Independent Study

Desheng Zheng

Zhihan Fang

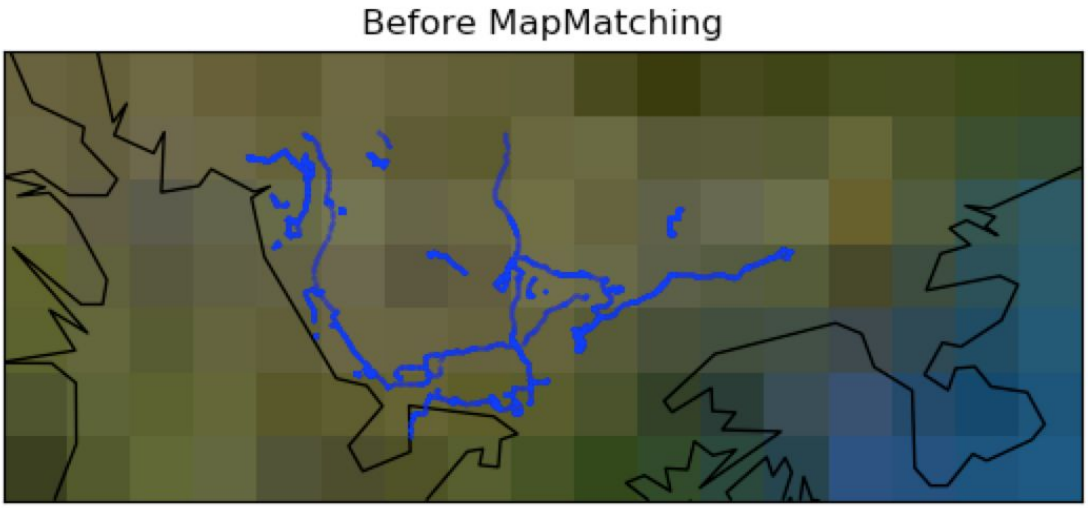
Diana Kris Navarro

Olaolu Emmanuel

Before and After Map Matching

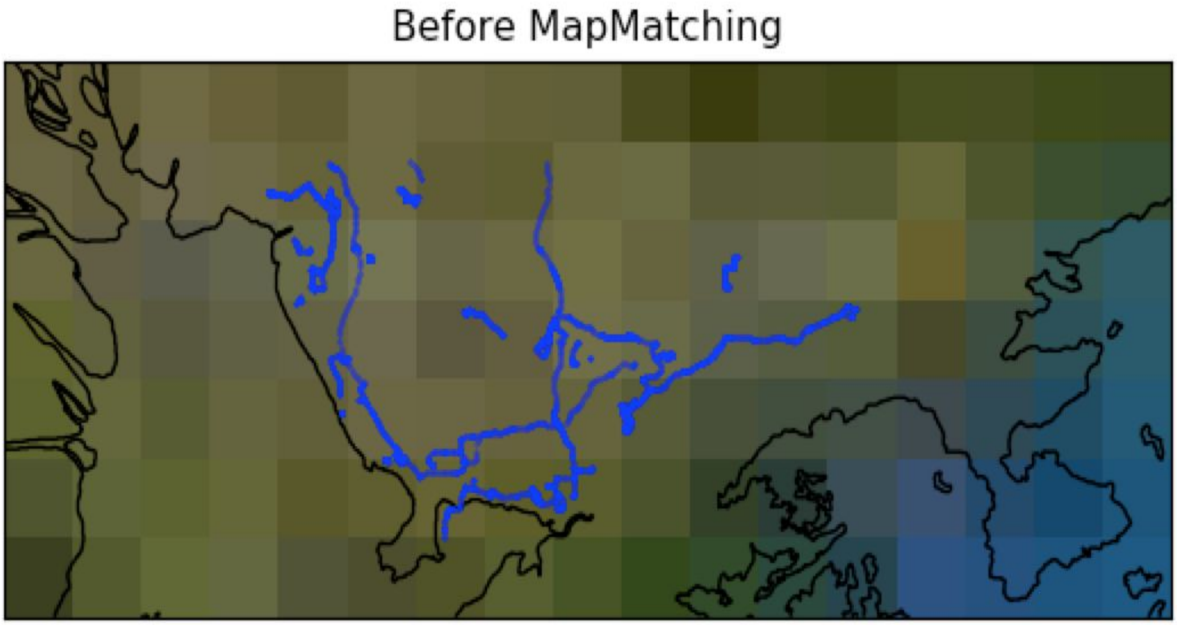
This past semester Biggie and Diana continued projects pertaining to the data collection from the city of Shenzhen. Our first project consisted of mapping coordinates before and after they were matched to form cleaner paths. We were given CSV files containing the points before map-matching as well as the points after map-matching. We first debated on what technologies to use to plot these points. Our knowledge of Python made the Matplotlib and Basemap libraries the best choice for us. We used smaller sets of the CSV to test our processing of the maps, not realizing the magnitude of processing our original CSV file which was half a gigabyte.

Matplotlib is a great library for processing medium sized amounts of data. Along with Matplotlib we used Basemap. Basemap allows one to plot points on real maps as

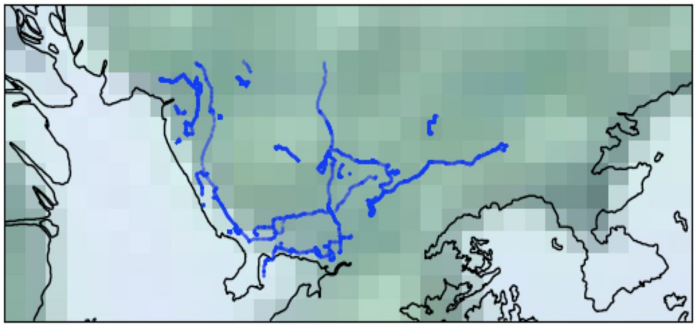


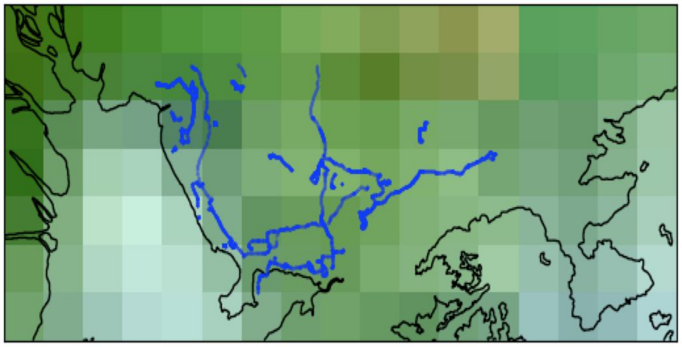
opposed to grids or graphs. We soon realized Matplotlib and Basemap are not the best for processing CSV files that are greater than 200MB like the data that we received. When first using small amounts of data we were able to process and visualize coordinates in roughly 2 minutes (without the background image).

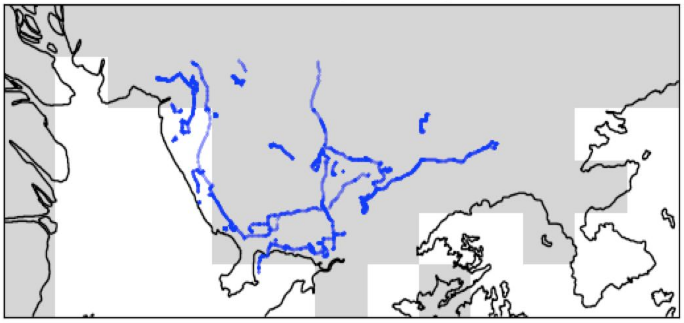
The map above displays the first 10,000 coordinates before map-matching from our CSV. The total time of processing the 10,000 points and visualizing them took roughly four minutes. Although it did what it was supposed to, this was not very good for the map of this clarity. We can clearly see the large pixels of the Earth (taken from Nasa's photo collection *Blue Marble*). Basemap has several tiers of resolution for their maps: c (crude), l (low), i (intermediate), h (high), f (full) or None. The map above is of intermediate resolution. After changing the clarity parameters for Blue Marble and Basemap we were able to process this image:



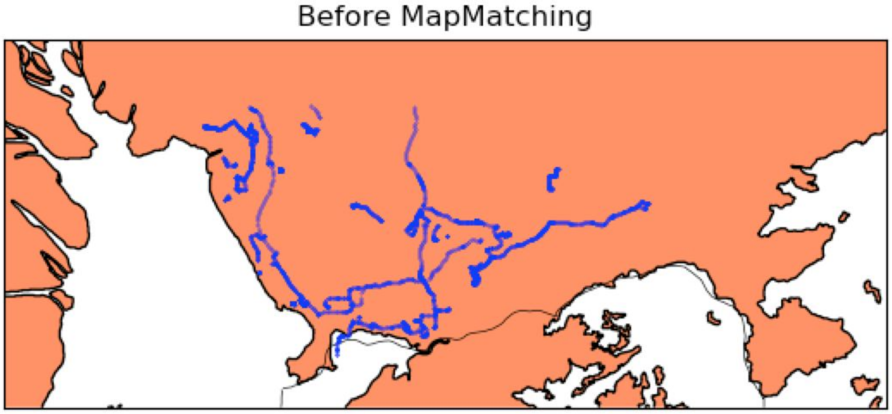
The above map is using resolution in full (f) and Blue Marble Earth image to it’s full resolution. There is unfortunately not a huge difference. We attempted to make the image less pixelated but we have zoomed beyond the intended use of the the Blue Marble Nasa image. What is noticeable is that the coast lines are much more detailed than before. There are many different maps we can use in Basemap. Unfortunately we faced the same issues with each map image that Basemap provided.


Before MapMatching

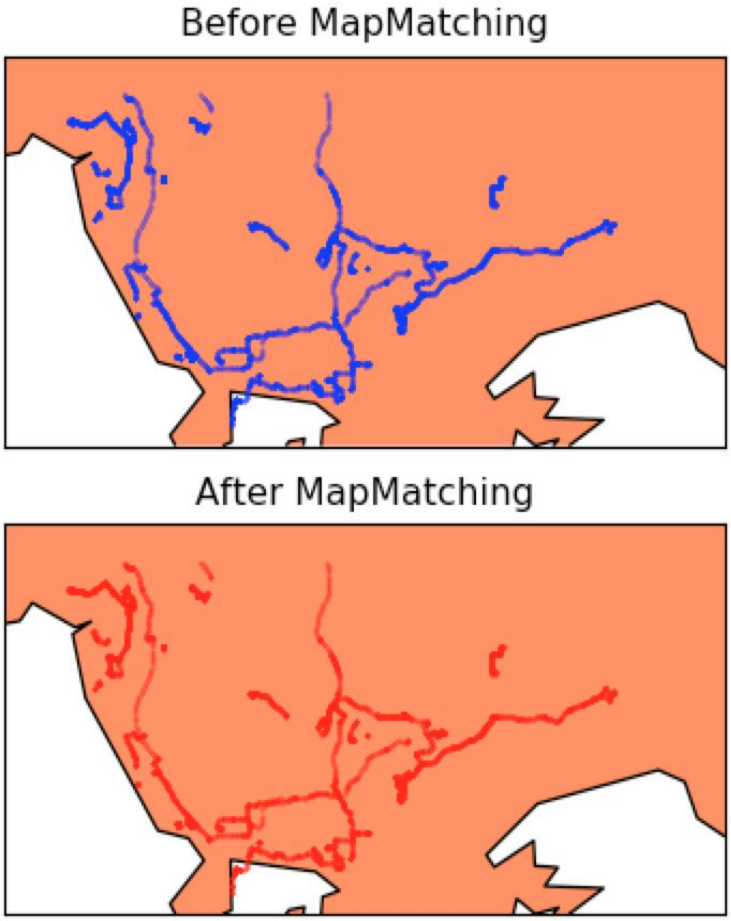

Before MapMatching


Before MapMatching

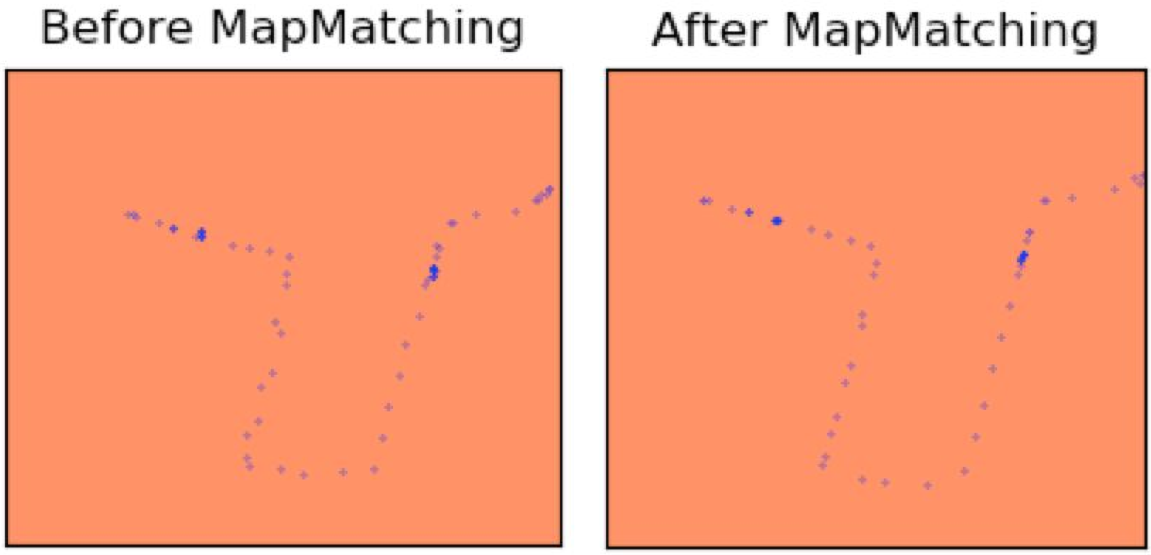
Along with taking a longer time to process, every other map image had issues with zooming in and zooming out. In order to see the the true results of the map-matching we decided to process colors instead of images.



With the above map we switched to using coral to represent land and white to represent the water.



The maps to the left are the results of before and after the points are matched. Although we do not see a significant change between these two maps the results are more noticeable once we zoom in on particular parts.



The above is a good example of points that were aligned to form a more distinct path of coordinates. Overall we learned a great deal about using libraries such as Basemap and Matplotlib, but we also learned about the setbacks of using those two libraries. One setback being that Matplotlib takes a great deal of time to plot and visualize points, so although both libraries were easy to use (especially for beginners) they were not able to handle the amount of data that we received. We were only be able to work with the first 10,000 points. When attempting to visualize the half a gigabyte of data, the process took longer than 4 hours.

road_classification

May 2, 2018

1 Road Classification

Another project that we faced was classifying the roads of Shenzhen given training data in the form of CSV files. The data we were given was private vehicle trips and truck trips with the following facets: coordinates of the trip and timestamps of the trip. From this data we could formulize feature vectors that contained the following data about a particular road: median speed, average speed, 1/4 speed, 3/4 speed, whether there has been a truck/private vehicle on the road, ratio of trucks to private vehicles.

1.1 Tools

The tools that we chose to accomplish this goal are as follows: - From the Python stlib we used - csv - to read the data - statistics - for handy stat functions like mean, median, etc.... - IntEnum - We defined each row of our data in an int enum to make data access as readable as possible - Third party - sklean - for machine learning algorithms - numpy - for easy data formatting i.e. ensuring contiguous arrays - dateutil - to reliably parse dates

```
In [ ]: """
        An SVC for road types.
        """
        import csv
        import collections
        import statistics
        from enum import IntEnum
        import math

        from sklearn import svm
        import numpy as np
        from dateutil import parser
        from tqdm import tqdm
        from sklearn.model_selection import train_test_split
```

1.2 Enum

What we learned from our previous work is that when reading code that access rows of data stored in lists, it can be very hard to follow if we simply use int indices. For this reason, we created an IntEnum to make our data accessing more readable as data[Column.DATE] probably gives more information than data[0].

```
In [ ]: class Column(IntEnum):
            DATE = 0
            TRIP_CHAR = 1
            ROAD_ID = 2
            LON = 3
            LAT = 4
            VEHICLE_TYPE = 5
            LABEL = 6
```

1.3 Global Variables

- CACHE_SIZE - this is the amount of memory that our SVM can use to process our data. We initially set this to 1000, but when we increased it, we found it was much faster, which makes sense.
- label_names - this is a list of the names of each label, or street name. Since we mapped our labels to ints, this would allow us to access the actual name if we needed it.

```
In [ ]: CACHE_SIZE = 4000
        label_names = None
```

This method was for getting the distance between to points in miles. We coul not find a library with an implementation of this so we opted to use this found on someones blog.

```
In [ ]: # distance between two points in miles
        # see: https://www.johndcook.com/blog/python_longitude_latitude/
        def distance_on_unit_sphere(lat1, long1, lat2, long2):
            # Convert latitude and longitude to
            # spherical coordinates in radians.
            degrees_to_radians = math.pi/180.0

            # phi = 90 - latitude
            phi1 = (90.0 - lat1)*degrees_to_radians
            phi2 = (90.0 - lat2)*degrees_to_radians

            # theta = longitude
            theta1 = long1*degrees_to_radians
            theta2 = long2*degrees_to_radians

            # Compute spherical distance from spherical coordinates.

            # For two locations in spherical coordinates
            # (1, theta, phi) and (1, theta', phi')
            # cosine( arc length ) =
            # sin phi sin phi' cos(theta-theta') + cos phi cos phi'
            # distance = rho * arc length

            cos = (math.sin(phi1)*math.sin(phi2)*math.cos(theta1 - theta2)
                   + math.cos(phi1)*math.cos(phi2))
            # sometimes cos > 1?
            if cos > 1:
                # ŕ\_()_/ŕ
                cos = 1

            arc = math.acos(cos)

            # Remember to multiply arc by the radius of the earth
            # in your favorite set of units to get length.
            return arc * 3960
```

This function is for reading our input data from csv(s) into a list. We ran into many complications while doing so because of assumptions we made about the data. For example, We assumed that the data would begin with the start of a trip, but we found that it actually started in the middle of trip so we omitted that trip. We also found multiple ‘s’ and ‘e’ chars together which did not make sense, so we removed the extras.

```
In [ ]: def get_data(limit):
            fnames = ['../data/pv/all.csv', '../data/truck/all.csv']
            data = []

            for fname in fnames:
                with open(fname) as f:
                    vehicle_type = fname.split('/')[2]
                    seen_s = False
                    prev = None

                    reader = tqdm(csv.reader(f, delimiter=","))
                    for count, datum in enumerate(reader):
                        if count >= limit:
                            break

                        reader.set_description_str('Processing row {}/{} in {}'
                                                   .format(count, limit, fname))

                        datum = [parser.parse(datum[3]), datum[6].lower(),
                                 int(datum[7]), float(datum[8]), float(datum[9]),
                                 vehicle_type, datum[15]]

                        # filter out remainder of trip at beginning of data
                        # (data should start with an 's')
                        if datum[Column.TRIP_CHAR] == 's':
                            seen_s = True
                        if not seen_s and datum[Column.TRIP_CHAR] != 's':
                            continue

                        # filter out extra 'S' and 'E' chars
                        is_start_or_end = (datum[Column.TRIP_CHAR] == 's'
                                           or datum[Column.TRIP_CHAR] == 'e')
                        same_char_as_prev = (prev
                                             and (prev[Column.TRIP_CHAR]
                                                  == datum[Column.TRIP_CHAR]))
                        if is_start_or_end and same_char_as_prev:
                            continue

                        data.append(datum)

                        prev = datum

            # map labels to ints
            global label_names
            # get a set of all unique labels then convert to a list
            # so that it is indexable
            label_names = list(set([datum[Column.LABEL] for datum in data]))
            for datum in data:
                datum[Column.LABEL] = label_names.index(
                    datum[Column.LABEL])

            return data
```

This is the function where we process the raw rows of data into feature vectors, one per road. D

The first part was creating a dictionary of metadata for each road. This includes the number of

Once this was complete, we just had to perform various stat functions on the speeds list and we

```
In [ ]: def get_features(data):
            roads = collections.defaultdict(
                lambda: {'num_truck': 0, 'num_pv': 0, 'speeds': [],
                         'label': -1})

            prev = None
            data_tqdm = tqdm(data)
            data_len = len(data)
            for idx, datum in enumerate(data_tqdm):
                data_tqdm.set_description_str('Processing datum {}/{}'
                                              .format(idx, data_len))

                date, trip_char, road_id, lon, lat, vehicle_type, label =\
                    datum

                roads[road_id]['label'] = label

                # add to vehicle count
                vehicle_key = 'num_truck' if vehicle_type == 'truck' else 'num_pv'
                roads[road_id][vehicle_key] += 1

                # add to speeds if not last point
```

```
    part_of_same_trip = (prev
                         and (prev[Column.TRIP_CHAR] == 's'
                              or prev[Column.TRIP_CHAR] == 'm')
                         and (trip_char == 'm' or trip_char == 'e'))
    if part_of_same_trip:
        time_diff = date - prev[Column.DATE]
        hours = time_diff.seconds / 60 / 60

        prev_lat = prev[Column.LAT]
        prev_lon = prev[Column.LON]
        miles = distance_on_unit_sphere(lat, lon, prev_lat,
                                        prev_lon)

        # things happen
        if miles == 0 or hours == 0:
            continue

        speed = miles / hours
        roads[road_id]['speeds'].append(speed)

    prev = datum

features = []
labels = []
roads_len = len(roads)
roads_tqdm = tqdm(enumerate(roads.items()))
for i, (k, v) in roads_tqdm:
    roads_tqdm.set_description_str('Creating feature vector {}/{}'
                                   .format(i, roads_len))

    num_truck, num_pv, speeds, label = (v['num_truck'], v['num_pv'],
                                        v['speeds'], v['label'])
    speeds.sort()

    # for some reason some roads have no speeds?
    if not speeds:
        continue

    features.append([
        # 1/4 speed
        speeds[int(len(speeds) / 4)],
        # median speed
        statistics.median(speeds),
        # 3/4 speed
        speeds[int((len(speeds) / 4) * 3)],
        # average speed
        statistics.mean(speeds),
        # existance of pv
```

```
                    1 if num_pv > 0 else 0,
                    # existance of truck
                    1 if num_truck > 0 else 0,
                    # percentage of pv on road
                    num_pv / (num_pv + num_truck),
                    # percentage of truck on road
                    num_truck / (num_pv + num_truck)])
                labels.append(label)

            return (features, labels)
```

This method was for splitting our data into training and testing data. This was important because if we simply trained on the data and then tested on it, our score would be extremly biased, or overfitted, because it was trained on what is is being told to predict. We chose to ues 60% for testing and 40% for training.

```
In [ ]: def get_train_test_data(features, labels):
            return [
                np.ascontiguousarray(arr, dtype=np.float32)
                for arr
                in train_test_split(features, labels, test_size=.4, random_state=0)]
```

Despite being the least amount of code, this method actually takes up the majority of the runtime. That is because this is the method which trains the classifier.

```
In [ ]: def get_classifier(features, labels):
            print('Fitting model.....')
            clf = svm.SVC(cache_size=CACHE_SIZE).fit(features, labels)
            print('Fitting model DONE')

            return clf
```

This lets us know how well we did. The best score we recieved was 30%. Despite how low this is, we are confident that with more training data, it could actually perform quite well. We only had 40,000 vectors, but I imagine we would need somthing in the millions.

```
In [ ]: def get_score(clf, features, labels):
            print('Calculating score.....')
            score = clf.score(features, labels)
            print('Calculating score DONE')

            return score
```

Runs the classifier

```
In [ ]: if __name__ == '__main__':
            data = get_data(1_000_000_000)
            features, labels = get_features(data)
            f_train, f_test, l_train, l_test = get_train_test_data(features, labels)
```

```
clf = get_classifier(f_train, l_train)
score = get_score(clf, f_test, l_test)
print('SCORE: {}%'.format(int(score * 100)))
```