, num_parallel_tree=None,
                     predictor=None, random_state=None, ...)
```

Figure 38. Simple XGBoost training

After training the model, we evaluated the XGBoost's R2 score, MSE (Mean-Squared Error) and RMSE (Root Mean-Squared Error) and plotted them using Seaborn. These evaluation metrics were chosen as they are the most used for regression models (Agrawal, 2021). Printed below are the evaluations' values and visualisations.

```
In [90]: # Evaluating model R2 score, Mean-Squared Error and Root-mean-squared error
r2 = r2_score(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)

# Visualising Evaluations
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(15,5))
sns.regplot(x=y_test, y=y_pred, ax=axes[0])
sns.barplot(x=["R2 Score", "MSE", "RMSE"], y=[r2, mse, rmse], ax=axes[1])
plt.tight_layout()
plt.show()
```

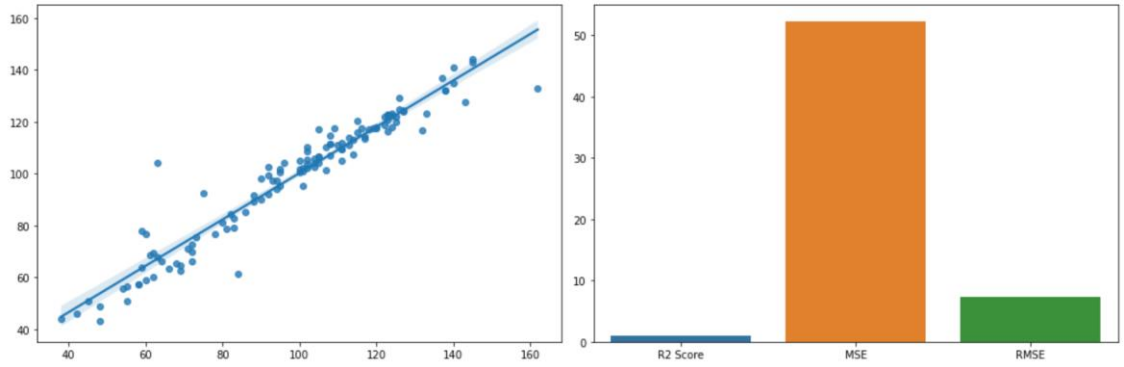

Figure 39. Simple XGBoost score visualization

```
In [10]: # Printing Evaluations
r2 = r2_score(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
print("R2 Score:", r2)
print("MSE:", mse)
print("RMSE:", rmse)

R2 Score: -35000.068091223606
MSE: 3483.6796580064497
RMSE: 59.02270459752289
```

Figure 40. Simple XGBoost score

Below we visualised the plotting of our predicted and actual values from the dataset. We had to refit the y_test values using a label encoder, as the current values would not fit with Seaborn visualisations.

```
In [11]:
# Plotting larger graph for MSE
fig, ax = plt.subplots(figsize=(10, 6))
sns.scatterplot(x=y_test, y=y_pred, ax=ax)
ax.plot(ax.get_xlim(), ax.get_ylim(), ls="--", c=".3")
ax.set_xlabel("Actual")
ax.set_ylabel("Predicted")
ax.set_title("Mean Squared Error: {:.2f}".format(mse))
ax.annotate("MSE = {:.2f}".format(mse), xy=(0.05, 0.95), xycoords='axes fraction',
            fontsize=12, ha='left', va='top', bbox=dict(boxstyle='square', facecolor='white', alpha=0.8))
plt.tight_layout()
plt.show()
```

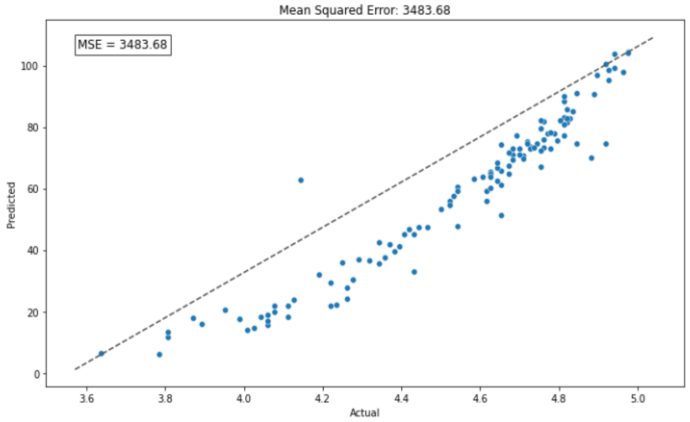

Figure 41. Simple XGBoost MSE score visualisation

To determine the best train, test split we attempted to use multiple (0.2, 0.15, 0.1, 0.05) testing values, and move forward with the best-performing result. From the visualisation below comparing the 20% and 10% testing split, we can see that the 10% split performs better with the MSE evaluation and RMSE. The difference in the R2 score is slight but not negligible.

```
# Creating train/test split
## Test size is 10%
## Random State is 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)
```

Figure 42. XGBoost train test split

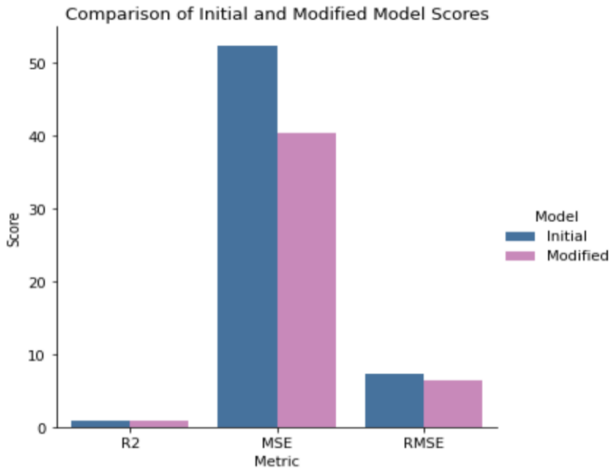

Figure 43. XGBoost comparison with initial model

XGBoost – GridSearchCV

We implemented GridSearchCV to tune hyperparameters to improve the XGBoost Regressor's performance. The parameters of 'learning_rate', 'max_depth', 'min_child_weight', 'subsample', and 'colsample_bytree', altered the decision trees used in the XGBoost model. 'n_estimators' and 'gamma', made no change to the model's performance, seen in our testing.

```
# Define hyperparameters for grid search
param_grid = {
    'learning_rate': [0.05, 0.1, 0.2],
    #'n_estimators': [100,300,600,1000],
    #'gamma': [0.1, 1, 2],
    'max_depth': [3, 5, 7],
    'min_child_weight': [1, 3, 5],
    'subsample': [0.8, 0.9, 1],
    'colsample_bytree': [0.8, 0.9, 1]
}
```

Figure 44. XGBoost parameters to tune

Cross-validation was used to maximise the data being used:

```
In [50]: # Using GridSearchCV with Cross Validation
         grid_search = GridSearchCV(model, param_grid, cv=5)
         grid_search.fit(X_train, y_train)

Out[50]: GridSearchCV(cv=5,
                      estimator=XGBRegressor(base_score=None, booster=None,
                                             callbacks=None, colsample_bylevel=None,
                                             colsample_bynode=None,
                                             colsample_bytree=None,
                                             early_stopping_rounds=None,
                                             enable_categorical=False, eval_metric=None,
                                             feature_types=None, gamma=None, gpu_id=None,
                                             grow_policy=None, importance_type=None,
                                             interaction_constraints=None,
                                             learning_rate=None, m...
                                             max_cat_to_onehot=None, max_delta_step=None,
                                             max_depth=None, max_leaves=None,
                                             min_child_weight=None, missing=nan,
                                             monotone_constraints=None, n_estimators=100,
                                             n_jobs=None, num_parallel_tree=None,
                                             predictor=None, random_state=None, ...),
                      param_grid={'colsample_bytree': [0.8, 0.9, 1],
                                  'learning_rate': [0.05, 0.1, 0.2],
                                  'max_depth': [3, 5, 7, 9, 11],
                                  'min_child_weight': [1, 3, 5],
                                  'subsample': [0.8, 0.9, 1]})
```

Figure 45. XGBoost tuning

Below are the hyperparameters found by GridSearchCV to yield the best results. These would be the parameters we go forward with in the XGBoost Regressor.

```
In [54]: print(best_params)
         {'colsample_bytree': 1, 'learning_rate': 0.2, 'max_depth': 3, 'min_child_weight': 3, 'subsample': 0.9}
```

Figure 46. XGBoost GridSearchCV best parameters

We saw a large increase in the performance of the model after using GridSearchCV.

```
R2 Score: 0.9697564286771346
MSE: 0.002995424205275527
RMSE: 0.05473046871054118
```

Figure 47. Best XGBoost GridSearchCV score

As seen in the visualisation below, our model was predicting slightly higher values than the actual values from the dataset. Still, the RMSE (as the MSE) score is highly reduced.

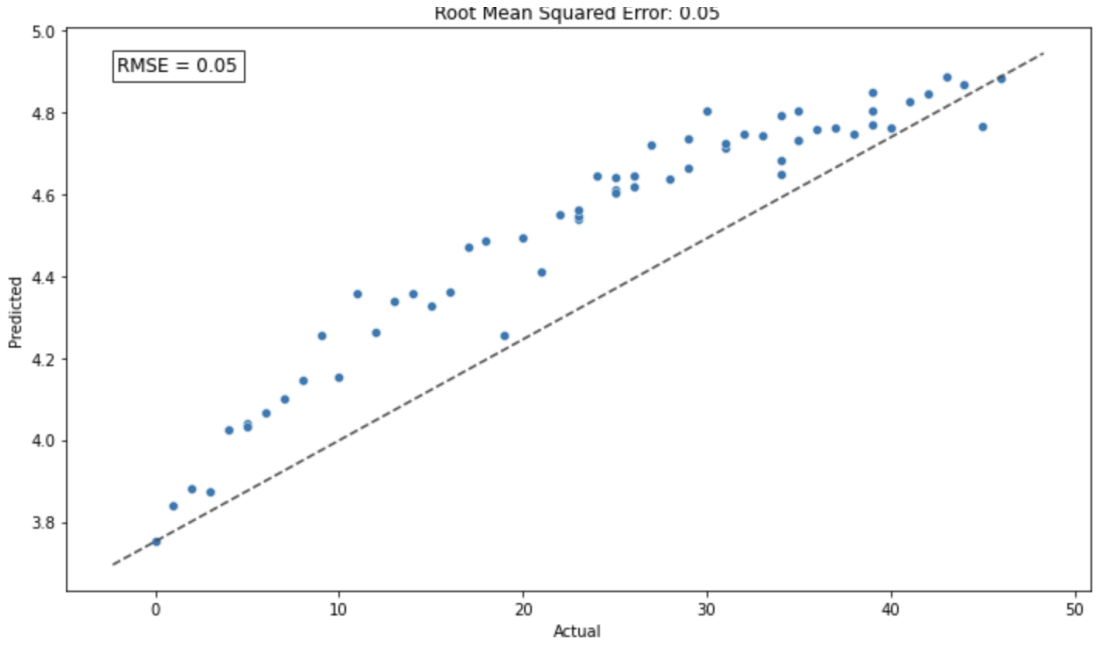

Figure 48. Best XGBoost GridSearchCV RMSE

XGBoost – L1 and L2 regularisation

In a final attempt to improve model performance, we used L1 and L2 regularisation.

For L1 Regularisation we added the alpha value to the Regressor. This minorly increased model performance:

```
In [104]: # L1 Regularisation
          model = xgb.XGBRegressor(objective='reg:squarederror', alpha=0.1, **best_params)

          model.fit(X_train, y_train)

          y_pred = model.predict(X_test)

In [105]: # Printing New Evaluations
          r2_l1 = r2_score(y_test, y_pred)
          mse_l1 = mean_squared_error(y_test, y_pred)
          rmse_l1 = np.sqrt(mse_l1)
          print("R2 Score:", r2_l1)
          print("MSE:", mse_l1)
          print("RMSE:", rmse_l1)

          R2 Score: 0.9703781677820037
          MSE: 0.0029338450900245777
          RMSE: 0.054164980291924576
```

Figure 49. XGBoost l1 optimization

For L2 Regularisation, we added the lambda value. This decreased model performance:

```
In [106]: # L2 Regularisation
          model = xgb.XGBRegressor(objective='reg:squarederror', reg_lambda=0.1, **best_params)

          model.fit(X_train, y_train)

          # Make predictions on the testing data
          y_pred = model.predict(X_test)

In [107]: # Printing New Evaluations
          r2_l2 = r2_score(y_test, y_pred)
          mse_l2 = mean_squared_error(y_test, y_pred)
          rmse_l2 = np.sqrt(mse_l2)
          print("R2 Score:", r2_l2)
          print("MSE:", mse_l2)
          print("RMSE:", rmse_l2)

          R2 Score: 0.9639284883522841
          MSE: 0.003572642858773646
          RMSE: 0.059771589060134966
```

Figure 50. XGBoost l2 optimization

Ending with below scores, we considered this to be a strong performance from the model:

- R2 score 0.964
- MSE score 0.003
- RMSE score 0.059

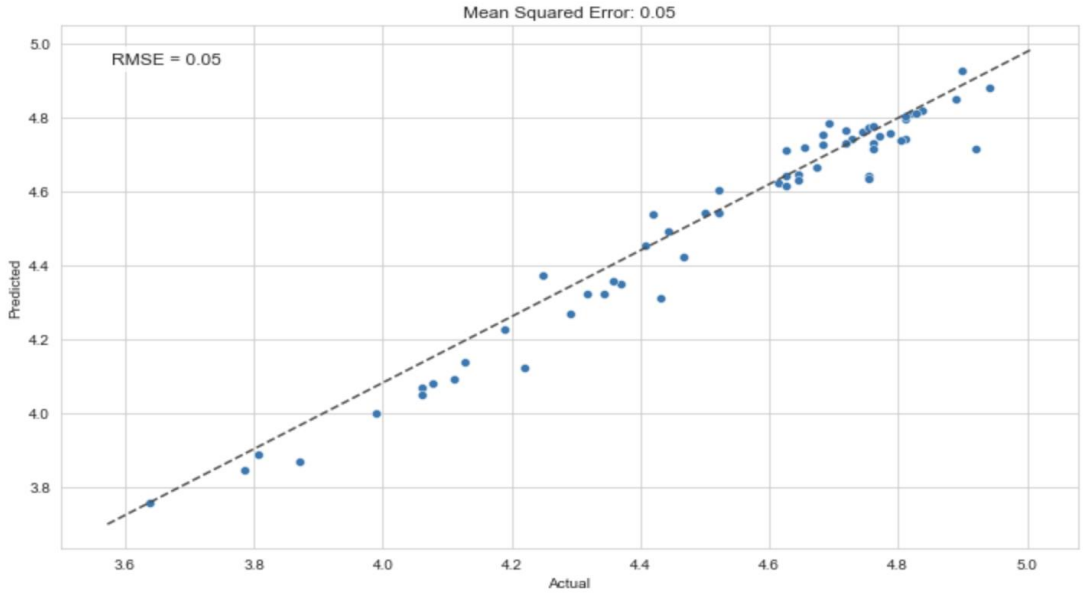

Figure 51. XGBoost l1 MSE score

In the visualisation below you can see the change in our model's RMSE scores after tuning and regularisation. The untuned model's RMSE score has been left out as it makes the tuned model's score unreadable.

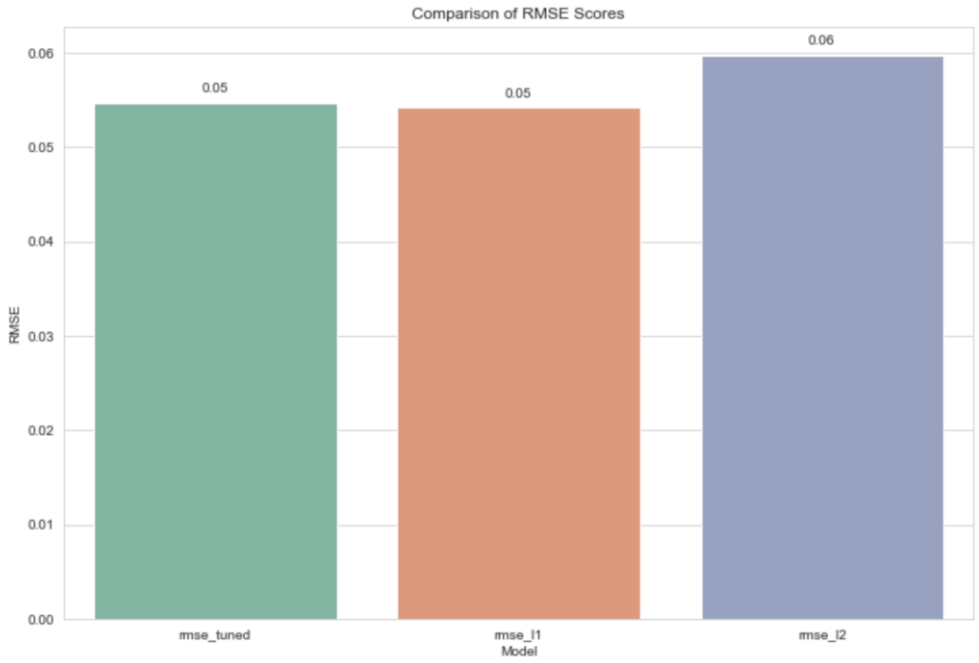

Figure 52. XGBoost models comparison

Neural Network – Training with no optimizations

Neural networks are subset of machine learning algorithm and at the heart of deep learning. They are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network. (IBM, 2023)

We initially built a very simple and standard neural network to see how performance would be.

Our starting points are:

- An input layer made of 25 units and an input dimension of 14 (due to the number of features).
- An activation set to “relu” and a kernel initializer set to “he_uniform” for the input layer.
- An output layer made of 1 unit (due to the regression problem).
- An activation set to “linear” for the output layer
- 100 epoches.
- A “mean_squared_error” loss function.
- An SDG optimizer with learning rate 0.01 and momentum 0.9.

The loss function is the only parameter that has been intentionally defined, as:

“MSE loss is the default loss to use for regression problems. Mathematically, it is the preferred loss function under the inference framework of maximum likelihood if the distribution of the target variable is Gaussian. It is the loss function to be evaluated first and only changed if you have a good reason.” (Jason Brownlee, 2019)

We now build and fit our Neural Network after scaling our dataset as done with Random Forest and XGBoost:

```
In [119]:
# define model
model = Sequential()
model.add(Dense(25, input_dim=(X_train.shape[1]), activation='relu',
kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))
```

```
In [123]:
opt = SGD(learning_rate=0.01, momentum=0.9)
model.compile(loss='mean_squared_error', optimizer=opt)

# fit model
history = model.fit(X_train, y_train, validation_data=(X_test,y_test), epochs=100)
```

Figure 53. Simple Neural Network training

Produced scores are:

- R2 Training score 0.755
- R2 Testing score 0.743
- MSE Training score 0.255
- MSE Testing score 0.233

A couple of graphs have also been built to visualize how the loss and MSE both change over time:

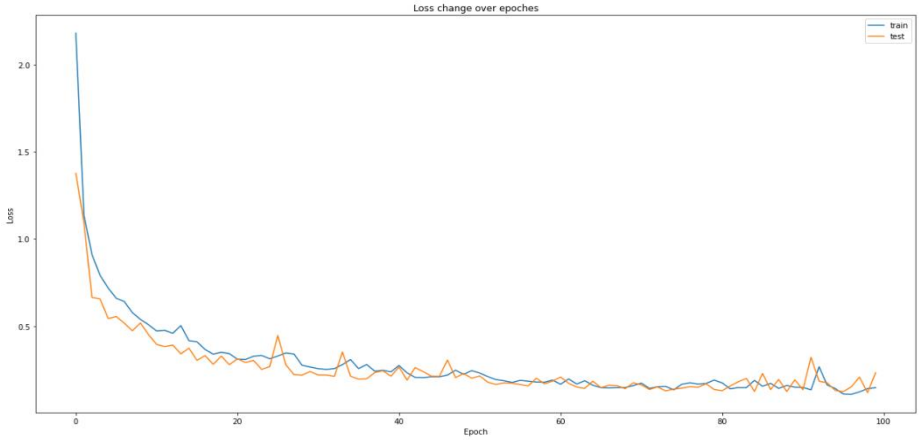

Figure 54. Simple Neural Network loss change over epochs

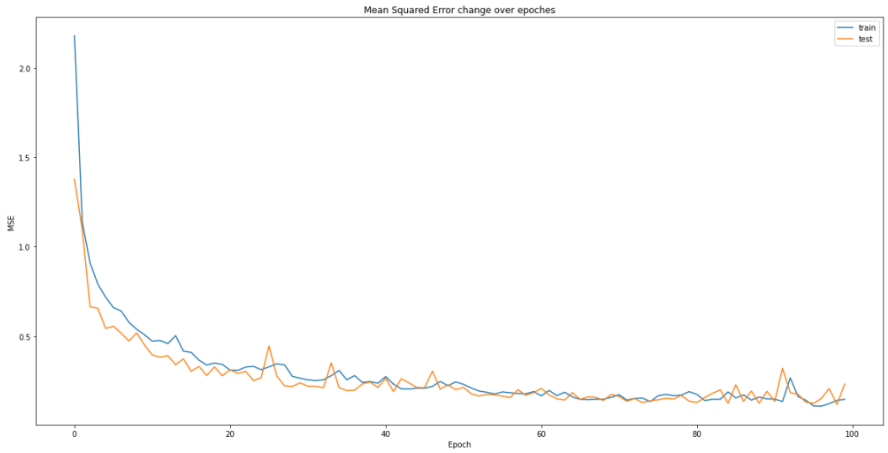

Figure 55. Simple Neural Network MSE change over epochs

Also Mean Squared Logarithmic Error Loss and Mean Absolute Error Loss have been tested as loss function on the same exact model, but the results didn't improve. For this reason, we proceeded with the optimization of the model keeping the MSE as loss function.

Neural Network – GridSearchCV

At this point, we want to optimize our model and we will specifically look at the following:

- Activation functions
- Init Mode
- Optimizer
- Learning rate
- Number of neurons
- Number of epochs

For hyperparameters selection Grid search cross-validation (GridSearchCV) is used. *"GridSearchCV process will construct and evaluate one model for each combination of parameters"*. (Jason Brownlee, 2016).

We will refer to this optimized model as "GridSearchCV Model".

All hyperparameters are first initialized:

GridSearchCV model

```
In [132]:
# activation options
activation = ['softmax', 'softplus', 'softsign', 'relu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear']

In [133]:
# epochs options
epochs = [10, 50, 100, 500]

In [134]:
# optimizer options
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']

In [135]:
# learning rate options
learn_rate = [0.001, 0.01, 0.1, 0.3]

In [136]:
# neurons options
neurons = [1, 5, 10, 20, 30, 50, 80, 100, 150, 200]

In [137]:
# initial mode options
init_mode = ['uniform', 'lecun_uniform', 'normal', 'zero', 'glorot_normal', 'glorot_uniform', 'he_normal', 'he_uniform']
```

Figure 56. Neural Network parameters to tune

Our strategy involves utilizing the GridSearchCV technique to evaluate a set of hyperparameter combinations multiple times, specifically five times. We will then select the hyperparameter that produces the best results most frequently and keep track of its frequency using a dictionary.

This approach aims to enhance the accuracy of our model and address the issue of overfitting. Our initial focus will be on evaluating the activation function for the input layer.

The following code is used:

In [138]:
```
# hyperparameter dictionary
result_activation = {
    'softmax': 0,
    'softplus': 0,
    'softsign': 0,
    'relu': 0,
    'tanh': 0,
    'sigmoid': 0,
    'hard_sigmoid': 0,
    'linear': 0
}

# parameters grid
param_grid = dict(
    model__activation = activation,
)
```

In [62]:
```
# Function to create model, required for KerasClassifier
def create_model(activation):
    # create model
    model = Sequential()
    model.add(Dense(25, input_dim=(X_train.shape[1]), activation=activation))
    model.add(Dense(1, activation='linear'))

    # Compile model
    model.compile(loss='mean_squared_error')
    return model

for i in range(5):
    # create model
    model = KerasRegressor(model=create_model, epochs=100, verbose=0)

    grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
    grid_result = grid.fit(X_train, y_train)

    result_activation[grid_result.best_params_["model__activation"]] += 1
    print("Cicle %i - Best: %f using %s" % (i+1, grid_result.best_score_, grid_result.best_params_))

result_activation
```

Figure 57. Neural Network code for hyperparameter tuning

Most of the GridSearchCV executions returned ‘relu’ as the best activation function:

```
Out[35]: {'softmax': 0,
          'softplus': 0,
          'softsign': 0,
          'relu': 4,
          'tanh': 1,
          'sigmoid': 0,
          'hard_sigmoid': 0,
          'linear': 0}
```

Figure 58. Neural Network best activation function

And the following scores were obtained:

- R2 Training score 0.619
- R2 Testing score 0.618
- MSE Training score 0.395
- MSE Testing score 0.345

Upon reviewing our results, we have observed that we have not made any significant progress in terms of improving scores from our previous model. The reason for this is that we had randomly chosen hyperparameters previously.

However, we are now in the process of systematically identifying the optimal hyperparameters step by step. In the interest of brevity, we will not include the code for each tuning, but we will report on the results we obtain as we tune different parameters. This will allow us to monitor the model's performance as it improves over time.

Step 1 – init_mode tuning

Best init_mode: normal
Best init mode scores:
- R2 Training score 0.641
- R2 Testing score 0.654
- MSE Training score 0.373
- MSE Testing score 0.313

Step 2 – optimizer tuning

Best optimizer: Adagrad
Best optimizer scores:
- R2 Training score 0.198
- R2 Testing score 0.179
- MSE Training score 0.832
- MSE Testing score 0.743

NB: score are now worst because no learning rate has been configured

Step 3 – learn_rate tuning

Best learn_rate: 0.3
Best learn_rate scores:
- R2 Training score 0.763
- R2 Testing score 0.720
- MSE Training score 0.246
- MSE Testing score 0.253

Step 4 – Number of neuros tuning

Best neurons: 50
Best neurons scores:
- R2 Training score 0.800
- R2 Testing score 0.802
- MSE Training score 0.207
- MSE Testing score 0.179

Step 5 – Number of epochs tuning

Best epochs: 500
Best epochs scores:
- R2 Training score 0.958
- R2 Testing score 0.923
- MSE Training score 0.043
- MSE Testing score 0.069

Let's also check how the loss is evolving with the increasing number of epochs:

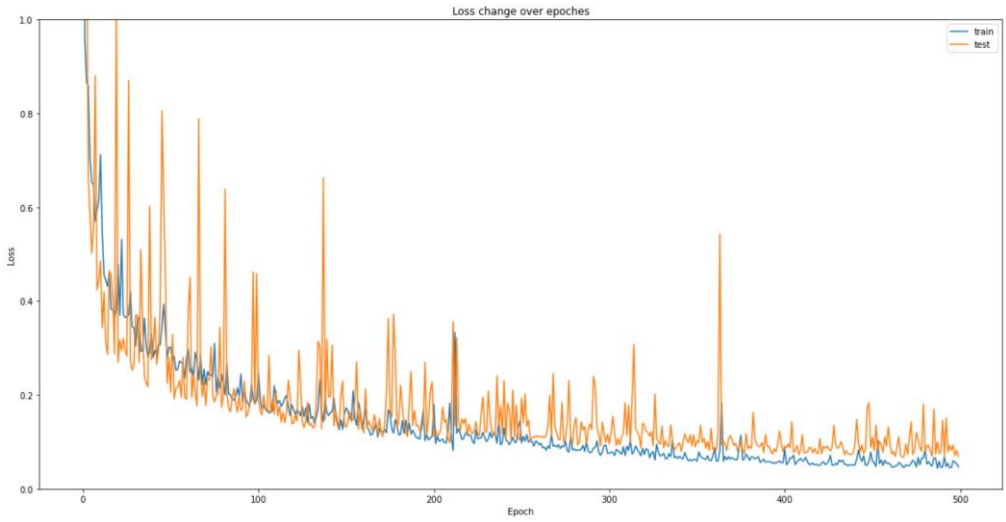

Figure 59. Best Neural Network loss change over epochs

Evaluation

The Evaluation phase looks more broadly at which model best meets the business and what to do next. (Nick Hotz, 2022)

Below table summarizes results we got with all models be trained:

	R2 Score	MSE Score
RF - No optimization	0.906	64002,00
RF - Dataset scaling	0.905	0.086
RF - GridSearchCV	0.907	0.084
XGBoost - No optimization	-35000.068	3483.679
XGBoost - GridSearchCV	0.969	0.003
XGBoost - L1 optimization	0.970	0.003
XGBoost - L2 optimization	0.964	0.003
NN - No optimization	0.743	0.233
NN - GridSearchCV	0.923	0.069

We must specify that we always started training the models performing no optimization.

This is because we wanted to be able to start from the simplest model possible, without scaling the dataset, to prove how results could be improved applying our understanding of the models implemented.

For this reason, it makes no sense to compare the non-optimized models because in some cases (RF and XGBoost) not scaling the dataset resulted in non-acceptable and comparable scoring results.

For the Random Forest model, we observed that cross-validation didn't improve our results. Our assumption is that the combination of two factors (how the random forest works and the limited dimension of the testing dataset where cross-validation was applied) didn't allow us to improve the model performance. In fact, running GridSearchCV multiple times resulted in different best hyperparameters selection that, when used to train a new model, didn't result in a significant score improvement.

For the XGBoost model, the combination of dataset scaling and cross-validation did significantly improve the model outcome. Furthermore, L1 regularization even more improved the model results.

Finally, the neural network we implemented was the most complex when going beyond the non-optimized NN. Given the huge amount of hyperparameters selection, we couldn’t run a single GridSearchCV evaluation across all hyperparameters. This probably resulted in the selection of very good parameters but possibly not the best.

Overall, if looking at the MSE and R2 scores, we can say that the best performing model is the XGBoost with hyperparameter selection using GridSearchCV and L1 regularization. The results obtained are:

- R2 score 0.97
- MSE score 0.003

It is worth nothing that result might be not completely comparable between XGBoost and Neural Network since one was trained using a 20% test split and the other a 30% test split. This might have resulted in an overfitting of the XGBoost model given the small dimension of the dataset we worked on.

Deployment

A model is not particularly useful unless the customer can access its results. (Nick Hotz, 2022)

This study is providing a step towards the right direction in predicting the efficacy and utility for a particular job offer, considering the country in which the offer is being released.

When deploying the model into production it would return the quality life index of a country given its commodity price features. This index must then be related to the salary offer with a mathematical formula – not described in this paper – to let the employee evaluate if he/she would live with a higher purchasing power that the current situation.

Overall, we believe that the model would be a good starting point when evaluating a new job offer but it is also important to consider how commodity prices might change in short periods of time. If for example, as it is currently happening, a period of high inflation would cause a data drift, meaning that MLOps principles need to be considered if the model must be used into production.

Reference List

Hotz, N. (2022). *CRISP-DM. Data Science Project Management*. Available at: https://www.datascience-pm.com/crisp-dm-2/ [Accessed: December 27, 2022].

OECD data (2022). *Prices - price level indices*. Available at: https://data.oecd.org/price/price-level-indices.htm [Accessed: December 27, 2022].

Eurostat (2022). *Purchasing power parities (PRC_PPP)*. Available at: https://ec.europa.eu/eurostat/cache/metadata/en/prc_ppp_esms.htm [Accessed: December 27, 2022].

Bakshi, C. (2020). *Random Forest Regression. Medium*. Available at: https://levelup.gitconnected.com/random-forest-regression-209c0f354c84 [Accessed: January 2, 2023].

IBM (2023). *What are Neural Networks?. IBM*. Available at: www.ibm.com. Available at: https://www.ibm.com/topics/neural-networks [Accessed: April 27, 2023].

Brownlee, J. (2019). How to Choose Loss Functions When Training Deep Learning Neural Networks. Machine Learning Mastery. Available at: https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/ [Accessed: April 27, 2023].

Agrawal, R. (2021). Know The Best Evaluation Metrics for Your Regression Model ! [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2021/05/know-the-best-evaluation-metrics-for-your-regression-model/ [Accessed 29 Apr. 2023].