National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Karthika Rajan
Student ID: X21122920

School of Computing
National College of Ireland

Supervisor: Muslim Jameel Syed

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Karthika Rajan
Student ID:	X21122920
Programme:	Data Analytics
Year:	2023
Module:	MSc Research Project
Supervisor:	Muslim Jameel Syed
Submission Due Date:	18/09/2021
Project Title:	Configuration Manual
Word Count:	3008
Page Count:	23

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Karthika Rajan
Date:	18th September 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

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

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

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

Configuration Manual

Karthika Rajan
X21122920

1 Introduction

The environment setup is very essential because it serves as the basis for the investigation and the analysis. The configuration manual provides a detailed description of how environment has been set up, including information on the programming language used, the system configuration, utilised libraries and the packages. The manual also provides step-by-step coverage of coding techniques used throughout the study, along with the explanations and screenshots for better comprehension.

2 Environmental Setup

Successful completion of a project often relies on the use of variety of tools and the software that automate certain processes, increase productivity and guarantee accuracy. A complete list of all tools and software used for the project is provided in this section.

2.1 Hardware specification

Specification of the machine used in this study are shown in figure 1.

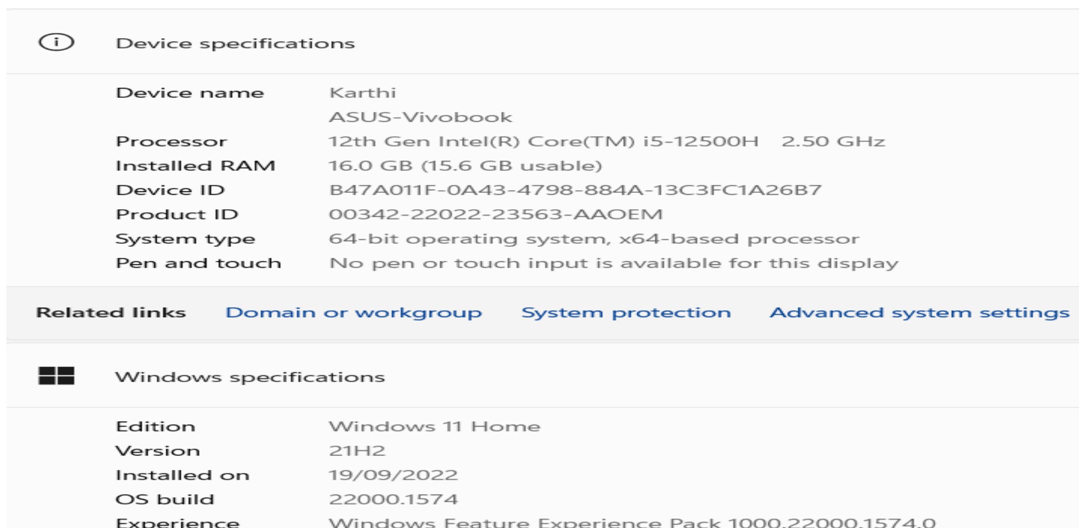

Figure 1: Hardware specification

The computer is powered by 2.50 GHz Intel(R) Core(TM) i5-12500H processor from the 12th generation. The 12th Generation Intel processors are renowned for the improved

performance and the efficiency, which makes them suitable for variety of computational tasks. The machine is equipped with a16 GB of installed RAM. The computer operates on a 64-bit operating system.

2.2 Software specification

2.2.1 Anaconda Navigator

For this research project, Anaconda [1] was installed on the computer so that Jupyter environment could be used. Anaconda is a complete platform that makes it easier to set up settings for the data science and machine learning. It comes with all tools and the packages needed to do research well. Figure 2 shows the home page of Anaconda Navigator.

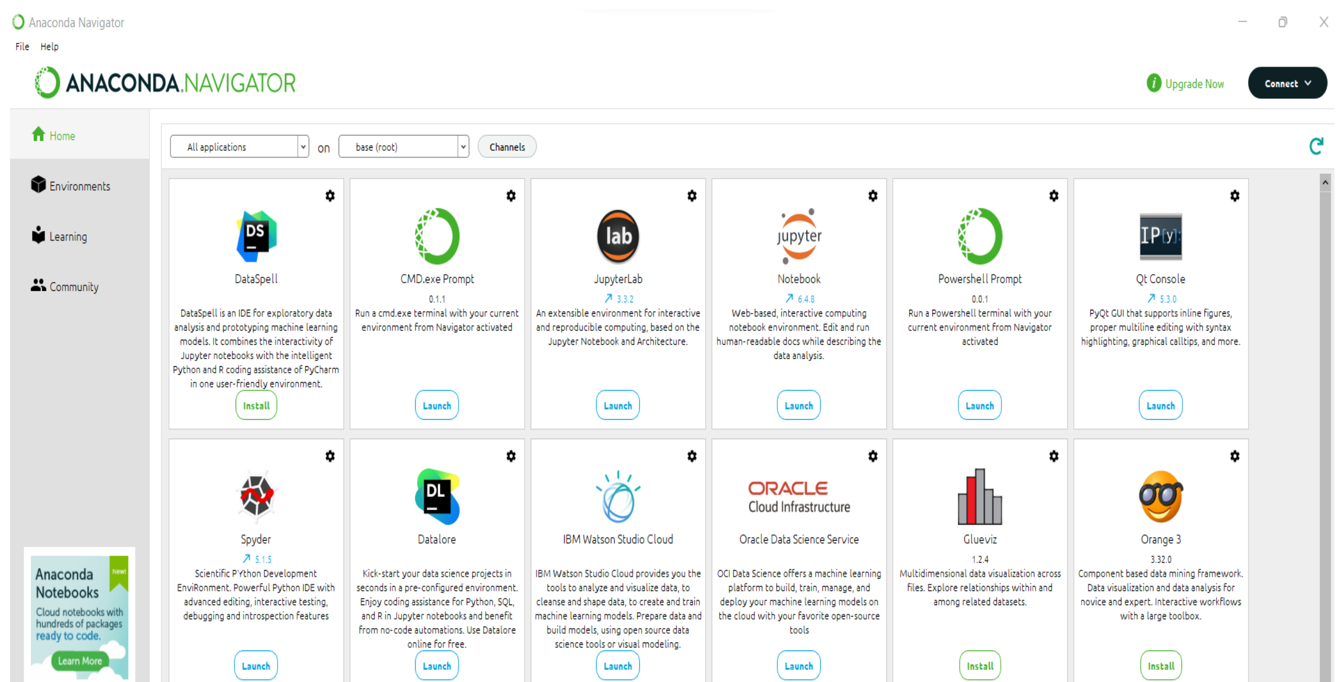

Figure 2: Software specification

2.2.2 Jupyter Notebook

As the main environment for writing and analysing the code, this study used Jupyter Notebook version 6.4.8. Figure 3 shows how to start a Jupyter notebook by choosing "Python 3" choice from "New" menu.This option is the starting point for undertaking multiple analyses in Jupyter environment. The research entails the creation of the three distinct notebooks, each of which corresponds to various objectives:

- Machine Learning Models with Default Parameters Notebook:The application of the machine learning models with their default settings is implemented in this notebook. It acts as starting point for evaluating the model behaviour and performance without any parameter adjustment.

[1] https://docs.anaconda.com/

- Machine Learning Models with Hyperparameter Optimization Notebook: In this notebook, hyperparameter optimisation is used to build the machine learning models.
- Deep Learning Model Notebook: The third notebook is designated for the deep learning models implementation.

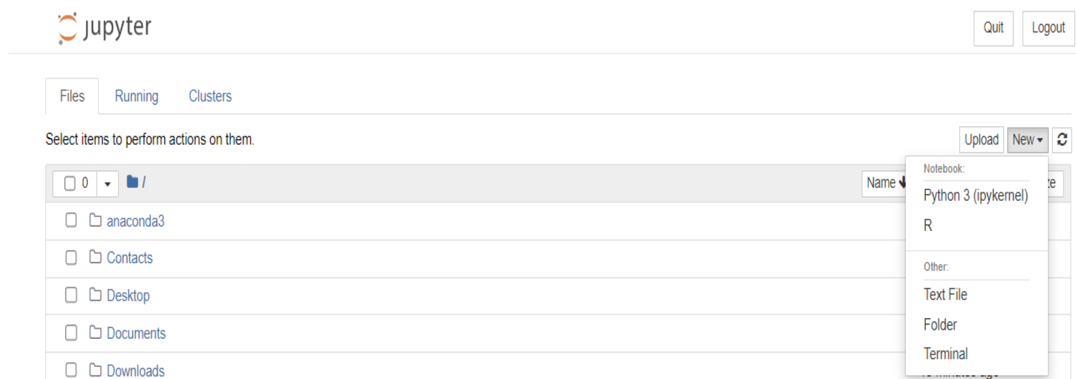

Figure 3: Jupyter Notebook

2.2.3 Python

Using Python programming language, machine learning and deep learning models were implemented. Python's strong ecosystem of libraries and tools and its user-friendliness provided a firm basis for creating and trying out with the various models.

3 Imported Packages

The figure 4 depicts the packages imported in this project.

- Pandas: This library is used to manipulate and analyse the data.
- Numpy: Necessary for numerical and array operations.
- Matplotlib: It is a popular plotting library that enables creation of visualisation.
- Tensorflow: It is an open-source deep learning framework.
- Random: This module provides random number generation and random operation capabilities.
- Xgboost: It is a scalable and effective gradient boosting library.
- RandomForestRegressor: It is a decision tree-based ensemble learning method for the regression tasks.
- DecisionTreeRegressor: It is a single decision tree-based regression model from scikit-learn library.

- KNeighborsRegressor : It is a k-nearest neighbors regression model from the scikit-learn.
- MinMaxScaler: It scales features to a particular range (usually 0 to 1).
- PolynomialFeatures, StandardScaler: StandardScaler standardises features by eliminating mean and scaling to unit variance. PolynomialFeatures generates polynomial features for polynomial regression.
- Ridge: It is a linear regression model that incorporates L2 regularisation as a means to mitigate issue of overfitting.
- Sequential: It is a neural network model in high-level neural networks API Keras.
- LabelEncoder: Converts categorical data into numerical labels.
- Train_test_split, GridSearchCV, cross_val_score: It is for dividing data into training and testing sets, tuning hyperparameters using grid search and performing the cross-validation.
- mean_squared_error, mean_absolute_error, r2_score: These metrics are used to evaluate regression model performance.
- make_pipeline: It creates a pipeline for data preprocessing and modelling .
- LSTM, Dense: LSTM is a kind of recurrent neural network layer utilised for sequence data and Dense represents a fully connected neural network layer.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import random
import xgboost as xgb
from sklearn.ensemble import RandomForestRegressor
import xgboost as xgb
from sklearn.tree import DecisionTreeRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import Ridge
from keras.models import Sequential
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.pipeline import make_pipeline
from keras.layers import LSTM, Dense
```

Figure 4: Imported Packages

4 Data Source and Description of Dataset

This study obtained the dataset used to predict Irish road accidents from the website of the Central Statistics Office (CSO)[2]. The dataset is in csv format.

Figure 5 depicts how the data is loaded and also shows a brief overview of the dataset's structure and contents. The dataset contains 9984 rows and 7 columns. The variables in tha dataset are: Statistic Label, Year, Age group, Sex, Road user type, unit and value. The head function is used to display first few entries of the dataset. By doing so, it provides a preview of actual data present in the DataFrame, providing a concise summary of its contents. The code also includes a concise summary of data types associated with the columns in the dataset.

```
data = pd.read_csv("C:\\Users\\karthi\\Downloads\\RESEARCH PROJECT\\full_data.csv")
```

```
data.shape
```

```
(9984, 7)
```

```
print(data.head())
```

```
     Statistic Label  Year Age Group   Sex  Road User Type    UNIT  VALUE
0  Killed Casualties  2005  All ages  Male  All road users  Number    286
1  Killed Casualties  2005  All ages  Male     Pedestrians  Number     51
2  Killed Casualties  2005  All ages  Male  Pedal cyclists  Number      7
3  Killed Casualties  2005  All ages  Male  Motor cyclists  Number     53
4  Killed Casualties  2005  All ages  Male   All Car users  Number    148
```

```
print(data.dtypes)
```

```
Statistic Label    object
Year                int64
Age Group          object
Sex                object
Road User Type     object
UNIT               object
VALUE               int64
dtype: object
```

Figure 5: Data loading and overview

5 Data Visualization

Data visualization is a powerful technique that converts the complex datasets into visually understandable representations, making information simpler to comprehend, interpret and communicate. Using diagrams, charts, maps, and other visual elements, data visualization permits the analysts to investigate the connections among variables, find

[2] https://data.cso.ie/table/ROA16

outliers and discover meaningful trends.

The code provided in figure 6 presents distribution of the road accidents by age group using bar chart. By grouping and displaying data, code helps develop a better comprehension of the accident patterns and potential areas of concern by providing insights into relative frequency of the crashes among various age groups.

```
#Age Group Distribution of People involved in Road Accidents

# Group the data by age group and calculate the total number of accidents for each group
age_group_counts = data.groupby('age_group')['VALUE'].sum()

# Plot the bar chart
plt.figure(figsize=(10, 6))
age_group_counts.plot(kind='bar', color='maroon')
plt.title('Age Group Distribution of People involved in Road Accidents')
plt.xlabel('Age Group')
plt.ylabel('Number of Accidents')
plt.xticks(rotation=45)
plt.show()
```

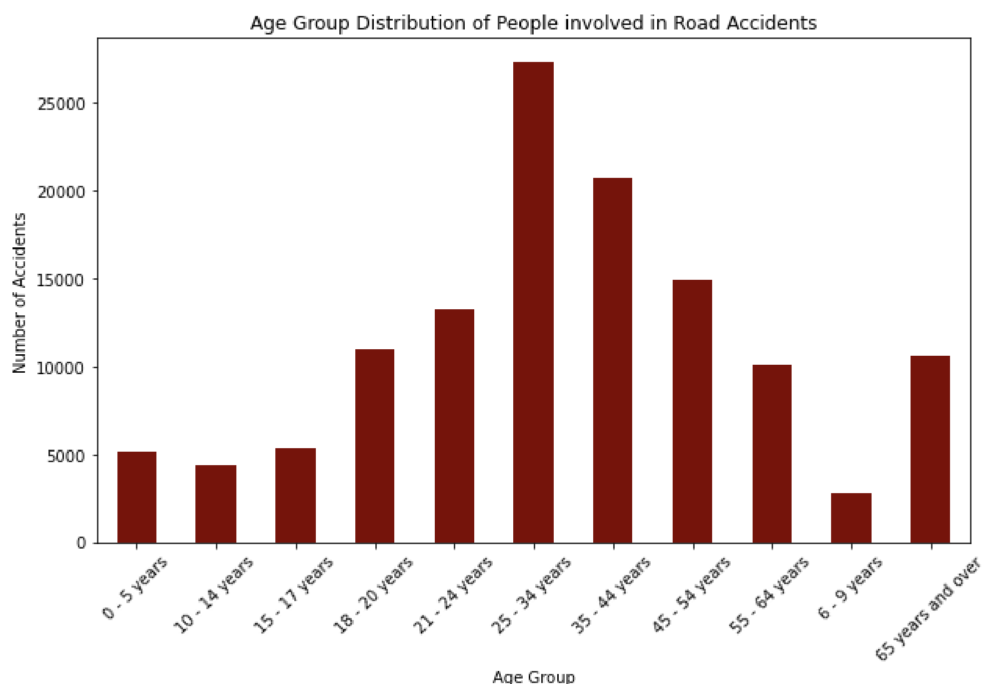

Figure 6: Age group distribution of people involved in road accidents

Using a pie chart, the code given in figure 7 illustrates gender distribution of individuals involved in the traffic accidents. The chart provides the quick visual representation of proportion of accidents linked to each gender, making it simple to compare relative impact of accidents on each gender.

Using a stacked bar chart, the code as shown in figure 8 effectively demonstrates pattern of road accidents by road user type and the gender. By classifying crashes according to these two factors and presenting them as stacked bars, the chart provides a clear and insightful means of comparing accident rates of various road user types and genders.

```
#Gender Distribution of  People involved in Road Accidents
gender_counts = data.groupby('Sex')['VALUE'].sum()
plt.figure(figsize=(8, 8))
plt.pie(gender_counts, labels=gender_counts.index, autopct='%1.1f%%', startangle=90)
plt.title('Gender Distribution of People involved in Road Accidents')
plt.axis('equal')
plt.show()
```

Gender Distribution of People involved in Road Accidents

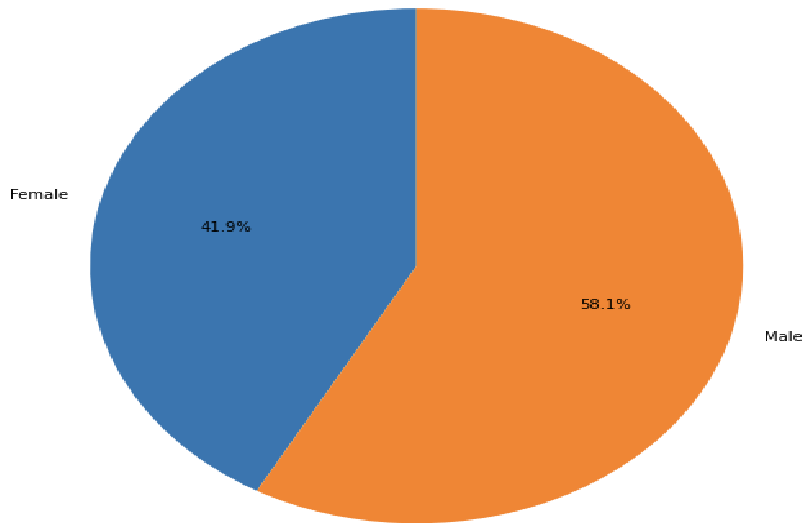

Figure 7: Gender distribution of people involved in road accidents

```
#Distribution of Road Accidents by Road User Type and Gender
# Group the filtered data by road user type and gender and calculate the count of accidents for each combination
grouped_data = data.groupby(['road_user_type', 'Sex'])['VALUE'].sum().unstack()

# Plot the stacked bar chart
plt.figure(figsize=(10, 6))
grouped_data.plot(kind='bar', color=['pink', 'lightblue', 'blue', 'yellow'])

# Set the x-axis label
plt.xlabel('Road User Type')

# Set the y-axis label
plt.ylabel('Number of Accidents')

# Set the chart title
plt.title('Distribution of Road Accidents by Road User Type and Gender')

# Add a legend
plt.legend(title='Gender')

# Show the plot
plt.tight_layout()
plt.show()
```

```
<Figure size 720x432 with 0 Axes>
```

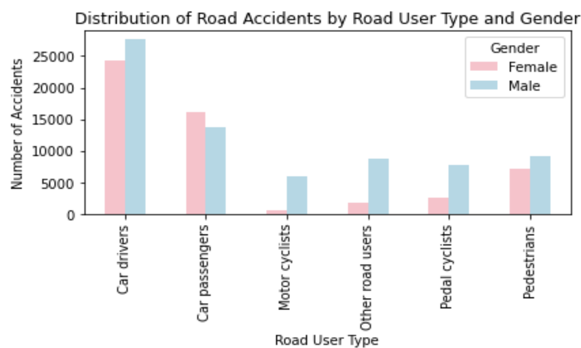

Figure 8: Distribution of Road Accidents by Road User Type and Gender

6 Data Prepartion

Data preparation is a fundamental and essential step in data analysis, involving data transformation, cleaning and arrangement of unprocessed data into suitable format for analysis, modelling, and interpretation. Raw data is frequently available in variety of formats and may contain inconsistencies, absent values or noise . The objective of data preparation is to address these challenges by refining data, guaranteeing its quality and arranging it in a coherent structure that facilitates informed decision-making.

6.1 Check for null values

```
print(data.isnull().sum())
```

```
statistic_label    0
Year               0
age_group          0
Sex                0
road_user_type     0
UNIT               0
VALUE              0
dtype: int64
```

Figure 9: Check for null values

Figure 9 depicts the code used to display the number of absent values in each column of the dataset. This code computes the number of missing values in each column and outputs the results, revealing the completeness of the dataset. The "data.isnull().sum()" uses pandas DataFrame method "isnull()" to construct boolean DataFrame where every record is either True (if the corresponding value is missing) or false (if the value is not missing). The ".sum()" function then calculates the sum of each column's tYes values, effectively tallying the missing values in each column. It is found that the dataset does not contain any null values.

6.2 Check for duplicate rows

```
duplicate_rows = data[data.duplicated()]
duplicate_rows
```

Statistic Label	Year	Age Group	Sex	Road User Type	UNIT	VALUE

Figure 10: Check for duplicate rows

Figure 10 depicts the code used to identify and remove duplicate entries from the dataframe. This code filters dataframe to include only the duplicate entries, which have

identical values across all the columns. There are no duplicate rows in the dataset.

6.3 Data Reduction

Summary or aggregate data frequently represents the consolidated information derived from the individual data points. This may consist of totals, averages or other aggregated metrics. Such data may not be appropriate for the comprehensive analysis, as it lacks individual-level information necessary to identify patterns, relationships or trends.

```
# Remove rows where road user type is all car users and all road users
data = data[~((data['road_user_type'] == 'All Car users') | (data['road_user_type'] == 'All road users'))]

# Remove rows where statistic is all injured
data = data[data['statistic_label'] != 'All Killed and Injured Casualties']

# Remove rows where age group is all ages
data = data[data['age_group'] != 'All ages']

# Print the updated DataFrame
print(data)
```

```
         statistic_label  Year    age_group     Sex    road_user_type  \
17      Killed Casualties  2005  0 - 5 years    Male       Pedestrians
18      Killed Casualties  2005  0 - 5 years    Male    Pedal cyclists
19      Killed Casualties  2005  0 - 5 years    Male    Motor cyclists
21      Killed Casualties  2005  0 - 5 years    Male       Car drivers
22      Killed Casualties  2005  0 - 5 years    Male    Car passengers
...                   ...   ...          ...     ...               ...
6650  Injured Cassualties  2020  Age unknown  Female    Pedal cyclists
6651  Injured Cassualties  2020  Age unknown  Female    Motor cyclists
6653  Injured Cassualties  2020  Age unknown  Female       Car drivers
6654  Injured Cassualties  2020  Age unknown  Female    Car passengers
6655  Injured Cassualties  2020  Age unknown  Female  Other road users

        UNIT  VALUE
17    Number      3
18    Number      0
19    Number      0
21    Number      0
22    Number      0
...      ...    ...
6650  Number      0
6651  Number      0
6653  Number      0
6654  Number      0
6655  Number      0

[4608 rows x 7 columns]
```

Figure 11: Code for data reduction

In the Irish road accident dataset, statistical label column contains three values: "All Killed and Injured Casualties," "Killed Casualties," and "Injured Casualties." "All Killed and Injured Casualties" represents overall number of the killed and the injured casualties. This aggregate figure is omitted from the dataset because it does not specifically describe any fatalities or injuries. Similarly, the value "All ages" from the column age group is removed, as it represents aggregated data for all age categories. The Road User type column contains another set of aggregated data. The value "All road users" is a collection of data containing counts of various road user categories, such as pedestrians, cyclists, motorcyclists, all vehicle users, and other road users. Therefore, "all road users" is also eliminated. "All Car Users" equals the sum of "Car drivers" and "Car Passengers" .As a result, this value is also excluded from the dataset. The code for this is given in figure 11.

6.4 Data Transformation

```
# Find the mode value of the "age" column
mode_value = data["age_group"].mode()[0]

# Replace "Age unknown" with the mode value
data.loc[data["age_group"] == "Age unknown", "age_group"] = mode_value
```

Figure 12: Code for data transformation

The code given in the figure 12 is intended to deal with the unknown age group values ("Age unknown") in the dataset by imputing them with median value of existing age groups. This imputation technique ensures that "age_group" column contains legitimate values for the further modelling and analysis.

7 Model Building and Evaluation

7.1 Machine Learning Models

There are 5 machine learning models implemented in this study and those are: Random Forest, Decision Tree, XGBoost, KNN and Ridge regression.

7.1.1 Random Forest with and without hyperparameter optimization

```
# Define the independent variables (features) and dependent variable (target)
features = ['statistic_label', 'Year', 'age_group', 'Sex', 'road_user_type']
target = 'VALUE'

# Prepare the data for training
X = data[features]
y = data[target]

# Convert categorical variables to numerical using one-hot encoding
X = pd.get_dummies(X)

# 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)

# Create a Random Forest regressor object
rf_regressor = RandomForestRegressor(n_estimators=100, max_depth=None, min_samples_split=2, random_state=42)

# Train the model
rf_regressor.fit(X_train, y_train)

# Make predictions on the test set
y_pred = rf_regressor.predict(X_test)


## Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)
print("R-squared Score:", r2)
```

```
Mean Squared Error (MSE): 141.35649777083486
Mean Absolute Error (MAE): 5.231602107220329
Root Mean Squared Error (RMSE): 11.889343874698673
R-squared Score: 0.9490940679968694
```

Figure 13: Code for Random Forest without hyperparameter optimization

Figure 13 shows the code for random forest model without hyperparameter optimization. First, the code defines features (independent variables) and target variable (dependent variable). Next the dataset is divided into training and testing sets. The code initializes a Random Forest regressor, trains it with training data and makes predictions on the test set. To assess the efficacy of a model, evaluation metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R-squared score are calculated.

```
#Define the independent variables (features) and dependent variable (target)
features = ['statistic_label','Year', 'age_group', 'Sex', 'road_user_type']
target = 'VALUE'

# Prepare the data for training
X = data[features]
y = data[target]

# Convert categorical variables to numerical using one-hot encoding
X = pd.get_dummies(X)

# 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)

# Create a Random Forest regressor object
rf_regressor = RandomForestRegressor(random_state=42)

# Define the hyperparameter grid for grid search
param_grid = {
    'n_estimators': [100, 200, 300],  # Number of trees in the forest
    'max_depth': [None, 5, 10],       # Maximum depth of the tree
    'min_samples_split': [2, 5, 10]   # Minimum number of samples required to split an internal node
}

# Perform grid search to find the best hyperparameters
grid_search = GridSearchCV(estimator=rf_regressor, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Get the best model from grid search
best_rf_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_pred = best_rf_model.predict(X_test)

## Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)
print("R-squared Score:", r2)
```

```
Mean Squared Error (MSE): 136.82676564861882
Mean Absolute Error (MAE): 5.188205843024172
Root Mean Squared Error (RMSE): 11.697297365144601
R-squared Score: 0.9507253353177374
```

Figure 14: Code for Random Forest with hyperparameter optimization

Figure 14 demonstrates the Random Forest algorithm's hyperparameter tuning and model evaluation processes. The hyperparameter grid for grid search is defined, which includes number of trees in the forest (n_estimators), the maximum depth of trees (max_depth) and minimal number of the samples necessary to split internal nodes (min_samples_split). The grid search is conducted using the 5 folds of cross-validation (CV) to identify optimal hyperparameter combination that delivers best model performance. GridSearchCV

evaluates multiple hyperparameter combinations and chooses one with the highest cross-validated score. The optimal model is then extracted from the grid search results. This model is employed to forecast the target variable on the test set. Same evaluation metrix are used to evaluate performance of the model.

7.1.2 XGBoost with and without hyperparameter optimization

```
features = ['statistic_label','Year', 'age_group', 'Sex', 'road_user_type']
target = 'VALUE'

# Prepare the data for training
X = data[features]
y = data[target]
# Convert categorical variables to numerical using one-hot encoding
X = pd.get_dummies(X)




# 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)

# Train an XGBoost model
model = xgb.XGBRegressor()
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

## Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)
print("R-squared Score:", r2)
```

```
Mean Squared Error (MSE): 101.79175604411736
Mean Absolute Error (MAE): 5.353884187149472
Root Mean Squared Error (RMSE): 10.089190058875754
R-squared Score: 0.9633422991275451
```

Figure 15: Code for XGBoost without hyperparameter optimization

Figure 15 shows the code for implementing XGBoost algorithm with default parameters. Using xgb.XGBRegressor class, XGBoost regression model is created. Using fit technique, the algorithm is then trained on training data (X_train and y_train). The trained model is then applied to test set (X_test) to predict the target values. The y_pred variable stores these predictions.

Figure 16 shows the code for XGBoost model with hyperparameter optimization.

```
# Define the features and target variable
features = ['statistic_label', 'Year', 'age_group', 'Sex', 'road_user_type']
target = 'VALUE'

# Prepare the data for training
X = data[features]
y = data[target]

# Convert categorical variables to numerical using one-hot encoding
X = pd.get_dummies(X)

# 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)

# Define the hyperparameter grid for grid search
param_grid = {
    'learning_rate': [0.1, 0.01, 0.001],
    'max_depth': [1, 3, 5],
    'n_estimators': [100, 300, 500]
}

# Create an XGBoost regressor
model = xgb.XGBRegressor()

# Perform grid search to find the best hyperparameters
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Get the best model from grid search
best_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_pred = best_model.predict(X_test)

# Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)
print("R-squared Score:", r2)
```

```
Mean Squared Error (MSE): 99.96085941986101
Mean Absolute Error (MAE): 5.797719987333043
Root Mean Squared Error (RMSE): 9.998042779457437
R-squared Score: 0.9640016497801784
```

Figure 16: Code for XGBoost with hyperparameter optimization

A hyperparameter grid is defined, detailing various hyperparameter combinations including learning rates, the maximal tree depths and number of estimators. These parameters affect the efficacy and the behaviour of XGBoost model. A regressor for XGBoost is created. Next, the code uses grid search technique (GridSearchCV) to explore hyperparameter grid systematically. Grid search identifies, using a 5-fold cross-validation strategy (cv=5), the hyperparameter combination that produces highest performance on the training data.The best_model stores the best model derived from grid search. This model contains the optimal hyperparameters for the provided data. Using this optimized

model, predictions are made on the unaltered test set (X_test). This method illustrates how hyperparameter tuning can enhance the performance of the model by identifying the parameter configuration that produces highest predictive accuracy.

7.1.3 Decision Tree with and without hyperparameter optimization

```
features = ['statistic_label','Year', 'age_group', 'Sex', 'road_user_type']
target = 'VALUE'


# Prepare the data for training
X = data[features]
y = data[target]

# Convert categorical variables to numerical using one-hot encoding
X = pd.get_dummies(X)

# 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)

# Create a Decision Tree regressor object
decision_tree = DecisionTreeRegressor(random_state=42)

# Train the model
decision_tree.fit(X_train, y_train)

# Make predictions on the test set
y_pred = decision_tree.predict(X_test)

## Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)
print("R-squared Score:", r2)
```

```
Mean Squared Error (MSE): 234.2125813449024
Mean Absolute Error (MAE): 6.344902386117137
Root Mean Squared Error (RMSE): 15.304005402014937
R-squared Score: 0.9156543213206202
```

Figure 17: Code for Decision Tree without hyperparameter optimization

Figure 17 shows the code for decision tree without hyperparameter optimization. Decision Tree regressor instance with a fixed random state is created for consistency. Afterward, model is trained with training data. Using the trained model, predictions are made on test data, resulting in the y_pred values.Using functions from sklearn.metrics library, multiple evaluation metrics such as MSE, RMSE, MAE and R-Squared score are calculated.

Figure 18 shows the code for decision tree with hyperparameter optimization.

```
# Define the features and target variable
features = ['statistic_label', 'Year', 'age_group', 'Sex', 'road_user_type']
target = 'VALUE'

# Prepare the data for training
X = data[features]
y = data[target]

# Convert categorical variables to numerical using one-hot encoding
X = pd.get_dummies(X)

# 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)

# Create a Decision Tree regressor object
dt_regressor = DecisionTreeRegressor()

# Define the hyperparameter grid for grid search
param_grid = {
    'max_depth': [None, 5, 10],        # Maximum depth of the tree
    'min_samples_split': [2, 5, 10]   # Minimum number of samples required to split an internal node
}

# Perform grid search to find the best hyperparameters
grid_search = GridSearchCV(estimator=dt_regressor, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Get the best model from grid search
best_dt_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_pred = best_dt_model.predict(X_test)

# Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)
print("R-squared Score:", r2)
```

```
Mean Squared Error (MSE): 206.46592552422268
Mean Absolute Error (MAE): 5.852675343456255
Root Mean Squared Error (RMSE): 14.368922211642134
R-squared Score: 0.9256465706816058
```

Figure 18: Code for Decision Tree with hyperparameter optimization

An instance of Decision Tree regressor is created. Afterwards a hyperparameter grid containing values for the maximum tree depth and minimum required samples for node separation is defined. The code uses a grid search technique (GridSearchCV) to systematically explore hyperparameter grid using a 5-fold cross-validation strategy (cv=5), ultimately identifying hyperparameter combination that produces the greatest performance on the training data.The best_dt_model is used to store the optimized model acquired through grid search. Using this best model, predictions are made on the test data, yielding y_pred values.

7.1.4 KNN with and without hyperparameter optimization

```
# Define the features and target variable
features = ['statistic_label', 'Year', 'age_group', 'Sex', 'road_user_type']
target = 'VALUE'

# Prepare the data for training
X = data[features]
y = data[target]

# Convert categorical variables to numerical using one-hot encoding
X = pd.get_dummies(X)

# 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)

# Create a KNN regression model
knn_model = KNeighborsRegressor(n_neighbors=5)  # Specify the number of neighbors

# Train the model
knn_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = knn_model.predict(X_test)
## Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)
print("R-squared Score:", r2)
```

```
Mean Squared Error (MSE): 844.1308459869847
Mean Absolute Error (MAE): 13.450325379609545
Root Mean Squared Error (RMSE): 29.053929957700813
R-squared Score: 0.6960078374520638
```

Figure 19: Code for KNN without hyperparameter optimization

Figure 19 shows the code for KNN model without hyperparameter optimization. A KNN regression model is built and set up with a predetermined number of neighbors - in this case, five to take into account while making predictions. Using training set of data (X_train and y_train) the model is trained and predictions are made for test set of data (X_test) using trained KNN model.

Figure 20 shows the code for KNN model with hyperparameter optimization. A KNN regressor instance is created and a hyperparameter grid with values for number of

neighbours to take into account (n_neighbors) and prediction weight function (weights) is defined. Grid search (GridSearchCV) and 5-fold cross-validation (cv=5) are used by the algorithm to systematically explore hyperparameter grid and identify the setting that yields best performance on the training data. The grid search's top model is saved under the name best_knn_model. This improved model is then used to make predictions on test data, producing y_pred values.

```
# Define the features and target variable
features = ['statistic_label', 'Year', 'age_group', 'Sex', 'road_user_type']
target = 'VALUE'

# Prepare the data for training
X = data[features]
y = data[target]

# Convert categorical variables to numerical using one-hot encoding
X = pd.get_dummies(X)

# 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)

# Create a KNN regressor object
knn_regressor = KNeighborsRegressor()

# Define the hyperparameter grid for grid search
param_grid = {
    'n_neighbors': [3, 5, 7],      # Number of neighbors to consider
    'weights': ['uniform', 'distance']   # Weight function used in prediction
}

# Perform grid search to find the best hyperparameters
grid_search = GridSearchCV(estimator=knn_regressor, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Get the best model from grid search
best_knn_model = grid_search.best_estimator_

# Make predictions on the test set using the best model
y_pred = best_knn_model.predict(X_test)

# Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)
print("R-squared Score:", r2)
```

```
Mean Squared Error (MSE): 627.2428099752194
Mean Absolute Error (MAE): 10.987255887469997
Root Mean Squared Error (RMSE): 25.044816029973536
R-squared Score: 0.7741145236505774
```

Figure 20: Code for KNN with hyperparameter optimization

7.1.5 Ridge Regression with and without hyperparameter optimization

Figure 21 shows the code for Ridge regression model without hyperparameter optimization.

```
# Define the features and target variable
features = ['statistic_label', 'Year', 'age_group', 'Sex', 'road_user_type']
target = 'VALUE'

# Prepare the data for training
X = data[features]
y = data[target]

# Convert categorical variables to numerical using one-hot encoding
X = pd.get_dummies(X)

# 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)

# Apply feature scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Apply polynomial features
poly = PolynomialFeatures(degree=2)
X_train_poly = poly.fit_transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)

# Apply Ridge Regression with cross-validation
ridge = Ridge(alpha=0.5)  # Adjust alpha as needed
ridge_scores = cross_val_score(ridge, X_train_poly, y_train, scoring='neg_mean_squared_error', cv=5)
ridge_rmse_scores = (-ridge_scores)**0.5
ridge_rmse_mean = ridge_rmse_scores.mean()

# Train the model
ridge.fit(X_train_poly, y_train)

# Make predictions on the test set
y_pred = ridge.predict(X_test_poly)

# Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics

print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)
print("R-squared Score:", r2)
```

```
Mean Squared Error (MSE): 646.3923235621941
Mean Absolute Error (MAE): 18.03978248620588
Root Mean Squared Error (RMSE): 25.42424676489343
R-squared Score: 0.7672183154682558
```

Figure 21: Code for Ridge Regression without hyperparameter optimization

The data is normalized via the feature scaling, which improves the model stability and the convergence. To transfer data, a StandardScaler object is used. To capture any potential nonlinear relationships in data, degree 2 polynomial features are then inserted. The PolynomialFeatures class is used to implement this transformation. The association between the characteristics and the target variable is modelled using Ridge

Regression and regularisation parameter (alpha) of the model is fine-tuned using cross-validation.

```
# Define the features and target variable
features = ['statistic_label', 'Year', 'age_group', 'Sex', 'road_user_type']
target = 'VALUE'

# Prepare the data for training
X = data[features]
y = data[target]

# Convert categorical variables to numerical using one-hot encoding
X = pd.get_dummies(X)

# 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)

# Define the pipeline with PolynomialFeatures and Ridge Regression
pipeline = make_pipeline(
    PolynomialFeatures(include_bias=False),
    StandardScaler(),
    Ridge()
)

# Define the hyperparameter search space
param_grid = {
    'polynomialfeatures__degree': [2, 3, 4],
    'ridge__alpha': [0.1, 0.5, 1.0]
}

# Perform grid search
grid_search = GridSearchCV(pipeline, param_grid, scoring='neg_mean_squared_error', cv=5)
grid_search.fit(X_train, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_

# Train the model with the best hyperparameters
model = grid_search.best_estimator_
model.fit(X_train, y_train)

# Make predictions using the model with the best hyperparameters
y_pred = model.predict(X_test)

# Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics with the best hyperparameters
print("Best Hyperparameters:", best_params)
print("Best Mean Squared Error (MSE):", mse)
print("Best Mean Absolute Error (MAE):", mae)
print("Best Root Mean Squared Error (RMSE):", rmse)
print("Best R-squared Score:", r2)
```

```
Best Hyperparameters: {'polynomialfeatures__degree': 4, 'ridge__alpha': 0.1}
Best Mean Squared Error (MSE): 152.91746897540602
Best Mean Absolute Error (MAE): 6.365535034581359
Best Root Mean Squared Error (RMSE): 12.365988307901433
Best R-squared Score: 0.9449306795194315
```

Figure 22: Code for Ridge Regression with hyperparameter optimization

Figure 22 shows the code for Ridge regression model with hyperparameter optimization. With multiple degrees for polynomial features and varied alpha values for Ridge Regression, a search space for hyperparameters is established. After that, grid search (GridSearchCV) is used to choose optimal configuration based on negative mean squared error by methodically examining the hyperparameter combinations and using 5-fold cross-validation.

7.2 Deep Learning Models

There are 2 deep learning models implemented in this study and those are: LSTM and FNN

7.2.1 Long Short-term Memory(LSTM)

Figure 23 and 24 shows the LSTM model.

```
# Set the seed values
seed_value = 42
np.random.seed(seed_value)
random.seed(seed_value)
tf.random.set_seed(seed_value)

# Define the features and target variable
features = ['statistic_label', 'Year', 'age_group', 'Sex', 'road_user_type']
target = 'VALUE'

# Prepare the data for training
X = data[features]
y = data[target]

# Convert categorical variables to numerical using one-hot encoding
X = pd.get_dummies(X)

# 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 input features to a range between 0 and 1
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Reshape the data to be 3D for input to LSTM (samples, timesteps, features)
X_train_reshaped = np.reshape(X_train_scaled, (X_train_scaled.shape[0], 1, X_train_scaled.shape[1]))
X_test_reshaped = np.reshape(X_test_scaled, (X_test_scaled.shape[0], 1, X_test_scaled.shape[1]))

# Build the LSTM model
model = Sequential()
model.add(LSTM(64, input_shape=(1, X_train_scaled.shape[1])))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')

# Train the LSTM model
model.fit(X_train_reshaped, y_train, epochs=50, batch_size=32, verbose=1)

# Make predictions on the test set
y_pred = model.predict(X_test_reshaped)

# Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)
print("R-squared Score:", r2)
```

```
Epoch 1/50
116/116 [==============================] - 2s 2ms/step - loss: 4340.3472
Epoch 2/50
116/116 [==============================] - 0s 2ms/step - loss: 4009.0288
Epoch 3/50
116/116 [==============================] - 0s 2ms/step - loss: 3648.1172
Epoch 4/50
116/116 [==============================] - 0s 2ms/step - loss: 3381.4065
Epoch 5/50
116/116 [==============================] - 0s 2ms/step - loss: 3176.7563
Epoch 6/50
116/116 [==============================] - 0s 2ms/step - loss: 3019.5171
Epoch 7/50
116/116 [==============================] - 0s 3ms/step - loss: 2880.0676
Epoch 8/50
116/116 [==============================] - 0s 3ms/step - loss: 2745.2783
```

Figure 23: Code for LSTM model

In order to guarantee consistency across runs, code starts by setting seed values. The data is transformed into a series of time steps and moulded into a suitable (3D) format

```
116/116 [==============================] - 0s 4ms/step - loss: 2002.5156
Epoch 18/50
116/116 [==============================] - 0s 4ms/step - loss: 1949.2672
Epoch 19/50
116/116 [==============================] - 0s 4ms/step - loss: 1897.5194
Epoch 20/50
116/116 [==============================] - 0s 4ms/step - loss: 1846.4785
Epoch 21/50
116/116 [==============================] - 0s 4ms/step - loss: 1797.1854
Epoch 22/50
116/116 [==============================] - 0s 4ms/step - loss: 1749.0046
Epoch 23/50
116/116 [==============================] - 0s 4ms/step - loss: 1704.4142
Epoch 24/50
116/116 [==============================] - 0s 4ms/step - loss: 1660.6367
Epoch 25/50
116/116 [==============================] - 0s 4ms/step - loss: 1619.2526
Epoch 26/50
116/116 [==============================] - 0s 4ms/step - loss: 1577.2148
Epoch 27/50
116/116 [==============================] - 0s 4ms/step - loss: 1536.0172
Epoch 28/50
116/116 [==============================] - 0s 4ms/step - loss: 1496.7986
Epoch 29/50
116/116 [==============================] - 0s 4ms/step - loss: 1459.9584
Epoch 30/50
116/116 [==============================] - 0s 4ms/step - loss: 1424.8490
Epoch 31/50
116/116 [==============================] - 0s 4ms/step - loss: 1389.9464
Epoch 32/50
116/116 [==============================] - 0s 4ms/step - loss: 1355.2529
Epoch 33/50
116/116 [==============================] - 0s 4ms/step - loss: 1318.8995
Epoch 34/50
116/116 [==============================] - 0s 4ms/step - loss: 1282.3666
Epoch 35/50
116/116 [==============================] - 0s 4ms/step - loss: 1246.9668
Epoch 36/50
116/116 [==============================] - 0s 4ms/step - loss: 1211.9849
Epoch 37/50
116/116 [==============================] - 0s 4ms/step - loss: 1177.7844
Epoch 38/50
116/116 [==============================] - 0s 4ms/step - loss: 1145.5763
Epoch 39/50
116/116 [==============================] - 0s 4ms/step - loss: 1114.8767
Epoch 40/50
116/116 [==============================] - 0s 4ms/step - loss: 1084.2734
Epoch 41/50
116/116 [==============================] - 0s 4ms/step - loss: 1056.2274
Epoch 42/50
116/116 [==============================] - 0s 4ms/step - loss: 1028.1912
Epoch 43/50
116/116 [==============================] - 0s 4ms/step - loss: 1001.1658
Epoch 44/50
116/116 [==============================] - 0s 4ms/step - loss: 976.6319
Epoch 45/50
116/116 [==============================] - 0s 4ms/step - loss: 951.9124
Epoch 46/50
116/116 [==============================] - 0s 4ms/step - loss: 928.5131
Epoch 47/50
116/116 [==============================] - 0s 4ms/step - loss: 905.0485
Epoch 48/50
116/116 [==============================] - 0s 4ms/step - loss: 884.9780
Epoch 49/50
116/116 [==============================] - 0s 4ms/step - loss: 863.4937
Epoch 50/50
116/116 [==============================] - 0s 4ms/step - loss: 844.6230
29/29 [==============================] - 1s 2ms/step
Mean Squared Error (MSE): 542.8327442699231
Mean Absolute Error (MAE): 9.38700058157687
Root Mean Squared Error (RMSE): 23.29877130386757
R-squared Score: 0.804512652728023
```

Figure 24: LSTM model

for implementing LSTM model. Using the Sequential API of Keras, an LSTM model is created that consists of an LSTM layer and dense output layer. A mean squared error loss function and the Adam optimizer are used in the model's construction. The LSTM model is trained using the rearranged training data, with batch size and the epoch count being specified. Using the trained LSTM model, predictions are made on the test data after training.

7.2.2 Feedforward Neural Network(FNN)

```
# Set the seed values
seed_value = 42
np.random.seed(seed_value)
random.seed(seed_value)
tf.random.set_seed(seed_value)

# Define the features and target variable
features = ['statistic_label', 'Year', 'age_group', 'Sex', 'road_user_type']
target = 'VALUE'

# Prepare the data for training
X = data[features]
y = data[target]

# Convert categorical variables to numerical using one-hot encoding
X = pd.get_dummies(X)

# 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 input features to a range between 0 and 1
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Build the FNN model
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],)))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))

# Compile the model
model.compile(loss='mean_squared_error', optimizer='adam')

# Train the FNN model
model.fit(X_train_scaled, y_train, epochs=50, batch_size=32, verbose=1)

# Make predictions on the test set
y_pred = model.predict(X_test_scaled)

# Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
print("Root Mean Squared Error (RMSE):", rmse)
print("R-squared Score:", r2)
```

```
Epoch 44/50
116/116 [==============================] - 0s 3ms/step - loss: 272.4097
Epoch 45/50
116/116 [==============================] - 0s 2ms/step - loss: 271.8281
Epoch 46/50
116/116 [==============================] - 0s 3ms/step - loss: 268.0157
Epoch 47/50
116/116 [==============================] - 0s 3ms/step - loss: 264.6234
Epoch 48/50
116/116 [==============================] - 0s 2ms/step - loss: 273.8737
Epoch 49/50
116/116 [==============================] - 0s 2ms/step - loss: 261.6113
Epoch 50/50
116/116 [==============================] - 0s 2ms/step - loss: 259.3347
29/29 [==============================] - 0s 2ms/step
Mean Squared Error (MSE): 201.37441846009952
Mean Absolute Error (MAE): 7.321110939901977
Root Mean Squared Error (RMSE): 14.19064545607773
R-squared Score: 0.9274801468983843
```

Figure 25: Code for FNN model

Figure 25 shows the code for FNN model. The input characteristics are scaled using

MinMaxScaler to normalise them between 0 and 1. Sequential API from Keras is used to build the FNN model. It is made up with the layers that are closely related and have activation features like "relu." The mean squared error loss function and the Adam optimizer are used in the model's construction. Fitting FNN model to the scaled training data includes determining the batch size and number of epochs. Predictions are made for the test data using the trained model after training.

8 Conclusion

The hardware and software requirements have been thoroughly documented in this configuration manual, ensuring a smooth setup for performing the models. The manual's detailed guide to data preparation ensures that data are optimally structured for training the model. The manual then delves into the construction of predictive models, highlighting various evaluation metrics and algorithms. This manual ultimately empowers users with knowledge and tools necessary to transition from raw data to the accurate predictions.