**CCT College Dublin**

**Assessment Cover Page**

|  |  |
| --- | --- |
| **Module Title:** | *Programming for DA*  *Statistics for Data Analytics*  *Machine Learning for Data Analysis*  *Data Preparation & Visualisation* |
| **Assessment Title:** | *MSC\_DA\_CA1* |
| **Lecturer Name:** | *Sam Weiss*  *John O’Sullivan*  *Muhammad Iqbal*  *David McQuaid* |
| **Student Full Name:** | Emre Kalem |
| **Student Number:** | 2023047 |
| **Assessment Due Date:** | 14/04/2023 |
| **Date of Submission:** | 14/04/2023 |

**Declaration**

|  |
| --- |
| By submitting this assessment, I confirm that I have read the CCT policy on Academic Misconduct and understand the implications of submitting work that is not my own or does not appropriately reference material taken from a third party or other source. I declare it to be my own work and that all material from third parties has been appropriately referenced. I further confirm that this work has not previously been submitted for assessment by myself or someone else in CCT College Dublin or any other higher education institution. |

**Abstract**

This study aims to analyze the housing market trends in Ireland from 1978 to 2019 using a dataset containing the number of new home registrations in each city on a yearly basis. The analysis includes data visualization, data distribution, and machine learning techniques. The data was processed using Python programming language.

The machine learning model trained with my current values can predict the number of new home registrations for future years with reasonable accuracy.

# import libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

# read the data from the csv file and store it in a dataframe called nhr\_df

nhr\_df = pd.read\_csv('newhouseregistration.csv')

I downloaded that data file from the data.gov.ie website and saved it to my computer under the name **“newhouseregistration.csv”**

In order to use this file as a data frame in my project, I take the data from the CSV file using the "pandas" library and define it as a data frame named **“nhr\_df”**

(\*NHR meaning = New House Registration)

# it will show a tuple of the number of rows and columns in the dataframe

nhr\_df.shape

(1134, 8)

I will use the 'shape' feature to learn the numbers of rows and columns in my dataset. The result I will get here will allow me to have no idea what kind of analytical research and what kind of visualizations I will use. It will also help me understand the size and structure of the data in my dataset.

# it will show the first 5 rows of the dataframe

nhr\_df.head()

|  | **STATISTIC** | **STATISTIC Label** | **TLIST(A1)** | **Year** | **C02339V02812** | **County** | **UNIT** | **VALUE** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | HSA10 | New House Registrations | 1978 | 1978 | - | All Counties | Number | 3781 |
| 1 | HSA10 | New House Registrations | 1978 | 1978 | 01 | Carlow | Number | 1 |
| 2 | HSA10 | New House Registrations | 1978 | 1978 | 02 | Dublin | Number | 1761 |
| 3 | HSA10 | New House Registrations | 1978 | 1978 | 03 | Kildare | Number | 154 |
| 4 | HSA10 | New House Registrations | 1978 | 1978 | 04 | Kilkenny | Number | 82 |

# it will show the last 5 rows of the dataframe

nhr\_df.tail()

|  | **STATISTIC** | **STATISTIC Label** | **TLIST(A1)** | **Year** | **C02339V02812** | **County** | **UNIT** | **VALUE** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1129 | HSA10 | New House Registrations | 2019 | 2019 | 22 | Roscommon | Number | 7 |
| 1130 | HSA10 | New House Registrations | 2019 | 2019 | 23 | Sligo | Number | 0 |
| 1131 | HSA10 | New House Registrations | 2019 | 2019 | 24 | Cavan | Number | 7 |
| 1132 | HSA10 | New House Registrations | 2019 | 2019 | 25 | Donegal | Number | 67 |
| 1133 | HSA10 | New House Registrations | 2019 | 2019 | 26 | Monaghan | Number | 5 |

# it will show the information about the dataframe

nhr\_df.info()

I used the “info()” method to gain a better understanding of the data types for each column, the number of non-null values, and the memory usage of the DataFrame. This helps us identify potential data quality issues and decide on appropriate data cleaning and preprocessing steps. It also allows us to optimize the memory usage of the DataFrame for more efficient data manipulation.

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 1134 entries, 0 to 1133

Data columns (total 8 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 STATISTIC 1134 non-null object

1 STATISTIC Label 1134 non-null object

2 TLIST(A1) 1134 non-null int64

3 Year 1134 non-null int64

4 C02339V02812 1134 non-null object

5 County 1134 non-null object

6 UNIT 1134 non-null object

7 VALUE 1134 non-null int64

dtypes: int64(3), object(5)

memory usage: 71.0+ KB

Since I have both numeric and object type values ​​in my dataset, I want to apply the “describe()” method in different ways. Because I know that the “describe()” method provides us with statistics of numeric data.

# it will return a new DataFrame that includes count, mean, standard deviation, minimum, maximum, and quartile information for each numerical column in nhr\_df.

nhr\_df.describe()

|  | **TLIST(A1)** | **Year** | **VALUE** |
| --- | --- | --- | --- |
| count | 1134.000000 | 1134.000000 | 1134.000000 |
| mean | 1998.500000 | 1998.500000 | 1353.728395 |
| std | 12.126266 | 12.126266 | 5045.279643 |
| min | 1978.000000 | 1978.000000 | 0.000000 |
| 25% | 1988.000000 | 1988.000000 | 51.250000 |
| 50% | 1998.500000 | 1998.500000 | 208.500000 |
| 75% | 2009.000000 | 2009.000000 | 831.750000 |
| max | 2019.000000 | 2019.000000 | 66649.000000 |

# it will return a new DataFrame that includes count, unique, top, and frequency information for each non-numerical column in nhr\_df.

nhr\_df.describe(include='object')

|  | **STATISTIC** | **STATISTIC Label** | **C02339V02812** | **County** | **UNIT** |
| --- | --- | --- | --- | --- | --- |
| count | 1134 | 1134 | 1134 | 1134 | 1134 |
| unique | 1 | 1 | 27 | 27 | 1 |
| top | HSA10 | New House Registrations | - | All Counties | Number |
| freq | 1134 | 1134 | 42 | 42 | 1134 |

Here, I observe that in the county column, they collect all county values ​​for each year and also create data under the name **“All Counties”** as a line. I want to remove these lines as I think this line is unnecessary for me and will also mislead processes. Because if I want to get the total value in years for all counties, I can do it myself with a different code.

# This code drops all rows in the nhr\_df DataFrame where the value in the 'County' column is equal to 'All Counties'. After running this code, the original DataFrame will be modified to exclude those rows.

nhr\_df.drop(nhr\_df[nhr\_df['County'] == 'All Counties'].index, inplace = True)

The given code drops all rows from the dataset that have a 'County' value of 'All Counties'. To achieve this, the 'drop' method is called on the “nhr\_df” dataframe with the argument being a boolean index that selects only the rows where the 'County' column is equal to 'All Counties'. The “inplace” parameter is set to “**True**” to apply the changes directly to the original dataframe.

# it will show the first 5 rows of the dataframe

nhr\_df.head()

|  | **STATISTIC** | **STATISTIC Label** | **TLIST(A1)** | **Year** | **C02339V02812** | **County** | **UNIT** | **VALUE** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | HSA10 | New House Registrations | 1978 | 1978 | 01 | Carlow | Number | 1 |
| 2 | HSA10 | New House Registrations | 1978 | 1978 | 02 | Dublin | Number | 1761 |
| 3 | HSA10 | New House Registrations | 1978 | 1978 | 03 | Kildare | Number | 154 |
| 4 | HSA10 | New House Registrations | 1978 | 1978 | 04 | Kilkenny | Number | 82 |
| 5 | HSA10 | New House Registrations | 1978 | 1978 | 05 | Laois | Number | 13 |

# it will show a tuple of the number of rows and columns in the dataframe

nhr\_df.shape

(1092, 8)

# This code checks for missing values in each column of the nhr\_df DataFrame and returns a count of the missing values for each column.

nhr\_df.isnull().sum()

STATISTIC 0

STATISTIC Label 0

TLIST(A1) 0

Year 0

C02339V02812 0

County 0

UNIT 0

VALUE 0

dtype: int64

The code "nhr\_df.isnull().sum()" is used to check the number of missing values in each column of the "nhr\_df" dataframe. It first applies the “isnull()” method to identify the missing values, which returns a boolean dataframe with "True" for missing values and "False" otherwise. The “sum()” method is then applied to count the number of missing values in each column, which is returned as a pandas series. This allows us to quickly identify if any columns have missing values, and if so, how many missing values they have.

# This code identifies and selects all rows in nhr\_df that have identical values in both the 'Year' and 'County' columns, indicating potential duplicates or data quality issues. The resulting rows are stored in the duplicate\_rows DataFrame.

duplicate\_rows = nhr\_df[nhr\_df.duplicated(['Year', 'County'])]

# it will show the number of duplicate rows in the dataframe

duplicate\_rows.shape

(0, 8) // Empty DataFrame

The code duplicate\_rows = nhr\_df[nhr\_df.duplicated(['Year', 'County'])] uses the Pandas duplicated() method to identify any duplicate rows in the nhr\_df DataFrame based on the Year and County columns. Specifically, it returns a new DataFrame called duplicate\_rows that contains all the rows in nhr\_df that have duplicate values in both the Year and County columns. This is accomplished by passing a list of column names ['Year', 'County'] to the duplicated() method as an argument. The resulting DataFrame can be used to examine and possibly remove any duplicate rows in the original DataFrame.

The dataset consists of **1134 entries and 8 columns**. The columns are 'STATISTIC', 'STATISTIC Label', 'TLIST(A1)', 'Year', 'C02339V02812', 'County', 'UNIT', and 'VALUE'.

The 'STATISTIC' and 'STATISTIC Label' columns provide information on the statistical measure and its label, respectively. The 'TLIST(A1)' column is an integer column that is not relevant to the analysis. The 'Year' column provides information on the year of the observation. The 'C02339V02812' column contains categorical data and provides information on the industry sector. The 'County' column provides information on the geographical location of the observation. The 'UNIT' column contains information on the unit of measurement for the statistical measure.

To summarize the dataset, I can use descriptive statistics to better understand the distribution of the data. Here are some relevant descriptive statistics for the **“VALUE”** column:

# calculate descriptive statistics of the Value column

mean = nhr\_df['VALUE'].mean()

std\_dev = nhr\_df['VALUE'].std()

minimum = nhr\_df['VALUE'].min()

maximum = nhr\_df['VALUE'].max()

median = nhr\_df['VALUE'].median()

# print the statistics

print('Mean:', mean)

print('Standard deviation:', std\_dev)

print('Minimum:', minimum)

print('Maximum:', maximum)

print('Median:', median)

I calculated the statistics of the column named VALUE from the columns in my dataframe nhr\_df, which is a pandas dataframe. The statistical information I calculate ices values ​​such as mean value, standard deviation value, minimum and maximum value, and median. When I calculated these values, I made a statistical calculation and gathered information about the properties of the data set.

The information I calculated here shows the distribution of the numerical data of the column named VALUE of my dataset, the central trainings and the endpoints. When I need this information in my project, I will be able to use it for various operations such as measurements, data exploration, hope testing or decision making.

Mean: 702.8974358974359

Standard deviation: 1636.359993950272

Minimum: 0

Maximum: 18714

Median: 191.0

Based on the values ​​I calculated here, I saw that the mean value was significantly larger than the median, which shows me that my data is right-skewed. Also, an excessively large standard deviation indicates that my data is highly dispersed.

# import plotly.express as px

import plotly.express as px

# select the rows where the 'County' column is equal to 'Dublin'

dublin\_df = nhr\_df[nhr\_df['County'] == 'Dublin']

# hover\_name and text are used to display the name of the county and the year when the user hovers over the line

fig = px.line(dublin\_df, x='Year', y='VALUE', title='Dublin New House Registration', color='County', hover\_name='County', text='Year')

# display the year below the line

fig.update\_traces(textposition="bottom right")

# display the plot

fig.show()

Graphical user interface, chart, line chart

Description automatically generated

To visualize the data for Dublin City on a yearly basis, I first filtered the dataframe to only include rows where the **“County”** column was equal to **“Dublin”**. Then, I grouped the filtered dataframe by the **“Year”** column and took the sum of the **“VALUE”** column for each year. I created a line chart using the Plotly express library. Here I wanted to use the records of the province of dublin from my dataset. I used YEAR as x and VALUE as y in my chart where the number of newly registered houses in the city of Dublin is visualized. This helped me understand the number of newly registered homes per year for the city of Dublin.

# This code create line plot

fig = px.line(nhr\_df, x='Year', y='VALUE', title='New House Registration', color='County', hover\_name='County', text='Year')

# display the year below the line

fig.update\_traces(textposition="bottom right")

# display the plot

fig.show()

Chart, line chart

Description automatically generated

# This code create bar plot

fig = px.bar(nhr\_df, x='Year', y='VALUE', title='New House Registration', color='County', hover\_name='County')

# display the plot

fig.show()

Chart

Description automatically generated with low confidence

Here again I wanted to take advantage of the great graphics of the plolty express library. This time I will use bar graph instead of line graph. And this time, I wanted to show the values ​​of all provinces of Ireland on year basis, not for Dublin province. We know that in my data set I have the number of newly registered houses from all provinces of Ireland on a yearly basis from 1978 to 2019. Now I will try to analyze them better by visualizing them with graphics. I think the Poisson distribution is suitable for measuring count samples where the occurrence features are constant over time and independent. I use the Poisson distribution to organize new housing records in each county over time. I assume that new home records are constant over time and that the records in each county are independent of each other.

**Distribution**

The Poisson distribution has the property to be discrete and only takes non-negative integer values, making it suitable values ​​for census data such as the number of new home registrations in each county. The mean of the Poisson distribution, the way it occurs, and the variance is the size of the mean; this is a useful feature for the count display example where variance is usually assets with mean.

In summary, the Poisson distribution is suitable for census view measures with a constant rate of occurrence and independent conditions, which apply to our Irish housing dataset. The discrete nature and useful properties of the Poisson distribution make it a suitable choice for modeling census data, such as the number of new housing registrations in each county over time.

I will be separating the records for Dublin city from the rest of the counties, and applying a Poisson distribution to model the number of new housing units registered in Dublin city over the given time period.

# estimate of the true mean

lambda\_hat = np.mean(dublin\_df["VALUE"] / 52)

print(lambda\_hat)

117.75228937728939

I calculate the lambda value to apply poisson for the city of Dublin. I calculate this by averaging the values ​​in the value column.

# import the poisson function from scipy.stats

from scipy.stats import poisson

# this code calculates the probability of observing exactly 120 new house registrations in Dublin, assuming that the rate of new house registrations follows a Poisson distribution with a mean parameter estimated from the data.

poisson.pmf(k=120, mu=lambda\_hat)

0.0356254036364004

The code calculates the probability mass function (PMF) of a Poisson distribution with a mean parameter of lambda\_hat = 117.75228937728939, where the random variable represents the number of new home registrations in Dublin. The PMF gives the probability of observing a specific number of new home registrations, given the mean parameter. In this case, the specific number is k = 120.

The Poisson distribution is commonly used to model rare events, and the mean parameter lambda\_hat represents the average number of events per unit of time or space. In this context, lambda\_hat represents the average number of new home registrations in Dublin.

The result of the code poisson.pmf(k=120, mu=lambda\_hat) gives the probability of observing k=120 new home registrations in Dublin, given the average number of registrations per unit of time or space is lambda\_hat. This probability can be interpreted as the likelihood of such an event occurring under the given conditions.

# k values from 100 to 150

k\_values = np.arange(80, 160, 1)

# calculate the pmf values

pmf\_values = poisson.pmf(k\_values, lambda\_hat)

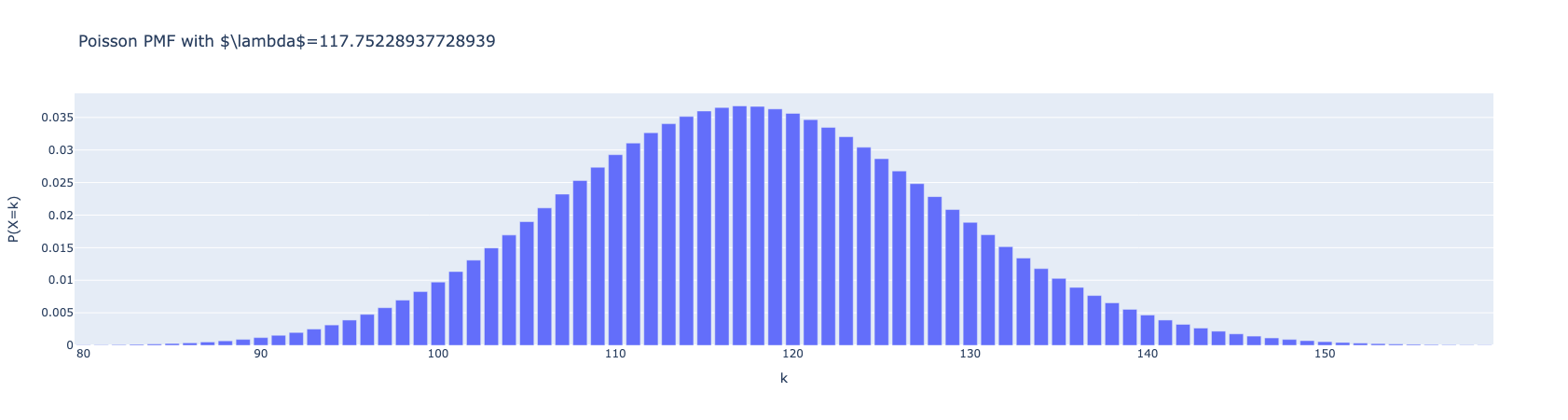
title = r'$\text{Poisson PMF with }\lambda \text{ =}$'

# plot the graph

fig = px.bar(x=k\_values, y=pmf\_values, labels={'x':'k', 'y':'P(X=k)'}, title=f"{title} + {lambda\_hat}")

# show the graph

fig.show()



The code first creates an array of k values from 80 to 160 using the numpy.arange() function with a step size of 1. These k values represent the number of new home registrations that may occur in Dublin city between 1978 and 2019.

Next, the code calculates the PMF values for each k value using the poisson.pmf() function with the k\_values array and lambda\_hat as arguments. The PMF gives the probability of the number of events that occur being exactly equal to a specified value.

Then, the code plots the PMF values against the k values using the matplotlib.pyplot.plot() function. The x-axis represents the possible k values, and the y-axis represents the corresponding probabilities of those k values occurring.

The code sets the x-label to 'k' using the xlabel() function, and the y-label to 'P(X=k)' using the ylabel() function. The code also sets the title of the plot to 'Poisson PMF with $\lambda$=lambda\_hat' using the title() function. The $\lambda$ symbol is a LaTeX symbol for lambda.

Finally, the code displays the plot using the show() function. The resulting plot shows the probability of each possible k value occurring in a Poisson distribution with an average value of lambda\_hat.

# calculate the cdf values

poisson.cdf(k=120, mu=lambda\_hat)

0.6054782053067237

Here we tried to calculate the number of new home registrations in the city of Dublin by the week with the poisson distribution. Now I used the CDF property of the poisson distribution to get the statistical result when this weekly number is less than or equal to 120.

# calculate the survival function values

poisson.sf(k=120, mu=lambda\_hat)

0.3945217946932763

The SF of a Poisson distribution gives the probability that the number of events that occur is greater than a specified value, in this case, 120.

The average number of new home registrations in Dublin city between 1978 and 2019 is denoted by lambda\_hat and its value is 117.75228937728939. This value is the expected number of new home registrations in Dublin city in a given year, assuming that the rate of new home registrations is constant over time and events occur independently.

The code poisson.sf(k=120, mu=lambda\_hat) calculates the survival function (SF) of the Poisson distribution. The SF of a Poisson distribution gives the probability that the number of events that occur is greater than a specified value, in this case, 120. Therefore, the code calculates the probability that the number of new home registrations in Dublin city between 1978 and 2019 is greater than 120.

In other words, the SF gives the complement of the cumulative distribution function (CDF) of the Poisson distribution. The CDF gives the probability that the number of events that occur is less than or equal to a specified value, whereas the SF gives the probability that the number of events that occur is greater than a specified value.

# calculate the quantile values

poisson.ppf(q=0.99, mu=lambda\_hat)

This code calculates the percent-point function (PPF) of a Poisson distribution with an average value of lambda\_hat. The PPF of a Poisson distribution gives the smallest integer value of k such that the cumulative probability up to k is greater than or equal to a specified quantile q, in this case, 0.99. Therefore, the code poisson.ppf(q=0.99, mu=lambda\_hat) calculates the number of new home registrations in Dublin city between 1978 and 2019 that will not be exceeded with at least a 99% probability.

In other words, the PPF gives the inverse of the cumulative distribution function (CDF) of the Poisson distribution. The CDF gives the probability that the number of events that occur is less than or equal to a specified value, whereas the PPF gives the value of k such that the probability of the number of events that occur being less than or equal to k is equal to or greater than the specified quantile q.

In summary, the code poisson.ppf(q=0.99, mu=lambda\_hat) calculates the number of new home registrations in Dublin city between 1978 and 2019 that will not be exceeded with at least a 99% probability, assuming that the rate of new home registrations is constant over time and events occur independently.

**Machine Learning**

# import libraries for LabelEncoder

from sklearn.preprocessing import LabelEncoder

# Check the data types of the columns

county\_column\_type = nhr\_df['County'].dtypes

# Check the column type if not int64

if county\_column\_type != 'int64':

# Initialize a LabelEncoder object

labelEncoder = LabelEncoder()

# Fit the LabelEncoder object to the 'County' column of the DataFrame and transform the column values

county\_codes = labelEncoder.fit\_transform(nhr\_df['County'])

# Replace county names with numeric codes in the 'County' column of the DataFrame

nhr\_df['County'] = county\_codes

In order to do machine learning processes, I first need to make my categorical column suitable for machine learning. I will use a label encoder for this. Thanks to this process, I update my column in my dataset with newly created numeric values ​​by creating numeric values ​​corresponding to the data in my column stored as text.

Now I wanted to check if my column has an int64 value. Because I wanted to do this check so that if the code is run again after doing this process, it will not fall into a process error.

# first 5 rows of the data

nhr\_df.head()

|  | **STATISTIC** | **STATISTIC Label** | **TLIST(A1)** | **Year** | **C02339V02812** | **County** | **UNIT** | **VALUE** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | HSA10 | New House Registrations | 1978 | 1978 | 01 | 0 | Number | 1 |
| 2 | HSA10 | New House Registrations | 1978 | 1978 | 02 | 5 | Number | 1761 |
| 3 | HSA10 | New House Registrations | 1978 | 1978 | 03 | 8 | Number | 154 |
| 4 | HSA10 | New House Registrations | 1978 | 1978 | 04 | 9 | Number | 82 |
| 5 | HSA10 | New House Registrations | 1978 | 1978 | 05 | 10 | Number | 13 |

After the operation, when I print the first 5 lines of my data set, you can see that our County column has changed from categorical data to numeric data.

Supervised learning is a type of machine learning in which a model is trained on a dataset containing labeled examples, that is, each sample contains both input properties and their corresponding output values. Supervised learning aims to learn a function that maps input properties to corresponding output values. This learned function can then be used to predict output values ​​for new input features that the model has not encountered before. Based on this information, I will try to explain using supervised learning and why I use it.

# Import necessary libraries

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

# Remove unnecessary columns

data = nhr\_df.drop(['STATISTIC', 'STATISTIC Label', 'TLIST(A1)', 'C02339V02812', 'UNIT'], axis=1)

# Split the data into features and target

X = data.drop(['VALUE'], axis=1)

y = data['VALUE']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Scale the data using StandardScaler

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

In this particular project, I aim to predict the number of new house registrations in Dublin on a yearly basis 2020. I can employ supervised learning since I have a labeled dataset that includes previous years' new house registration counts in Dublin as input features, along with their corresponding output values. By training a model on this labeled dataset, I can learn a function that maps the input features to the output values, which I can then use to forecast the number of new house registrations for future years.

There are various types of supervised learning algorithms, such as regression and classification. In this case, I can use regression algorithms since I aim to predict a numerical value, specifically, the number of new house registrations for each year.

# Split the data into features and target

X = data.drop(['VALUE'], axis=1)

y = data['VALUE']

In order to use it in machine learning, I start my operations by specifying a data variable and removing the columns that I do not need from my data set. I will use the year county and value columns. I removed all columns except these from my dataset. Then I want to use the other 2 columns by removing the value column from the year county and value columns in the new yield data to define my variable, X.

# Import necessary libraries

from sklearn.linear\_model import LinearRegression

from sklearn.tree import DecisionTreeRegressor

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean\_squared\_error

from sklearn.metrics import r2\_score

# Initialize the models

lin\_reg = LinearRegression()

tree\_reg = DecisionTreeRegressor(random\_state=42)

forest\_reg = RandomForestRegressor(random\_state=42)

# Train the models

lin\_reg.fit(X\_train, y\_train)

tree\_reg.fit(X\_train, y\_train)

forest\_reg.fit(X\_train, y\_train)

# Make predictions on the test set

lin\_reg\_pred = lin\_reg.predict(X\_test)

tree\_reg\_pred = tree\_reg.predict(X\_test)

forest\_reg\_pred = forest\_reg.predict(X\_test)

# Evaluate the models using mean squared error

lin\_reg\_mse = mean\_squared\_error(y\_test, lin\_reg\_pred)

tree\_reg\_mse = mean\_squared\_error(y\_test, tree\_reg\_pred)

forest\_reg\_mse = mean\_squared\_error(y\_test, forest\_reg\_pred)

lin\_reg\_r2 = r2\_score(y\_test, lin\_reg\_pred)

tree\_reg\_r2 = r2\_score(y\_test, tree\_reg\_pred)

forest\_reg\_r2 = r2\_score(y\_test, forest\_reg\_pred)

print('Linear Regression R2 score:', lin\_reg\_r2)

print('Decision Tree Regression R2 score:', tree\_reg\_r2)

print('Random Forest Regression R2 score:', forest\_reg\_r2)

print('Linear Regression MSE:', lin\_reg\_mse)

print('Decision Tree Regression MSE:', tree\_reg\_mse)

print('Random Forest Regression MSE:', forest\_reg\_mse)

Linear Regression R2 score: 0.025357299200511663

Decision Tree Regression R2 score: 0.6269475625901089

Random Forest Regression R2 score: 0.8673648173222261

Linear Regression MSE: 2262020.721933107

Decision Tree Regression MSE: 865806.8675799087

Random Forest Regression MSE: 307829.25007123285

# Import necessary libraries

import matplotlib.pyplot as plt

# Compare the results

models = ['Linear Regression', 'Decision Tree Regression', 'Random Forest Regression']

# Mean Squared Error (MSE) and R2 Score Comparison

mse\_scores = [lin\_reg\_mse, tree\_reg\_mse, forest\_reg\_mse]

r2\_scores = [lin\_reg\_r2, tree\_reg\_r2, forest\_reg\_r2]

# Plot the results in a bar chart for mse scores

plt.bar(models, mse\_scores, color='red')

plt.title('Mean Squared Error (MSE) Comparison')

plt.ylabel('MSE')

# xticks rotation 90 degrees

plt.xticks(rotation=20)

plt.show()

# Plot the results in a bar chart for r2 scores

plt.bar(models, r2\_scores)

plt.title('R2 Score Comparison')

plt.ylabel('R2 Score')

# xticks rotation 90 degrees

plt.xticks(rotation=20)

plt.show()

Chart, bar chart

Description automatically generatedChart, bar chart

Description automatically generated

According to R2 scores and mean squared error (MSE) values, I can see that the Random Forest Regression model performs best. I can easily see the comparison of these values ​​in the table.

The fact that the R2 score for the Linear Regression model is quite low indicates that the model is not suitable for the data. The Decision Tree Regression model performed better with an R2 score of 0.627, but I can understand that it still needs improvement. I can see that the Random Forest Regression model is the most suitable model for the data, with the highest R2 score of 0.867.

The MSE values also support this conclusion, as the Random Forest Regression model had the lowest MSE, followed by the Decision Tree Regression model and then the Linear Regression model. This means that the Random Forest Regression model had the smallest errors when predicting the number of new house registrations for each county and year.

it is always a good idea to fine-tune the model to improve its performance. There are several hyperparameters that can be tuned for a Random Forest Regression model, including:

**n\_estimators**: the number of trees in the forest

**max\_depth**: the maximum depth of the trees

**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

**max\_features**: the number of features to consider when looking for the best split

I can use GridSearchCV or RandomizedSearchCV to search for the best combination of hyperparameters.

# Import necessary libraries

from sklearn.model\_selection import GridSearchCV

# define the hyperparameter grid

param\_grid = {

'n\_estimators': [50, 100, 150],

'max\_depth': [5, 10, 20],

'min\_samples\_split': [2, 5, 10],

'min\_samples\_leaf': [1, 2, 4],

'max\_features': ['auto', 'sqrt', 'log2']

}

# perform grid search

grid\_search = GridSearchCV(estimator=forest\_reg, param\_grid=param\_grid, cv=5, n\_jobs=-1)

grid\_search.fit(X\_train, y\_train)

# print the best hyperparameters

print("Best hyperparameters:", grid\_search.best\_params\_)

Best hyperparameters: {'max\_depth': 20, 'max\_features': 'auto', 'min\_samples\_leaf': 1, 'min\_samples\_split': 5, 'n\_estimators': 50}

Now that I have fine-tuned the hyperparameters of our Random Forest model, I can retrain the model using these hyperparameters to get the best results.

# Import necessary libraries

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import r2\_score, mean\_squared\_error

from sklearn.model\_selection import train\_test\_split

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Let's retrain our Random Forest Regressor with the best hyperparameters

ft\_rf\_reg = RandomForestRegressor(n\_estimators=50, max\_depth=20, max\_features='auto', min\_samples\_leaf=1, min\_samples\_split=5)

ft\_rf\_reg.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = ft\_rf\_reg.predict(X\_test)

# Evaluate the model using mean squared error

ft\_rf\_r2 = r2\_score(y\_test, y\_pred)

ft\_rf\_mse = mean\_squared\_error(y\_test, y\_pred)

print("Random Forest Regression R2 score (fine tuned):", ft\_rf\_r2)

print("Random Forest Regression MSE (fine tuned):", ft\_rf\_mse)

Random Forest Regression R2 score (fine tuned): 0.9491328294709911

Random Forest Regression MSE (fine tuned): 118056.17967316514

Imports the required RandomForestRegressor, r2\_score, mean\_squared\_error, and train\_test\_split libraries from the sklearn.ensemble, sklearn.metrics, and sklearn.model\_selection modules, respectively.

It splits data into training and test sets using the train\_test\_split() function with a test size of 0.2 and a random state of 42. This is done to evaluate the performance of the model on data on which it was not trained.

Initializes a RandomForestRegressor() object with the best hyperparameters from tuning and assigns it to a variable named ft\_rf\_reg. This creates an instance of the random forest regression model with the specified hyperparameters.

Fits the ft\_rf\_reg object to the training data using the fit() method. This trains the model on the training data.

Makes predictions on the test set using the predict() method. This produces the predicted target values ​​for the test set using the trained model.

It evaluates the performance of the model using the r2\_score() and mean\_squared\_error() functions and assigns the corresponding scores to the variables ft\_rf\_r2 and ft\_rf\_mse. The R2 score measures the proportion of variance in the target variable explained by the model. The mean squared error measures the mean squared difference between the predicted and actual target values.

Prints the R2 score and mean square error of the fine-tuned random forest regression model to the test set. This provides insights into the accuracy of model predictions on new data.

Overall, this code block fine-tunes a RandomForestRegressor() model using the best hyperparameters and evaluates its performance on the test set using the R2 score and mean squared error. The output of this code block provides insights into the accuracy of model predictions on new data.

# Import necessary libraries

import matplotlib.pyplot as plt

# Data and titles for the plot

r2\_scores = [lin\_reg\_r2, tree\_reg\_r2, forest\_reg\_r2, ft\_rf\_r2]

mse\_scores = [lin\_reg\_mse, tree\_reg\_mse, forest\_reg\_mse, ft\_rf\_mse]

models = ['Linear Regression', 'Decision Tree Regression', 'Random Forest Regression', 'Fine-Tuned Random Forest Regression']

# Plot the results in a bar chart for mse scores

plt.bar(models, mse\_scores, color='red')

plt.title('Mean Squared Error (MSE) Comparison')

plt.ylabel('MSE')

# xticks rotation 20 degrees

plt.xticks(rotation=20)

plt.show()

# Plot the results in a bar chart for r2 scores

plt.bar(models, r2\_scores)

plt.title('R2 Score Comparison')

plt.ylabel('R2 Score')

# xticks rotation 20 degrees

plt.xticks(rotation=20)

plt.show()

Chart, bar chart, histogram

Description automatically generatedChart, bar chart

Description automatically generated

|  |  |  |
| --- | --- | --- |
| Model | R2 Score | MSE |
| Linear Regression | 0.0254 | 2,262,021 |
| Decision Tree Regression | 0.6269 | 865,807 |
| Random Forest Regression | 0.8674 | 307,829 |
| Fine-tuned Random Forest Regression | 0.9491 | 118,056 |

# get the dublin data

dublin\_encoded = labelEncoder.transform(['Dublin'])[0]

# get the year data

year = 2020

prediction\_data = pd.DataFrame({'Year': [year], "County": dublin\_encoded})

predicted\_value = ft\_rf\_reg.predict(prediction\_data)[0]

print(f"Predicted number of new house registrations in {year}: {predicted\_value}")

Predicted number of new house registrations in 2020: 4281.685619047617

In summary, I use supervised learning in this project as I have a labeled dataset and want to predict a numerical value. By training a model on this dataset, I can learn a function that maps the input features to the output values, which can be utilized to make predictions for future years.

The CRISP-DM framework is a widely recognized and established approach to data mining projects. Its structured and systematic process maximizes the chances of success by ensuring that the objectives of the project are achieved efficiently and effectively. This framework consists of six phases, which include Understanding the Business, Understanding the Data, Preparing the Data, Modeling, Evaluating and Deploying. Each stage has a specific set of tasks that must be completed before moving on to the next stage. Also, the framework is iterative, allowing us to adapt and make changes as needed throughout the project lifecycle.

For this project, I chose to use the CRISP-DM framework because of its well-defined process and structure. By following this framework, I can make sure I have a clear understanding of the problem I am trying to solve, the data I am working on, and the modeling techniques I will use to achieve our goals. This will increase our chances of success in predicting annual new home registrations for Dublin from 2020.

Additionally, the CRISP-DM framework provides a comprehensive approach to model evaluation, ensuring that I adequately evaluate models and that they are suitable for deployment. By applying this framework, I can trust the quality and accuracy of our estimates.

In summary, using the CRISP-DM framework for this project provides a clear roadmap and structure that allows me to follow a systematic process to achieve our goals. This approach can maximize our chances of success by helping us fully understand the problem, data, and modeling techniques, and allowing me to adequately evaluate our models.

**Data Analytics Programming Languages**

**Why Python for Data Processing**

Why you should choose Python for your data-processing needs is a little more involved. For a start, there are good alternatives as far as data processing is concerned. Let's deal with a few candidates for the job, starting with the enterprise behemoth Java.

**Java**

Among the other main, general-purpose programming languages, only Java offers anything like the rich ecosystem of libraries that Python does, with considerably more native speed too. But while Java is a lot easier to program in than languages Fike C++, it isn't, in my opinion, a particularly nice language to program in, having rather too much in the way of tedious boilerplate code and excessive verbiage. This sort of thing starts to weigh heavily after a while and makes for a hard slog at the code face. As for speed, Python's default interpreter is slow, but Python is a great glue language that plays nicely with other languages. This ability is demonstrated by the big Python data-processing libraries like NumPy (and its dependent, pandas), SciPy, and the like, which use Ctt and Fortran libraries to do the heavy lifting while providing the ease of use of a simple scripting language.

**R**

The venerable R has, until recently, been the tool of choice for many data scientists and is probably Python's main competitor in the space. Like Python, R benefits from a very active community, some great tools like the plotting library ggplot2, and a syntax specially crafted for data science and statistics. But this specialism is a double-edged sword. Because R was developed for a specific purpose, it means that if, for example, you wish to write a web server to serve your R-processed data, you have to skip out to another language with all the attendant learning overheads or try to hack something together in a round-hole/ square-peg sort of way. Python's general-purpose nature and its rich ecosystem mean one can do pretty much everything required of a data-processing pipeline US visuals aside) without having to leave its comfort zone. Personally, it is a small sacrifice to pay for a little syntactic clunkiness.

**Others**

There are other alternatives to doing your data processing with Python, but none of them come close to the flexibility and power afforded by a general-purpose, easy-to-use programming language with a rich ecosystem of libraries. For example, mathematical programming environments such as Matlab and Mathematica have active communities and a plethora of great libraries, but they hardly count as general purpose, because they are designed to be used within a closed garden. They are also proprietary, which means a significant initial investment and a different vibe to Python's resoundingly open-source environment. GUI-driven dataviz tools like Tableau are great creations but will quickly frustrate someone used to freedom in programming. They tend to work great as long as you are singing from their songsheet, as it were. Deviations from the designated path get painful very quickly.

**Python’s Getting Better All the Time**

As things stand, I think a very good case can be made for Python being the budding data scientist's language of choice. But things are not standing still; in fact, Python's capabilities in this area are growing at an astonishing rate. To put it in perspective, I have been programming in Python for over 20 years and have grown used to being surprised if I can't find a Python module to help solve a problem at hand, but I find myself sur prised at the growth of Python's data-proce ssing abilities, with a new, powerful library appearing weekly. To give an example, Python has traditionally been weak on statistical analysis libraries, with R being far ahead. Recently a number of powerful modules, such as statsmodels, have started to close this gap fast.

Dataviz on the web is an exciting place to be right now, with innovations in interactive visualizations coming thick and fast, and many (if not most) of them being developed with D3. JavaScript is the only browser-based language, so the cool visuals are by necessity being coded in it (or converted into it). But JavaScript lacks the tools or environment necessary for the less dramatic but just as a vital element of modern dataviz: the aggregation, curation, and processing of the data. This is where Python rules the roost, providing a general-purpose, concise, and eminently readable programming language with access to an increasing stable of first-class data-processing tools. Many of these tools leverage the power of very fast, low-level libraries, making Python data processing fast as well as easy. This book introduces some of those heavyweight tools, as well as a host of other smaller but equally vital tools. It also shows how Python and JavaScript in concert represent the best dataviz stack out there for anyone wishing to deliver their visualizations to the internet.

Python

By far, the best command-line Python interpreter is IPython, which comes in three shades: the basic terminal version, an enhanced graphical version, and a browser-based notebook. Since IPython version 4.0, the latter two have been spun out into Project Jupyter. The Jupyter Notebook is a rather brilliant and fairly recent innovation, providing a browser-based interactive computational environment. The great boon of the notebook is session persistence and the possibility of web access.4 The ease with which one can share programming sessions, complete with embedded data visualizations, makes the notebook a fantastic teaching tool as well as a great way to recover programming context. That's why the Python chapters of this book have accompanying Jupyter notebooks.

There are lots of options for trying out JavaScript code without starting a server, though the latter isn't that difficult. Because the JavaScript interpreter comes embedded in all modern web browsers, there are a number of sites that let you try out bits of JavaScript along with HTML and CSS and see the results. CodePen is a good option. These sites are great for sharing code and trying out snippets and usually allow you to add libraries such as D3.js with a few mouse clicks. If you want to try out code one-liners or quiz the state of live code, browser-based consoles are your best bet. With Chrome, you can access the console with the key combo Ctrl-Shift-J (Command + Option + J on a Mac). As well as trying little Js snippets, the console allows you to drill down into any objects in scope, revealing their methods and properties. This is a great way to quiz the state of a live object and search for bugs.

**Summary**

In this study, I worked with a data set containing newly registered house count data from all provinces of Ireland between 1978 and 2019. I showed the year-based analyzes according to the distribution of provinces by visualizing and using statistical data. I made various operations for statistical scales. For the following years, I trained a machine that could predict as accurately as possible. This machine was trained using the data of the previous years in the data set I have. He then made predictions to get reasonable results. This study will help investors and real estate agents to have an idea about the housing market.

# Bibliography

Dale, K., 2022. *Data Visualization with Python and JavaScript.* 2 ed. s.l.:O'Reilly Media, Inc..