Taxi Trajectory Prediction

Karl Krauth (z3416790)
David McKinnon (z3421068)

June 10, 2015

1 Important please read

Note, this report describes our initial project, we later switched to a project involving reinforcement learning due to the poor performance on Kaggle. Only evaluate this if you have already evaluated the reinforcement learning project and would like to also include this in your evaluation. If this would lead to lost marks do not evaluate anything in this folder.

2 Introduction

Taxis nowadays use electronic dispatch systems for scheduling pick-ups, but they do not usually enter their drop-off locations. Therefore, when a call comes for a taxi, it is difficult for dispatchers to know which taxi to contact.

To improve the efficiency of electronic taxi dispatching systems it is important to be able to predict the final destination of a taxi while it is in service. Since there is often a taxi whose current ride will end near a requested pick up location from a new passenger, it would be useful to know approximately where each taxi is likely to end so that the system can identify the best taxi to assign to each new pickup request. This lowers the waiting time for new passengers and allows the taxi system to operate more efficiently.

This project occurs in the context of a Kaggle competition hosted by ECML/PKDD. The goal of this competition is to predict the destination of taxis travelling in Porto, Portugal given specific data about the current trip. To aid with this a dataset of around 1.7 million complete trips is provided.

3 Methods

3.1 Regression Trees and Boosting

The second method that was attempted was the use of regression trees, both used by itself and augmented by AdaBoost. Before being able to train on the dataset we first went through a pre-processing step. The following features were modified as follows:

- **ORIGIN_CALL:** If set to a NULL value instead set it to 0.
- **ORIGIN_STAND:** If set to a NULL value instead set it to 0.
- **CALL_TYPE:** Remove this column since ORIGIN_CALL and ORIGIN_STAND give us the information we need about the CALL_TYPE.
- **DAYTYPE:** Remove the DAYTYPE since they're all set to the same value.
- **MISSING_DATA:** Remove any rows with missing data (only 10 datapoints have missing data)
- **POLYLINE:** Keep only the starting point, a randomly selected point in the middle and the end point (as the target value).

The idea behind the modification of POLYLINE is that we would like to transform a variable length vector into a fixed length vector, we also would like to modify polyline to only be a partial path since a complete path would not help with prediction.

After having modified the dataset we then train a regression tree on the generated training set. A maximum depth of 20 leads to good performance. We also train a regression tree used in conjunction with AdaBoost, with 5 instances and a max depth of 20 for better performance.

3.2 Frechet Distance and KNN

3.2.1 Frechet Distance

The final and best performing method that was evaluated only made use of the paths. It used a similarity measure between two oriented curves called the frechet distance. The frechet between two curves A and B is defined as follow:

$$\inf_{\alpha,\beta} \max_{t\in[0,1]} \left(d(A(\alpha(t)), B(\beta(t)))\right).$$

Where d in our case is the haversine distance between two GPS coordinates. Intuitively the frechet distance can be thought as being the minimum length of a leash required to connect a dog and its owner constrained on the two seperate paths, as they walk along with no backtracking. In our case we are dealing with polygonal line segments and thus only need to consider the curves at each vertex, an approach using dynamic programming is given in the implementation.

3.2.2 Truncating paths.

We could simply compare the current test data's path with all training paths and pick the destination of the training path with the shortest frechet distance as our prediction. However this does not work well in practice since this favours predictions that assume that the test path is nearly complete (which it is not in the majority of situations). To remedy this we instead chose to truncate training paths that are longer than the test path before comparing them. We define the length of a polygonal path as the haversine distance between every pair of adjacent coordinates in the path.

3.2.3 K nearest neighbours

Calculating the Frechet distance is computationally expensive, given a path of length m and a path of length n, it takes $O(mn)$ time to compute the distance. As such, computing the Frechet distance for between every test path against every training path is computationally infeasible. Instead we would like to quickly eliminate the majority of training paths without taking their Frechet distance from the test path. Solutions to the k nearest neighbours problem such as the ball tree naturally lend themselves to this. Unfortunately we can't build a ball tree using our distance function as a metric since we truncate paths before applying the Frechet distance, our distance function does not induce a metric space [1] making it unsuitable for KNN.

[1]Specifically, the distance function does not satisfy the identity of indiscernible and triangle inequality.

Instead we use the distance from the starting point of two paths as our metric in the ball tree. We then grab the 10,000 paths with starting point closest to our test path and get the path with the closest Frechet distance of the test path out of all those paths. In practice this works very well since it gives the starting point of a path a higher precedence than other points.

3.3 Memory and time constraints

To reduce the running time of our algorithm we wished to keep disk I/O to a minimum, hence we loaded every single data-point into memory before operating on it. Unfortunately this meant that python's native datatypes could no longer be used as our program would require greater than 12Gb to run. We thus had to make use of efficient arrays and take care not to accidentally expand too much data into python's native floats at any one point.

4 Results

We ran a 10 fold cross validation test on a regression tree with maximum depth of: 5, 20, and 40 and on AdaBoost with 5 instances of a regression tree of max depth of 20. Our test/training set were the generated truncated paths. Our performance measure was the mean distance of the predictions from the true destination. The baseline simply involves picking the last point on the partial path as our prediction. The results were as follows:

Algorithm	Tree 5	Tree 20	Tree 40	AdaBoost	Baseline
Error(km)	2.30	1.73	2.23	1.66	4.20

We were not able to test the frechet predictor using cross validation due to the slow runtime. Instead we directly submitted to the Kaggle competition which evaluated the submission based on 300 test points that were not in the training set. We did not submit the worse performing algorithms due to a 2 submission a day limit. The baseline involves picking all predictions as the most visited point (downtown). The results were as follows:

Algorithm	Tree 20	AdaBoost	Frechet	Baseline
Error(km)	3.62	3.54	3.30	3.67

Upon submission the Frechet predictor was ranked in 60^{th} out of 193 (top 30%). We do make a note that the performance was worse on the test set than on our generated test set. Upon further inspection we see that the test set involves many more outliers than our generated test set.

5 Conclusion

The Frechet predictor ended up performing best on Kaggle's test set. Unfortunately the performance left a lot to be desired compared to our generated test data. Incorporating more features in our frechet predictor would likely mitigate this. For example having a recommender system that looks at a caller's previous trips and determines future trips based on their and other caller's prior trips depending on how similar trips are.

The runtime of the frechet predictor also leaves a lot to be desired with. It takes on the order of 3 hours to train the predictor on 1.7 million datapoints. Using an approximation to the frechet distance would see a large reduction in runtime.