Iteration 1:

Cleaning initial datasets:

* Remove empty rows.
* Filter data for locations (i.e. Europe).
* Removing unneeded rows.
* Removing all data with latitudes and longitudes over a threshold.

Put all of this into a DataCleaner class.

import csv  
  
  
class DataCleaner:  
  
 def \_\_init\_\_(self, input\_filename, output\_filename):  
 self.input\_filename = input\_filename  
 self.output\_filename = output\_filename  
  
 with open(self.input\_filename, mode='r', newline='', encoding='utf-8') as infile:  
 reader = csv.reader(infile)  
 self.header = next(reader)  
 self.data = [row for row in reader]  
  
 def save(self):  
 with open(self.output\_filename, mode='w', newline='', encoding='utf-8') as outfile:  
 writer = csv.writer(outfile)  
 writer.writerow(self.header)  
 writer.writerows(self.data)  
  
 def remove\_irrelevant\_data(self, rows\_and\_headers\_to\_keep):  
 new\_rows = []  
 for row in self.data:  
 new\_row\_data = []  
 for index in rows\_and\_headers\_to\_keep:  
 new\_row\_data.append(row[index])  
 new\_rows.append(new\_row\_data)  
  
 new\_header = []  
 for index in rows\_and\_headers\_to\_keep:  
 new\_header.append(self.header[index])  
  
 self.data = new\_rows  
 self.header = new\_header  
  
 def remove\_if\_row\_empty(self, row\_index):  
 self.data = [row for row in self.data if row[row\_index] != '']  
  
 def filter\_for\_locations(self, row\_index, filter\_data):  
 self.data = [row for row in self.data if row[row\_index] in filter\_data]  
  
 def remove\_all\_over\_threshold(self, row\_index, threshold):  
 self.data = [row for row in self.data if float(row[row\_index]) < threshold]

Iteration 2:

Loading datasets into memory.

Obtaining array of only latitudes and longitudes of data from the datasets.

dataset1\_path = 'Data/operations\_cleaned.csv'  
dataset2\_path = 'Data/worldcitiespop.csv'  
  
dataset1 = pd.read\_csv(dataset1\_path)  
dataset2 = pd.read\_csv(dataset2\_path)  
  
dataset1.dropna(subset=['Target Latitude', 'Target Longitude'], inplace=True)  
dataset2.dropna(subset=['Latitude', 'Longitude'], inplace=True)  
  
locations1 = dataset1[['Target Latitude', 'Target Longitude']]  
locations2 = dataset2[['Latitude', 'Longitude']]

Iteration 3:

Cluster data and fit it with European locations dataset.

Predict the bombing locations on it.

Plot data on a map.

Initially, I had clusters size be the length of the European locations dataset but that was never going to finish and it wouldn’t give anything. Then ran it with 100, 1000 and 2000.

kmeans = KMeans(n\_clusters=n\_clusters, random\_state=42, verbose=2)  
kmeans.fit(locations2)  
  
joblib.dump(kmeans, 'kmeans\_model.pkl')  
print("Model saved to 'kmeans\_model.pkl'.")  
  
# centroids = kmeans.cluster\_centers\_  
closest\_centroids = kmeans.predict(locations1)  
  
lat\_min, lat\_max = 12.0, 72.0  
lon\_min, lon\_max = -25.0, 60.0  
  
centroids = kmeans.cluster\_centers\_[closest\_centroids]  
centroids\_filtered = np.array([c for c in centroids if lat\_min <= c[0] <= lat\_max and lon\_min <= c[1] <= lon\_max])  
  
map\_emea = folium.Map(location=[34.0, 9.0], zoom\_start=3)  
  
# Add markers for the filtered centroids  
for lat, lon in centroids\_filtered:  
 folium.Marker([lat, lon]).add\_to(map\_emea)  
  
# Save or display the map  
map\_emea.save('Data/emea\_matched\_locations\_map.html')

Iteration 4:

Rework existing code into new DataProcessing file as it was messy.

Validate all was still working.

Add plotting to scatter diagrams.

Added filtering to significant centroids i.e. removed any clusters with less than 10 bombings to have occurred there.

import csv  
import os  
  
import folium  
import joblib  
import numpy as np  
from matplotlib import pyplot as plt  
from sklearn.cluster import KMeans  
  
from Final.DataCleaner import DataCleaner  
  
os.environ['LOKY\_MAX\_CPU\_COUNT'] = '4'  
  
def main():  
 # Cleaning the original csv files.  
 should\_load\_kmeans = True  
 clusters\_count = 10  
 bombing\_dataset\_name = 'Data/DataSets/operations\_cleaned.csv'  
 world\_locations\_dataset\_name = 'Data/DataSets/emea\_locations.csv'  
 kmeans\_model\_name = f'Data/kmeans\_bombing\_model\_clusters\_{clusters\_count}.pkl'  
 scatter\_diagram\_name = f'Data/Images/bombing\_scatter\_clusters\_{clusters\_count}.png'  
 map\_name = f'Data/Images/map\_of\_locations\_clusters\_{clusters\_count}.html'  
  
 print('Cleaning original csv files.')  
 # clean\_bombing\_operations()  
 # clean\_world\_locations()  
  
 # Loading the files into memory and filters for relevant rows.  
 \_, bombing\_locations\_full = load\_file('Data/DataSets/operations\_cleaned.csv')  
 \_, world\_locations\_full = load\_file('Data/DataSets/emea\_locations.csv')  
 print('Loading the files into memory and filtering for relevant rows.')  
 \_, bombing\_locations\_full = load\_file(bombing\_dataset\_name)  
 \_, world\_locations\_full = load\_file(world\_locations\_dataset\_name)  
  
 bombing\_locations = filter\_for\_latitude\_and\_longitude(bombing\_locations\_full, 4, 5)  
 world\_locations = filter\_for\_latitude\_and\_longitude(world\_locations\_full, 3, 4)  
  
 print('Beginning KMeans.')  
 bombings\_per\_centroid\_threshold = 10  
  
 if should\_load\_kmeans:  
 kmeans = joblib.load(kmeans\_model\_name)  
 else:  
 kmeans = cluster\_and\_fit\_data(world\_locations, clusters\_count)  
  
 print("Fitting bombing locations to a world location.")  
 nearest\_locations\_indices = kmeans.predict(bombing\_locations)  
  
 print(  
 f'Filtering cluster locations to only include ones that have more bombings than '  
 f'{bombings\_per\_centroid\_threshold}.')  
 significant\_centroids = filter\_significant\_centroids(kmeans, nearest\_locations\_indices,  
 bombings\_per\_centroid\_threshold)  
  
 if not should\_load\_kmeans:  
 print(f'Saving KMeans model to {kmeans\_model\_name}.')  
 save\_kmeans\_model(kmeans, kmeans\_model\_name)  
 print('Ending KMeans.')  
  
 print('Beginning display data.')  
 print('Rounding for significant centroids.')  
 unique\_significant\_centroids = round\_significant\_centroids(significant\_centroids)  
  
 print('Plotting on scatter diagram.')  
 plot\_and\_save\_on\_scatter\_diagram(unique\_significant\_centroids, scatter\_diagram\_name)  
  
 print("Plotting on map.")  
 plot\_and\_save\_on\_map(unique\_significant\_centroids, map\_name)  
  
def clean\_bombing\_operations(input\_name='Data/DataSets/operations.csv',  
 @ @ -99, 5 +145, 109 @ @ def filter\_for\_latitude\_and\_longitude(data, latitude\_index,  
 longitude\_index):  
 return np.array([[row[latitude\_index], row[longitude\_index]] for row in data])  
  
def cluster\_and\_fit\_data(data, number\_clusters):  
 *"""  
 Clusters the data and fits the world locations dataset to it.  
 Args:  
 data: World location dataset.  
 number\_clusters: Number of clusters.  
 Returns: KMeans object.  
 """* return KMeans(n\_clusters=number\_clusters, n\_init=10).fit(data)  
  
def filter\_significant\_centroids(kmeans, nearest\_locations\_indices,  
 bombings\_per\_centroid\_threshold=10):  
 *"""  
 Filters the centroids from the kmeans and excludes any that have less bombings fitted to than the threshold.  
 Args:  
 kmeans: KMeans object.  
 nearest\_locations\_indices: Indices of the nearest locations of the bombings  
 bombings\_per\_centroid\_threshold: Threshold of how many bombings have to have been fit to the centroid to not  
 be excluded.  
 Returns: The significant clusters that have the necessary bombings in it.  
 """* bombings\_per\_centroid = np.bincount(nearest\_locations\_indices)  
 significant\_centroids\_indices = np.where(bombings\_per\_centroid > bombings\_per\_centroid\_threshold)[0]  
 significant\_centroids = kmeans.cluster\_centers\_[significant\_centroids\_indices]  
  
 return significant\_centroids  
  
def round\_significant\_centroids(centroids):  
 *"""  
 Used for processing on the graphs. Gets only one value for each centroid to avoid duplication for the graphs.  
 Prevents delays on the map. Credit: ChatGPT, I couldn't figure out how to do it the best.  
 Args:  
 centroids: Centroids from KMeans.  
 Returns: List of unique centroids.  
 """* rounded\_significant\_centroids = np.round(centroids, decimals=5)  
 dtype = np.dtype(','.join(['f8'] \* rounded\_significant\_centroids.shape[1]))  
 \_, unique\_indices = np.unique(rounded\_significant\_centroids.view(dtype), return\_index=True, axis=0)  
 unique\_significant\_centroids = centroids[unique\_indices]  
  
 return unique\_significant\_centroids  
  
def save\_kmeans\_model(kmeans, name):  
 *"""  
 Saves the model to a file.  
 Args:  
 kmeans: KMeans object.  
 name: File path to save location.  
 """* joblib.dump(kmeans, name)  
  
def plot\_and\_save\_on\_scatter\_diagram(centroids, file\_name):  
 *"""  
 Plots the KMeans data on a scatter diagram.  
 Args:  
 centroids: Data to plot.  
 file\_name: File path to save location.  
 """* latitude = centroids[:, 0]  
 longitude = centroids[:, 1]  
  
 plt.scatter(longitude, latitude)  
 plt.xlabel('Longitude')  
 plt.ylabel('Latitude')  
 plt.title('Geographical Scatter Plot')  
  
 print(f"Saving diagram to {file\_name}")  
 plt.savefig(file\_name)  
 plt.close()  
  
def plot\_and\_save\_on\_map(centroids, file\_name):  
 *"""  
 Plots the KMeans data on a map of Europe.  
 Args:  
 centroids: Data to plot.  
 file\_name: File path to save location.  
 """* map\_center = np.mean(centroids, axis=0)  
  
 map = folium.Map(location=map\_center, zoom\_start=4)  
  
 for location in centroids:  
 folium.Marker([location[0], location[1]]).add\_to(map)  
  
 print(f"Saving map to {file\_name}")  
 map.save(file\_name)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

Iteration 5:

Added saving of kmeans so it doesn’t need to be run each time.

Added method to find the closest world location. This is done with a new dataset which contains all villages, towns and cities with a population more than 1000. Saves this to a file.

Note: above code includes this

Iteration 6:

Added to find closest world location method to first check another new dataset of capitals. If the capitals are within a reasonable distance (35km) it will use that. This is because in the previous closest locations dataset for capitals they are broken up into subdivisions, so it first checks for a capital and if not, it will default to the closest world location dataset.

def find\_closest\_world\_locations(centroids, world\_locations):  
 *"""  
 Finds the closest world location to each centroid.  
 Args:  
 centroids: The centroids from KMeans.  
 world\_locations: The world locations to compare against.  
 Returns:  
 A list of tuples containing the centroid and its closest world location.  
 """* closest\_locations = []  
 for centroid in centroids:  
 closest\_location = min(world\_locations,  
 key=lambda loc: np.linalg.norm(  
 centroid - np.array(  
 [float(loc[19].split(', ')[0]), float(loc[19].split(', ')[1])])))  
 closest\_locations.append((centroid, closest\_location))  
 return closest\_locations  
  
def save\_closest\_locations\_to\_csv(closest\_locations, file\_name):  
 *"""  
 Saves the closest world locations to each centroid to a CSV file.  
 Args:  
 closest\_locations: A list of tuples containing the centroid and its closest world location.  
 file\_name: The name of the file to save the data.  
 """* with open(file\_name, mode='w', newline='', encoding='utf-8') as file:  
 writer = csv.writer(file)  
 writer.writerow(  
 ['Centroid Latitude', 'Centroid Longitude', 'Name', 'Country Name', 'Closest Location Latitude',  
 'Closest Location Longitude'])  
 for centroid, location in closest\_locations:  
 writer.writerow([centroid[0], centroid[1], location[2], location[7], location[19].split(', ')])

Iteration 7:

Added plotting of the previous data on the map. Labels were also added to the markers. They contained data such as location name, latitude and longitude and so on. A blue label was made for the cluster location and a red label was made for the closest location.

This also meant adjustments to methods such as the filter significant centroids method as it now also kept count of how many centroids were present while excluding lesser ones.

Adjusted the scatter diagram to also generate one for the closest locations.

Final Code:

import csv  
import os  
  
import folium  
import joblib  
import numpy as np  
from geopy.distance import geodesic  
from matplotlib import pyplot as plt  
from sklearn.cluster import KMeans  
  
from NN.DataCleaner import DataCleaner  
  
os.environ['LOKY\_MAX\_CPU\_COUNT'] = '4'  
  
  
def main():  
 *"""  
  
 First, it cleans the data. Next, it loads all the required files into memory. There are 4 csv files, with them being  
 the dataset of Allied bombings that happened in europe during WW2, the next is a dataset of every village, town and  
 city in Europe. The next is the same, but it only includes locations with a population greater than 1000, and  
 finally a dataset of all European capitals.  
  
 Next it creates kmeans for each location in europe and fits the bombings to each of those locations. It then removes  
 any that have 10 or fewer bombings that occurred there. It then links them to either the closest significant  
 location (pop > 1000) or the closest capital. The issue is, capitals in the other datasets are split into suburbs,  
 so I used another dataset to get the capital locations and if it is within the reasonable distance (I found 35km to  
 be the best) it will select that, or else the closest significant location.  
  
 It then saves the results into a csv file, two scatter diagrams and plots them on a map. Each map contains two  
 labels for each cluster, with one in blue being the cluster location, and the other in red being the closest  
 significant location. Each one contains a label which has an ID to link it to the other, as well as location  
 information and how many bombings took place in that location.  
  
 """* should\_load\_kmeans = False  
 clusters\_count = 1000  
 bombing\_dataset\_name = 'Data/DataSets/operations\_cleaned.csv'  
 world\_locations\_dataset\_name = 'Data/DataSets/emea\_locations.csv'  
 europe\_cities\_dataset\_name = 'Data/DataSets/europe\_locations\_pop\_greater\_1000.csv'  
 europe\_capitals\_dataset\_name = 'Data/DataSets/europe\_capitals.csv'  
 kmeans\_model\_name = f'Data/kmeans\_bombing\_model\_clusters\_{clusters\_count}.pkl'  
 scatter\_centroids\_diagram\_name = f'Data/Images/bombing\_scatter\_centroids\_clusters\_{clusters\_count}.png'  
 scatter\_closest\_location\_diagram\_name = \  
 f'Data/Images/bombing\_scatter\_closest\_location\_clusters\_{clusters\_count}.png'  
 map\_name = f'Data/Images/map\_of\_locations\_clusters\_{clusters\_count}.html'  
  
 print('Cleaning original csv files.')  
 # clean\_bombing\_operations()  
 # clean\_world\_locations()  
 # clean\_country\_capitals()  
  
 print('Loading the files into memory and filtering for relevant rows.')  
 \_, bombing\_locations\_full = load\_file(bombing\_dataset\_name)  
 \_, world\_locations\_full = load\_file(world\_locations\_dataset\_name)  
 \_, europe\_cities\_full = load\_file(europe\_cities\_dataset\_name)  
 \_, europe\_capitals\_full = load\_file(europe\_capitals\_dataset\_name)  
  
 bombing\_locations = filter\_for\_latitude\_and\_longitude(bombing\_locations\_full, 4, 5)  
 world\_locations = filter\_for\_latitude\_and\_longitude(world\_locations\_full, 3, 4)  
 europe\_cities = split\_and\_filter\_for\_latitude\_and\_longitude(europe\_cities\_full, 19, ', ')  
  
 print('Beginning KMeans.')  
 bombings\_per\_centroid\_threshold = 10  
  
 if should\_load\_kmeans:  
 kmeans = joblib.load(kmeans\_model\_name)  
 else:  
 kmeans = cluster\_and\_fit\_data(world\_locations, clusters\_count)  
  
 print("Fitting bombing locations to a world location.")  
 nearest\_locations\_indices = kmeans.predict(bombing\_locations)  
  
 print(  
 f'Filtering cluster locations to only include ones that have more bombings than '  
 f'{bombings\_per\_centroid\_threshold}.')  
 significant\_centroids = filter\_significant\_centroids(kmeans, nearest\_locations\_indices,  
 bombings\_per\_centroid\_threshold)  
  
 if not should\_load\_kmeans:  
 print(f'Saving KMeans model to {kmeans\_model\_name}.')  
 save\_kmeans\_model(kmeans, kmeans\_model\_name)  
 print('Ending KMeans.')  
  
 print('Beginning display data.')  
 print('Rounding for significant centroids.')  
 unique\_significant\_centroids = combine\_significant\_centroids(significant\_centroids)  
  
 print("Finding closest world locations to centroids.")  
 closest\_locations = find\_closest\_world\_locations(unique\_significant\_centroids, europe\_cities\_full,  
 europe\_capitals\_full)  
  
 print("Saving closest locations to CSV.")  
 closest\_locations\_csv\_name = f'Data/Closest\_Locations\_to\_Centroids\_{clusters\_count}.csv'  
 save\_closest\_locations\_to\_csv(closest\_locations, closest\_locations\_csv\_name)  
  
 print('Plotting centroids on scatter diagram.')  
 centroids\_without\_count = np.array([centroid[0] for centroid in unique\_significant\_centroids])  
 plot\_and\_save\_on\_scatter\_diagram(centroids\_without\_count, scatter\_centroids\_diagram\_name)  
  
 print('Plotting closest locations on scatter diagram.')  
 closest\_locations\_lat\_long = filter\_for\_latitude\_and\_longitude\_with\_additional\_index(closest\_locations, 1, 2, 3)  
 plot\_and\_save\_on\_scatter\_diagram(closest\_locations\_lat\_long, scatter\_closest\_location\_diagram\_name)  
  
 plot\_and\_save\_on\_map(closest\_locations, map\_name)  
  
  
def clean\_bombing\_operations(input\_name='Data/DataSets/operations.csv',  
 output\_name='Data/DataSets/operations\_cleaned.csv'):  
 *"""  
 Cleans the original bombing dataset and removes all irrelevant data before outputting the cleaned data into a new  
 csv file. Limits the data to Europe, Middle East and North Africa.  
 Args:  
 input\_name: Name of the original file.  
 output\_name: Name of the new file.  
  
 """* emea\_countries\_in\_dataset = ['POLAND', 'AUSTRIA', 'YUGOSLAVIA', 'ITALY', 'SICILY', 'BULGARIA', 'FRANCE',  
 'GREAT BRITAIN', 'PANTELLARIA', 'SARDINIA', 'ROMANIA', 'HOLLAND OR NETHERLANDS',  
 'GERMANY', 'ALBANIA', 'LUXEMBOURG', 'CYPRUS', 'CRETE', 'DENMARK', 'HUNGARY', 'CORSICA',  
 'SWITZERLAND', 'NORWAY', 'BELGIUM', 'GREECE', 'CZECHOSLOVAKIA']  
 rows\_and\_headers\_to\_keep = [1, 3, 14, 15, 19, 20]  
  
 data\_cleaner = DataCleaner(input\_name, output\_name)  
 data\_cleaner.remove\_if\_row\_empty(3)  
 data\_cleaner.remove\_if\_row\_empty(19)  
 data\_cleaner.remove\_if\_row\_empty(20)  
 data\_cleaner.filter\_for\_locations(14, emea\_countries\_in\_dataset)  
 data\_cleaner.remove\_irrelevant\_data(rows\_and\_headers\_to\_keep)  
 data\_cleaner.remove\_all\_over\_threshold(5, 80)  
 data\_cleaner.save()  
  
  
def clean\_world\_locations(input\_name='Data/DataSets/world\_locations.csv',  
 output\_name='Data/DataSets/emea\_locations.csv'):  
 *"""  
 Cleans the original locations dataset and removes all irrelevant data before outputting the cleaned data into a new  
 csv file. Limits the data to Europe, Middle East and North Africa.  
 Args:  
 input\_name: Name of the original file.  
 output\_name: Name of the new file.  
  
 """* emea\_county\_codes\_in\_dataset = ['lt', 'al', 'mt', 'cz', 'ch', 'hr', 'ad', 'pl', 'ro', 'by', 'is', 'cy', 'lu', 'at',  
 'sk', 'gb', 'me', 'be', 'ru', 'fr', 'gr', 'ee', 'pt', 'ua', 'mk', 'se', 'bg', 'sm',  
 'de', 'es', 'ba', 'rs', 'ie', 'hu', 'li', 'mc', 'md', 'si', 'fi', 'it', 'no', 'nl',  
 'lv', 'dk']  
  
 rows\_and\_headers\_to\_keep = [0, 1, 2, 5, 6]  
  
 data\_cleaner = DataCleaner(input\_name, output\_name)  
 data\_cleaner.filter\_for\_locations(0, emea\_county\_codes\_in\_dataset)  
 data\_cleaner.remove\_irrelevant\_data(rows\_and\_headers\_to\_keep)  
 data\_cleaner.save()  
  
  
def clean\_country\_capitals(input\_name='Data/DataSets/country\_capitals.csv',  
 output\_name='Data/DataSets/europe\_capitals.csv'):  
 *"""  
 Cleans the country capitals dataset to only include europe capitals  
 Args:  
 input\_name: Name of the original file.  
 output\_name: Name of the new file.  
  
 """* european\_countries = [  
 "Albania", "Andorra", "Armenia", "Austria", "Azerbaijan", "Belarus", "Belgium", "Bosnia and Herzegovina",  
 "Bulgaria", "Croatia", "Cyprus", "Czechia", "Denmark", "Estonia", "Finland", "France", "Georgia", "Germany",  
 "Greece", "Holy See", "Hungary", "Iceland", "Ireland", "Italy", "Kazakhstan", "Latvia", "Liechtenstein",  
 "Lithuania", "Luxembourg", "Malta", "Moldova", "Monaco", "Montenegro", "Netherlands", "North Macedonia",  
 "Norway", "Poland", "Portugal", "Romania", "Russia", "San Marino", "Serbia", "Slovakia", "Slovenia", "Spain",  
 "Sweden", "Switzerland", "Turkey", "Ukraine", "United Kingdom"]  
  
 data\_cleaner = DataCleaner(input\_name, output\_name)  
 data\_cleaner.filter\_for\_locations(0, european\_countries)  
 data\_cleaner.save()  
  
  
def load\_file(file\_name):  
 *"""  
 Opens a csv file.  
 Args:  
 file\_name: Name of the file.  
  
 Returns: The header and the rows.  
  
 """* with open(file\_name, mode='r', newline='', encoding='utf-8') as infile:  
 reader = csv.reader(infile)  
 header = next(reader)  
 rows = [row for row in reader]  
  
 return header, rows  
  
  
def filter\_for\_latitude\_and\_longitude(data, latitude\_index, longitude\_index):  
 *"""  
 Filters the data and only gets the latitude and longitude data.  
 Args:  
 data: Raw data from the csv.  
 latitude\_index: Index of the row for latitude.  
 longitude\_index: Index of the row for longitude.  
  
 Returns: Numpy array of latitude and longitudes.  
  
 """* return np.array([[row[latitude\_index], row[longitude\_index]] for row in data])  
  
  
def filter\_for\_latitude\_and\_longitude\_with\_additional\_index(data, contained\_data, latitude\_index, longitude\_index):  
 *"""  
 Filters the data and only gets the latitude and longitude data.  
 Args:  
 data: Raw data from the csv.  
 contained\_data: Index of the object within.  
 latitude\_index: Index of the row for latitude.  
 longitude\_index: Index of the row for longitude.  
  
 Returns: Numpy array of latitude and longitudes.  
  
 """* return np.array(  
 [[float(row[contained\_data][latitude\_index]), float(row[contained\_data][longitude\_index])] for row in data])  
  
  
def split\_and\_filter\_for\_latitude\_and\_longitude(data, index, split\_value):  
 *"""  
 Filters the data and only gets the latitude and longitude data.  
 Args:  
 data: Raw data from the csv.  
 index: Index of the row.  
 split\_value: Value to split the index by.  
  
 Returns: Numpy array of latitude and longitudes.  
  
 """* return np.array([[row[index].split(split\_value)[0], row[index].split(split\_value)[1]] for row in data])  
  
  
def cluster\_and\_fit\_data(data, number\_clusters):  
 *"""  
 Clusters the data and fits the world locations dataset to it.  
 Args:  
 data: World location dataset.  
 number\_clusters: Number of clusters.  
  
 Returns: KMeans object.  
  
 """* return KMeans(n\_clusters=number\_clusters, n\_init=10).fit(data)  
  
  
def filter\_significant\_centroids(kmeans, nearest\_locations\_indices,  
 bombings\_per\_centroid\_threshold=10):  
 *"""  
 Filters the centroids from the kmeans and excludes any that have less bombings fitted to than the threshold.  
 Args:  
 kmeans: KMeans object.  
 nearest\_locations\_indices: Indices of the nearest locations of the bombings  
 bombings\_per\_centroid\_threshold: Threshold of how many bombings have to have been fit to the centroid to not  
 be excluded.  
  
 Returns: The significant clusters that have the necessary bombings in it.  
  
 """* bombings\_per\_centroid = np.bincount(nearest\_locations\_indices)  
 significant\_centroids\_indices = np.where(bombings\_per\_centroid > bombings\_per\_centroid\_threshold)[0]  
 significant\_centroids = [(kmeans.cluster\_centers\_[index], bombings\_per\_centroid[index])  
 for index in significant\_centroids\_indices]  
  
 return significant\_centroids  
  
  
def combine\_significant\_centroids(centroids\_with\_counts):  
 *"""  
 Combine the centroid coordinates.  
 Args:  
 centroids\_with\_counts: List of tuples with each centroid and the count of bombings.  
  
 Returns: List of unique rounded centroids with associated bombing counts.  
  
 """* rounded\_centroids\_with\_counts = []  
 seen = set()  
  
 for centroid, count in centroids\_with\_counts:  
 rounded\_centroid = tuple(np.round(centroid, decimals=5))  
 if rounded\_centroid not in seen:  
 seen.add(rounded\_centroid)  
 rounded\_centroids\_with\_counts.append((rounded\_centroid, count))  
  
 return rounded\_centroids\_with\_counts  
  
  
def save\_kmeans\_model(kmeans, name):  
 *"""  
 Saves the model to a file.  
 Args:  
 kmeans: KMeans object.  
 name: File path to save location.  
  
 """* joblib.dump(kmeans, name)  
  
  
def plot\_and\_save\_on\_scatter\_diagram(points, file\_name):  
 *"""  
 Plots the KMeans data on a scatter diagram.  
 Args:  
 points: Data to plot.  
 file\_name: File path to save location.  
  
  
 """* latitude = points[:, 0]  
 longitude = points[:, 1]  
  
 plt.scatter(longitude, latitude)  
 plt.xlabel('Longitude')  
 plt.ylabel('Latitude')  
 plt.title('Geographical Scatter Plot')  
  
 print(f"Saving diagram to {file\_name}")  
 plt.savefig(file\_name)  
 plt.close()  
  
  
def plot\_and\_save\_on\_map(closest\_locations, file\_name):  
 map\_center = np.mean([loc[0][:2] for loc in closest\_locations], axis=0)  
 map = folium.Map(location=map\_center, zoom\_start=4)  
  
 for i, (centroid, location, count) in enumerate(closest\_locations):  
 folium.Marker(  
 centroid[:2],  
 popup=f"Centroid ID: {i}, Bombings Count: {count}<br>Details:<br>Name: {location[0]}, Country: "  
 f"{location[1]}, Lat: {centroid[0]}, Long: {centroid[1]}",  
 icon=folium.Icon(color='blue')  
 ).add\_to(map)  
 folium.Marker(  
 [float(location[2]), float(location[3])],  
 popup=f"Centroid ID: {i}, Closest Location<br>Name: {location[0]}, Country: {location[1]}, Lat: "  
 f"{location[2]}, Long: {location[3]}",  
 icon=folium.Icon(color='red')  
 ).add\_to(map)  
  
 map.save(file\_name)  
  
  
def find\_closest\_world\_locations(centroids, world\_locations, capitals, reasonable\_distance\_km=35):  
 *"""  
 Finds the closest world location to each centroid.  
 Returns:  
 A list of tuples containing the centroid, its closest world location, and bombings count.  
 """* closest\_locations = []  
 for centroid, count in centroids:  
 nearest\_capital = min(capitals, key=lambda loc: geodesic(centroid[:2], loc[2:4]).kilometers)  
 distance\_to\_capital = geodesic(centroid[:2], nearest\_capital[2:4]).kilometers  
  
 if distance\_to\_capital <= reasonable\_distance\_km:  
 closest\_location = nearest\_capital  
 location\_details = [closest\_location[1], closest\_location[0], closest\_location[2], closest\_location[3]]  
 else:  
 closest\_location = min(world\_locations,  
 key=lambda loc: np.linalg.norm(centroid[:2] - np.array(  
 [float(loc[19].split(', ')[0]), float(loc[19].split(', ')[1])])))  
 split\_location = closest\_location[19].split(', ')  
 location\_details = [closest\_location[2], closest\_location[7], split\_location[0], split\_location[1]]  
  
 closest\_locations.append((centroid, location\_details, count))  
 return closest\_locations  
  
  
def save\_closest\_locations\_to\_csv(closest\_locations, file\_name):  
 *"""  
 Saves the closest world locations to each centroid to a CSV file.  
  
 Args:  
 closest\_locations: A list of tuples containing the centroid and its closest world location.  
 file\_name: The name of the file to save the data.  
 """* with open(file\_name, mode='w', newline='', encoding='utf-8') as file:  
 writer = csv.writer(file)  
 writer.writerow(['Centroid Latitude', 'Centroid Longitude', 'Name', 'Country Name', 'Closest Location Latitude',  
 'Closest Location Longitude', 'Number of Bombing At Location'])  
 for centroid, location, bombings\_count in closest\_locations:  
 writer.writerow([centroid[0], centroid[1], location[0], location[1], location[2], location[3],  
 bombings\_count])  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()