National College of Ireland

Configuration Manual

MSc Research Project
MSc in Data Analytics

Sinchana Jyothilinga
Student ID: x21128952

School of Computing
National College of Ireland

Supervisor: Dr. Cristina Hava Muntean

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Sinchana Jyothilinga
Student ID:	x21128952
Programme:	MSc in Data Analytics
Year:	2023
Module:	MSc Research Project
Supervisor:	Dr. Cristina Hava Muntean
Submission Due Date:	01/02/2023
Project Title:	Configuration Manual
Word Count:	973
Page Count:	11

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:	Sinchana Jyothilinga
Date:	1st February 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

Sinchana Jyothilinga
x21128952

1 Introduction

This configuration manual provides extensive information regarding the system configuration, the software and hardware specifications, as well as the steps that were taken to carry out the Research project, Analysis and Prediction of Terrorist Attacks Using Supervised Machine Learning and Deep Learning Techniques.

The information on software and hardware specs may be found in Section 2 of this handbook. The section 3 includes environment configuration, data gathering and preparation, library importation, and the mounting of Google Drive. The different stages of data preparation are broken down into their constituent parts and discussed in Section 4. Section 5 explains the design and execution of the models.

2 System Configuration

This section provides the details of hardware and software configurations utilized for the implementation of this project.

2.1 Hardware Requirements

Table 1: Hardware Configuration

Operating System	Windows 10
RAM	27.3 GB (Google Colab Pro)
Disk Space	210 GB (Google Colab Pro)
Runtime Model Name	Intel(R) Xeon(R) CPU @ 2.20GHz

2.2 Software Requirements

Table 2: Software Configuration

Programming Language	Python 3.8.16
IDE	Google Colab Pro
Database Management	Google Drive
Web Browser	Google Chrome
Email Account	Gmail account for Google Drive and Colab
Other Softwares	Microsoft Office and Overleaf

3 Environment Setup

This section describes the steps for setting up the environment and data collection and mounting the notebook to the drive.

3.1 Google Colab Pro Environment Setup

Google Colab was upgraded to Google Colab Pro for the implementation of this project since the size of the dataset is huge. Google Colab Pro provides more memory and disk space (Figure 1) for faster processing.

```
from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('Not using a high-RAM runtime')
else:
  print('You are using a high-RAM runtime!')
```

```
Your runtime has 27.3 gigabytes of available RAM

You are using a high-RAM runtime!
```

Figure 1: Memory provided in Google Colab Pro

Before starting the implementation, the notebook setting in Google Colab Pro is changed as shown in Figure 2. The Hardware accelerator is set to None since the dataset is a CSV file and the Runtime shape is changed from Standard to High-RAM to access additional memory provided in Colab Pro.

3.2 Data Collection

The dataset for this research work is obtained from the Global Terrorism Database (GTD) [1], which is an open-source dataset maintained by the University of Maryland as shown in Figure 3.

Once the CSV file is retrieved from the source, it is stored in the Google Drive associated with the account being used for further processing as shown in Figure 4.

[1]https://www.start.umd.edu/gtd/

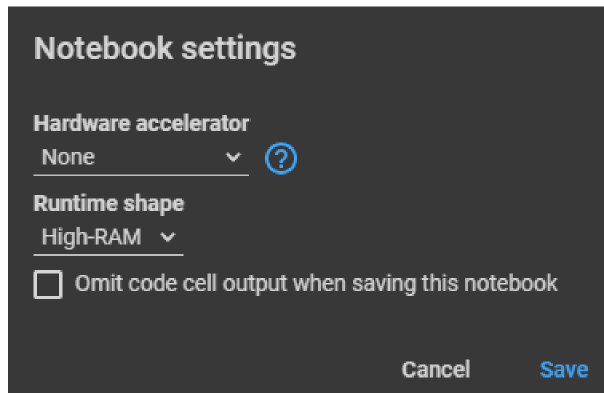

Figure 2: Google Colab Pro Notebook Settings

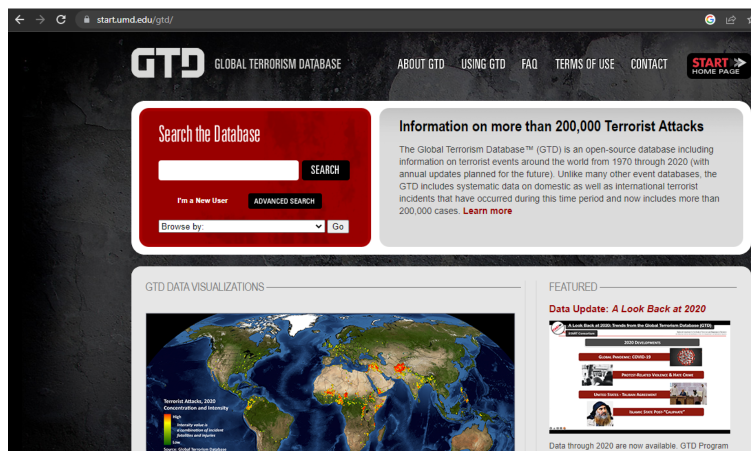

Figure 3: GTD website

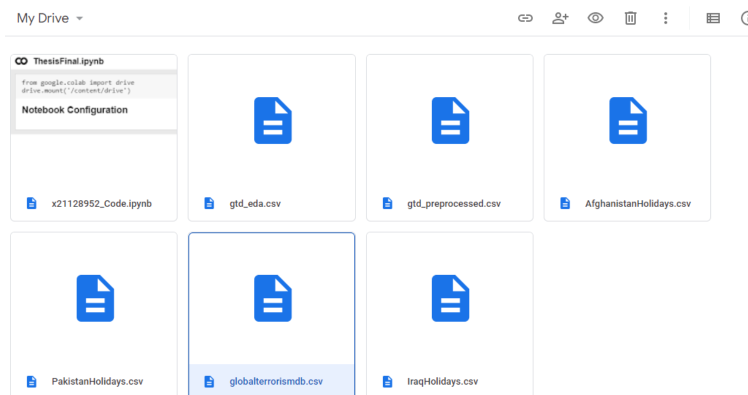

Figure 4: GTD dataset stored in Google Drive

3.3 Mounting Google Drive

In order to be able to use the dataset from the Google Drive, the drive needs to be mounted with the notebook as shown in Figure 5.

```
from google.colab import drive
drive.mount('/content/drive')

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Figure 5: Mounting Google Drive with the Notebook

3.4 Importing Python Libraries

Once the notebook is mounted with Google Drive, the python libraries required for the implementation are imported. The code for importing the libraries is shown in Figure 6. The libraries required for the implementation include: Numpy[2], Pandas[3], SkLearn[4], Matplotlib[5], Seaborn[6], NLTK[7], Plotly[8], and Folium[9].

```
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import matplotlib.gridspec as gridspec
import seaborn as sns
import time
import collections
from chart_studio import plotly
import chart_studio.plotly as py                # interactive, publication-quality graphs online
import plotly.graph_objs as go
from plotly.offline import download_plotlyjs,init_notebook_mode,plot,iplot
import string
import nltk                               # libraries and programs for symbolic and statistical natural language processing for English
from nltk.corpus import stopwords         # stop word is a commonly used word (such as "the", "a", "an", "in")
from nltk.tokenize import word_tokenize   # divides a string into substrings
from wordcloud import WordCloud, STOPWORDS
init_notebook_mode(connected=True)
import warnings; warnings.simplefilter('ignore')

from sklearn.preprocessing import scale
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score

from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

from imageio import imread                # provides an easy interface to read and write a wide range of image data, including animated images, volumetric data
import codecs                             # encoders and decoders
from subprocess import check_output       # allows you to spawn new processes
import folium                             # data wrangling strengths of the Python ecosystem and the mapping strengths of the Leaflet.js library
from folium import plugins
from folium.plugins import HeatMap
```

Figure 6: Code for importing Python Libraries

3.5 Loading the Dataset

After the prerequisite libraries have been imported, the next step is to get the dataset from Google Drive so that it may be processed further. The code to load the dataset is shown in Figure 7.

[2] https://numpy.org/
[3] https://pandas.pydata.org/
[4] https://scikit-learn.org/stable/
[5] https://matplotlib.org/
[6] https://seaborn.pydata.org/
[7] https://www.nltk.org/
[8] https://plotly.com/
[9] https://python-visualization.github.io/folium/

```
# Load GTD
gtd_df = pd.read_csv('/content/drive/MyDrive/globalterrorismdb.csv', low_memory=False, index_col = 0,
                     na_values=[''])
```

Figure 7: Loading the GTD dataset

4 Data Preparation

The dataset is imbalanced with a large number of missing values. Hence the data needs to be cleaned and prepared before implementing any models. The dataset was cleaned by following the below steps.

The dataset contains 2,09,706 entries which contains many missing values. The below function shown in Figure 8 checks for the number of missing values in each attribute and computes the percentage.

```
# Check the number of missing values in each attribute
count = gtd_df.isnull().sum()
percent = round(count / 209706 * 100, 2)
series = [count, percent]
result = pd.concat(series, axis=1, keys=['Count','Percent'])
result.sort_values(by='Count', ascending=False)
```

	Count	Percent
gsubname3	209683	99.99
weapsubtype4_txt	209636	99.97
weapsubtype4	209636	99.97
weaptype4_txt	209633	99.97
weaptype4	209633	99.97
claimmode3_txt	209566	99.93
claimmode3	209566	99.93
gsubname2	209522	99.91

Figure 8: Checking for Missing values

The attributes which contain more than 35% missing values are dropped in the first run to handle the missing values as shown in Figure 9.

For filling in the missing values in numeric attributes, the Mean/Median imputation method is used. The Impute function used is shown in Figure 10.

The following codes are mapped to the labels as shown in Figure 11. Categorical numbers are shown by the value -1, and categorical text values are shown by the value UNKNOWN. If a coded value is missing for a numeric characteristic, NAN is used instead. In many characteristics, the values 1, 0, and -1 are used for Yes, No, and Unknown.

5 Models Implementation

The codes for implementing the predictive models are described in the following section.

```python
target_attrs = result[result['Percent'] < 35.0]
keep_attrs = target_attrs.index.values

# The nperps attribute contain 18.91% blank values.  However, an additional 64.31% are
# coded (-99, -9) as unknown.
keep_attrs = keep_attrs[keep_attrs != 'nperps']
keep_attrs

# Remove attributes that duplicate another attribute
keep_attrs = keep_attrs[keep_attrs != 'country']
keep_attrs = keep_attrs[keep_attrs != 'region']
keep_attrs = keep_attrs[keep_attrs != 'attacktype1']
keep_attrs = keep_attrs[keep_attrs != 'targtype1']
keep_attrs = keep_attrs[keep_attrs != 'targsubtype1']
keep_attrs = keep_attrs[keep_attrs != 'natlty1']
keep_attrs = keep_attrs[keep_attrs != 'weaptype1']
keep_attrs = keep_attrs[keep_attrs != 'weapsubtype1']
```

```
array(['iyear', 'imonth', 'iday', 'extended', 'country', 'country_txt',
       'region', 'region_txt', 'provstate', 'city', 'latitude',
       'longitude', 'specificity', 'vicinity', 'summary', 'crit1',
       'crit2', 'crit3', 'doubtterr', 'multiple', 'success', 'suicide',
       'attacktype1', 'attacktype1_txt', 'targtype1', 'targtype1_txt',
       'targsubtype1', 'targsubtype1_txt', 'corp1', 'target1', 'natlty1',
       'natlty1_txt', 'gname', 'guncertain1', 'individual', 'nperpcap',
       'claimed', 'weaptype1', 'weaptype1_txt', 'weapsubtype1',
       'weapsubtype1_txt', 'nkill', 'nkillus', 'nkillter', 'nwound',
       'nwoundus', 'nwoundte', 'property', 'ishostkid', 'scite1',
       'dbsource', 'INT_LOG', 'INT_IDEO', 'INT_MISC', 'INT_ANY'],
      dtype=object)
```

Figure 9: Drop columns with maximum Missing values

```python
# Function to impute either the median or mean
def fill_value(attr):
    fill = 0.0
    threshold = 3
    attr_clean = attr.dropna()
    attr_std = attr_clean.std()
    outliers = attr_clean[attr_clean > (threshold * attr_std)]

    if (outliers.count() > 0):
        fill = attr_clean.median()
    else:
        fill = attr_clean.mean()

    return fill
```

Figure 10: Impute function to fill Missing values

```python
# Map the codes to labels
ynu_map = {1: 'YES', 0: 'NO', -1: 'UNKNOWN'}

# List of target attributes to map
ynu_attrs =['extended', 'vicinity', 'crit1', 'crit2', 'crit3', 'doubtterr', 'multiple',
            'success', 'suicide', 'guncertain1', 'individual', 'claimed', 'property',
            'ishostkid', 'INT_LOG', 'INT_IDEO', 'INT_MISC', 'INT_ANY']

# Iterate over each target attribute and map it
for att in ynu_attrs:
    att_txt = att + '_txt'
    subset_df[att_txt] = subset_df[att].map(ynu_map)

# Get the list of attributes, dropping the coded for labeled attributes
final_attrs = []

for attr in subset_df.columns.values:
    if attr not in ynu_attrs:
        final_attrs.append(attr)

subset_df2 = subset_df.loc[:, final_attrs]
subset_df2.info(verbose = True)
```

Figure 11: Map Codes to Labels

5.1 Weapon Classification using kNN

For classifying the weapons, the original dataset is split into 80% training and 20% testing sets as shown in Figure 12.

```
# Seed for reproducible results
seed = 1009

# Predictor variables with one hot encoding
X = pd.get_dummies(gtd_df[['country_txt', 'region_txt', 'attacktype1_txt', 'nkill']],
                   drop_first = True)

# Labels
y = gtd_df['weaptype1_txt']

# Create an 80/20 split for training and testing data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = seed, stratify = y)
```

Figure 12: Train-Test Split

A kNN classifier using 12 neighbors is created as shown in Figure 13.

```
start = time.time()

# Create the classifier
knn1 = KNeighborsClassifier(n_neighbors = 12)
print("The KNN classifier parameter:\n")
print(knn1)

# Fit it using the training data
knn1.fit(X_train, y_train)

# Predict the lables using the test dataset
pred_lables1 = knn1.predict(X_test)

# Display a sample of the predictions
print("\nTest set predictions:\n {}".format(pred_lables1))

# Calculate the accuracy
score1 = accuracy_score(y_test, pred_lables1)
print("\nAccuracy: {}".format(score1))

end = time.time()
print("\nExecution Seconds: {}".format((end - start)))
```

```
The KNN classifier parameter:

KNeighborsClassifier(n_neighbors=12)
KNeighborsClassifier(n_neighbors=12)
Test set predictions:
 ['Explosives' 'Explosives' 'Explosives' ... 'Explosives' 'Firearms'
 'Explosives']

Accuracy: 0.9225963488843814

Execution Seconds: 1.9045171737670898
```

Figure 13: kNN Classifier with 12 neighbors

Iterating from 1 to 12 to find the best value for K as shown in Figure 14.

Creating a KNN classifier using the best K of 11 neighbors from the previous test as shown in Figure 15.

5.2 Perpetrator Classification using Decision Tree and MLP Classifiers

For categorizing the perpetrator groups, Decision Tree and MLP classifier models are used. The code for implementing Decision Tree classifier is shown in Figure 16.

The code for implementing Neural Network MLPClassifier is shown in Figure 17.

```
[ ]  start = time.time()

     # Setup arrays to store train and test accuracies
     neighbors = np.arange(1, 12)
     train_accuracy = np.empty(len(neighbors))
     test_accuracy = np.empty(len(neighbors))

     # Loop over different values of k
     for i, k in enumerate(neighbors):
         # Setup a k-NN Classifier with k neighbors: knn
         knn2 = KNeighborsClassifier(n_neighbors=k)

         # Fit the classifier to the training data
         knn2.fit(X_train, y_train)

         #Compute accuracy on the training set
         train_accuracy[i] = knn2.score(X_train, y_train)

         #Compute accuracy on the testing set
         test_accuracy[i] = knn2.score(X_test, y_test)


     end = time.time()
     print("Execution Seconds: {}".format((end - start)))

     KNeighborsClassifier(n_neighbors=1)KNeighborsClassifier(n_neighbors=2)KNeig
```

Figure 14: Finding the Best K

```
start = time.time()

# Create the classifier
knn3 = KNeighborsClassifier(n_neighbors = 11)
print("The KNN classifier parameter:\n")
print(knn3)

# Fit it using the training data
knn3.fit(X_train, y_train)

# Predict the lables using the test dataset
pred_lables3 = knn3.predict(X_test)

# Calculate the accuracy
score3 = accuracy_score(y_test, pred_lables3)
print("\nAccuracy: {}".format(score3))

end = time.time()
print("\nExecution Seconds: {}".format((end - start)))

The KNN classifier parameter:

KNeighborsClassifier(n_neighbors=11)
KNeighborsClassifier(n_neighbors=11)
Accuracy: 0.9225152129817444

Execution Seconds: 1.8322088718414307
```

Figure 15: kNN Classifier with best K value

```
#Decision Tree Classifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import plot_confusion_matrix

import time
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier()

param_grid = {'criterion':['gini','entropy'],'min_samples_leaf':[1, 2, 3, 4, 5, 10, 25, 50, 100],'max_depth':np.arange(2, 10)}

grid_dtc = GridSearchCV(dtc,param_grid,scoring='accuracy',cv = 10,refit = True)
```

```
# Don't need to standardize variables here
start_time = time.time()

grid_dtc.fit(x_train,y_train)
print("Best Score ==> ", grid_dtc.best_score_)
print("Tuned Paramerers ==> ", grid_dtc.best_params_)
print("Accuracy on Train set ==> ", grid_dtc.score(x_train,y_train))
print("Accuracy on Test set ==> ", grid_dtc.score(x_test,y_test))

end_time = time.time()
run_time = end_time - start_time

print('Runtime of the Grid Search for the Decision Tree Algorithm: {:.5f}s'.format(run_time))
```

```
GridSearchCV(cv=10, estimator=DecisionTreeClassifier(),
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': array([2, 3, 4, 5, 6, 7, 8, 9]),
                         'min_samples_leaf': [1, 2, 3, 4, 5, 10, 25, 50, 100]},
             scoring='accuracy')Best Score ==>  0.9074877904383696
Tuned Paramerers ==>  {'criterion': 'entropy', 'max_depth': 8, 'min_samples_leaf': 3}
Accuracy on Train set ==>  0.9116443100950942
Accuracy on Test set ==>  0.9083161915956509
Runtime of the Grid Search for the Decision Tree Algorithm: 55.39397s
```

Figure 16: Decision Tree Classifier

```
#Neural Network
from sklearn.neural_network import MLPClassifier
```

```
start_time = time.time()

mlp = MLPClassifier(hidden_layer_sizes=(35,35,35), max_iter=250, random_state=42)
mlp.fit(x_train_std, y_train)

print("Accuracy on Train set ==> ", mlp.score(x_train_std,y_train))
print("Accuracy on Test set ==> ", mlp.score(x_test_std,y_test))

end_time = time.time()
mlp_run_time = end_time - start_time

print('Runtime of the Neural Network: {:.5f}s'.format(mlp_run_time))
```

```
MLPClassifier(hidden_layer_sizes=(35, 35, 35), max_iter=250, random_state=42)Accuracy on Train set ==>  0.9275773033566346
Accuracy on Test set ==>  0.904202174551866
Runtime of the Neural Network: 48.09412s
```

Figure 17: MLP Classifier

5.3 Time Series Analysis for forecasting future terrorist attacks

For forecasting the terrorist attacks in Iraq, Afghanistan, and Pakistan, time series analysis is utilized with Exponential Weighted Moving Average. Initially, the data frame is reindexed to include all days for the 10 year period and fill added days with zero. Smoothing is applied using exponential weighted moving average. A modified dataset is created to comply with the Facebook Prophet requirements. Then the time series model is created and the holidays dataset of that nation is factored. Finally, predictions are done for 365 days after last the data point. The code used for implementing these steps is shown in Figure 18.

```
idx = pd.date_range('2013-01-01', '2020-12-31')

iraq_ts = iraq_counts.set_index('incident_date')

iraq_ts = iraq_ts.reindex(idx, fill_value=0)
iraq_ts.head()
```

```
[ ]  # Use a 30 day span for averaging
     iraq_ewm = iraq_ts.ewm(span=30, adjust=False).mean()

     iraq_ewm.head()
```

```
[ ]  import fbprophet

     iraq_fb = iraq_ts.copy()
     iraq_fb['index1'] = iraq_fb.index
     iraq_fb.columns = ['y', 'ds']

     iraq_fb.head()
```

```
# Make the prophet model and fit on the data
prophet1 = fbprophet.Prophet(changepoint_prior_scale=0.15, holidays=iraq_holidays)
prophet1.fit(iraq_fb)
```

```
# Specify 365 days out to predict
future_data = prophet1.make_future_dataframe(periods=365, freq = 'D')

# Predict the values
forecast_data = prophet1.predict(future_data)
forecast_data[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()
#forecast_data.tail()
```

Figure 18: Time Series Analysis

5.4 Regression Analysis for predicting casualties

One machine learning regression model and one deep learning regression model is used for predicting the fatalities of an attack. For implementing the MLPRegressor, the features are scaled using StandardScaler function as shown in Figure 19.

The code for implementing the MLPRegressor is shown in Figure 20.

The code for implementing the RandomForestRegressor algorithm is shown in Figure 21. Looking at the MSE values obtained by these models, it is evident that Random Forest outperforms MLP model.

```
[ ] # Feature Scaling
    from sklearn.preprocessing import StandardScaler
    sc = StandardScaler()
    X = X.astype('float64')
    X = sc.fit_transform(X)
```

Figure 19: Feature Scaling for MLPRegressor

```
[ ] from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state = 0)

[ ] from sklearn.neural_network import MLPRegressor
    from sklearn.model_selection import GridSearchCV
    mlp = MLPRegressor()
    parameters = {'activation' : ['tanh'],
                  'solver': ['sgd'],
                  'hidden_layer_sizes':np.arange(1, 18)}

    mlpregressor = GridSearchCV(mlp, parameters, scoring='neg_mean_squared_error', cv=5)
    mlpregressor.fit(X_train, y_train)

    GridSearchCV(cv=5, estimator=MLPRegressor(),
                 param_grid={'activation': ['tanh'],
                             'hidden_layer_sizes': array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17]),
                             'solver': ['sgd']},
                 scoring='neg_mean_squared_error')

[ ] print(mlpregressor.best_params_)
    print(mlpregressor.best_score_)

    {'activation': 'tanh', 'hidden_layer_sizes': 15, 'solver': 'sgd'}
    -106.68446992920929

[ ] from sklearn.metrics import mean_squared_error

    y_pred = mlpregressor.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    print('MSE on MLP model =', mse)
    mse_arr = []
    mse_arr.append(mse)
    mse_arr
    mlp_mse = mse

    MSE on MLP model = 60.18700196046154
    [60.18700196046154]
```

Figure 20: Predicting casualties using MLPRegressor

```
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor()

parameters = {'max_depth': range(3,5),
              'n_estimators': (40, 50,),}

rfr = GridSearchCV(rf, parameters, scoring='neg_mean_squared_error', cv=5)

rfr.fit(X_train, y_train)

print(rfr.best_params_)
print(rfr.best_score_)

GridSearchCV(cv=5, estimator=RandomForestRegressor(),
             param_grid={'max_depth': range(3, 5), 'n_estimators': (40, 50)},
             scoring='neg_mean_squared_error'){'max_depth': 4, 'n_estimators': 50}
-59.71564837325705

[ ] y_pred = rfr.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    print('MSE on random_forset model =', mse)
    mse_arr.append(mse)
    mse_arr
    rf_mse = mse

    MSE on random_forset model = 20.52859215086285
    [60.18700196046154, 20.52859215086285]
```

Figure 21: Predicting casualties using RandomForestRegressor