Taxi Trip Time Prediction

COMS 4771

Junke HE (jh3488)
Ning Lu
Xinghai Zhang
Yanyu Zheng

Nakul Verma $\boxed{\mathcal{IN}}$

Contents

1 Project Description

To improve the efficiency of electronic taxi dispatching systems it is important to be able to predict how long a driver will have his taxi occupied. If a dispatcher knew approximately when a taxi driver would be ending their current ride, they would be better able to identify which driver to assign to each pickup request.

In this challenge, we will build a predictive framework that is able to infer the trip time of taxi rides in Porto, Portugal based on their (initial) partial trajectories.

2 Dataset

1. TRIPID: (String) It contains an unique identifier for each trip;
2. CALLTYPE: (char) It identifies the way used to demand this service. It may contain one of three possible values:
   a. A if this trip was dispatched from the central;
   b. B if this trip was demanded directly to a taxi driver on a specific stand;
   c. C otherwise (i.e. a trip demanded on a random street).
3. ORIGINCALL: (integer) It contains an unique identifier for each phone number which was used to demand, at least, one service. It identifies the trips customer if CALLTYPE=A. Otherwise, it assumes a NULL value;
4. ORIGINSTAND: (integer): It contains an unique identifier for the taxi stand. It identifies the starting point of the trip if CALLTYPE=B. Otherwise, it assumes a NULL value;
5. TAXIID: (integer): It contains an unique identifier for the taxi driver that performed each trip;
6. TIMESTAMP: (integer) Unix Timestamp (in seconds). It identifies the trips start;
7. DAYTYPE: (char) It identifies the daytype of the trips start. It assumes one of three possible values:
   a. B if this trip started on a holiday or any other special day (i.e. extending holidays, floating holidays, etc.);
   b. C if the trip started on a day before a type-B day;
   c. A otherwise (i.e. a normal day, workday or weekend).
8. POLYLINE: (String): It contains a list of GPS coordinates (i.e. WGS84 format) mapped as a string. Here is an example:

   $$"[[-8.585676, 41.148522], [-8.585712, 41.148639], ..., [-8.584884, 41.146623]]"$$

   Each pair of coordinates is also identified by the same brackets as [LONGITUDE, LATITUDE]. This list contains one pair of coordinates for each 15 seconds of trip. For training data their POLYLINE describes their complete trajectory, but for test data we only have records of partial trajectory;

3 Feature Selection

In order to define similarity between observations, we want to see which are the most contributing factors. Intuitively, features such as TAXIID and TRIPID may not be important in terms of making prediction of the trip time. However, we want to verify that the others are necessary by running feature selection algorithms. The first algorithm we try is Principal Component Analysis, which transfers the original Euclidean Space in which the data lives to a lower dimensional feature space, while preserving the dominant patterns. (S Wold, 1987). The result of PCA tends to create new features by combining old ones, and thus not suitable for our propose. Then we use CFS, which stands for Correlation-Based Feature Selection. CFS chooses good feature sets that contain features that are highly correlated with the class, yet uncorrelated with each other. (M.A. Hall, 1999). The features returned by CFS are CALLTYPE, ORIGINALSTAND and TAXIID. Notice that CFS is ran on the original dataset, without those features we generated from the polyline.

Results of running CFS using Weka Explorer graphical interface:

```
Search Method:
        Best first.
        Start set: no attributes
        Search direction: forward
        Stale search after 5 node expansions
        Total number of subsets evaluated: 13
        Merit of best subset found:    0.071

Attribute Subset Evaluator (supervised, Class (numeric): 5 Time):
        CFS Subset Evaluator
        Including locally predictive attributes

Selected attributes: 1,3,4 : 3
                     Call_Type
                     Origin_Stand
                     Taxi_ID
```

4 Kernel Method and Analysis

4.1 Local Learning Algorithms

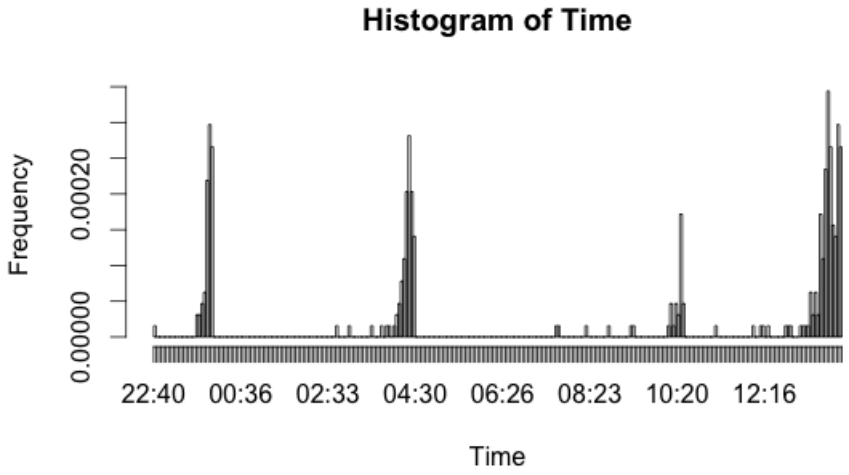
The main algorithm we developed is based on the idea of Local Learning. Local learning algorithms attempt to locally adjust the capacity of the training system to the properties of the training set in each area of the input space. (L Bottou, 1992) In our case, it is equivalent to create a nearest neighbor smoother as our kernel. For each observation in the testing data, we want to find similar observations in the training data, and build nonparametric regression model only around those instances within the kernel. We also borrow the concept of kernel regression, which is assigning weights to each of the training data that is in the neighborhood.

4.2 Neighbor Searching

As is mentioned in part 3: Feature Selection, we found that CALLTYPE, TAXIID and ORIGINALSTAND may significantly influence the trip time of taxi. However, we wont choose TAXIID as criteria, as it will sharply reduce the size of a local neighborhood, which can affect the credibility of prediction. So, our first step is to make sure the neighbors have the same value in these two covariates as the test data.

Based on the prior knowledge, we assume that in a certain range of time, the road conditions are almost the same. So for each test data, we expand its starting time to a time band, that is to add in or subtract from the starting time by 30 min. All training data included in this time band will be treated as local neighbors. One thing need to mention is that there is only one variable, TIMESTAMP, identifying the trips start with Unix Timestamp (in seconds). We convert it to UTC time at first, and then exclude the date, only preserve the time for each data point. As you can imagine, even if on different days, the road conditions at a certain time are quite similar, like every 9 oclock in the morning, Manhattan is always crowded, and you will expect a much longer trip even your destination is not that far. We dont need to consider if its holiday or weekend, as all the data has the same day type, as described in one of the covariates.

Another thing worth mentioning is that the training set has 1.7 million rows (2.9 Gb), which is very hard to manipulate. As we found from the test data, the data points can be well separated into 4 groups by the the starting time, as is shown in the graph below.



We exclude the data from training set that is impossible to be contained in the time band for each test data, and separate the training data into four groups corresponding to the test data. We successfully reduce the training set we will use from 2.9 Gb to 1.2 Gb totally, and for each test data the searching space are largely shrunken.

To simplify our calculation, we want to make sure the distance of boarding location between the training data and the test data is so closed that we can treat as 0. This is also a reasonable assumption, which will be shown later. The boarding location can be found at the beginning of POLYLINE, and we calculate the mean of the GPS coordinates in a neighborhood as their common start. After these three steps, the training data in a local neighborhood will have the same CALLTYPE, the same CALLSTAND, a similar starting time and a similar boarding location (the definition of similar is controlled by parameters time.band and region.band in our code) as its test data.

4.3 Weights Calculation

After we find the neighbors for each test data, we move to the next step, calculating the weights. The variable we use in this step is POLYLINE, which is the record of the path of a taxi.

4.3.1 Preprocess POLYLINE

POLYLINE is a series of GPS coordinates, which is recorded every 15 seconds. For test data not only does this variable contain the information of taxi path, but also it tells us the time that is already taken. Thus, for each training data existing in the local neighborhood, we only calculate weights for those with longer POLYLINE. By doing so, we estimate the trip time of a test data based on its past time, which is a conditional expectation.

When we calculate the weights, we hope that training data with similar routine like test data will have larger weights. We trim the POLYLINE of training data to the same length as test data at first. Thus, for each POLYLINE in a local neighborhood, all could be considered as a set of vectors, having the format as

$$\{(x_1^*, y_1^*), (x_2, y_2), (x_3, y_3), , (x_n, y_n)\}(1),$$

where (x_1^*, y_1^*) is the common start in this band and n is the number of GPS coordinates of the test data. Then we convert this GPS information to a record of path by calculating

$$\{(x_2 - x_1^*, y_2 - y_1^*), (x_3 - x_2, y_3 - y_2), , (x_n - x_{n-1}, y_n - y_{n-1})\}(2).$$

Add up all the vectors in the path set with (x_1^*, y_1^*), we will reach the last GPS point of the trimmed POLYLINE.

As we assume the beginning of POLYLINE in a local neighborhood are the same, we only need to measure the significance of a training data by comparing the path set with test data.

4.3.2 Kernel Function

A vector has two components, direction and length. We want to create a function with two vector arguments that will generate a relatively low value if these two vectors have similar directions and small difference between their lengths.

Assume there are two vectors, x_1 and x_2, we capture the similarity of directions of these two vecotrs based on cosine similarity,

$$(1 + \frac{\vec{x_1} \cdot \vec{x_2}}{\|\vec{x_1}\|\|\vec{x_2}\|})^3,$$

this ensure that the value is always positive and if the vectors are opposite in directions ($\cos(\vec{x_1}, \vec{x_2}) < 0$) the value it generates will be much smaller if the vectors are in the same directions ($\cos(\vec{x_1}, \vec{x_2}) > 0$), as it is a three degree polynomial.

To capture the difference of length, we want to have such a function that it is always between 0 and 1. If the difference of distances of vectors is small, the value will be closer to 1:

$$1 - \frac{|\|\vec{x_1}\| - \|\vec{x_2}\||}{\|\vec{x_1}\| + \|\vec{x_2}\|}.$$

When we combine these two parts, we want the length part and have the same importance as the direction part in our function, so will multiply it by 8, as the direction part is ranging from 0 to 8. So our function to measure the similarity is

$$f(\vec{x_1}, \vec{x_2}) = (1 + \frac{\vec{x_1} \cdot \vec{x_2}}{\|\vec{x_1}\|\|\vec{x_2}\|})^3 \cdot 8 \cdot (1 - \frac{|\|\vec{x_1}\| - \|\vec{x_2}\||}{\|\vec{x_1}\| + \|\vec{x_2}\|})$$

.

4.3.3 Weighting Method

Assume that for a test data the path set we mentioned in 4.3.1 (2) is

$$\begin{aligned} X_{test} &= \{(x_2 - x_1^*, y_2 - y_1^*), (x_3 - x_2, y_3 - y_2), , (x_n - x_{n-1}, y_n - y_{n-1})\} \quad (1) \\ &= \{\vec{p_1}, \vec{p_2}, ..., \vec{p_{n-1}}\} \quad (2) \end{aligned}$$

and for a training data the set is

$$\begin{aligned} X_{train}^{(i)} &= \{(x_2^{(i)} - x_1^*, y_2^{(i)} - y_1^*), (x_3^{(i)} - x_2^{(i)}, y_3^{(i)} - y_2^{(i)}, , (x_n - x_{n-1}^{(i)}, y_n - y_{n-1}^{(i)}) \quad (3) \\ &= \{\vec{p_1}^{(i)}, \vec{p_2}^{(i)}, ..., \vec{p_{n-1}}^{(i)}\} \quad (4) \end{aligned}$$

We calculate the values of our kernel function for each pair of the vectors in these path set, and sum them up,to have get the weight value for a training data

$$K_i = \sum_{j=1}^{n-1} f(\vec{p_j}, \vec{p_j}^{(i)}),$$

Thus the weight for a training data is

$$w_i = \frac{K_i}{\sum_j K_j},$$

where training data j is any data points in the local neighborhood.

After we calculate the weights for each training data, we can estimate the trip time of test data by calculating

$$\sum_j w_j \cdot y_j,$$

where y_j is the jth trip time in the local training data set.

5 Model Comparison

The No Free Lunch theorem for machine learning states that there does not exist an optimal algorithm for a given problem. Because an algorithm that performs well on a certain class of problems will necessarily degraded performance on the set of all remaining problems. (DH Wolpert, 1996) Therefore, though we focus mainly on tuning the Kernel Regression, it is still worth the effort to explore other models.

Weka is a open source software that contains a collection of machine learning algorithms for data mining tasks. It is efficient and user-friendly in terms of testing different algorithms. The training data we use consists CALLTYPE, Starting Coordinates, Number of GPS points so far and Average Speed. All the models are validated using 10-fold validation.

Table of models examined and results

Method	Mean Absolute Error	Relative Absolute Error (%)
Zero R	694	100
Single Conjunctive Rule	576	64
M5 Rule	446	82.8
Decision Stamp	575	64.1
M5P Tree	445	65.37
REP Tree	453	69
SMOREG	482	67.3
IBK (k=3)	512	73.8
IBK(k=5)	485	69.9
Additive Regression w/ M5P	445	64.19
Bagging w/ MSP	439	63.32

ZeroR is a lazy classifier that always predicts the mean for numeric class. It is often used as a branch mark and the performance is pretty bad with 100

As we can see from the table, M5P tree seems to have the lowest mean absolute error and relative absolute error among single classifiers. M5P is tree-based piecewise linear models that are similar to CART, but generally have smaller trees in terms of size for regression problem. (J.R. Quinlan, 1992). The two ensemble methods are both built on M5P as the underlying classifier, but the performance does not have significant improvement in this case. As of the time this report is written, we exhaust the chances for submission to see how M5Ps result compares to the kernel regression model, but it is something we should definitely consider.

6 Future Improvement

It's the first time for all our team members to participate in a machine learning competition, all of us have learned a lot from it. We think we can keep improving our predition performance mainly from two parts: applying cross validation and using existing path clustering method.

6.1 Cross Validation

As in our neighbor searching step, we set up some parameters like 'time.band' and 'region.band' to help decide the local neighborhood. We just arbitrarily assign values to those parameters without using cross validation to find out the optimal value.

6.2 Trajectory Clustering

In our project, we are trying to create a method to cluster the taxi path by our own. During this process, we think our knowledge preparation is not enough.

Actually, when I google trajectory clustering, there are many papers on this topic. Next time when we take part into the kaggle competition, the first step would be reading relative papers.

7 Reference

1. Wold, Svante, Kim Esbensen, and Paul Geladi. "Principal component analysis." Chemometrics and intelligent laboratory systems 2.1 (1987): 37-52.

2. Hall, Mark A. Correlation-based feature selection for machine learning. Diss. The University of Waikato, 1999.

3. Bottou, Lon, and Vladimir Vapnik. "Local learning algorithms." Neural computation 4.6 (1992): 888-900.

4. Wolpert, David H. "The lack of a priori distinctions between learning algorithms." Neural computation 8.7 (1996): 1341-1390.

5. Quinlan, John R. "Learning with continuous classes." 5th Australian joint conference on artificial intelligence. Vol. 92. 1992.

8 Appendix-R Code

8.1 Funtions

```
DATASET_MATCH <- function(train_set, test_set, time_band=1800){
  n_test <- dim(test_set)[1]
  match_ind = list()
  for (i in 1:n_test) {
    test_data <- test_set[i,]
    test_time <- test_data$STARTIME
    test_time <- as.POSIXct(test_time, tz='GMT', format='%H:%M:%S')
    time_low <- test_time - time_band
    time_high <- test_time + time_band
    time_low <- format(time_low, '%H:%M:%S')
    time_high <- format(time_high, '%H:%M:%S')
    test_cri <- test_data$CALL_STAND
    if (class(test_cri) == 'factor'){
      cri_ind_temp <- which(train_set$CALL_STAND == as.character(test_cri))
    } else {
      cri_ind_temp <- which(train_set$CALL_STAND == test_cri)
    }
    train_set_i <- train_set$STARTIME[cri_ind_temp]
    m_ind_i <- cri_ind_temp[which(train_set_i >= time_low & train_set_i <= time_hig
    match_ind[[i]] <- m_ind_i
  }
  print('The size of the 10 smallest trainset that matched with the test data:')
  print(sort(do.call('c', lapply(match_ind, 'length')))[1:10])
  return (match_ind)

}


GEN_TIME <- function(train_polyline){
  n_poly_points <- strsplit(train_polyline, ',')
  n_poly_points <- do.call('c', lapply(n_poly_points, length))
```

```
  ind_na <- which(n_poly_points == 1)
  if (length(ind_na) > 0) {
    n_poly_points <- n_poly_points[-ind_na]
    TRIPTIME <- (n_poly_points/2 - 1) * 15
  } else {
    TRIPTIME <- (n_poly_points/2 - 1) * 15
  }
  results <- list('TRIPTIME'=TRIPTIME, 'NAs'=ind_na)

}

# convert POLYLINE to a set of gps points
# make sure train data have the same length as test data
POLYLINE_PRE <- function(train_poly, test_poly, region_band, sample_size){
  polyline <- c(test_poly, train_poly)
  polyline <- strsplit(polyline, ',')
  len_low <- length(polyline[[1]])
  len_all <- do.call('c', lapply(polyline, 'length'))
  ind_aban <- which(len_all < len_low)
  polyline <- lapply(polyline,'[', 1:len_low)
  polyline <- lapply(polyline, 'gsub', pattern='[][]', replacement='')
  polyline <- as.numeric(unlist(polyline))
  polyline <- matrix(polyline, ncol=len_low, byrow=TRUE)
  start_x <- polyline[,1]
  start_y <- polyline[,2]
  s_x_low <- start_x[1] - region_band
  s_x_high <- start_x[1] + region_band
  s_y_low <- start_y[1] - region_band
  s_y_high <- start_y[1] + region_band
  s_x_outre <- which(start_x < s_x_low | start_x > s_x_high)
  s_y_outre <-  which(start_y < s_y_low | start_y > s_y_high)
  outre <- union(s_x_outre, s_y_outre)
  ind_aban <- union(ind_aban, outre)
  if (length(ind_aban) >= 1){
    if (length(ind_aban) < (length(len_all)-1)) {
      polyline <- polyline[-ind_aban,]
      polyline <- t(polyline)
      poly_test_vec <- polyline[,1]
      poly_train_mat <- polyline[,-1]
    } else {
      poly_test_vec <- polyline[-ind_aban,]
      poly_train_mat <- NA
    }
  } else {
    polyline <- t(polyline)
    poly_test_vec <- polyline[,1]
    poly_train_mat <- polyline[,-1]
  }
```

```
  ind_aban <- ind_aban - 1
  result = list('TEST'=poly_test_vec, 'TRAIN'=poly_train_mat, 'NAs'=ind_aban, 'TEST
  return (result)

}


POLY_WEIGHT <- function(train_poly, test_poly, region_band=0.0001, sample_size=50){
  polytrans <- POLYLINE_PRE(train_poly, test_poly, region_band, sample_size)
  test_vec <- polytrans$TEST
  train_mat <- polytrans$TRAIN
  na_ind <- polytrans$NAs
  test_len <- polytrans$TESTLEN
  if (is.na(train_mat)[1]){
    na_ind <- NULL
    poly_weights <- rep(1/length(train_poly), length(train_poly))
  } else {
    if (length(train_poly) - length(na_ind) > 1){
      test_X1 <- test_vec[seq(from=1, to=(test_len-1), by=2)]
      test_X2 <- test_vec[seq(from=2, to=(test_len), by=2)]
      train_X1 <- train_mat[seq(from=1, to=(test_len-1), by=2),]
      train_X2 <- train_mat[seq(from=2, to=(test_len), by=2),]
      n_path <- length(test_X1)
      if (n_path > 1) {
        mean_start_X1 <- mean(c(test_X1[1], train_X1[1,]))
        mean_start_X2 <- mean(c(test_X2[1], train_X2[1,]))
        final_test_X1 <- test_X1[n_path]
        final_test_X2 <- test_X2[n_path]
        final_train_X1 <- train_X1[n_path,]
        final_train_X2 <- train_X2[n_path,]
        path_test_X1 <- final_test_X1 - mean_start_X1
        path_test_X2 <- final_test_X2 - mean_start_X2
        path_train_X1 <- final_train_X1 - mean_start_X1
        path_train_X2 <- final_train_X2 - mean_start_X2
        inner_pro <- path_test_X1*path_train_X1 + path_test_X2*path_train_X2
        norm_test <- sqrt(path_test_X1^2 + path_test_X2^2)
        norm_train <- sqrt(path_train_X1^2 + path_train_X2^2)
        cos_simi <- inner_pro / (norm_test*norm_train)
        weight_value <- ((cos_simi + 1)^3) * (8 * (1 - abs(norm_train - norm_test)
      } else {
        weight_value <- sqrt((train_X1 - test_X1)^2 + (train_X2 - test_X2)^2)
      }
      poly_weights <- weight_value / sum(weight_value)
    } else {
      poly_weights <- 1
    }

  }
```

```
  result <- list('WEIGHT'=poly_weights, 'NAs'=na_ind)
  return (result)

}
```

8.2 Main

```
####################
# Author: Junke HE
# Date: June 28th, 2015
##################################
#
run_start_time <- Sys.time()
## set up for test data
setwd('/Users/ZacharyHE/Documents/CU/COMS S 4771/Project/Data/Raw_Data')
#
## import test data
testset <- read.csv('test.csv', header=T, colClasses=c(NA, NA, 'NULL', NA, 'NULL',
testset <- transform(testset, POLYLINE=as.character(POLYLINE))
testset <- subset(testset, select=-MISSING_DATA)
#
## generate the start time (without date) in testset
start_time <- as.POSIXct(testset$TIMESTAMP, tz='GMT', origin='1970-01-01')
start_time <- format(start_time, '%H:%M:%S')
# devide the test set into four different sets based on the start time
testset1_ind <- which(start_time >= '02:40:17' & start_time <= '03:59:28')
testset2_ind <- which(start_time >= '06:44:32' & start_time <= '08:29:44')
testset3_ind <- which(start_time >= '11:39:46' & start_time <= '14:29:19')
testset4_ind <- which(start_time >= '15:12:05' & start_time <= '17:59:53')
#
testset1 <- cbind(testset[testset1_ind,], STARTIME=start_time[testset1_ind])
testset1 <- transform(testset1, STARTIME=as.character(STARTIME))
testset2 <- cbind(testset[testset2_ind,], STARTIME=start_time[testset2_ind])
testset2 <- transform(testset2, STARTIME=as.character(STARTIME))
testset3 <- cbind(testset[testset3_ind,], STARTIME=start_time[testset3_ind])
testset3 <- transform(testset3, STARTIME=as.character(STARTIME))
testset4 <- cbind(testset[testset4_ind,], STARTIME=start_time[testset4_ind])
testset4 <- transform(testset4, STARTIME=as.character(STARTIME))
#
n_test_obs <- c(dim(testset1)[1], dim(testset2)[1], dim(testset3)[1], dim(testset4)
## set up for train data
setwd('/Users/ZacharyHE/Documents/CU/COMS S 4771/Project/Data')
trainset1 <- read.csv('train1.csv', header=TRUE, colClasses=c('NULL', NA, 'NULL', N
names(trainset1)[6] <- 'STARTIME'
trainset1$STARTIME <- as.POSIXct(trainset1$STARTIME, tz='GMT', format='%Y-%m-%d %H:
trainset1$STARTIME <- format(trainset1$STARTIME, '%H:%M:%S')
trainset2 <- read.csv('train2.csv', header=TRUE, colClasses=c('NULL', NA, 'NULL', N
names(trainset2)[6] <- 'STARTIME'
```

```
trainset2$STARTIME <- as.POSIXct(trainset2$STARTIME, tz='GMT', format='%Y-%m-%d %H:
trainset2$STARTIME <- format(trainset2$STARTIME, '%H:%M:%S')
trainset3 <- read.csv('train3.csv', header=TRUE, colClasses=c('NULL', NA, 'NULL', N
names(trainset3)[6] <- 'STARTIME'
trainset3$STARTIME <- as.POSIXct(trainset3$STARTIME, tz='GMT', format='%Y-%m-%d %H:
trainset3$STARTIME <- format(trainset3$STARTIME, '%H:%M:%S')
trainset4 <- read.csv('train4.csv', header=TRUE, colClasses=c('NULL', NA, 'NULL', N
names(trainset4)[6] <- 'STARTIME'
trainset4$STARTIME <- as.POSIXct(trainset4$STARTIME, tz='GMT', format='%Y-%m-%d %H:
trainset4$STARTIME <- format(trainset4$STARTIME, '%H:%M:%S')
#

## set up for user defined function
setwd('/Users/ZacharyHE/Documents/CU/COMS S 4771/Project/Code')
source('TRAIN_PRE.R')
# generate trip time and drop NAs from train set
train1_pre <- GEN_TIME(trainset1$POLYLINE)
train1_trip_time <- train1_pre$TRIPTIME
train1_trip_na <- train1_pre$NAs
trainset1 <- cbind(trainset1[-train1_trip_na,], TRIPTIME=train1_trip_time)
train2_pre <- GEN_TIME(trainset2$POLYLINE)
train2_trip_time <- train2_pre$TRIPTIME
train2_trip_na <- train2_pre$NAs
trainset2 <- cbind(trainset2[-train2_trip_na,], TRIPTIME=train2_trip_time)
train3_pre <- GEN_TIME(trainset3$POLYLINE)
train3_trip_time <- train3_pre$TRIPTIME
train3_trip_na <- train3_pre$NAs
trainset3 <- cbind(trainset3[-train3_trip_na,], TRIPTIME=train3_trip_time)
train4_pre <- GEN_TIME(trainset4$POLYLINE)
train4_trip_time <- train4_pre$TRIPTIME
train4_trip_na <- train4_pre$NAs
trainset4 <- cbind(trainset4[-train4_trip_na,], TRIPTIME=train4_trip_time)
#
testset1$CALL_STAND <- paste0(testset1$CALL_TYPE, testset1$ORIGIN_STAND)
testset1 <- subset(testset1, select=-c(CALL_TYPE, ORIGIN_STAND))
testset2$CALL_STAND <- paste0(testset2$CALL_TYPE, testset2$ORIGIN_STAND)
testset2 <- subset(testset2, select=-c(CALL_TYPE, ORIGIN_STAND))
testset3$CALL_STAND <- paste0(testset3$CALL_TYPE, testset3$ORIGIN_STAND)
testset3 <- subset(testset3, select=-c(CALL_TYPE, ORIGIN_STAND))
testset4$CALL_STAND <- paste0(testset4$CALL_TYPE, testset4$ORIGIN_STAND)
testset4 <- subset(testset4, select=-c(CALL_TYPE, ORIGIN_STAND))
trainset1$CALL_STAND <- paste0(trainset1$CALL_TYPE, trainset1$ORIGIN_STAND)
trainset1 <- subset(trainset1, select=-c(CALL_TYPE, ORIGIN_STAND))
trainset2$CALL_STAND <- paste0(trainset2$CALL_TYPE, trainset2$ORIGIN_STAND)
trainset2 <- subset(trainset2, select=-c(CALL_TYPE, ORIGIN_STAND))
trainset3$CALL_STAND <- paste0(trainset3$CALL_TYPE, trainset3$ORIGIN_STAND)
trainset3 <- subset(trainset3, select=-c(CALL_TYPE, ORIGIN_STAND))
trainset4$CALL_STAND <- paste0(trainset4$CALL_TYPE, trainset4$ORIGIN_STAND)
```

```
trainset4 <- subset(trainset4, select=-c(CALL_TYPE, ORIGIN_STAND))
## match each test data with a specific train set
source('TRAIN_TEST_MATCH.R')
match_1 <- DATASET_MATCH(trainset1, testset1)
match_2 <- DATASET_MATCH(trainset2, testset2)
match_3 <- DATASET_MATCH(trainset3, testset3)
match_4 <- DATASET_MATCH(trainset4, testset4)
# combine all train sets and test sets into lists for later loops
train <- list(trainset1, trainset2, trainset3, trainset4)
test <- list(testset1, testset2, testset3, testset4)
remove(trainset1, trainset2, trainset3, trainset4)
remove(testset1, testset2, testset3, testset4)
match <- list(match_1, match_2, match_3, match_4)
#
## predict the trip time for test data
source('POLYLINE_FUN.R')
pred <- list()
for (i in 1:4){
  nobs_i <- n_test_obs[i]
  match_temp <- match[[i]]
  train_temp <- train[[i]]
  test_temp <- test[[i]]
  pre_temp <- vector()
  n_estobs_i <- vector()
  for (j in 1:nobs_i){
    ind_i_j <- match_temp[[j]]
    poly_i_j <- train_temp$POLYLINE[ind_i_j]
    poly_testi_j <- test_temp$POLYLINE[j]
    weight_na_i <- POLY_WEIGHT(poly_i_j, poly_testi_j)
    weight_i <- weight_na_i$WEIGHT
    nas_i <- weight_na_i$NAs
    n_estobs_i[j] <- length(weight_i)
    if (length(nas_i) >= 1) {
      testi_prej <- sum(train_temp$TRIPTIME[ind_i_j[-nas_i]] * weight_i)
    } else {
      testi_prej <- sum(train_temp$TRIPTIME[ind_i_j] * weight_i)
    }
    pre_temp[j] <- testi_prej
  }
  pred[[i]] <- data.frame('TRIP_ID'=test_temp$TRIP_ID, 'TRAVEL_TIME'=pre_temp, 'SA
}

test_pred <- rbind(pred[[1]], pred[[2]], pred[[3]], pred[[4]])
test_pred <- transform(test_pred, TRIP_ID=factor(TRIP_ID, levels=paste0('T', 1:327)
test_pred <- test_pred[order(test_pred$TRIP_ID),]
test_submi <- subset(test_pred, select=-SAMPLE_SIZE)
run_end_time <- Sys.time()
running_time <- run_end_time - run_start_time
```

```
#
## output prediction
setwd('/Users/ZacharyHE/Desktop')
write.csv(test_submi, 'Submission.csv', row.names=FALSE)
```