Eliminating GPS Uncertainty

Rushill Shah[1]

Abstract— This research has been conducted to create a filtration algorithm in order to remove outliers from a given set of GPS data by creating parameters and using mathematical and logistic regression. To a small extent, trial and error too has been used, in order to ensure that the system is functional for all GPS points in the JSON format. Other methods discussed in this Python (3.6)-based research report involve usage of Google's 'SnapToRoads' API and curve smoothing in order to better eliminate outliers.

Index Terms— API, GPS, GPS outliers, Outliers, Python, Regression, Smoothing, Standard Deviation, Uncertainty

1 Introduction

During a two-week internship with a tech start-up that provides driving behavior analytics and fleet management through software to businesses, working on GPS based systems used as part of the software, this research was conducted.

A part of research was looking into the uncertainty of GPS data provided by the phone, a primary source of location data for the company. GPS points often showed the location of the signal source to be miles away from the actual position.

In an attempt to eliminate such uncertainty and make the software more reliable, the algorithm created in this project uses mathematical and logical means to refine GPS data. This serves to augment pre-existing systems for elimination of outliers by adding a three-tier filtration system. The project builds upon and uses pre-existing systems as well as creates its own that are in accordance to the many unique factors of the software such as frequency of input, sending data and location which will be further discussed.

2 Aim

To build a basic filtering algorithm based on probability and logistic regression to detect and remove outliers from a given set of GPS points.

3 Background

3.1 Background Information

- **Data Set:** Although this program will work for every set of GPS points in a tuple format, the sample data that will be used is as in Appendix **Section 1**.
- **Iterations:** Using python 'for loops' this program involves the use of an iterative algorithm for practical use. All the code can be found in Appendix **Section 2.**
- **Data Format:** The data received by the software is stored in the JSON[i] format. JSON[ii] (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for human interpretation. It is easy for machines to parse and generate. It's universal and the de facto data format of the Web 2.0. In this case, it is presented in an array data type. Additionally, the data in this research is extracted from a tuple.
- **Distance (Haversine formula):** This formula is used to calculate the distance between two points.

A = sin({difference between lat}/2)**2+cos(lat1)*cos(lat2)*sin({ difference between lon}/ 2)**2

C = 2 * atan2 (sqrt(a), sqrt(1 - a))

Distance = R * C * 1000 [R = radius of earth = 6373] (1)

- **Time:** Time is in the epoch time format or ''timestamp'', which shows the current time as the number of milliseconds from January 1, 1970. For instance, 6:30 pm on July 2, 2017 would be 1498977000 seconds in the timestamp format.
- **Snap To Roads API:** The Google Snap To Roads API works by interpolating fed in values with data in its registry, 'snapping' or corresponding them. It plots the input to a road and eliminates the outlier points.

3.2 Initial Calculations

At the beginning, the JSON file must be imported from the root directory and queried. Only after this can the data in the file by used.

Using aforementioned Haversine formula, after importing the required functions or using NumPy, the distance between the two points is calculated between a pair of GPS points, for all the given data points in the array.

Additionally, the program extracts the time in seconds from the given 'timestamp', in order to use for the velocity calculations. The following code can be used:

SecondsTime = (EpochTime1 - EpochTime2)/1000 (2)
SetTime.append(SecondsTime)

Velocity is calculated by using the formula v = d/t and iterat-

[1] Rushill Shah is an 11th grade student at the Dhirubhai Ambani School in Mumbai

ing the process for each coordinate in the array.

For a greater range of data to work with, the program also calculates the average distance and velocity.

AverageVelocity = TotalDistance/ TotalTime (3)
AverageDistance = TotalDistance/Recordings

4 Elimination Process

The method used in this project is threefold-

- Standard deviation
- Curve Smoothing
- API

4.1 Standard Deviation

Standard Deviation of a set of data is a measure of how widely spread the data points in a set of data are. It is used to determine whether values are in or out of range, or in essence, the deviation of a data point from the mean.
Using the 'std' function in NumPy, the standard deviation is calculated for velocity and distance. As according to the norm, the deviation from average is calculated for each element of the array. If the element is at a greater deviation than the standard deviation, it is appended to an outlier set.

Sample Code for Standard Deviation

```
Std= sqrt (AvDist*(abs(distance -(distance*AvDist))**2))
stdVel = sqrt(AvVel*(abs(Vel-(Vel*AvVel))**2))
```
(4)

Since the distance and velocity tend to vary largely in a trip in India due to traffic, traffic lights, possible accidents, slopes, speed bumps and so on, the standard deviation is not a very reliable way to eliminate outliers in this context unless accompanied by support systems.
The values that have a greater deviation from the mean than the standard deviation are appended to an Outlier array, while the indices of the rest are appended to the 'In' array. These indices will be used towards the end of the program to acquire the final outliers

Sample code for indices

```
SetTrue = []
ValSet = [1343, 555, 3342]
if True:
  SetTrue.append(Valset.index)
```
(5)

In this block of code SetTrue is a set of values that are not outliers

The output on running the code is given in Appendix **Section 3**.


```
{
  "snappedPoints": [
    {
      "location": {
        "latitude": 12.919082345679861,
        "longitude": 77.651684714045615
      },
      "originalIndex": 0,
      "placeId": "ChIJ6yP7JIIUrjsRpHSPhEcWRHc"
    },
    {
      "location": {
        "latitude": 12.918915069015311,
        "longitude": 77.651690053806533
      },
      "originalIndex": 1,
      "placeId": "ChIJ6yP7JIIUrjsRpHSPhEcWRHc"
    },
    {
      "location": {
        "latitude": 12.91865394531316,
        "longitude": 77.651699184979108
```


Fig.1. The portion webage of the SnapToRoads API once the request has been sent.

4.2 Curve Smoothing

Curve smoothing, or Data Smoothing is a method used to remove uncertain or outlying data points involving the use of an algorithm to remove noise from a data set, allowing important patterns to stand out.
The algorithm consists of the creation of an approximate function by removal of any data points that extend far too beyond the derivative or trend. It involves the use of trial and error as well as prior results from the software.
Another method that can be used is based on the derivative, in a manner similar to the difference method used above. Parameters for what magnitude of difference between derivatives gives an outlier too can be set using precedents and trial and

error, which have not been conducted in this project since it was a secondary method.

Curve Smoothing has been done on velocity values solely for the reason that latitude and longitude values cannot be smoothed without losing actual data from the set.

Moreover, smoothing cannot be done for the first two and last three terms of a set, due to the absence of adjacent data points Moreover, smoothing cannot be done for the first two and last three terms of a set, due to the absence of adjacent data points. These have been assumed to be true and added to the set nonetheless. Here too an index set has been created that stores the indices of the valid elements of the data set.

Sample Code for Curve Smoothing:

SetVel, OutVel are predefined sets of velocity and velocity outliers between 2 points

```
OutSmooth = []
```

(6)

```
IndexOut = []
IndSmooth = []
length = len(SetVel)
for i in range(2, length-3):
   diffnext = abs(SetVel[i+2])- (SetVel[i+1])
   diffprev = abs(SetVel[i-1])-(SetVel[i-2])
   diff1 = abs(SetVel[i]-SetVel[i-1])
   diff2 = abs(SetVel[i+1]-SetVel[i])
   if diff1 > 1.5*diffprev or diff2 > 1.5*diffnext:
      IndSmooth.append(SetVel.index(SetVel[i]))
   else:
      OutSmooth.append(SetVel.index(SetVel[i]))

for i in range (0, 2):
   IndSmooth.append(i)
for i in range (24, length):
   IndSmooth.append(i)
```

While the sample code accomplishes its purpose and it explanatory, a more efficient block is used in the actual code in Appendix **Section 1** .The output for the code can be found in appendix **Section 4**.

4.3 SnapToRoads API

The Google SnapToRoads API applies a set of data to Google's existing registry and matches it to a road. The API interpolates data into the roads it possesses in its registry and returns back the points that are closest to its data points.

From Python, this program makes an HTTP request and sends data to the API, and on receipt adds the outliers indicated by the API to a separate array. An HTTP request can be made in python by using the request.() command.

Sample Code for SnapToRoads request:

(7)

```
results = requests.get("API Key")
snappoints = results.json()['snappedPoints']
myset = set()
for point in snappoints:
   snapdata.add((point['location']['latitude'],
   point['location']['longitude']))
print(snapdata)
```

The code here extracts the original index from the webpage. This index is the index of the data point, which was sent with the request, and has been confirmed by the API to be valid. The 'originalIndex' data is requested and then added to a different array.

This is the third array of indices that will be used to get the final result through intersection.

As visible in the Figure, the originalIndex gives us the indices of the accepted values in the data set sent with the request, after having snapped it. The 'latitude' and 'longitude' values show us to what data point has the request been snapped to.

5 Final Outliers

Final Outliers:

Now that the program has arrays of outliers using three different methods, we can use it to filter out unreliable values. The program finally makes a set of outliers that is an intersection of all four sets (standard deviation of distance and velocity, Smoothing and SnapToRoads), thus excluding values most likely to be outliers and accounting for uncertainty in all three methods.

However, the final output given by the set of code does not provide the values in the format of the input, but provides it in the form of a sub-array tuple, which can be easily converted to other formats.

Sample code for the intersection:

(8)

```
Set1 = [1, 2, 3]
Set2 = [4, 5, 6]
SetIntersect = list(set(Set1) & set(Set2))
Print (SetIntersect)
OUT: [1, 2, 3, 4, 5, 6]
```

The final set of outliers can be found in **Section 5** of the Appendix

6 Conclusion

Through this research, a lot was learned about iterations, python features, and regression. A number of mathematical methods used also served to teach me about uncertainty elimination in all measurements. Above all, this research taught me about APIs, making http requests, and methods such as curve smoothing and standard deviation, skills I view as important with regards to my future career.

Thus with the use of mathematical and logical regression, API, and general trial and error a reliable filtration system has been created for the exclusion of unreliable values.

7 Acknowledgments

I would like to thank Mr. Shivalik Sen, CEO of Vahanalytics for allowing me to work as an intern and conduct research with the company, and guiding me through the challenging steps of this research.

8 Appendix

8.1 Section 1

[

{
"timestamp": 1499159501922,
"coordinates": [
12.919082641601562,
77.65169525146484
]
},
{
"timestamp": 1499159503930,
"coordinates": [
12.918915748596191,
77.6517105102539
]
},
{
"timestamp": 1499159506936,
"coordinates": [
12.918656349182129,
77.65177154541016
]
},
{
"timestamp": 1499159509002,
"coordinates": [
12.918524742126465,
77.6517562866211
]
},
{
"timestamp": 1499159511984,
"coordinates": [
12.918295860290527,
77.65178680419922
]
},
{
"timestamp": 1499159513206,
"coordinates": [
12.918216705322266,
77.65177154541016
]
},
{
"timestamp": 1499159515914,
"coordinates": [
12.918027877807617,
77.65178680419922
]
},
{
"timestamp": 1499159517944,
"coordinates": [
12.917914390563965,
77.65178680419922
]
},
{
"timestamp": 1499159520903,
"coordinates": [
12.917774200439453,
77.65178680419922
]
},
{
"timestamp": 1499159522917,
"coordinates": [
12.917659759521484,
77.65179443359375
]
},
{
"timestamp": 1499159524954,
"coordinates": [
12.917553901672363,
77.65180969238281
]
},
{
"timestamp": 1499159526928,
"coordinates": [
12.917448043823242,
77.6518325805664
]
},
{
"timestamp": 1499159529917,
"coordinates": [
12.917227745056152,
77.65177917480469
]
},
{
"timestamp": 1499159532923,
"coordinates": [
12.91706657409668,
77.65178680419922
]
},
{
"timestamp": 1499159534932,
"coordinates": [
12.916943550109863,
77.65178680419922
]
},
{
"timestamp": 1499159537974,
"coordinates": [
12.916749000549316,
77.65178680419922
]
},
{
"timestamp": 1499159539937,

"coordinates": [
12.916621208190918,
77.65179443359375
]
},
{
"timestamp": 1499159541962,
"coordinates": [
12.91647720336914,
77.65180206298828
]
},
{
"timestamp": 1499159542240,
"coordinates": [
12.91647720336914,
77.65180206298828
]
},
{
"timestamp": 1499159544921,
"coordinates": [
12.916269302368164,
77.65177154541016
]
},
{
"timestamp": 1499159546928,
"coordinates": [
12.916149139404297,
77.65178680419922
]
},
{
"timestamp": 1499159548959,
"coordinates": [
12.916014671325684,
77.65177917480469
]
}
]

8.2 Section 2

```
import requests

import json

with open('strings.json') as data_file:
    data = json.load(data_file)

from math import sin, cos, sqrt, atan2, radians

R = 6373.0  # approximate radius of earth in km
SetDist = []
api = []
SetDat =[]
SetTime=[]
SetVel = []
length = len(data)
for i in range(0, length-1):
    dataPoint = data[i]
    dataPoint1 = data [i+1]
    coordinate = dataPoint['coordinates']
    coordinate1 = dataPoint1['coordinates']
    SetDat.append(coordinate)
    x = coordinate[0]
    y = coordinate[1]
    x1 = coordinate1[0]
    y1 = coordinate1[1]
    import math
    lat1 = math.cos(math.radians(x)) #converts degrees of long
or latitude into rads
    lon1 = math.cos(math.radians(y))
    lat2 = math.cos(math.radians(x1))
    lon2 = math.cos(math.radians(y1))
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat / 2)**2 +  cos(lat1) * cos(lat2) * sin(dlon / 2)**2  #
formula for distance
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    distance = R * c * 1000
    Time = dataPoint['timestamp']
    Time = dataPoint['timestamp']
    TimePoint = dataPoint['timestamp']
    TimePoint1 = dataPoint1["timestamp"]
    SecTime = (TimePoint1 - TimePoint)/1000
    SetTime.append(SecTime)
    print("The time between the 2 points is-", SecTime, "s")
    SetDist.append(distance)

    print("distance between the 2 points:", distance, "m")
    AvDist = sum(SetDist)/len(SetDist)
    Velocity = SetDist[i]/SetTime[i]
    Velocity = Velocity*(18/5)
    SetVel.append(Velocity)
    AvVel = sum(SetVel)/len(SetVel)

    print ('Speed of the car is', Velocity, "m/s")
    print ('Speed of the car is', Velocity, "km/hr")
print ('The array of raw GPS coordinates is:-', SetDat)
api1 = []
for i in range(0, length-1):
    dataPoint = data[i]
    dataPoint1 = data [i+1]
    coordinate = dataPoint['coordinates']
    coordinate1 = dataPoint1['coordinates']
    x = coordinate[0]
    y = coordinate[1]
    x1 = coordinate1[0]
```

```
    y1 = coordinate1[1]

    str1 = str(x)
    str2 = str(y)
    str3 = '|'
    apiData = str1 + ',' + str2 + str3

    api1.append(apiData)
    i +=1
print (' The data points to be sent for a SnapToRoads request i
n correct format:', api1)
requestdat                                                    =
'https://roads.googleapis.com/v1/snapToRoads?path='      +
apiData +        '&key=AIzaSyAmplaUG26XJGwPrLbky2bHQ-
eBmQvZUVU'
print("The Average Distance between 2 points is:", AvDist, "m
etres" + '\n')

print(" The average velocity during the trip is", AvVel, "km/hr
")
std= sqrt(AvDist*(abs(distance - (distance*AvDist))**2))
stdVel = sqrt(AvVel*(abs(Velocity - (Velocity*AvVel))**2))
print ('The standard deviation in distance is:', std, '\n')
print ('The standard deviation in velocity is:', stdVel, '\n')

ValRange = []
Outliers = []
lenn = len(SetDist)
IndDist = []
lenn = len(SetDist)

#STANDARD DEVIATION

for r in range (0, lenn-1):
    dev = SetDist[r]-AvDist
    if abs(dev) <= std:
        IndDist.append(SetDist.index(SetDist[r]))
    else:
        Outliers.append(SetDist[r])
print ("The set of values in range for distance =", ValRange)
print ("The outliers are=", Outliers)
print('The valid indices of distance data as processed by stand
ard deviation are:', IndDist)

VelRange = []
OutVel = []
length = len(SetVel)
IndVel = []
for r in range (0, length-1):
    dev = SetVel[r]-AvVel
    if abs(dev) <= stdVel:
        IndVel.append(SetVel.index(SetVel[r]))
    else:
        pass

print ('Standard Deviation in Speed is ', dev)
print('The valid indices in the velocity data as processed are:', I
ndVel)

#CURVE SMOOTHING

SetVel = [math.sin(math.pi*(n % 10 - 5)/5) for n in range(20)]

diffs = [[ e1 - e0, e2 - e1, e3 - e2, e4 - e3 ]
        for e0, e1, e2, e3, e4
            in zip(SetVel,
                SetVel[1:],
                SetVel[2:],
                SetVel[3:],
                SetVel[4:])]

OutSmooth = []
IndSmooth = []
delta = 1.5

for i, d in enumerate(diffs):
    if d[1] < delta*d[0] or d[2] < delta*d[3]:
        IndSmooth.append(i+2)
    else:
        OutSmooth.append(SetVel[i+2])

for i in range (0, 2):
    IndSmooth.append(i)
for i in range (24, length):
    IndSmooth.append(i)

print ('The Indices of the values in Range after Curve Smoothi
ng are:', IndSmooth)
print("The values of Velocity that do not satisfy the conditions
are:", OutSmooth)

#HTTP REQUEST
results = requests.get(requestdat)
snappoints = results.json()['snappedPoints']
snapdata = []
for point in snappoints:
    # this is each individual element in snappedPoints array
    snapdata.append(point['originalIndex'])
print ('The indices of data that is valid as requested from the S
napToRoads API is:', snapdata)

#RESULT

InFinal = []
InFin-
al = list(set(snapdata) & set(IndSmooth) & set(IndDist) & set(I
ndVel))
print('The final set of indices for the data points that are not ou
tliers is:', InFinal)

length = len(InFinal)
FinalData = []
for i in InFinal:
    FinalData.append(SetDat[i])
```

print ('The result of the program, on eliminating all outliers is: ', FinalData)

7.3 Section 3

[{'timestamp': 1499159501922, 'coordinates': [12.919082641601562, 77.65169525146484]}, {'timestamp': 1499159503930, 'coordinates': [12.918915748596191, 77.6517105102539]}, {'timestamp': 1499159506936, 'coordinates': [12.918656349182129, 77.65177154541016]}, {'timestamp': 1499159509002, 'coordinates': [12.918524742126465, 77.6517562866211]}, {'timestamp': 1499159511984, 'coordinates': [12.918295860290527, 77.65178680419922]}, {'timestamp': 1499159513206, 'coordinates': [12.918216705322266, 77.65177154541016]}, {'timestamp': 1499159515914, 'coordinates': [12.918027877807617, 77.65178680419922]}, {'timestamp': 1499159517944, 'coordinates': [12.917914390563965, 77.65178680419922]}, {'timestamp': 1499159520903, 'coordinates': [12.917774200439453, 77.65178680419922]}, {'timestamp': 1499159522917, 'coordinates': [12.917659759521484, 77.65179443359375]}, {'timestamp': 1499159524954, 'coordinates': [12.917553901672363, 77.65180969238281]}, {'timestamp': 1499159526928, 'coordinates': [12.917448043823242, 77.6518325805664]}]

7.4 Section 4

Section 4.1 :Indices of values in range

The Indices of the values **in** Range after Curve Smoothing are: [2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 0, 1, 24, 25, 26]

Section 4.2: Data points in range

[{'timestamp': 1499159501922, 'coordinates': [12.919082641601562, 77.65169525146484]}, {'timestamp': 1499159503930, 'coordinates': [12.918915748596191, 77.6517105102539]}, {'timestamp': 1499159506936, 'coordinates': [12.918656349182129, 77.65177154541016]}, {'timestamp': 1499159509002, 'coordinates': [12.918524742126465, 77.6517562866211]}, {'timestamp': 1499159511984, 'coordinates': [12.918295860290527, 77.65178680419922]}, {'timestamp': 1499159513206, 'coordinates': [12.918216705322266, 77.65177154541016]}, {'timestamp': 1499159515914, 'coordinates': [12.918027877807617, 77.65178680419922]}, {'timestamp': 1499159517944, 'coordinates': [12.917914390563965, 77.65178680419922]}, {'timestamp': 1499159520903, 'coordinates': [12.917774200439453, 77.65178680419922]}, {'timestamp': 1499159522917, 'coordinates': [12.917659759521484, 77.65179443359375]}]

7.5 Section 5

Section 5.1: Indices of data points snapped by API

The indices of data that is valid as requested from the SnapTo Roads API is: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]

Section 5.2: Data points snapped

{(12.911924499999998, 77.649052), (12.91247152863899, 77.64863494367839), (12.915622446484862, 77.65178829923374), (12.915129444333664, 77.65181069088683), (12.912473433376999, 77.64883380796768), (12.915808049228117, 77.65177986934125), (12.912650843379408, 77.65188984884699), (12.913309683275038, 77.65187592806588), (12.912472259151805, 77.64871121105774), (12.915683362257036, 77.65178553250118), (12.915475867907952, 77.65179495667844), (12.911938955387066, 77.64983043148027), (12.911924574576156, 77.64906257055475), (12.913995558775671, 77.65186064430418), (12.911930136083416, 77.6495503015075), (12.915263648397163, 77.65180459545144), (12.912360523136746, 77.64899217857234), (12.912149064679113, 77.65189661764867), (12.911992536413095, 77.65165489457297), (12.912469264125377, 77.64839860755507), (12.918214811587053, 77.65171454089874), (12.91802528954814, 77.65172119976897), (12.917911987935893, 77.65172590496891), (12.917772027120648, 77.65173171726882)}

7.6 Section 6

[[12.919082641601562, 77.65169525146484], [12.918915748596191, 77.6517105102539], [12.918656349182129, 77.65177154541016], [12.918524742126465, 77.6517562866211], [12.918295860290527, 77.65178680419922], [12.918027877807617, 77.65178680419922], [12.917914390563965, 77.65178680419922], [12.917774200439453, 77.65178680419922], [12.917659759521484, 77.65179443359375], [12.917553901672363, 77.65180969238281], [12.917448043823242, 77.6518325805664], [12.917227745056152, 77.65177917480469], [12.91706657409668, 77.65178680419922], [12.916943550109863, 77.65178680419922], [12.916621208190918, 77.65179443359375], [12.91647720336914, 77.65180206298828], [12.915775299072266, 77.65180969238281], [12.915729522705078, 77.65179443359375]]

8 References

- "Introducing JSON." JSON. Accessed July 27, 2017. http://www.json.org/.
- "Introducing JSON." JSON. Accessed July 27, 2017. http://www.json.org/.
- "Haversine Formula in Python (Bearing and Distance between two GPS points)." Haversine Formula in Python (Bearing and Distance between two GPS points) - Stack Overflow. Accessed July 27, 2017. https://stackoverflow.com/questions/4913349/haversine-formula-in-python-bearing-and-distance-between-two-gps-points.
- Google. Accessed July 27, 2017. https://developers.google.com/maps/documentation/roads/snap.
- "Epoch & Unix Timestamp Conversion Tools." Epoch Converter. Accessed July 27, 2017. https://www.epochconverter.com/.

- Standard Deviation and Variance. Accessed July 27, 2017. http://www.mathsisfun.com/data/standard-deviation.html.

- "Numpy.std¶." Numpy.std — NumPy v1.13 Manual. Accessed July 27, 2017. https://docs.scipy.org/doc/numpy/reference/generated/numpy.std.html.

- "Fit." Smooth response data - MATLAB smooth - MathWorks India. Accessed July 27, 2017. https://in.mathworks.com/help/curvefit/smooth.html.

- "Requests: HTTP for Humans¶." Requests: HTTP for Humans — Requests 2.18.2 documentation. Accessed July 27, 2017. http://docs.python-requests.org/.

i