Los Angeles safe walking

Module code: CMP6202
Module title: Artificial Intelligence and Machine Learning
Coordinator name: Dr. Hossein Ghomeshi

Student name: Zan Zver
Student ID: 18133498
Word count: 3249

Date: 12.12.2021

Table of contents

Table of images

Abstract

Technology is something that is improving our lives on daily basis. One field that can help us with is safety, and that is what this project is about – improving user safety (in Las Angeles) with the help of machine learning.

Introduction, aims and objectives

Introduction

The goal of this research is to join power of machine learning and data to increase safety in the city, in this case the city is Las Angeles. Approaches applied to this can be implemented to any other city, therefore safety of its citizens would have increased.

Aims and objectives

The idea is to train the model on the dataset provided by the local police department, in this case it is Los Angels Police Department (LAPD). With that, model can show you safe routes and “hot spots” on the map so you as a user can avoid potential crime.

Acknowledgement

GitHub

GitHub was used in the time of writing this. Code is still available on it if it needs to be checked. Reason for using GitHub is for source control.
Code that is submitted (to BCU) is mostly the same compared to GitHub. Due to security reasons, API file (creds.py) on GitHub does not contain any credentials but they are included in the code submitted. You can check the GitHub on under (ZanZver/AI_and_ML_assessment, 2021).

API

There are two APIs that are used in the project:

- Openroute service (Openrouteservice, n.d.),
- Geocodio (LLC, D, n.d.)

Openroute service

Openroute service is used for routing. When web GUI is displayed to you with a route between point A and point B, this was done with help of this API. It is free to use, so anybody can use it.
Example of where you could see result of Openroute service:

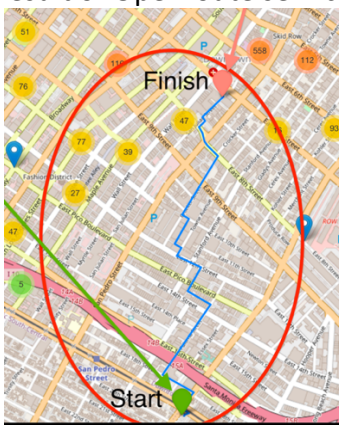

Figure 1: Example of Openroute service

This route was created by Openroute but drawn locally by Folium.
Uses per day:
Standard plan has 2000 free API calls per day (for directions). For testing of this project, academic version was requested and Openroute service approved it. Academic version has 10000 free API calls per day. (Openrouteservice, n.d.)
If you want to recreate this project, standard account is enough. Below is 1 month history of API calls shown.

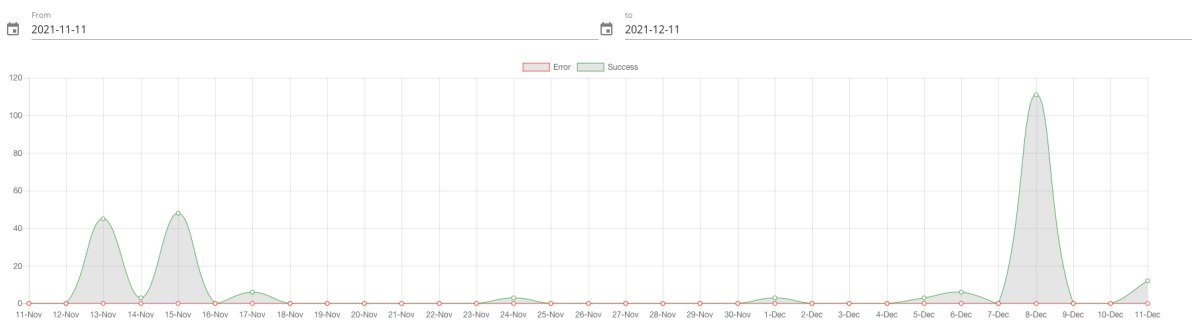

Figure 2: Openroute service API calls

Geocodio

Geocodio is used for translation of location (string) to LAT/LON. When web GUI is displayed, you can enter origin and destination point. Instead of entering coordinates, Geocodio will transform string you have entered to LAT/LON.

Example of where you could see result of Geocodio:

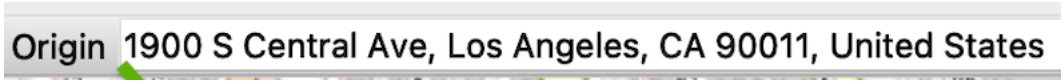

Figure 3: Example of Geocodio

When you enter the address in the app (at the start), it is converted to LAT and LON.

Uses per day:
For this project, plan "Pay as you go" was used. It has 2500 free calls per day. So, in other words, you can use 2500 API calls for free (it is enough) and any additional calls are to be paid. Despite the price of their calls being inexpensive, usage limit was set to block any more calls after 2500. (LLC, D, n.d.)

Here you can see graph of API calls for a December.

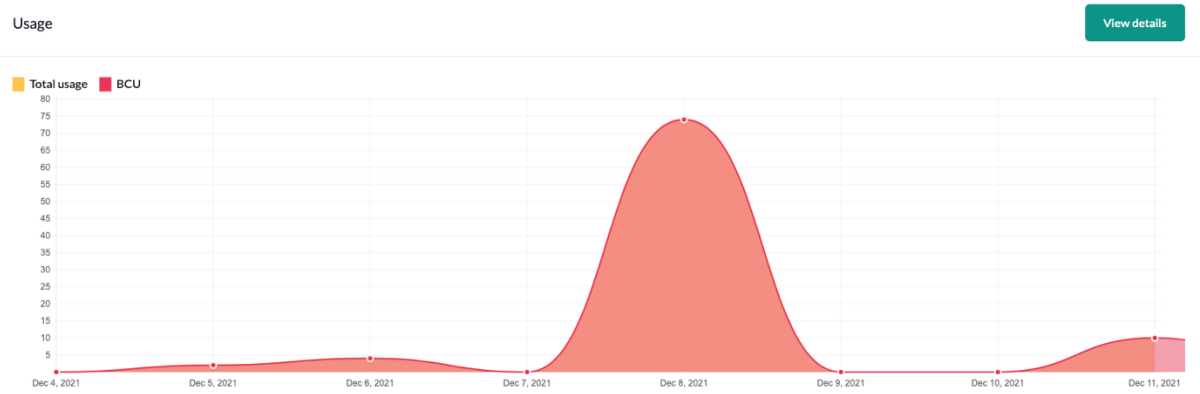

Figure 4: Geocodio API calls

Note

APIs will be unavailable after the 30.1.2021. If you chose to replicate the project after that period of time, you can request both APIs from the original source for free.

Dataset description

About dataset

Dataset that has been selected is called “Los Angeles crime dataset”. Source of dataset is publicly available (Lacity, 2021a). Selected dataset has 2.06M rows and 28 columns (at the time of writing), therefore it is a great candidate for AI and ML tasks due to the size of data. What the project is mostly relying on the dataset is latitude (LAT) and longitude (LON), since it is location based.

Data folder:
In the submitted folder is a subfolder called “Data”. It contains the following datasets:

- Crime_Data_from_2010_to_2019.csv
  - Main dataset from (Lacity, 2021a).
- Crime_Data_from_2010_to_2019-lite.csv
  - Subset of main dataset (only 5k rows), used for testing
- cleanData.csv
  - Same data as in main (or lite) dataset, but cleaned
- Reporting_Districts_data.csv
  - Information on reporting districts of Las Angeles
  - On the site of main data, there is a PDF that is describing reporting districts called “Reporting_Districts_data.csv” (Lacity, 2021a). Unfortunately, it does not provide any data, therefore data is gathered from external source (of same domain) (Socrata, n.d.)
- MO_CODES_Numerical_20191119.json
  - MO codes IDs and descriptions
  - Data is gathered from PDF “UCR-COMPSTAT062618.pdf” available on the site of original data (Lacity, 2016)
- UCR-COMPSTAT062618.json
  - UCR codes IDs and descriptions
  - Data is gathered from PDF “UCR-COMPSTAT062618.pdf” available on the site of original data (Lacity, 2021b)
- PDFs
  - List of PDFs that describe the dataset(s) in further details
- trees
  - After successful run of model prediction, image of the tree is saved there.

Problem to be addressed

There is a lot of crime in LA (NeighborhoodScout, n.d.). Crime does usually happen on the same locations (National Institute of Justice, n.d.) due to bad neighbourhoods (Areavibes, n.d.)or crime preferred locations in general (National Institute of Justice, n.d.).
The dataset used in this scenario (Lacity, 2021a) can help us identify what could happen to us with the help of machine learning.
Machine learning is great on finding patterns (Deeb A.E., 2017), and crimes do happen on the same pattern (National Institute of Justice, n.d.). Therefore, if we connect those two together, we could predict the outcome of the path we are planning to take.
Example:
User is planning to go from the bar to carpark at night. With the help of this program, we could make user aware what could happen to him/her. So, if user wants to take a risk of being robed due to taking shorter path, that is up to him/her, but they can also take a longer path that will result in different crimes happening in that aria (like assault)

This project could be transferred to other locations as well. Police departments in USA do have similar data collection (City-data, n.d.), therefore it could cover other cities with not much more work.

Machine learning model

Summary of the approach

There are 7 Python scripts that are used in the code. Here are the names and descriptions of them:

- main.py
  - Joining everything together
- startup.py
  - Importing pip libraries and CSVs
- data_processing.py
  - Cleaning and processing data
- machine_learning.py
  - Doing ML stuff
- web_app.py
  - Used for creating GUI and displaying HTML content
- creds.py
  - API keys are stored in here
- remove.py
  - Used for creating “sub dataset”. This needs to be triggered manually

How code is executed:

1. main.py is the only file that needs to be executed. It calls startup.py to load all needed dependencies (as “prerequisites”).
2. data_processing.py is called by main.py to clean the data
3. web_app.py is called by main.py to show user interface (UI)
4. web_app.py calls machine_learning.py once user has entered the data

Data pre-processing, visualisation, feature selection

When main.py is called/executed, one of the first things that happens is python creating an instance of clean data.

```
d2 = startup.DP.DataProcessing(startup.mainData)
errorCode, data = d2.getCleanData()
if(int(errorCode) == int(1)):
    print("Error getting the clean data. Program has tried 3 times and failed 3 times. Stopping the program now.")
    sys.exit(1)
```

Figure 5: DataProcessing object creation

In this case, start-up is building data processing object with the main data inside. After the creation of the object, function is called to get the clean data and error code with it.
Once data has been processed, it returns error code of 0 (if none) or 1 if an error has occurred along the way.

Once we are in getClean data function, it is first checked if there is already CSV with that name in there. If it is, it will try to read it and return it (to save time). Note that this is a recursive function. It will try to call itself 3 times (that is the reason for tryCount), if it fails it will return an error code.

```python
def getCleanData(self, tryCount = 0):
        '''
        Input:
            No input is required, tryCount is here for recursive reasons to break out of recursive function
            after 3 attempts
        Return:
            Two things are returned: error code and data (if it is any)
            Error codes: 0 (no errors) and 1 (general error)
        Function:
            Try to return a clean dataset
        '''
        try: # Check if there is already a file there and read it if possible. This helps with speed.
            cleanDataFrame = startup.pd.read_csv("Data/cleanData.csv", index_col=0)
            print("Clean data have been successfully loaded.")
            return(int(0), cleanDataFrame)
        except FileNotFoundError: # If file is not there, attempt to create one
            errorCode, cleanDataFrame, tryCount= self.builder(tryCount) # Call builder class,
            if(int(errorCode) == int(0)):
                return(int(0), cleanDataFrame)
            elif(int(errorCode) == int(1)):
                return(int(1), None)
            elif(int(tryCount) >= int(3)):
                return(int(1), None)
        except startup.pd.errors.EmptyDataError: #remove the file, create new one
            if(int(tryCount) >= int(3)):
                return(int(1), None)
            print("File is empty, removing the file and creating new file.")
            startup.os.remove("Data/cleanData.csv")
            tryCount += 1
            return(self.getCleanData(tryCount))
        except startup.pd.errors.ParserError: #remove the file, create new one
            if(int(tryCount) >= int(3)):
                return(int(1), None)
            print("Error parsing the file, removing the file and creating new file.")
            startup.os.remove("Data/cleanData.csv")
            tryCount += 1
            return(self.getCleanData(tryCount))
        except Exception as e:
            print("Exceptrion: " + str(e))
            return(int(1), None)
```

Figure 6: How data is getting cleaned

Function getData is relying on builder function for CSV creation if it is not there. What builder does is simply creates a clean CSV from scratch. It creates two arrays (one with main data and one that is empty).
After arrays are created, it iterates across full array. Each item from the row is inserted into the “constructor” to be checked if requirements are met and if so, item is appended to the new (empty) list. Based on this, new array is filled.
For every item that fails the “check” in the constructor, error code is returned (from constructor) and row is not appended to new list.
Example of what constructors are most commonly checking and doing:

- Missing values
    - There are some cases where missing values are accepted
- Data types
    - Data types are converted from string to something else (if possible).
    - Example of this is string to integer or 12h time (as a string) to 24h time as a date time type

- Data values
    - One logical example is age: it is checking if age of the victim is between 10 and 150 years. There is not a lot of young people that are victims in the dataset (and the ones that are, are 0 years) and the maximum human life expectancy is 150 years.
    - It is also checking facts from sub datasets (Reporting_Districts_data.csv, MO_CODES_Numerical_20191119.json, UCR-COMPSTAT062618.json) if police data is valid. This is mostly to make sure crime that happened exists and that police district is valid.

```
def builder(self, tryCount):
    '''
    Input:
        tryCount - number of attempts
    Return:
        Error code, data (if any), tryCount (doesn't change in here)
        Error codes: 0 (no errors) and 1 (general error)
    Function:
        Build a clean dataset based with constructors for each row
    '''
    arr = startup.np.stack(( ⋯
    )

    arr2 = [ ⋯

    count = int(0) #
    itemsRemoved = int(0)

    # Iterate across full array if to fill new array
    for item in arr: ⋯


    arr2 = startup.np.delete(arr2, (0), axis=0) # Remove first item from array since it is empty
    print("Items removed: " + str(itemsRemoved)) # Let the user know how may items have been removed
    columnNames = [ ⋯
    ]

    # Create a new datafreame and return it
    df = startup.pd.DataFrame()
    try: ⋯
    except: ⋯

    try: ⋯
    except: ⋯

    try: ⋯
    except: ⋯
```

Figure 7: Example of Builder function

After the data has been cleaned, new object of data visualisation class is created. Class constructor takes in the arguments of clean data.
When the object construction is done, start function is called, which launches the webpage that displays the data.

```
d1 = startup.DV.DataVisualization(
    drNo = data["DR_NO"].values,
    dateRptd = data["Date Rptd"].values,
    dateOcc = data["Date OCC"].values,
    timeOcc = data["Time OCC"].values,
    area = data["Area"].values,
    areaName = data["Area name"].values,
    rptDistNo = data["Rpt Dist No"].values,
    part1or2 = data["Part 1-2"].values,
    crmCd = data["Crm Cd"].values,
    crmCdDesc = data["Crm Cd Desc"].values,
    mocodes = data["Mocodes"].values,
    victAge = data["Vict age"].values,
    victSex = data["Vict sex"].values,
    victDescent = data["Vict descent"].values,
    premisCd = data["Premis Cd"].values,
    premisDesc = data["Premis Desc"].values,
    weaponUsedCd = data["Weapon Used Cd"].values,
    weaponDesc = data["Weapon Desc"].values,
    status = data["Status"].values,
    statusDesc = data["Status Desc"].values,
    crmCd1 = data["Crm Cd 1"].values,
    crmCd2 = data["Crm Cd 2"].values,
    crmCd3 = data["Crm Cd 3"].values,
    crmCd4 = data["Crm Cd 4"].values,
    location = data["Location"].values,
    crossStreet = data["Cross Street"].values,
    locLAT = data["LAT"].values,
    locLON = data["LON"].values
    )

startup.WB.start(d1)

```

Figure 8: DataVisualization object creation

Data is shown with the PyQT5 (TechVidvan, 2021) in the HTML config with markers defining the crime location. If there is a single marker, that means only one thing happened there, but if you zoom in the red circle (group of markers) they would spread out. Red is the "most" and it goes to green as "the least" amount of crime.

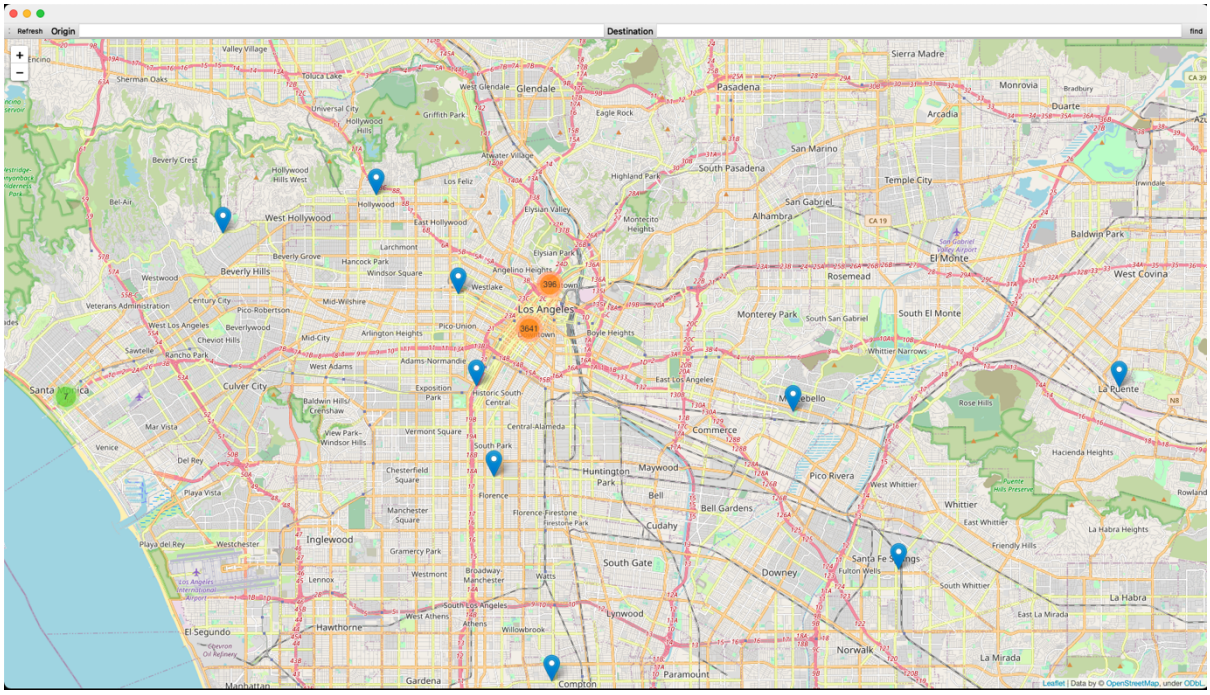

Figure 9: Example of Folium markers showing data

There is an option to put in the origin and destination on top for path search. Green marker is displaying origin on the map and light red marker is displaying destination. Fastest walking route is shown on the map with blue line. When this is displayed, model prediction is showing in the terminal.

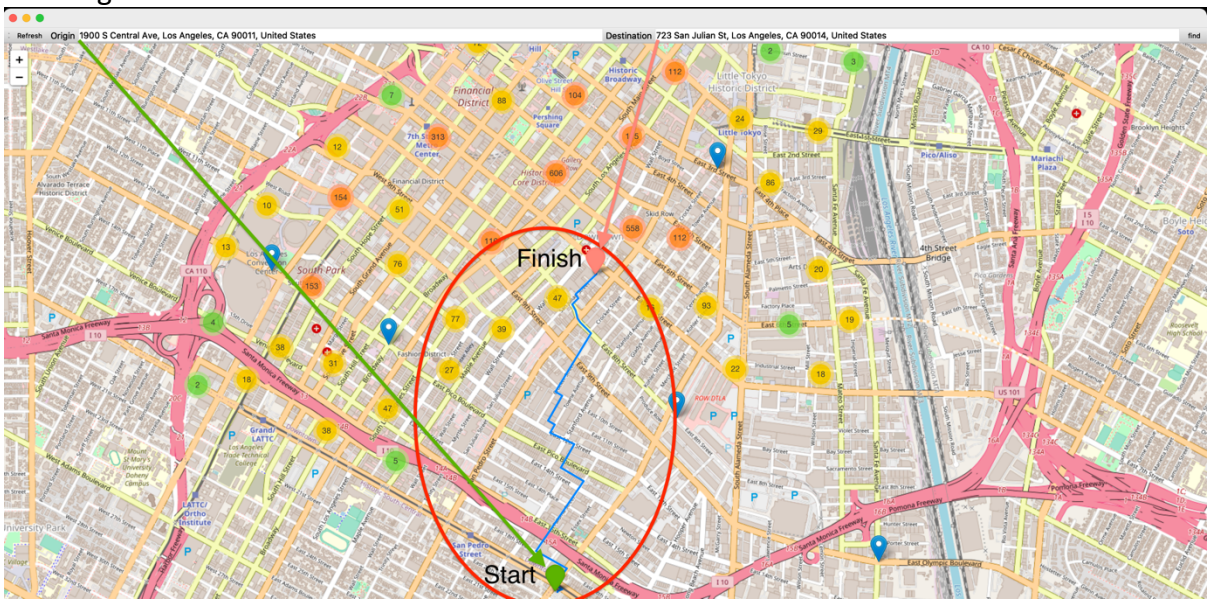

Figure 10: Example of routing from point A to point B

Model training and evaluation

Model algorithm

Model training and evaluation is done in the “machine_learning.py” section of the code. Whole process is done in 4 steps (as img shows):

1. Getting the path data
   Path data is passed to the function. This is in a form of a dataframe and has the complete route where user needs to go if he/she wants to come to destination.
2. Matching data
   LAT and LON from path data are matched with clean dataset. If an instance of LAT and LON has been found, it is then passed to a new dataframe called crimeBasedOnLocation.
3. Location prediction
   Based on the new dataframe (from point 2), model is created for every row and prediction is done with what could happen to you. Variable time now is important, since if a lot of crime is happening at time X and it is currently time Y, it (crime) is not affecting the end user.

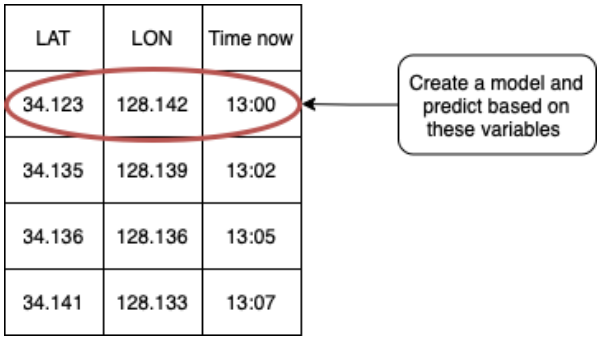

Figure 11: Location prediction variables

4. Grouping data
   In point 3, prediction is done for every location based on time and prediction is added to the dataframe. Now the script will print us the most occurring crimes on that route and what to watch for.

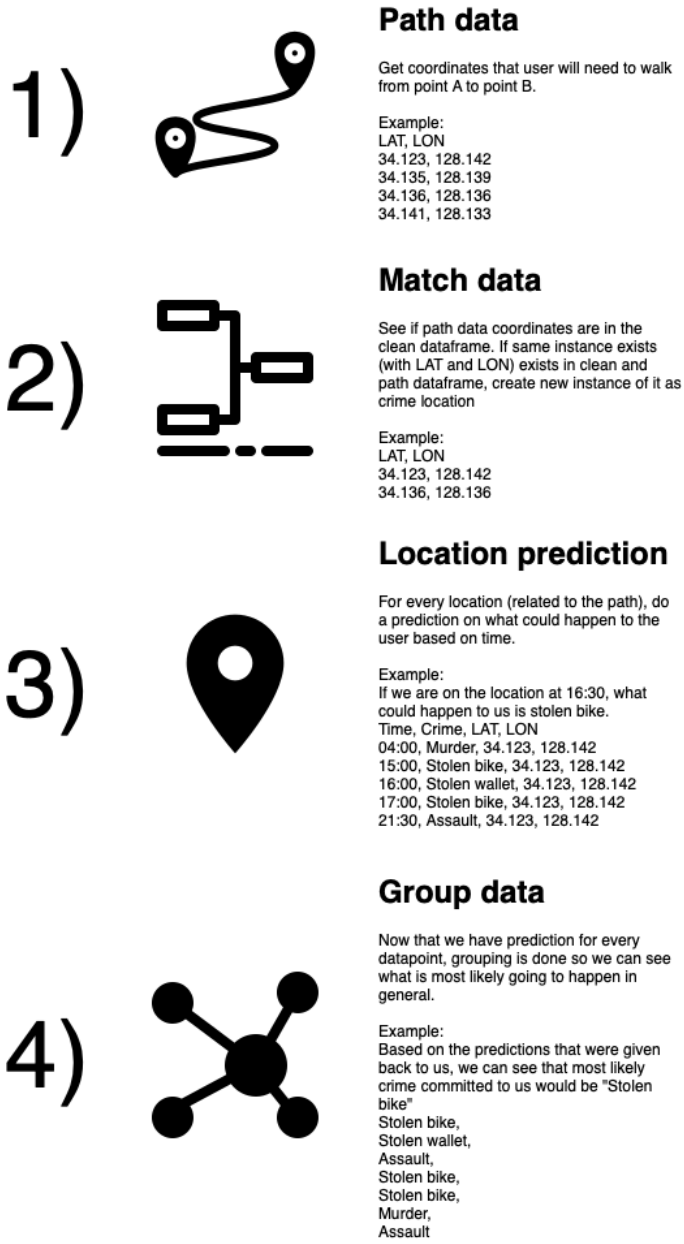

Figure 12: Model algorithm

Model code

In the section above, it is explained how algorithm works. In this section, code will be presented and explained. Note that all of the functions are from “machine_learning.py” script:

I. findCoords

This function is the “main” function of the algorithm since it connects all the things together.

It starts with point 1, which is assigning the given dataset.

```
def findCoords(dataset):
    '''
    Input:
        dataset - path of the route user wants to get from location A to location B
    Return:
        none (at the moment), prediction is printed in the terminal
    Function:
        based on the input dataset, create DF that matches locations in crime DF and do prediction on it
        with what could happen to the user based on LAT, LON and time.now

    Test variables:
    origin: 1900 S Central Ave, Los Angeles, CA 90011, United States
    destination: 723 San Julian St, Los Angeles, CA 90014, United States
    '''

    path = dataset
    cleanDataDf = startup.mainData
```

Figure 13: Showcase of findCoords function

Next step (point 2) is to match the data. Firstly, LAT an LON are "cut down" in length with lambda functions. LAT is limited to 2 tenths and 4 decimal numbers. LON is limited to 4 tenths and 3 decimal numbers.
Reasoning for the difference in our case:

- tenths
  - LAT will always have 2 numbers and LON will always have 3 numbers with a symbol (which takes one more space) in LA (34,-118)
- decimals
  - Testing was done with the LON being 4 decimals and LAT being 3 decimals, and it was accurate (in terms of closest crimes). It was also tested with different parameters and different decimals (e.g.: LON 3 decimals, LAT 3 decimals) and nothing was as accurate as now.

Now that datasets are ready to be used, third dataframe is created. New dataframe takes the values of the clean dataset and it encodes LAT and LON from path dataframe. Encoding is done to check if LAT and LON are in the dataframe. Now based on the encoding, all of the rows that don't have equal LAT and LON are removed.

```
cleanDataDf['LAT'] = cleanDataDf['LAT'].map(lambda x: '%2.4f' % x)
cleanDataDf['LON'] = cleanDataDf['LON'].map(lambda x: '%4.3f' % x)

pathDf = startup.pd.DataFrame(path, columns = ['LON','LAT'])
pathDf['LAT'] = pathDf['LAT'].map(lambda x: '%2.4f' % x)
pathDf['LON'] = pathDf['LON'].map(lambda x: '%4.3f' % x)

crimeBasedLocationPath = cleanDataDf.assign(IncleanDataDfLAT=cleanDataDf.LAT.isin(.LAT).astype(bool))
crimeBasedLocationPath = crimeBasedLocationPath.assign(IncleanDataDfLON=cleanDataDf.LON.isin(pathDf.LON).astype(bool))

crimeBasedLocationPath = ((crimeBasedLocationPath.loc[(crimeBasedLocationPath['IncleanDataDfLAT'] == True)
                                                       & (crimeBasedLocationPath['IncleanDataDfLON'] == True)]))
```

Figure 14: How data is matched

With new dataset, script iterates across the rows (point 3) and creates model (with LAT and LON). Model is then asked to do a prediction (random forest in our case) and prediction is appended to a dataframe.

```
for item, row in crimeBasedLocationPath.iterrows():
    print("Processing item: " + str(processedItem) + "/" + str(len(crimeBasedLocationPath)))
    X_train,X_test,y_train,y_test,predModel = splitData(row[26], row[27]) #add predDf
    df2 = df2.append(trainRandomForest(X_train,X_test,y_train,y_test,predModel,processedItem))
    processedItem += 1
```

Figure 15: Call splitData and trainRandomForest

With predictions done, we reach the final stage of the function (point 4), and this is results of the model. Script prints out top 5 most occurring items in the dataframe.

```
print("==================")
print("Prediction:")
print(df2['Desc'].value_counts()[:5].index.tolist())
print("==================")
```

Figure 16: Print out model prediction

II. splitData

Function accepts two inputs (LAT, LON – fist point of the route). At the start of the function, it gets the main data and then drops some columns from dataframe that are not needed.

```
def splitData(LAT, LON):
    '''
    Input:
        LAT and LON
    Return:
        X_train, X_test, y_train, y_test and encoded time now (as dfCurrentTime)
    Function:
        Creates a model based on startup.mainData, values that are inside are LAT, LON, time and crime ID (so what could happen to you)
    '''
    df = startup.mainData
    df = df.drop_duplicates()

    del df['DR_NO']
    del df['Date Rptd']
    del df['Date OCC']
    del df['Area name']
    del df['Crm Cd Desc']
    del df['Mocodes']
    del df['Vict sex']
    del df['Vict descent']
    del df['Premis Desc']
    del df['Weapon Desc']
    del df['Status']
    del df['Status Desc']
    del df['Crm Cd 1']
    del df['Crm Cd 2']
    del df['Crm Cd 3']
    del df['Crm Cd 4']
    del df['Location']
    del df['Cross Street']
```

Figure 17: splitData function

Next, function creates a new empty dataframe with same labels as the one above. This dataframe has current time inside of it. The reason for this is that current time can now be encoded (Pluralsight n.d.) with the model data and then separated (so results are not affected).

```
now = startup.datetime.now()

# df2 is just encoding of the time now (so it can be later used), df is actual model encoding. Due to time / performance efficiency,
# time is encoded together (so time now and other time), but it is later split
dfCurrentTime = startup.pd.DataFrame(startup.np.insert(df.values, 0, values=[now.strftime('%H:%M:%S'),
None, None,None,None,None,None,None,LAT,LON], axis=0)) # Labels in the df: Time OCC,Area,Rpt Dist No,Part 1-2,Crm Cd,Vict age,Weapon Used Cd,LAT,LON
```

Figure 18: Dataframe with current time

After now, two dataframes are merged together (dfCurrentTime is first row of df) and binary encoding is done for the time. Hours are spitted to 0-23 segments and minutes are spitted to 5 min intervals.

```
dfCurrentTime.columns = df.columns
df = dfCurrentTime

# Encode the time (binary)
df['Time OCC'] = startup.pd.to_datetime(df['Time OCC'])
df['Hour OCC'] = startup.pd.DatetimeIndex(df['Time OCC']).hour
df['Minute OCC'] = startup.pd.DatetimeIndex(df['Time OCC']).minute
df['Second OCC'] = startup.pd.DatetimeIndex(df['Time OCC']).second

df = df.drop(['Time OCC'], axis = 1) # Time is now encoded, so label "Time OCC" can be dropped
df = startup.pd.get_dummies(df, columns=['Hour OCC'], drop_first=False, prefix='Hour')
df = startup.pd.get_dummies(df, columns=['Minute OCC'], drop_first=False, prefix='Minute')

'''
How time is encoded:
Hour:
    24 new columns are created (name 0-23), so if the hour is 15, column with 15 will have 1 in it and other columns will have 0
Minute:
    60 new columns are created (name 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55). If the minute is 42, column 40 will have 1 in it, others will have 0
Second:
    60 new columns are created (name 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55). If the second is 42, column 40 will have 1 in it, others will have 0
'''
```

Figure 19: Binary encoding

After the encoding, current time is separated again from main data. “Crm Cd” (which means crime id number) is selected as a label that will be predicted from df dataframe.

```
dfCurrentTime = df.head(1) # Get the time now encoded
df = df.iloc[1:] # Drop the first row, it is not needed

y = df["Crm Cd"] # Have crime Cd (ID num) as predicted number output
```

Figure 20: Split dataframe

Some of the columns are not needed for this prediction, therefore they can be removed. The reason why they are removed at the end is for modularity. If something (let's say "Area") is interesting later on, it can quickly be modified here. But note, that all modifications that are made do need to be the same between two dataframes.

```
# Drop other unnecessary data
del df['Area']
del df['Rpt Dist No']
del df['Part 1-2']
del df['Crm Cd']
del df['Vict age']
del df['Premis Cd']
del df['Weapon Used Cd']
x = df # Save model to x

# Drop the same data as in df
del dfCurrentTime['Area']
del dfCurrentTime['Rpt Dist No']
del dfCurrentTime['Part 1-2']
del dfCurrentTime['Crm Cd']
del dfCurrentTime['Vict age']
del dfCurrentTime['Premis Cd']
del dfCurrentTime['Weapon Used Cd']
```

Figure 21: Remove columns

At the end, spit is done. 80% of the data is set to training and 20% is set to testing. This is also what it gets returned from the function.

```
X_train,X_test,y_train,y_test=startup.train_test_split(x,y,test_size=0.2) # Split to to 80% train and 20% test
return (X_train,X_test,y_train,y_test,dfCurrentTime) # Return test, train and encoded time now
```

Figure 22: Train, test split

III. trainDecisionTree

Decision tree was one of the first algorithms that was implemented as a test case. It is still in the code but results that are provided from it are not as effective as from random forest. This is due to random tree forest being better in general compared to decision tree. Just note that it is mostly a “placeholder” in the code, that meaning it will not be called, but it can be easily added back if needed.

```
def trainDecisionTree(X_train,X_test,y_train,y_test):
    dtree = startup.DecisionTreeRegressor(max_depth=8, min_samples_leaf=0.13, random_state=3)
    dtree.fit(X_train, y_train)

    startup.DecisionTreeRegressor(criterion='mse', max_depth=8, max_features=None,
            max_leaf_nodes=None, min_impurity_decrease=0.0,
            min_impurity_split=None, min_samples_leaf=0.13,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
                random_state=3, splitter='best')

    # Fit the regression tree 'dtree1' and 'dtree2'
    dtree1 = startup.DecisionTreeRegressor(max_depth=2)
    dtree2 = startup.DecisionTreeRegressor(max_depth=5)
    dtree1.fit(X_train, y_train)
    dtree2.fit(X_train, y_train)

    # Predict on training data
    tr1 = dtree1.predict(X_train)
    tr2 = dtree2.predict(X_train)

    # Predict on testing data
    y1 = dtree1.predict(X_test)
    y2 = dtree2.predict(X_test)

    print("+++++++++++++++")
    print("DecisionTree results train model:")
    print("Sqrt: " + str(startup.np.sqrt(startup.mean_squared_error(y_train,tr1))))
    print("R2 score: " + str(startup.r2_score(y_train, tr1)))

    # Print RMSE and R-squared value for regression tree 'dtree1' on testing data
    print("Results results test model:")
    print("Sqrt: " + str(startup.np.sqrt(startup.mean_squared_error(y_test,y1))))
    print("R2 score: " + str(startup.r2_score(y_test, y1)))
```

Figure 23: Example of decision tree

IV. trainRandomForest

As noted from above, random tree forest has better (in general) performance (Sharma A., 2020) therefore it was chosen for this task. When we get into the function, model is created with Scikit (Scikit-learn, 2018) random forest algorithm. For this case, there n_estimators (number of trees) are set to 5000, oob is set to True since this can give us score of the model later on and number of bootstrapping samples (random_state) is set to 100 (Pluralsight n.d.)
Then model is fitted with training data (X and y) that were passed into the function.

```
def trainRandomForest(X_train,X_test,y_train,y_test,predictDataset):
    '''
    Input:
        X_train,X_test,y_train,y_test,predictDataset (LAT, LON, time now encoded)
    Return:
        array of top 5 situations / crimes that you could encounter at that location (LAT, LON) now
    Function:
        it creates a model of random forest, train it and predict it
    '''
    model_rf = startup.RandomForestRegressor(n_estimators=5000, oob_score=True, random_state=100)
    model_rf.fit(X_train, y_train)
```

Figure 24: Random forest model creation

When model is created, prediction is done with prediction dataset. This is the dataset that has current time (encoded), LAT and LON of the route.

```
pred = (model_rf.predict(predictDataset.values))
```

Figure 25: Model prediction

For each run, R2 score, and square root are printed. With this, you can see how well the model did. More information about this can be found under results section

```
print("+++++++++++++++")
print("RandomForest results train model:")
pred_train_rf = model_rf.predict(X_train)
print("Sqrt: " + str(startup.np.sqrt(startup.mean_squared_error(y_train, pred_train_rf))))
print("R2 score: " + str(startup.r2_score(y_train, pred_train_rf)))
print("Results test model:")
pred_test_rf = model_rf.predict(X_test)
print("Sqrt: " + str(startup.np.sqrt(startup.mean_squared_error(y_test, pred_test_rf))))
print("R2 score: " + str(startup.r2_score(y_test, pred_test_rf)))
print("+++++++++++++++")
```

Figure 26: Model performance calculation

At the end of the function, local instance of UCR dataframe is created. This dataframe is used to translate UCR predicted codes to valid strings that are encoded in the UCR file. At the end, top 5 most occurring crimes are selected and returned.

```
ucrDF = startup.pd.DataFrame(startup.ucrData) #get the UCR codes from main
retArr = (ucrDF.iloc[(ucrDF['ID']-pred).abs().argsort()[:5]]) #translate UCR codes from prediction and get top 5 occuring
return retArr
```

Figure 27: Return results

Results and discussion

As noted in the section above, the terminal returns the prediction at the end. In the prediction (as seen in the image bellow), the listed items are the ones occurring the most in the dataframe. From the example, out of top 5 things that could happen to us, “Stolen Vehicle” is the most likely thing.

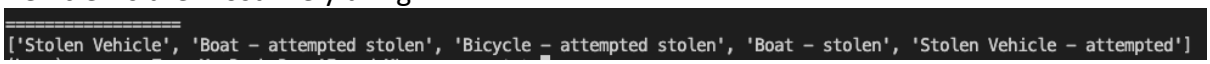

Figure 28: Example of prediction

The image above notes the Sqrt (as mean squared error) and R2 score. Train model scores are reasonable, with 98 and 79%, but test score is something that is out of the charts. This is something, that stands out and needs to be double checked since it seems wrong. After changing dataset split from 80-20 to even 60-40, changing number of trees, and random state as well, this problem was present at all times. Due to this, please note that model is not 100% tuned. At the end, it could be that data needs some further processing or it could just be a typo in the code.

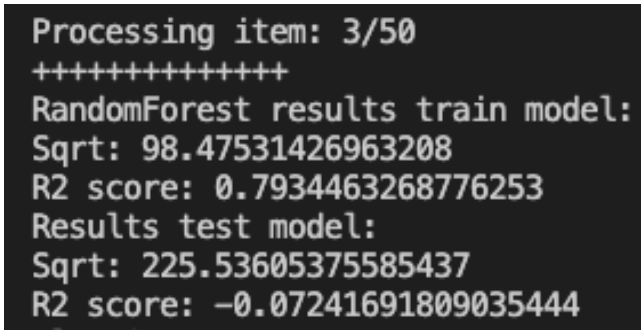

Figure 29: train/test results

As noted in the model section, all of the results are printed in this section of trainRandomForest function.

```
1  print("++++++++++++++")
2  print("RandomForest results train model:")
3  pred_train_rf = model_rf.predict(X_train)
4  print("Sqrt: " + str(startup.np.sqrt(startup.mean_squared_error(y_train, pred_train_rf))))
5  print("R2 score: " + str(startup.r2_score(y_train, pred_train_rf)))
6  print("Results test model:")
7  pred_test_rf = model_rf.predict(X_test)
8  print("Sqrt: " + str(startup.np.sqrt(startup.mean_squared_error(y_test, pred_test_rf))))
9  print("R2 score: " + str(startup.r2_score(y_test, pred_test_rf)))
```

Figure 30: Model performance calculation v2

For every model that gets generated, a tree is going to be saved to your local directory (under Data/trees). The submitted file is going to contain 2 example trees to look at, but if you wish to have a look at more, execute the code and trees shall be generated as .dot file and .png as well.

```
try:
    print("Plotting tree"+str(processedItem))
    filename = str('Data/trees/tree' + str(processedItem) + '.dot')
    startup.export_graphviz(model_rf.estimators_[0],
                    feature_names=X_train.columns,
                    filled=True,
                    rounded=True,
                    out_file=filename)
    startup.graphviz.render('dot', 'png', filename)
except Exception as e:
    print("Error plotting tree"+str(processedItem))
    print(e)
```

Figure 31: Creation of tree image

Here is an example of “tree1.dot.png” (it is not readable in Word). When you open the file, you can see the whole vector image and how the decisions were made.

Figure 32: tree1.dot.png example

After the code has finished creating all the trees, size of the folder can be quite large. Figure 33 is an example of 50 trees in a folder.

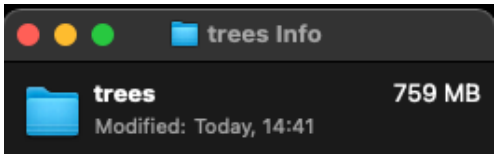

Figure 33: Example of 50 trees - folder size

Conclusion, recommendations and future work

Conclusion

At the end, user has a way of determining if a path is safe or not for him. Machine learning (or technology in general) can help with safety of individuals without spying, therefore this seems like a great example of something that is easy and functional for the user.

Recommendations and Future work

This project has potential to be expended and used in any kind of city (that has data to back it up). Due to that, viewing this report as an alpha stage of the project could open the possibilities for expansion.
Here are some of the big tasks that could drastically improve the project:

- Client-Server environment
    - At the moment all the processing is done on the client's machine. Performance would be increase if client could just send out an API request to a pretrained model for better result.
- Multi route
    - At the moment, GUI is only displaying what is the fastest route. It would be convenient to have multiple routes based on crime prediction. With this, user could simply choose what is best for him/her.
- Different routing
    - At the moment, routing is only done for walking. This could be expended in the future with adding support for cars or combination it.
      Example of this would be:
      User wants to go from point A to point B (with car). Based on the available parking spots and crime committed around that, user could choose where to park the car.
- Grouping based on location
    - At the moment, only one city (LA) is represented. It does have quite a number of crimes, therefore looking on the map can be a bit "scarry". One way to counter it would be to show crime only around users' radios (based on his/her location). Once path is shown, crime around it can be displayed as well.
- Google maps
    - This project could be integrated or working with a popular map platform such as Google maps. A lot of users do have Google maps already installed; therefore, it would be just a bonus feature they can use.

References

Areavibes (n.d.) Know Your Neighbourhood: Crime Statistics by City [online]
Available at: https://www.areavibes.com/library/neighborhood-crime-statistics/
[Accessed 12 Dec. 2021].

City-data (n.d.) Crime in New York, New York (NY): crime map [online]
Available at: https://www.city-data.com/crime/crime-New-York-New-York.html
[Accessed 12 Dec. 2021].

Deeb, A.E. (2017) So what is Machine Learning good for anyway? [online] Rants on Machine Learning
Available at: https://medium.com/rants-on-machine-learning/so-what-is-machine-learning-good-for-anyway-3cb38621383f
[Accessed 12 Dec. 2021].

Lacity (2016) MO_CODES_Numerical_20191119 [online]
Available at: https://data.lacity.org/api/views/63jg-8b9z/files/e14442b9-a6b8-4531-83f3-f7ba980b1377?download=true&filename=MO_CODES_Numerical_20191119.pdf
[Accessed 12 Dec. 2021].

Lacity (2021a) Crime Data from 2010 to 2019 | Los Angeles - Open Data Portal [online]
Available at: https://data.lacity.org/Public-Safety/Crime-Data-from-2010-to-2019/63jg-8b9z
[Accessed 12 Dec. 2021].

Lacity (2021b) UCR-COMPSTAT062618 [online]
Available at: https://data.lacity.org/api/views/63jg-8b9z/files/fff2caac-94b0-4ae5-9ca5-d235b19e3c44?download=true&filename=UCR-COMPSTAT062618.pdf
[Accessed 12 Dec. 2021].

LLC, D. (n.d.) Geocodio [online] Geocodio
Available at: https://www.geocod.io
[Accessed 12 Dec. 2021].

National Institute of Justice (n.d.) Why Crimes Occur in Hot Spots [online]
Available at: https://nij.ojp.gov/topics/articles/why-crimes-occur-hot-spots
[Accessed 12 Dec. 2021].

NeighborhoodScout (n.d.) Los Angeles Crime Rates and Statistics - NeighborhoodScout [online]
Available at: https://www.neighborhoodscout.com/ca/los-angeles/crime
[Accessed 12 Dec. 2021].

Openrouteservice (n.d.) Openrouteservice [online] Openroute service
Available at: https://openrouteservice.org
[Accessed 12 Dec. 2021].

Pluralsight (n.d.). Machine Learning with Time Series Data in Python | Pluralsight. [online]
Available at: https://www.pluralsight.com/guides/machine-learning-for-time-series-data-in-python
[Accessed 12 Dec. 2021].

Scikit-learn (2018) 3.2.4.3.2. sklearn.ensemble.RandomForestRegressor - scikit-learn 0.20.3 documentation [online]
Available at: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
[Accessed 12 Dec. 2021].

Sharma, A. (2020) Decision Tree vs. Random Forest - Which Algorithm Should you Use? [online] Analytics Vidhya
Available at: https://www.analyticsvidhya.com/blog/2020/05/decision-tree-vs-random-forest-algorithm/
[Accessed 12 Dec. 2021]

Socrata (n.d.) Reporting Districts | LAC Open Data [online]
Available at: https://data.lacounty.gov/GIS-Data/Reporting-Districts/kvwy-dqs6
[Accessed 12 Dec. 2021].

TechVidvan. (2021). How to Create a Web Browser with Python and PyQT. [online]
Available at: https://techvidvan.com/tutorials/create-web-browser-python-pyqt/
[Accessed 12 Dec. 2021].

Zver, Z. (2021) ZanZver/AI_and_ML_assessment [online] GitHub.
Available at: https://github.com/ZanZver/AI_and_ML_assessment
[Accessed 12 Dec. 2021].