NYC Taxi Big data
Bonifica, Elaborazione,
Analisi e Data-mining con RevoScale R

Data Science case study for a Microsoft 213x Class Project

Lorenzo Negri, April 2018

Applications used: Microsoft R Open, Visual Studio, RevoScaleR libraries

Obbiettivi

Lo scopo di questo progetto è analizzare e applicare algoritmi di previsioni in ambiente Microsoft R con dati CSV di capienza superiore ai 6GB. Il set di dati (TLC Trip Record Data), che riguardano la telemetria, le tempistiche e le operazioni di pagamento, di tutti i clienti Taxi della città di New York, è reso pubblicamente disponibile dal dipartimento dei trasporti della città: NYC.gov Taxi and Limousine Commission (TLC). Per questo progetto verranno utilizzati i dati relativi ai primi sei mesi dell'anno 2016. Ogni file CSV grezzo ha una dimensione di circa 2GB. Il totale di 12GB di dati solitamente riempirebbero solo loro la metà della memoria disponibile su un singolo personal computer da ufficio. Un server può avere una capacità di memoria molto più grande, ma in Cloud Computing o su un server utilizzato da molti utenti contemporaneamente, R Studio può esaurire molto rapidamente la memoria. Questo documento ha lo scopo di illustrare le tecniche utilizzate per poter elaborare comodamente dati di grandi dimensioni utilizzando RevoScale R con MS R Open e Visual studio, praticamente ovunque.

Panoramica

Ogni record (dati riga) nel file mostra un viaggio in taxi "giallo" a New York, con le seguenti variabili (dati colonna) registrate: *VendorID,* Un codice che indica il provider TPEP che ha fornito il record (1 = Creative Mobile Technologies, LLC; 2 = VeriFone Inc); *tpep_pickup_datetime* & *tpep_dropoff_datetime*, la data e l'ora in cui il/i passeggero/i sono stati prelevati e lasciati (in formato m/d/yyyy h:mm); *Passenger_count*, il numero di passeggeri per viaggio; *Trip_distance*, la distanza percorsa (in miglia come registrato dal tassametro); *Pickup_longitude* & *Pickup_latitude*, *Dropoff_longitude* & *Dropoff_ latitude*, la latitudine e la longitudine

in cui i passeggeri sono stati prelevati e fatti scendere; le informazioni di pagamento quali il tipo di pagamento (*Payment_type*, 1=carta di credito, 2=contanti, 3=Nessun addebito, 4=Controversia, 5= sconosciuto, 6=scatto annullato) e il costo del viaggio (*Fare_amount*) suddiviso per tipologia di tariffa (*RateCodeID*, 1=tariffa standard, 2=JFK, 3=Newark, 4=Nassau o Westchester, 5=tariffa negoziata, 6=giro in gruppo); *Store_and_fwd_flag*, indicatore binario se il record di viaggio è stato trattenuto nella memoria del veicolo prima di inviarlo al venditore, ovvero "salva e inoltra", perché il veicolo non aveva una connessione al server (Y or N); *Extra*, extra e supplementi vari (include l'ora di punta +$ 0,50 e +$ 1 per la corsa notturna); MTA_tax, Tassa MTA di $ 0,50 che viene esclusa automaticamente quando la tipologia di cliente non è soggetta; *Improvement_surcharge*, una sovrattassa di miglioramento di $ 0,30 per la valutazione del viaggio; *Tip_amount*, la mancia: questo campo viene popolato automaticamente per le mance da carta di credito (le mance in contanti non sono incluse); *Tolls_amount*, importo totale di tutti i pedaggi pagati in viaggio; *Total_amount*, l'importo totale addebitato ai passeggeri (escluso le mance in contanti).

Il *dataset* è composto di un totale di circa settanta milioni di osservazioni (dati riga) con 19 variabili (dati colonna).

Esplorazione, Trasformazione e Integrazione Dati

L'esplorazione iniziale dei dati avviene caricando i file in R. Per preparare l'ambiente di lavoro in modo da poter analizzare i dati, abbiamo bisogno di un set di third-party packages. Il pacchetto `RevoScaleR` è preinstallato con Microsoft R Server (MRS) installato in Visual Studio, poiché non può essere scaricato da CRAN, tutti gli altri pacchetti mostrati di seguito invece sono pacchetti di terze parti che possono essere scaricati e installati da CRAN utilizzando il comando `install.packages`.

Inoltre, mentre carichiamo i pacchetti con la linea di comando, definiamo già alcune opzioni per rendere successivamente più semplice la visualizzazione di dati o i risultati.

```
options(max.print = 1000, scipen = 999, width = 100)
library(RevoScaleR)
rxOptions(reportProgress = 1) # riduce la qtà di output che RevoScaleR produce
library(dplyr)
options(dplyr.print_max = 200)
options(dplyr.width = Inf) # mostra tutte le colonne di un oggetto tbl_df
library(stringr)
library(lubridate)
library(rgeos) # spatial package
library(sp) # spatial package
```

```
library(maptools) # spatial package
library(ggmap)
library(ggplot2)
library(gridExtra) # per mettere insieme i grafici
library(ggrepel) # evitare la sovrapposizione di testo nei grafici
library(tidyr)
library(seriation) # pacchetto per il riordino di una distance matrix
```

Carico adesso in R le prime 1000 righe dei dati utilizzando la funzione `read.table`. E per evitare conversioni di fattori non necessarie, avendo già esaminato le variabili, assegno le tipologie di classi alle colonne archiviandole in un oggetto chiamato `col_classes` che poi passeremo attraverso `read.table`.

```
col_classes <- c('VendorID' = "factor",
                 'tpep_pickup_datetime' = "character",
                 'tpep_dropoff_datetime' = "character",
                 'passenger_count' = "integer",
                 'trip_distance' = "numeric",
                 'pickup_longitude' = "numeric",
                 'pickup_latitude' = "numeric",
                 'RateCodeID' = "factor",
                 'store_and_fwd_flag' = "factor",
                 'dropoff_longitude' = "numeric",
                 'dropoff_latitude' = "numeric",
                 'payment_type' = "factor",
                 'fare_amount' = "numeric",
                 'extra' = "numeric",
                 'mta_tax' = "numeric",
                 'tip_amount' = "numeric",
                 'tolls_amount' = "numeric",
                 'improvement_surcharge' = "numeric",
                 'total_amount' = "numeric")
```

È buona norma caricare un piccolo campione di dati come `data.frame` in R su cui eseguire delle verifiche. Quando vogliamo applicare una funzione ai dati XDF, possiamo prima applicarla a `data.frame` dove è più facile e veloce individuare errori prima di applicarlo a tutto il dataset.

```
input_csv <- 'yellow_tripdata_2016-01.csv'
# prendiamo una parte dei dati e li carichiamo come data.frame (per fare test)
nyc_sample_df <- read.csv(input_csv, nrows = 1000, colClasses = col_classes)
head(nyc_sample_df)
```

```
VendorID tpep_pickup_datetime tpep_dropoff_datetime passenger_count trip_distance
1        2  2016-01-01 00:00:00   2016-01-01 00:00:00               2          1.10
2        2  2016-01-01 00:00:00   2016-01-01 00:00:00               5          4.90
3        2  2016-01-01 00:00:00   2016-01-01 00:00:00               1         10.54
4        2  2016-01-01 00:00:00   2016-01-01 00:00:00               1          4.75
5        2  2016-01-01 00:00:00   2016-01-01 00:00:00               3          1.76
6        2  2016-01-01 00:00:00   2016-01-01 00:18:30               2          5.52
  pickup_longitude pickup_latitude RatecodeID store_and_fwd_flag dropoff_longitude
1        -73.99037        40.73470          1                  N         -73.98184
2        -73.98078        40.72991          1                  N         -73.94447
3        -73.98455        40.67957          1                  N         -73.95027
4        -73.99347        40.71899          1                  N         -73.96224
5        -73.96062        40.78133          1                  N         -73.97726
6        -73.98012        40.74305          1                  N         -73.91349
  dropoff_latitude payment_type fare_amount extra mta_tax tip_amount tolls_amount
1         40.73241            2         7.5   0.5     0.5          0            0
2         40.71668            1        18.0   0.5     0.5          0            0
3         40.78893            1        33.0   0.5     0.5          0            0
4         40.65733            2        16.5   0.0     0.5          0            0
5         40.75851            2         8.0   0.0     0.5          0            0
6         40.76314            2        19.0   0.5     0.5          0            0
  improvement_surcharge total_amount
1                   0.3          8.8
2                   0.3         19.3
3                   0.3         34.3
4                   0.3         17.3
5                   0.3          8.8
6                   0.3         20.3
```

Possiamo quindi vedere le prime righe dei dati e sembra che tutto sia stato caricato correttamente.

Ora carico tutti i dati usando MRS. MRS ha due modi per gestire i file:

1. Può funzionare direttamente con i file, il che significa che può leggere e scrivere direttamente in file.
2. Può convertire i file in un formato chiamato XDF (XDF sta per *external data frame*).

Scelgo la seconda opzione. Per convertire file in XDF, usiamo la funzione `rxImport`. Ponendo `append="rows",` possiamo anche combinare più file in un singolo file XDF.

```
input_xdf <- 'yellow_tripdata_2016.xdf'
library(lubridate)
most_recent_date <- ymd("2016-07-01") # il giorno dei mesi è irrilevante

st <- Sys.time()
for(ii in 1:6) { # i dati di ogni mese aggiunti ai dati del primo mese
  file_date <- most_recent_date - months(ii)
  input_csv <- sprintf('yellow_tripsample_%s.csv', substr(file_date, 1, 7))
  append <- if (ii == 1) "none" else "rows"
  rxImport(input_csv, input_xdf, colClasses = col_classes, overwrite = TRUE,
append = append)
  print(input_csv)
}
Sys.time() - st # memorizza il tempo necessario per importare i dati
```

```
Rows Processed: 10906858
[1] "yellow_tripdata_2016-01.csv"
Rows Processed: 11382049
[1] "yellow_tripdata_2016-02.csv"
Rows Processed: 12210952
[1] "yellow_tripdata_2016-03.csv"
Rows Processed: 11934338
[1] "yellow_tripdata_2016-04.csv"
Rows Processed: 11836853
[1] "yellow_tripdata_2016-05.csv"
Rows Processed: 11135470
[1] "yellow_tripdata_2016-06.csv"
```

Ho usato il pacchetto *lubridate* per semplificare in modo automatico il loop tra i file CSV.

Summary Statistics

Possiamo ora vedere un riepilogo delle statistiche utilizzando la funzione `rxSummary` con `nyc_xdf`. La funzione `rxSummary` si applica con la classica notazione utilizzata da molte funzioni R. In questo caso, per esempio la formula ~ *fare_amount* significa che vogliamo vedere un riepilogo solo per quella colonna, aggiungendo con + altre variabili, possiamo aggiungere tutte le voci che ci interessano. Proprio come la funzione di riepilogo in R base, `rxSummary` ci mostrerà un output diverso a seconda del tipo di colonna. In questo caso vado ad inserire tutte le colonne numeriche che interessano e una categorica (*payment_type*).

```
input_xdf <- 'yellow_tripdata_2016.xdf'
nyc_xdf <- RxXdfData(input_xdf)
system.time(
  rxsum_xdf <- rxSummary(~passenger_count
                         + trip_distance
                         + fare_amount
                         + extra
                         + mta_tax
                         + tip_amount
                         + tolls_amount
                         + improvement_surcharge
                         + total_amount
                         + payment_type, nyc_xdf) # statistical summaries
)
rxsum_xdf
```

```
Rows Processed: 3467953
   user  system elapsed
   0.00    0.00    0.92
Call:
rxSummary(formula = ~passenger_count + trip_distance + fare_amount +
    extra + mta_tax + tip_amount + tolls_amount + improvement_surcharge +
    total_amount + payment_type, data = nyc_xdf)

Summary Statistics Results for: ~passenger_count + trip_distance + fare_amount +
    extra + mta_tax + tip_amount + tolls_amount + improvement_surcharge +
    total_amount + payment_type
Data: nyc_xdf (RxXdfData Data Source)
File name: yellow_tripdata_2016.xdf
Number of valid observations: 3467953

 Name                Mean       StdDev       Min    Max         ValidObs MissingObs
 passenger_count     1.6602820    1.31044643    0.0        9.00 3467953  0
 trip_distance       6.5103520 6445.86584647    0.0 12000004.50 3467953  0
 fare_amount         12.8835511  11.39395462 -376.0     2550.20 3467953  0
```

```
 extra                 0.3324486    0.43799020   -4.5       50.01 3467953  0
 mta_tax               0.4974139    0.03861298   -1.0        3.00 3467953  0
 tip_amount            1.8009263    2.64878399  -35.0      998.14 3467953  0
 tolls_amount          0.3159671    1.58133619  -10.5      613.50 3467953  0
 improvement_surcharge 0.2996687    0.01449831   -0.3       11.64 3467953  0
 total_amount         16.1304688   13.93443837 -376.3     2551.00 3467953  0

Category Counts for payment_type
Number of categories: 4
Number of valid observations: 3467953
Number of missing observations: 0

 payment_type Counts
 2            1148718
 1            2300794
 3              13696
 4               4745
```

Possiamo notare come vi siano degli errori nelle rilevazioni, ad esempio la distanza massima percorsa per una corsa taxi di milioni di miglia, oppure la tariffa di una corsa in dollari in negativo. Probabilmente sono operazioni effettuate manualmente dai taxisti, oppure errori del sistema di telemetria e GPS. Salta all'occhio anche la mancia massima di $ 998.14 e il costo del pedaggio totale per una corsa di $ 613.50.

I metodi di pagamento più utilizzati sono stati 1=carta di credito e 2=contanti. Quasi 14000 di nessun addebito e quasi 5000 controversie in sei mesi di servizio taxi a New York.

Nel set di dati dei Taxi di New York, certi outlier posso essere imputati a: (1) Un passeggero potrebbe prendere un taxi e usarlo tutto il giorno per fare più commissioni, chiedendo all'autista di aspettarlo. (2) Un passeggero potrebbe voler dare 5 dollari in di mancia e l'autista premendo per sbaglio due volte 5, aggiunge 55 dollari a un viaggio che ne costa 40 dollari. (3) Un passeggero potrebbe litigare con un autista e andarsene senza pagare. (4) Viaggi con più passeggeri potrebbero avere una persona che paga per tutti o ognuno paga per se stesso, con alcuni che pagano con una carta e altri che usano denaro contante. (5) Un autista può accidentalmente mantenere il contatore in funzione

dopo aver fatto scendere qualcuno. (6) Gli errori di registrazione della macchina possono comportare l'assenza di dati o di dati errati. In tutti questi casi un outlier potrebbe essere rumore per un'analisi e un punto di interesse per qualcos'altro.

Bonifica e integrazione dei dati

Le attività di preparazione dei dati comunemente riguardano la gestione dei valori mancanti, gestione dei valori anomali, determinazione del livello di granularità dei dati, ad esempio le variabili temporali possono essere in secondi, minuti o ore, ecc. - decidere quali funzionalità aggiungere o estrarre in base alle funzionalità esistenti per rendere l'analisi più interessante o più facile da interpretare.

Come prima cosa è buona idea controllare le tipologie delle variabili e accertarsi che nulla di anomalo venga riscontrato. Oltre alla tipologia dei valori in colonna, la funzione `rxGetInfo` mostra anche i minimi e i massimi per tutte le variabili, che possono essere utili ad identificare ulteriormente i valori anomali ed eseguire controlli di integrità. Con l'argomento `numRows = 10`, possiamo esaminare anche le prime 10 righe dei dati.

```
# tipologia di colonna e le prime 10 righe dei dati #
rxGetInfo(nyc_xdf, getVarInfo = TRUE, numRows = 10)
```

```
File name: C:\Data\NYC_taxi\01\yellow_tripdata_2016.xdf
Number of observations: 3467953
Number of variables: 20
Number of blocks: 6
Compression type: zlib
Variable information:
Var 1: VendorID 2 factor levels: 2 1
Var 2: tpep_pickup_datetime, Type: character
Var 3: tpep_dropoff_datetime, Type: character
Var 4: passenger_count, Type: integer, Low/High: (0, 9)
Var 5: trip_distance, Type: numeric, Low/High: (0.0000, 12000004.5000)
Var 6: pickup_longitude, Type: numeric, Low/High: (-121.9333, 0.0000)
Var 7: pickup_latitude, Type: numeric, Low/High: (0.0000, 53.4046)
Var 8: RatecodeID, Type: integer, Low/High: (1, 99)
```

```
Var 9: store_and_fwd_flag 2 factor levels: N Y
Var 10: dropoff_longitude, Type: numeric, Low/High: (-121.9333, 0.0000)
Var 11: dropoff_latitude, Type: numeric, Low/High: (0.0000, 50.7979)
Var 12: payment_type  4 factor levels: 2 1 3 4
Var 13: fare_amount, Type: numeric, Low/High: (-376.0000, 2550.2000)
Var 14: extra, Type: numeric, Low/High: (-4.5000, 50.0100)
Var 15: mta_tax, Type: numeric, Low/High: (-1.0000, 3.0000)
Var 16: tip_amount, Type: numeric, Low/High: (-35.0000, 998.1400)
Var 17: tolls_amount, Type: numeric, Low/High: (-10.5000, 613.5000)
Var 18: improvement_surcharge, Type: numeric, Low/High: (-0.3000, 11.6400)
Var 19: total_amount, Type: numeric, Low/High: (-376.3000, 2551.0000)
Var 20: u, Type: numeric, Low/High: (0.0000, 0.0500)
Data (5 rows starting with row 1):
VendorID tpep_pickup_datetime tpep_dropoff_datetime passenger_count trip_distance
1        2  2016-01-01 00:00:00   2016-01-01 00:00:00               2          1.10
2        2  2016-01-01 00:00:00   2016-01-01 00:00:00               5          4.90
3        2  2016-01-01 00:00:00   2016-01-01 00:00:00               1         10.54
4        2  2016-01-01 00:00:00   2016-01-01 00:00:00               1          4.75
5        2  2016-01-01 00:00:00   2016-01-01 00:00:00               3          1.76
  pickup_longitude pickup_latitude RatecodeID store_and_fwd_flag dropoff_longitude
1        -73.99037        40.73470          1                  N         -73.98184
2        -73.98078        40.72991          1                  N         -73.94447
3        -73.98455        40.67957          1                  N         -73.95027
4        -73.99347        40.71899          1                  N         -73.96224
5        -73.96062        40.78133          1                  N         -73.97726
  dropoff_latitude payment_type fare_amount extra mta_tax tip_amount tolls_amount
1         40.73241            2         7.5   0.5     0.5          0            0
2         40.71668            1        18.0   0.5     0.5          0            0
3         40.78893            1        33.0   0.5     0.5          0            0
4         40.65733            2        16.5   0.0     0.5          0            0
5         40.75851            2         8.0   0.0     0.5          0            0
  improvement_surcharge total_amount
1                   0.3          8.8
2                   0.3         19.3
3                   0.3         34.3
```

```
4                   0.3         17.3
5                   0.3          8.8
```

Una volta che i dati sono stati importati e controllati, possiamo iniziare a pensare alle interessanti / rilevanti caratteristiche che rientrano nella nostra analisi. L'obiettivo è principalmente esplorativo: vogliamo raccontare una storia basata sui dati. In questo progetto, qualsiasi informazione contenuta nei dati può essere utile. Inoltre, nuove informazioni (o caratteristiche) possono essere estratte da dati esistenti. Non è solo importante pensare a quali funzionalità estrarre, ma anche a come devono essere categorizzate le colonne, in modo che le analisi successive siano eseguite in modo appropriato e alcune anomalia vengano escluse. Come una semplice trasformazione, ad esempio, per estrarre la percentuale di mancia che i passeggeri hanno lasciato per il viaggio.

La trasformazione verrà calcolata su di una nuova colonna denominata *tip_percent* basata sulla seguente logica:

- SE *fare_amount* > zero & *tip_amount* < *fare_amount*, calcoleremo (*tip_amount* / *fare_amount*)* 100. E lo arrotonderemo a un numero intero, che sarà infine il valore di *tip_percent*.
- SE la condizione precedente non è soddisfatta per qualche motivo. Assegneremo NA al valore di *tip_percent*.

```
rxDataStep(nyc_xdf, nyc_xdf,
           transforms = list(tip_percent = ifelse(fare_amount > 0 &
                                tip_amount < fare_amount,
                                round(tip_amount * 100 / fare_amount, 0), NA)),
           overwrite = TRUE)
rxSummary(~tip_percent, nyc_xdf)
```

```
Call:
rxSummary(formula = ~tip_percent, data = nyc_xdf)

Summary Statistics Results for: ~tip_percent
Data: nyc_xdf (RxXdfData Data Source)
File name: yellow_tripdata_2016.xdf
Number of valid observations: 3467953
```

```
 Name        Mean     StdDev   Min Max ValidObs MissingObs
 tip_percent 13.96739 11.88536 0   99  3462221  5732
```

Possiamo vedere che in media i clienti hanno pagato circa il 14% del valore della corsa in mancia. La deviazione standard è leggermente alta. Possiamo vedere il minimo e il massimo valore percentuale. E possiamo notare che abbiamo parecchi valori mancanti. Questi sono i valori mancanti che vengono generati dalla nostra formula, e che vanno a escludere quei valori anomali che volevamo evitare.

Ora vogliamo l'interazione tra mese e anno, e ottenere dei conteggi da esso estrapolando i dati dalla colonna che ci interessa. Userò la colonna *pickup_datetime*, che è la prima delle colonne con valori di tipo *character*.

```
rxCrossTabs(~ month:year, nyc_xdf,
            transforms = list(
            date = ymd_hms(tpep_pickup_datetime),
            year = factor(year(date), levels = 2014:2016),
            month = factor(month(date), levels = 1:12)),
            transformPackages = "lubridate")
```

```
Call:
rxCrossTabs(formula = ~month:year, data = nyc_xdf, transforms = list(date =
ymd_hms(tpep_pickup_datetime),
    year = factor(year(date), levels = 2014:2016), month = factor(month(date),
        levels = 1:12)), transformPackages = "lubridate")

Cross Tabulation Results for: ~month:year
Data: nyc_xdf (RxXdfData Data Source)
File name: yellow_tripdata_2016.xdf
Number of valid observations: 3467953
Number of missing observations: 0
Statistic: counts

month:year (counts):
     year
```

```
month 2014 2015   2016
   1     0    0 544498
   2     0    0 568348
   3     0    0 610255
   4     0    0 596987
   5     0    0 591985
   6     0    0 555880
   7     0    0      0
   8     0    0      0
   9     0    0      0
   10    0    0      0
   11    0    0      0
   12    0    0      0
```

Per questo progetto sui taxi di New York, siamo interessati a confrontare i viaggi in base al giorno della settimana e all'ora del giorno. Queste due colonne non esistono ancora, ma possiamo estrarle dalla data e ora di partenza e dalla data e ora di arrivo. Per estrarre le funzionalità di cui sopra, utilizziamo il pacchetto *lubridate*, che ha funzioni utili per gestire le colonne di data e ora. Per eseguire queste trasformazioni, usiamo una funzione di trasformazione chiamata `xforms`.

```
xforms <- function(data)
{ # funzione di trasformazione per l'estrazione di parametri di data e ora

  weekday_labels <- c('Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat')
  cut_levels <- c(1, 5, 9, 12, 16, 18, 22)
  hour_labels <- c('1AM-5AM', '5AM-9AM', '9AM-12PM', '12PM-4PM',
                   '4PM-6PM', '6PM-10PM', '10PM-1AM')

  pickup_datetime <- ymd_hms(data$tpep_pickup_datetime, tz = "UTC")
  pickup_hour <- addNA(cut(hour(pickup_datetime), cut_levels))
  pickup_dow <- factor(wday(pickup_datetime),
                       levels = 1:7,
                       labels = weekday_labels)
  levels(pickup_hour) <- hour_labels
```

```
  dropoff_datetime <- ymd_hms(data$tpep_dropoff_datetime, tz = "UTC")
  dropoff_hour <- addNA(cut(hour(dropoff_datetime), cut_levels))
  dropoff_dow <- factor(wday(dropoff_datetime),
                        levels = 1:7,
                        labels = weekday_labels)
  levels(dropoff_hour) <- hour_labels

  data$pickup_hour <- pickup_hour
  data$pickup_dow <- pickup_dow
  data$dropoff_hour <- dropoff_hour
  data$dropoff_dow <- dropoff_dow
  data$trip_duration <- as.integer(as.duration(dropoff_datetime-pickup_datetime))

  data
}
```

Prima di applicare la trasformazione a tutti i dati, di solito è una buona idea testarlo e assicurarsi che funzioni. A tal fine, mettiamo da parte un campione dei dati come `data.frame`. L'esecuzione della funzione di trasformazione su `nyc_sample_df` dovrebbe restituire i dati originali con le nuove colonne.

```
library(lubridate)
Sys.setenv(TZ = "US/Eastern") # non importante per questo df
head(xforms(nyc_sample_df)) # testa la funzione sul data.frame
```

```
VendorID tpep_pickup_datetime tpep_dropoff_datetime passenger_count trip_distance
1        2  2016-01-01 00:00:00   2016-01-01 00:00:00               2          1.10
2        2  2016-01-01 00:00:00   2016-01-01 00:00:00               5          4.90
3        2  2016-01-01 00:00:00   2016-01-01 00:00:00               1         10.54
4        2  2016-01-01 00:00:00   2016-01-01 00:00:00               1          4.75
5        2  2016-01-01 00:00:00   2016-01-01 00:00:00               3          1.76
6        2  2016-01-01 00:00:00   2016-01-01 00:18:30               2          5.52
  pickup_longitude pickup_latitude RatecodeID store_and_fwd_flag dropoff_longitude
```

```
1        -73.99037        40.73470          1                  N         -73.98184
2        -73.98078        40.72991          1                  N         -73.94447
3        -73.98455        40.67957          1                  N         -73.95027
4        -73.99347        40.71899          1                  N         -73.96224
5        -73.96062        40.78133          1                  N         -73.97726
6        -73.98012        40.74305          1                  N         -73.91349
  dropoff_latitude payment_type fare_amount extra mta_tax tip_amount tolls_amount
1         40.73241            2         7.5   0.5     0.5          0            0
2         40.71668            1        18.0   0.5     0.5          0            0
3         40.78893            1        33.0   0.5     0.5          0            0
4         40.65733            2        16.5   0.0     0.5          0            0
5         40.75851            2         8.0   0.0     0.5          0            0
6         40.76314            2        19.0   0.5     0.5          0            0
  improvement_surcharge total_amount pickup_hour pickup_dow dropoff_hour
1                   0.3          8.8    10PM-1AM        Fri     10PM-1AM
2                   0.3         19.3    10PM-1AM        Fri     10PM-1AM
3                   0.3         34.3    10PM-1AM        Fri     10PM-1AM
4                   0.3         17.3    10PM-1AM        Fri     10PM-1AM
5                   0.3          8.8    10PM-1AM        Fri     10PM-1AM
6                   0.3         20.3    10PM-1AM        Fri     10PM-1AM
  dropoff_dow trip_duration
1         Fri             0
2         Fri             0
3         Fri             0
4         Fri             0
5         Fri             0
6         Fri          1110
```

Tutto sembra funzionare bene. Ciò non garantisce che l'esecuzione della funzione di trasformazione sull'intero set di dati abbia esito positivo, ma rende meno probabile il fallimento soprattutto se l'operazione sui dati richiede diverso tempo di elaborazione. Se la trasformazione funziona sul campione `data.frame`, come sopra, ma fallisce quando la eseguiamo sull'intero *dataset*, di solito è a causa di qualcosa nel set di dati che causa il fallimento (come i valori mancanti) che non erano presenti nei dati di esempio. Ora eseguiamo la trasformazione su tutto il set di dati.

```
st <- Sys.time()
rxDataStep(nyc_xdf, nyc_xdf, overwrite = TRUE, transformFunc = xforms, transformPackages = "lubridate")
Sys.time() - st
```

```
Time difference of 39.28945 secs
```

Esamino le nuove colonne create per assicurarmiche la trasformazione sia più o meno funzionante. Uso la funzione `rxSummary` per ottenere alcuni riepiloghi statistici dei dati. La funzione `rxSummary` è simile alla funzione di `summary` R base (a parte il fatto che il `summary` funziona solo su un `data.frame`):

- Fornisce riepiloghi per colonne numeriche (eccetto per percentili, per i quali uso la funzione `rxQuantile`).
- Fornisce i conteggi per ogni livello delle factor columns.

Utilizzo la stessa notazione formula usata da molte altre funzioni di modellazione in R o di plotting, per specificare in quali colonne vogliamo i riepiloghi. Ad esempio, qui vogliamo vedere i riepiloghi per *pickup_our* e *pickup_dow* (entrambi i factors) e *trip_duration* (numerico, in secondi).

```
rxs1 <- rxSummary( ~ pickup_hour + pickup_dow + trip_duration, nyc_xdf)
# possiamo aggiungere una colonna per le proporzioni accanto ai conteggi
rxs1$categorical <- lapply(rxs1$categorical, function(x) cbind(x, prop = round(prop.table(x$Counts), 2)))
rxs1
```

```
Call:
rxSummary(formula = ~pickup_hour + pickup_dow + trip_duration,
    data = nyc_xdf)

Summary Statistics Results for: ~pickup_hour + pickup_dow + trip_duration
Data: nyc_xdf (RxXdfData Data Source)
File name: yellow_tripdata_2016.xdf
Number of valid observations: 69406520
```

```
 Name          Mean     StdDev   Min         Max      ValidObs MissingObs
 trip_duration 933.9168 119243.5 -631148790 11538803 69406520 0

Category Counts for pickup_hour
Number of categories: 7
Number of valid observations: 69406520
Number of missing observations: 0

 pickup_hour Counts   prop
 1AM-5AM      3801430 0.05
 5AM-9AM     10630653 0.15
 9AM-12PM     9765429 0.14
 12PM-4PM    13473045 0.19
 4PM-6PM      7946899 0.11
 6PM-10PM    16138968 0.23
 10PM-1AM     7650096 0.11

Category Counts for pickup_dow
Number of categories: 7
Number of valid observations: 69406520
Number of missing observations: 0

 pickup_dow Counts   prop
 Sun         9267881 0.13
 Mon         8938785 0.13
 Tue         9667525 0.14
 Wed         9982769 0.14
 Thu        10398738 0.15
 Fri        10655022 0.15
 Sat        10495800 0.15
```

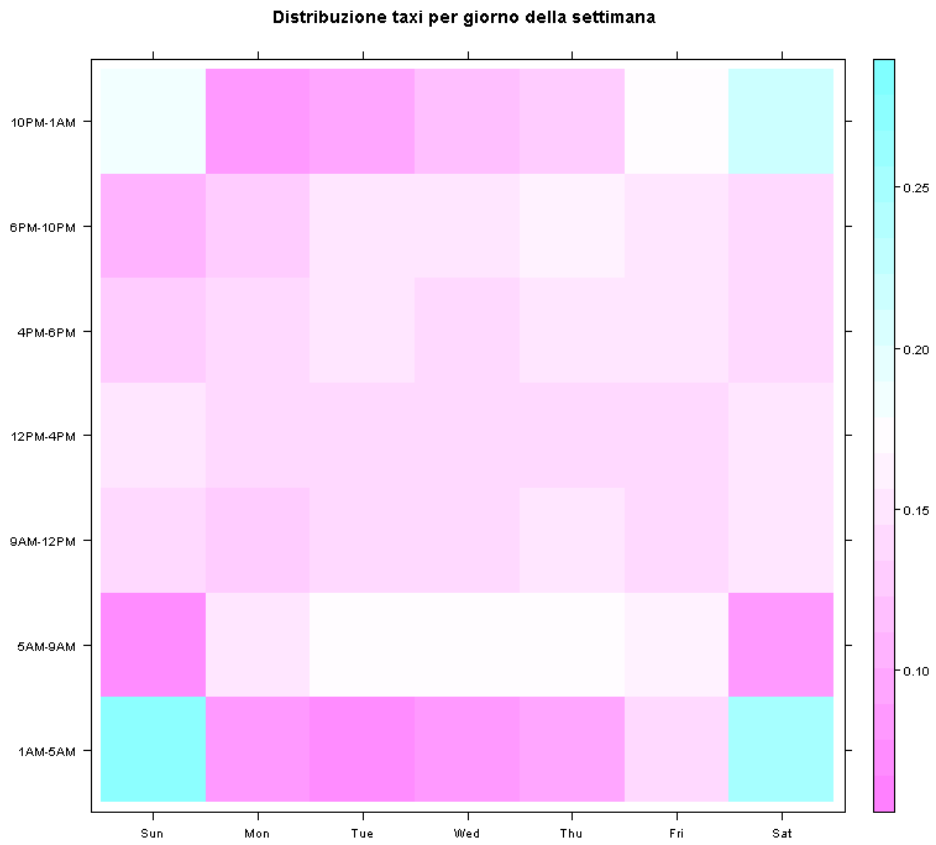
Si possono ottenere dati per ciascuna combinazione dei livelli delle colonne dei due fattori, invece dei soli dati individuali.

```
rxs2 <- rxSummary( ~ pickup_dow:pickup_hour, nyc_xdf)
rxs2 <- tidyr::spread(rxs2$categorical[[1]],
                      key = 'pickup_hour',
                      value = 'Counts')
row.names(rxs2) <- rxs2[ , 1]
rxs2 <- as.matrix(rxs2[ , -1])
rxs2
```

```
    1AM-5AM 5AM-9AM 9AM-12PM 12PM-4PM 4PM-6PM 6PM-10PM 10PM-1AM
Sun 1040233  740157  1396409  1980752 1032434  1697529  1380367
Mon  304474 1630951  1268326  1838143 1133728  2096219   666944
Tue  278407 1840134  1382381  1882356 1151837  2390506   741904
Wed  313809 1854757  1417953  1880896 1142071  2508618   864665
Thu  354646 1871828  1428985  1922502 1165535  2634023  1021219
Fri  553159 1766482  1406979  1922542 1173163  2516285  1316412
Sat  956702  926344  1464396  2045854 1148131  2295788  1658585
```

Ma in questo caso, i conteggi individuali non sono così utili come le proporzioni degli stessi, e per fare un confronto tra diversi giorni della settimana, vorrei che le proporzioni siano basate sui totali di ogni colonna, non sull'intera tabella. Posso quindi visualizzare visivamente le proporzioni usando la funzione `levelplot`.

```
levelplot(prop.table(rxs2, 2),
          cuts = 4,
          xlab = "",
          ylab = "",
          main = "Distribuzione taxi per giorno della settimana")
```



Il grafico mostra alcune informazioni interessanti:

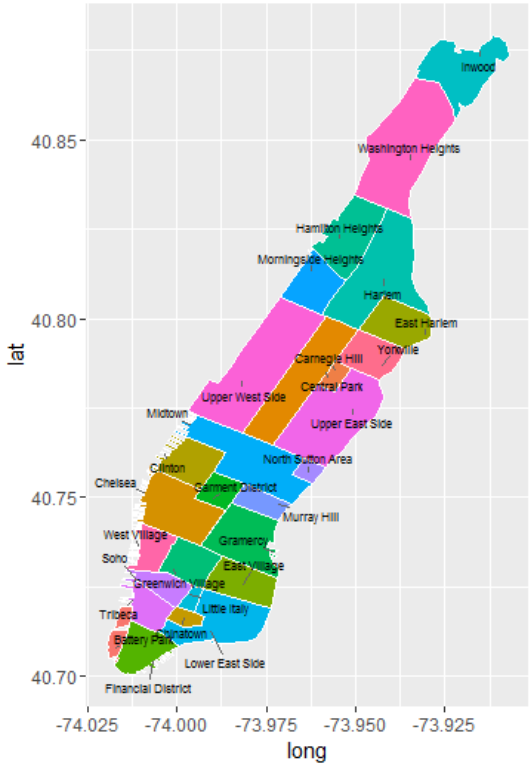
1. La mattina presto (tra le 5:00 e le 9:00) le corse in taxi sono prevedibilmente al minimo durante il fine settimana e leggermente basse anche il lunedì.
2. Durante l'orario lavorativo (tra le 9:00 e le 18:00), avviene circa la stessa percentuale di viaggi in taxi (dal 42 al 45% circa) per ogni giorno della settimana, compresi i fine settimana. In altre parole, indipendentemente da quale sia il giorno della settimana, un po' meno della metà di tutti i viaggi avviene tra le 9:00 e le 18:00.
3. Possiamo vedere un picco di viaggi in taxi tra le 18:00 e le 22:00 il giovedì e il venerdì sera, e un picco tra le 22:00 e l'1:00 di venerdì e soprattutto il sabato sera. Viaggi in taxi tra 1AM e 5AM con un picco al sabato (rientri dal venerdì sera) e ancor più la domenica (rientri del sabato sera). Cadono poi bruscamente negli altri giorni, ma leggermente si riprendono al venerdì (in tarda serata). In altre parole, molte persone escono di giovedì ma non restano fuori fino a tardi, altre persone escono il venerdì e rimangono anche più tardi, ma il sabato è il giorno in cui la maggior parte delle persone sceglie per una serata di ore piccole.

Ora aggiungo un altro set di dati: i quartieri di partenza e arrivo. Ottenere informazioni sui quartieri da longitudine e latitudine non è qualcosa che possiamo codificare facilmente, possiamo però usare alcuni pacchetti GIS e uno *shapefile* (per gentile concessione di Zillow). Uno *shapefile* è un file che contiene informazioni geografiche al suo interno, incluse informazioni sui confini che separano le aree geografiche. Il file *ZillowNeighborhoods-NY.shp* contiene informazioni sui quartieri di New York.

Iniziamo disegnando una mappa dei quartieri di Manhattan, così possiamo vedere i confini del quartiere e familiarizzare con i loro nomi.

```
library(rgeos)
library(sp)
library(maptools)
library(rgdal)

nyc_shapefile<-readShapePoly('ZillowNeighborhoods-NY/ZillowNeighborhoods-NY.shp')
mht_shapefile <- subset(nyc_shapefile, str_detect(CITY, 'New York City-Manhattan'))

mht_shapefile@data$id <- as.character(mht_shapefile@data$NAME)
mht.points <- fortify(gBuffer(mht_shapefile,
                              byid = TRUE,
                              width = 0), region = "NAME")
mht.df <- inner_join(mht.points, mht_shapefile@data, by = "id")

library(dplyr)
mht.cent <- mht.df %>%
  group_by(id) %>%
  summarize(long = median(long), lat = median(lat))

library(ggrepel)
ggplot(mht.df, aes(long, lat, fill = id)) +
    geom_polygon() +
    geom_path(color = "white") +
    coord_equal() +
```

```
    theme(legend.position = "none") +
    geom_text_repel(aes(label = id), data = mht.cent, size = 2)
```



Possiamo visualizzare quindi la mappa di Manhattan, possiamo vedere i quartieri che sono codificati in diversi colori con il nome del quartiere che mostra più o meno il quartiere stesso. Tutte queste informazioni erano codificate nello *shapefile* che ho appena caricato. In questo caso, abbiamo usato le informazioni per disegnare una mappa. Adesso bisogna codificare una trasformazione complessa che andrà a interagire con le coordinate per la partenza e arrivo, che abbiamo nei nostri dati. E troverà così il quartiere corrispondente al *pickup* e il *drop-off*.

Ho quindi archiviato lo *shapefile* per Manhattan in un oggetto chiamato `mht_shapefile`, che abbiamo usato per tracciare una mappa dei quartieri di Manhattan. Vediamo ora come possiamo usare lo stesso *shapefile* per individuare i quartieri di *pick-up* e *drop-off* per ogni corsa. Possiamo farlo con un `data.frame` (utilizzando il *sample* dei dati dei taxi di New York che abbiamo preso in precedenza).

Per aggiungere la colonna dei quartieri di *pick-up* a `nyc_sample_df`, dobbiamo procedere come segue: - sostituire gli `NA` in *pickup_latitude* e *pickup_longitude* con `0`, altrimenti otterremo un errore – utilizzando poi la funzione coordinate per specificare

che le due colonne precedenti rappresentano le coordinate geografiche in dati: uso poi la funzione `over` per ottenere quartieri in base alle coordinate specificate sopra, che restituirà un `data.frame` con informazioni sul quartiere – verranno allegati poi i risultati ai dati originali usando `cbind` dopo aver assegnato loro i nomi corretti.

```
# prendere solo le colonne con coordinate e sostituisce NA con 0
data_coords <- transmute(nyc_sample_df,
  long = ifelse(is.na(pickup_longitude), 0, pickup_longitude),
  lat = ifelse(is.na(pickup_latitude), 0, pickup_latitude)
)

# specifichiamo le colonne che corrispondono alle coordinate
coordinates(data_coords) <- c('long', 'lat')
head(data_coords)

# restituisce il nome dei quartieri in base alle coordinate
nhoods <- over(data_coords, mht_shapefile)
head(nhoods)

# rinomina le colonne in nhoods
names(nhoods) <- paste('pickup', tolower(names(nhoods)), sep = '_')
# combina le informazioni sul quartiere con i dati originali
nyc_sample_df <- cbind(nyc_sample_df, nhoods[, grep('name|city', names(nhoods))])
head(nyc_sample_df)
```

```
SpatialPoints:
          long      lat
[1,] -73.99406 40.71999
[2,] -73.98166 40.77374
[3,] -73.96375 40.77116
[4,] -73.97602 40.74456
[5,] -73.94185 40.82948
[6,] -73.97367 40.78452
Coordinate Reference System (CRS) arguments: NA
  STATE   COUNTY                    CITY            NAME REGIONID              id
```

```
1    NY New York New York City-Manhattan Lower East Side   270875 Lower East Side
2    NY New York New York City-Manhattan Upper West Side   270958 Upper West Side
3    NY New York New York City-Manhattan Upper East Side   270957 Upper East Side
4    NY New York New York City-Manhattan        Gramercy   273860        Gramercy
5    NY New York New York City-Manhattan          Harlem   195267          Harlem
6    NY New York New York City-Manhattan Upper West Side   270958 Upper West Side
  VendorID tpep_pickup_datetime tpep_dropoff_datetime passenger_count
1        2  2016-01-01 00:00:00   2016-01-01 00:00:00               2
2        2  2016-01-01 00:00:00   2016-01-01 00:00:00               5
3        2  2016-01-01 00:00:00   2016-01-01 00:00:00               1
4        2  2016-01-01 00:00:00   2016-01-01 00:00:00               1
5        2  2016-01-01 00:00:00   2016-01-01 00:00:00               3
6        2  2016-01-01 00:00:00   2016-01-01 00:18:30               2
  trip_distance pickup_longitude pickup_latitude RatecodeID store_and_fwd_flag
1          1.10        -73.99037        40.73470          1                  N
2          4.90        -73.98078        40.72991          1                  N
3         10.54        -73.98455        40.67957          1                  N
4          4.75        -73.99347        40.71899          1                  N
5          1.76        -73.96062        40.78133          1                  N
6          5.52        -73.98012        40.74305          1                  N
  dropoff_longitude dropoff_latitude payment_type fare_amount extra mta_tax
1         -73.98184         40.73241            2         7.5   0.5     0.5
2         -73.94447         40.71668            1        18.0   0.5     0.5
3         -73.95027         40.78893            1        33.0   0.5     0.5
4         -73.96224         40.65733            2        16.5   0.0     0.5
5         -73.97726         40.75851            2         8.0   0.0     0.5
6         -73.91349         40.76314            2        19.0   0.5     0.5
  tip_amount tolls_amount improvement_surcharge total_amount
1          0            0                   0.3          8.8
2          0            0                   0.3         19.3
3          0            0                   0.3         34.3
4          0            0                   0.3         17.3
5          0            0                   0.3          8.8
6          0            0                   0.3         20.3
             pickup_city       pickup_name
```

```
1 New York City-Manhattan Greenwich Village
2 New York City-Manhattan      East Village
3                     <NA>              <NA>
4 New York City-Manhattan   Lower East Side
5 New York City-Manhattan   Upper East Side
6 New York City-Manhattan          Gramercy
```

Per eseguire la trasformazione sopra riportata a tutti i dati, dobbiamo prendere il codice sopra riportato e inserirlo in una funzione di trasformazione da passare a `rxDataStep` attraverso l'argomento `transfromFunc`. Chiamo poi la funzione di trasformazione `find_nhoods`. La funzione di trasformazione aggiungerà sia il quartiere di *pick-up* che il quartiere di *drop-off*.

```
find_nhoods <- function(data) {

  # estrae pick-up lat e long e trova i il quartiere corrispondente
  pickup_longitude<-ifelse(is.na(data$pickup_longitude),0,data$pickup_longitude)
  pickup_latitude<-ifelse(is.na(data$pickup_latitude),0,data$pickup_latitude)
  data_coords <- data.frame(long = pickup_longitude, lat = pickup_latitude)
  coordinates(data_coords) <- c('long', 'lat')
  nhoods <- over(data_coords, shapefile)

  ## aggiunge i quartieri di partenza e quartieri della città ai dati
data$pickup_nhood <- nhoods$NAME
  data$pickup_borough <- nhoods$CITY

  # estrae il drop-off lat e long e trova il quartiere corrispondente
  dropoff_longitude <- ifelse(is.na(data$dropoff_longitude),
                                    0, data$dropoff_longitude)
  dropoff_latitude <- ifelse(is.na(data$dropoff_latitude),
                                   0, data$dropoff_latitude)
  data_coords <- data.frame(long = dropoff_longitude, lat = dropoff_latitude)
  coordinates(data_coords) <- c('long', 'lat')
  nhoods <- over(data_coords, shapefile)
```

```
  ## aggiunge i quartieri di arrivo e quartieri della città ai dati
  data$dropoff_nhood <- nhoods$NAME
  data$dropoff_borough <- nhoods$CITY

  ## restituisce i dati con le nuove colonne aggiunte
  data
}
```

Ora che abbiamo la nostra funzione, è il momento di testarla. Possiamo farlo eseguendo `rxDataStep` sui dati di esempio `nyc_sample_df`. Questo è un buon modo per assicurarsi che la trasformazione funzioni prima di eseguirla sul file XDF `nyc_xdf`. A volte i messaggi di errore che otteniamo sono più informativi quando applichiamo la trasformazione a un `data.frame`, ed è più facile rintracciarli e debuggarli.

```
# testa la funzione su un data.frame usando rxDataStep
head(rxDataStep(nyc_sample_df, transformFunc = find_nhoods, transformPackages =
c("sp", "maptools"),
                transformObjects = list(shapefile = mht_shapefile)))
```

```
VendorID tpep_pickup_datetime tpep_dropoff_datetime passenger_count trip_distance
1        2  2016-01-01 00:00:00   2016-01-01 00:00:00               2          1.10
2        2  2016-01-01 00:00:00   2016-01-01 00:00:00               5          4.90
3        2  2016-01-01 00:00:00   2016-01-01 00:00:00               1         10.54
4        2  2016-01-01 00:00:00   2016-01-01 00:00:00               1          4.75
5        2  2016-01-01 00:00:00   2016-01-01 00:00:00               3          1.76
6        2  2016-01-01 00:00:00   2016-01-01 00:18:30               2          5.52
  pickup_longitude pickup_latitude RatecodeID store_and_fwd_flag dropoff_longitude
1        -73.99037        40.73470          1                  N         -73.98184
2        -73.98078        40.72991          1                  N         -73.94447
3        -73.98455        40.67957          1                  N         -73.95027
4        -73.99347        40.71899          1                  N         -73.96224
5        -73.96062        40.78133          1                  N         -73.97726
6        -73.98012        40.74305          1                  N         -73.91349
  dropoff_latitude payment_type fare_amount extra mta_tax tip_amount tolls_amount
1         40.73241            2         7.5   0.5     0.5          0            0
```

```
2         40.71668           1        18.0   0.5     0.5          0           0
3         40.78893           1        33.0   0.5     0.5          0           0
4         40.65733           2        16.5   0.0     0.5          0           0
5         40.75851           2         8.0   0.0     0.5          0           0
6         40.76314           2        19.0   0.5     0.5          0           0
  improvement_surcharge total_amount      pickup_nhood          pickup_borough
1                   0.3          8.8 Greenwich Village New York City-Manhattan
2                   0.3         19.3      East Village New York City-Manhattan
3                   0.3         34.3       Boerum Hill  New York City-Brooklyn
4                   0.3         17.3   Lower East Side New York City-Manhattan
5                   0.3          8.8   Upper East Side New York City-Manhattan
6                   0.3         20.3          Gramercy New York City-Manhattan
             dropoff_nhood         dropoff_borough
1                 Gramercy New York City-Manhattan
2                     <NA>                    <NA>
3                Yorkville New York City-Manhattan
4                     <NA>                    <NA>
5                  Midtown New York City-Manhattan
6 Astoria-Long Island City    New York City-Queens
```

Le ultime quattro colonne corrispondono ora ai quartieri che volevamo. Quindi la trasformazione non dovrebbe avere problemi ad essere applicata anche su `nyc_xdf`.

```
st <- Sys.time()
rxDataStep(nyc_xdf, nyc_xdf,
           overwrite = TRUE,
           transformFunc = find_nhoods,
           transformPackages = c("sp", "maptools", "rgeos"),
           transformObjects = list(shapefile = mht_shapefile))
Sys.time() - st
rxGetInfo(nyc_xdf, numRows = 5)
```

```
Time difference of 30.77251 mins

File name: C:\Data\NYC_taxi\yellow_tripdata_2016.xdf
Number of observations: 69406520
```

```
Number of variables: 29
Number of blocks: 141
Compression type: zlib
Data (5 rows starting with row 1):
  VendorID     tpep_pickup_datetime     tpep_dropoff_datetime     passenger_count
trip_distance
1         2  2016-01-01 00:00:00   2016-01-01 00:00:00               2          1.10
2         2  2016-01-01 00:00:00   2016-01-01 00:00:00               5          4.90
3         2  2016-01-01 00:00:00   2016-01-01 00:00:00               1         10.54
4         2  2016-01-01 00:00:00   2016-01-01 00:00:00               1          4.75
5         2  2016-01-01 00:00:00   2016-01-01 00:00:00               3          1.76
  pickup_longitude pickup_latitude RatecodeID store_and_fwd_flag dropoff_longitude
1        -73.99037        40.73470          1                  N         -73.98184
2        -73.98078        40.72991          1                  N         -73.94447
3        -73.98455        40.67957          1                  N         -73.95027
4        -73.99347        40.71899          1                  N         -73.96224
5        -73.96062        40.78133          1                  N         -73.97726
  dropoff_latitude payment_type fare_amount extra mta_tax tip_amount tolls_amount
1         40.73241            2         7.5   0.5     0.5          0            0
2         40.71668            1        18.0   0.5     0.5          0            0
3         40.78893            1        33.0   0.5     0.5          0            0
4         40.65733            2        16.5   0.0     0.5          0            0
5         40.75851            2         8.0   0.0     0.5          0            0
  improvement_surcharge total_amount tip_percent pickup_hour pickup_dow
1                   0.3          8.8           0    10PM-1AM        Fri
2                   0.3         19.3           0    10PM-1AM        Fri
3                   0.3         34.3           0    10PM-1AM        Fri
4                   0.3         17.3           0    10PM-1AM        Fri
5                   0.3          8.8           0    10PM-1AM        Fri
  dropoff_hour dropoff_dow trip_duration      pickup_nhood         pickup_borough
1     10PM-1AM         Fri             0 Greenwich Village New York City-Manhattan
2     10PM-1AM         Fri             0      East Village New York City-Manhattan
3     10PM-1AM         Fri             0       Boerum Hill  New York City-Brooklyn
4     10PM-1AM         Fri             0   Lower East Side New York City-Manhattan
5     10PM-1AM         Fri             0   Upper East Side New York City-Manhattan
```

```
  dropoff_nhood         dropoff_borough
1      Gramercy New York City-Manhattan
2          <NA>                    <NA>
3     Yorkville New York City-Manhattan
4          <NA>                    <NA>
5       Midtown New York City-Manhattan
```

La funzione ha eseguito il suo lavoro su tutto il *dataset* in poco più di 30 minuti.

Esaminazione e visualizzazione dati

Oltre a chiedere se i dati abbiano un senso logico, spesso è una buona idea controllare anche se che i dati abbiano un senso pratico o affaristico. Ciò può aiutarci a rilevare determinati errori, ad esempio dati errati o attribuiti all'insieme di funzioni errate. Se non vengono rilevati, tali errori possono avere un impatto negativo sull'analisi generale.

```
system.time(
rxs_all <- rxSummary( ~ ., nyc_xdf)
)
```

```
Rows Processed: 69406520
   user  system elapsed
   0.05    0.02   85.16
```

```
head(rxs_all$sDataFrame)
```

```
                   Name       Mean      StdDev          Min          Max ValidObs
1             VendorID         NA          NA           NA           NA 69406520
2  tpep_pickup_datetime         NA          NA           NA           NA        0
3 tpep_dropoff_datetime         NA          NA           NA           NA        0
4       passenger_count   1.660674    1.310478       0.0000       9.0000 69406520
5         trip_distance   4.850022 4044.503422 -3390583.8000 19072628.8000 69406520
6      pickup_longitude -72.920469    8.763351     -165.0819     118.4089 69406520
  MissingObs
1          0
```

```
2          0
3          0
4          0
5          0
6          0
```

```
nhoods_by_borough <- rxCrossTabs( ~ pickup_nhood:pickup_borough, nyc_xdf)
nhoods_by_borough <- nhoods_by_borough$counts[[1]]
nhoods_by_borough <- as.data.frame(nhoods_by_borough)

# ottenere I quartieri
lnbs <- lapply(names(nhoods_by_borough), function(vv) subset(nhoods_by_borough,
nhoods_by_borough[ , vv] > 0, select = vv, drop = FALSE))
lapply(lnbs, head)
```

```
[[1]]
[1] Albany
<0 rows> (or 0-length row.names)

[[2]]
[1] Buffalo
<0 rows> (or 0-length row.names)

[[3]]
             New York City-Bronx
Baychester                   125
Bedford Park                1413
City Island                   52
Country Club                 354
Eastchester                   98
Fordham                     1243

[[4]]
                   New York City-Brooklyn
Bay Ridge                            3378
```

Bedford-Stuyvesant	54269
Bensonhurst	1159
Boerum Hill	76404
Borough Park	8762
Brownsville	2757

[[5]]

	New York City-Manhattan
Battery Park	643283
Carnegie Hill	807204
Central Park	936840
Chelsea	4599098
Chinatown	211229
Clinton	2050545

[[6]]

	New York City-Queens
Astoria-Long Island City	303231
Auburndale	464
Clearview	152
College Point	1
Corona	1496
Douglastown-Little Neck	937

[[7]]

	New York City-Staten Island
Annandale	6
Ardon Heights	22
Bloomfield-Chelsea-Travis	26
Charlestown-Richmond Valley	7
Clifton	525
Ettingville	13

[[8]]

[1] Rochester

```
<0 rows> (or 0-length row.names)

[[9]]
[1] Syracuse
<0 rows> (or 0-length row.names)
```

Siccome la maggior parte dei viaggi in taxi si svolge a Manhattan, focalizziamo la nostra attenzione solo su Manhattan e ignoriamo gli altri quattro quartieri. A tale scopo, creo due nuove colonne chiamate *pickup_nb* e *dropoff_nb* basate sulle colonne originali *pickup_nhood* e *dropoff_nhood*, tranne per il fatto che i loro livelli di fattore sono limitati ai quartieri di Manhattan (qualsiasi altro livello fattore verrà sostituito con un NA).

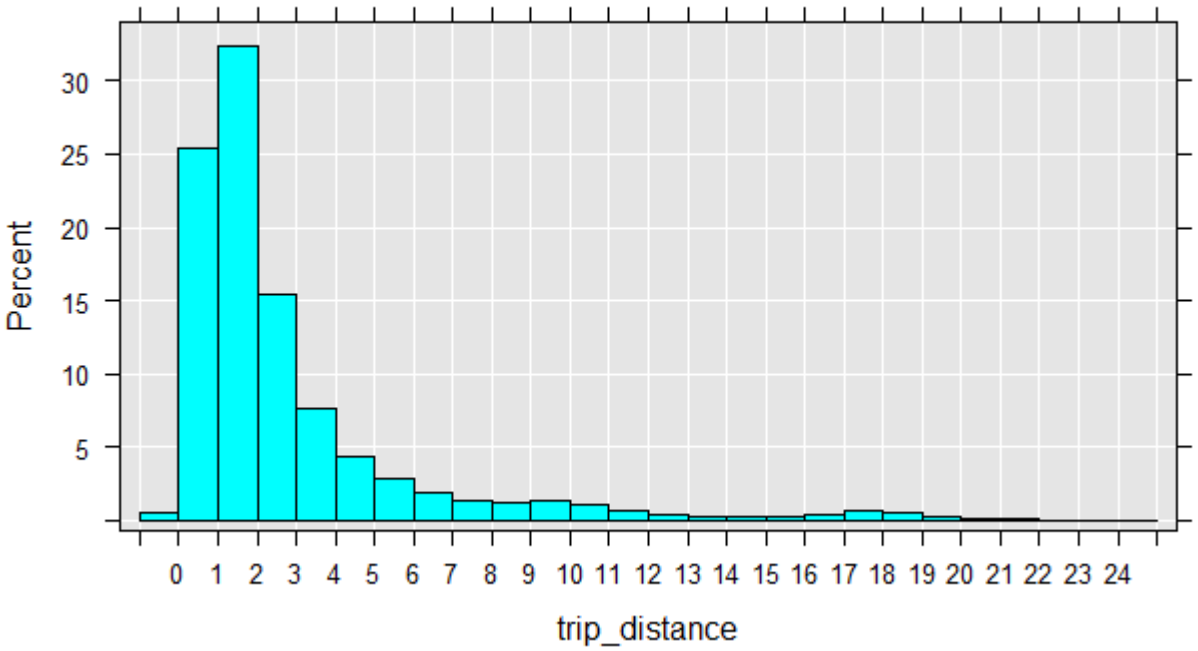
```
manhattan_nhoods <- rownames(nhoods_by_borough)
                    [nhoods_by_borough$`New York City-Manhattan` > 0]

refactor_columns <- function(dataList) {
  dataList$pickup_nb = factor(dataList$pickup_nhood, levels = nhoods_levels)
  dataList$dropoff_nb = factor(dataList$dropoff_nhood, levels = nhoods_levels)
  dataList
}

rxDataStep(nyc_xdf, nyc_xdf,
           transformFunc = refactor_columns,
           transformObjects = list(nhoods_levels = manhattan_nhoods),
           overwrite = TRUE)

rxs_pickdrop <- rxSummary( ~ pickup_nb:dropoff_nb, nyc_xdf)
head(rxs_pickdrop$categorical[[1]])
```

```
Rows Processed: 69406520

      pickup_nb   dropoff_nb Counts
1  Battery Park Battery Park  19876
2 Carnegie Hill Battery Park   2699
```

```
3  Central Park Battery Park   3479
4       Chelsea Battery Park  61024
5     Chinatown Battery Park   3813
6       Clinton Battery Park  23962
```

Osservando i risultati ottenuti finora e le istantanee dei dati, cerchiamo ora di identificare anomalie o outliers. Ad esempio, possiamo tracciare un istogramma di *trip_distance* e notare che quasi tutti i viaggi hanno percorso una distanza inferiore alle 20 miglia, con la grande maggioranza a meno di 5 miglia.

```
rxHistogram( ~ trip_distance,
               nyc_xdf,
               startVal = 0,
               endVal = 25,
               histType = "Percent",
               numBreaks = 20)
```



C'è un secondo picco intorno ai viaggi di 16 e 20 miglia, che vale la pena esaminare ulteriormente. Possiamo verificare ciò osservando in quali quartieri i passeggeri viaggiano da e per.

```
rxs <- rxSummary( ~ pickup_nhood:dropoff_nhood,
                  nyc_xdf,
                  rowSelection = (trip_distance > 15 & trip_distance < 22))
head(arrange(rxs$categorical[[1]], desc(Counts)), 10)
```

```
      pickup_nhood            dropoff_nhood Counts
1           Midtown Gravesend-Sheepshead Bay   2517
2    Upper East Side Gravesend-Sheepshead Bay   1090
3           Midtown  Douglastown-Little Neck   1013
4           Midtown                  Midtown    978
5  Garment District Gravesend-Sheepshead Bay    911
6           Midtown              Bensonhurst    878
7          Gramercy Gravesend-Sheepshead Bay    784
8           Jamaica          Upper West Side    775
9           Chelsea Gravesend-Sheepshead Bay    729
10          Midtown                Bay Ridge    687
```

Come possiamo vedere, *Gravesend-Sheepshead Bay* appare spesso come destinazione ma non come punto di partenza. Possiamo anche vedere viaggi da *Jamaica*, che è il quartiere più vicino all'aeroporto JFK.

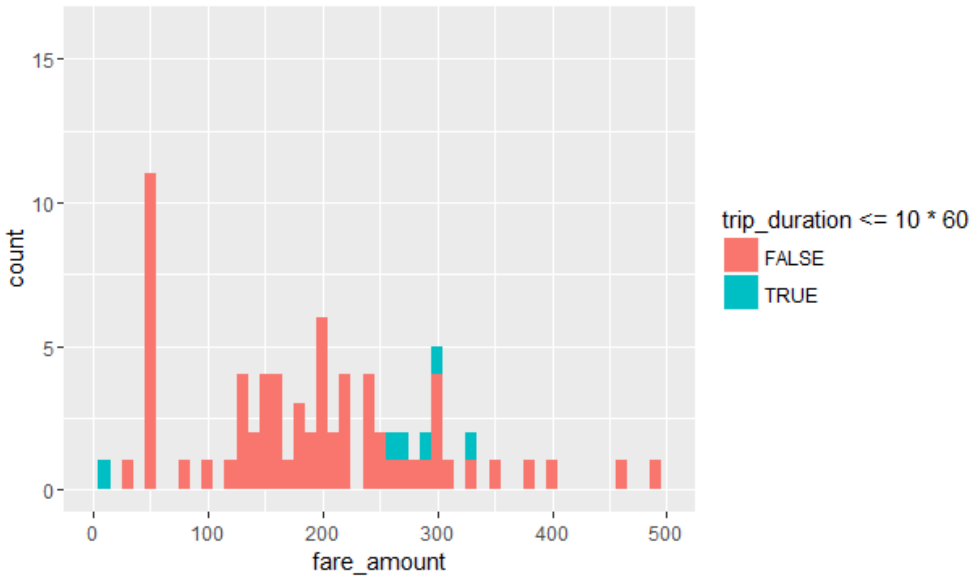
Uso `rxDataStep` e il suo `rowSelection` per estrarre tutti i punti di dati che sono valori anomali secondo certi criteri. Con `outFile`, emettiamo il set di dati risultante in un `data.frame` che chiamiamo `odd_trips`. Infine, se siamo troppo pignoli nei nostri criteri di selezione degli outlier, il `data.frame` potrebbe ancora contenere troppe righe (che potrebbero intasare la memoria e rallentare la produzione di grafici e altri riepiloghi). Quindi creiamo una nuova colonna u e la popoliamo con numeri casuali uniformi tra 0 e 1, e aggiungiamo u <.05 ai nostri criteri rowSelection. Possiamo regolare questo valore per alleggerire il `data.frame` (con una soglia più vicina a 0) o un `data.frame` più grande (con soglia più vicina a 1).

```
odd_trips <- rxDataStep(nyc_xdf, rowSelection = (
  u < .05 & ( # possiamo regolare il valore se vogliamo velocizzare il calcolo
    (trip_distance > 50 | trip_distance <= 0) |
    (passenger_count > 5 | passenger_count == 0) |
    (fare_amount > 5000 | fare_amount <= 0)
)), transforms = list(u = runif(.rxNumRows)))

print(dim(odd_trips))
```

```
[1] 93750    32
```

Ora limito gli *odd_trips* ai casi in cui è stata percorsa una distanza > 50 miglia, traccio un istogramma dell'importo pagato dal passeggero e coloro in base al fatto che il viaggio sia durato più o meno di 10 minuti.

```
odd_trips %>%
  filter(trip_distance > 50) %>%
  ggplot() -> p
p + geom_histogram(aes(x = fare_amount,
                       fill = trip_duration <= 10*60),
                       binwidth = 10) +
  xlim(0, 500) + coord_fixed(ratio = 25)
```



Come possiamo vedere, la maggior parte dei viaggi che hanno percorso più di 50 miglia non costano nulla o quasi nulla, anche se la maggior parte di questi viaggi ha impiegato 10 minuti o più. Non è chiaro se tali viaggi siano il risultato di errori umani o di errori della telemetria, ma se per esempio questa analisi fosse rivolta alla compagnia che possiede i taxi, questa constatazione meriterebbe ulteriori indagini.

Ora limito il campo concentrando i dati sui viaggi che si sono svolti solo all'interno di Manhattan, e solo a quelli che incontrino criteri "ragionevoli" per un viaggio. Poiché abbiamo aggiunto nuove funzionalità ai dati, possiamo anche eliminare alcune vecchie colonne di variabili in modo che il *dataset* possa essere elaborato più velocemente.

```
input_xdf <- 'yellow_tripdata_2016_manhattan.xdf'
mht_xdf <- RxXdfData(input_xdf)


rxDataStep(nyc_xdf, mht_xdf,
           rowSelection = (
             passenger_count > 0 &
               trip_distance >= 0 & trip_distance < 30 &
               trip_duration > 0 & trip_duration < 60*60*24 &
               str_detect(pickup_borough, 'Manhattan') &
               str_detect(dropoff_borough, 'Manhattan') &
               !is.na(pickup_nb) &
               !is.na(dropoff_nb) &
               fare_amount > 0),
           transformPackages = "stringr",
           varsToDrop = c('extra',
                          'mta_tax',
                          'improvement_surcharge',
                          'total_amount',
                          'pickup_borough',
                          'dropoff_borough',
                          'pickup_nhood',
                          'dropoff_nhood'),
           overwrite = TRUE)
```

E poiché ho limitato i dati, potrebbe essere una buona idea creare un campione dei nuovi dati (come `data.frame`). Il nostro ultimo campione, `nyc_sample_df` non era un buon esempio di dati, siccome prendevamo solo le prime 1000 righe di dati. Questa volta, utilizzo `rxDataStep` per creare un campione casuale dei dati, contenente solo l'1% delle righe dal set originale.

```
mht_sample_df <- rxDataStep(mht_xdf, rowSelection = (u < .01),
                            transforms = list(u = runif(.rxNumRows)))


dim(mht_sample_df)
```

```
Rows Processed: 57493035
WARNING: The number of rows (574832) times the number of columns (24)
exceeds the 'maxRowsByCols' argument (3000000). Rows will be truncated.

> dim(mht_sample_df)
[1] 125000     24
```

Quindi, abbiamo i dati in un formato pulito. Ho creato nuove colonne dati per rendere il set più completo. E ora si può analizzare più seriamente. Possiamo iniziare a farci domande più serie riguardo i dati, possiamo iniziare a guardare più visualizzazioni e vedere che tipo di storie possono raccontarci.

Cerco ora patterns tra i quartieri di partenza e arrivo e altre variabili come l'importo delle tariffe, la distanza percorsa, il traffico e le mance. Per avere una stima del traffico osservando il rapporto tra la durata del viaggio e la distanza del viaggio, supponendo che il traffico sia il motivo più comune per richiedere più tempo di viaggio.

Per l'analisi, uso `rxCube` e `rxCrossTabs` che sono entrambi molto simili a `rxSummary` ma restituiscono meno riepiloghi statistici e quindi funzionano più velocemente. Con `y ~ u: v` come formula, `rxCrossTabs` restituisce i conteggi e le somme e `rxCube` restituisce i conteggi e le medie per la colonna y suddivisi in base a qualsiasi combinazione di colonne `u` e `v`.

Comincio usando `rxCrossTabs` per ottenere somme e conteggi per *trip_distance*, suddivisi in *pickup_nb* e *dropoff_nb*. Posso poi dividere immediatamente le somme per i conteggi e ottenere le medie. Il risultato è una matrice di distanze e può essere inviata alla funzione `seriate` nella libreria di `seriation` per ordinarle in modo che i quartieri più vicini appaiano uno accanto all'altro (all'inizio i quartieri sono ordinati alfabeticamente, che è ciò che R fa di default, a meno che specificato diversamente).

```
rxct <- rxCrossTabs(trip_distance ~ pickup_nb:dropoff_nb, mht_xdf)
res <- rxct$sums$trip_distance / rxct$counts$trip_distance

library(seriation)
res[which(is.nan(res))] <- mean(res, na.rm = TRUE)
nb_order <- seriate(res)
```

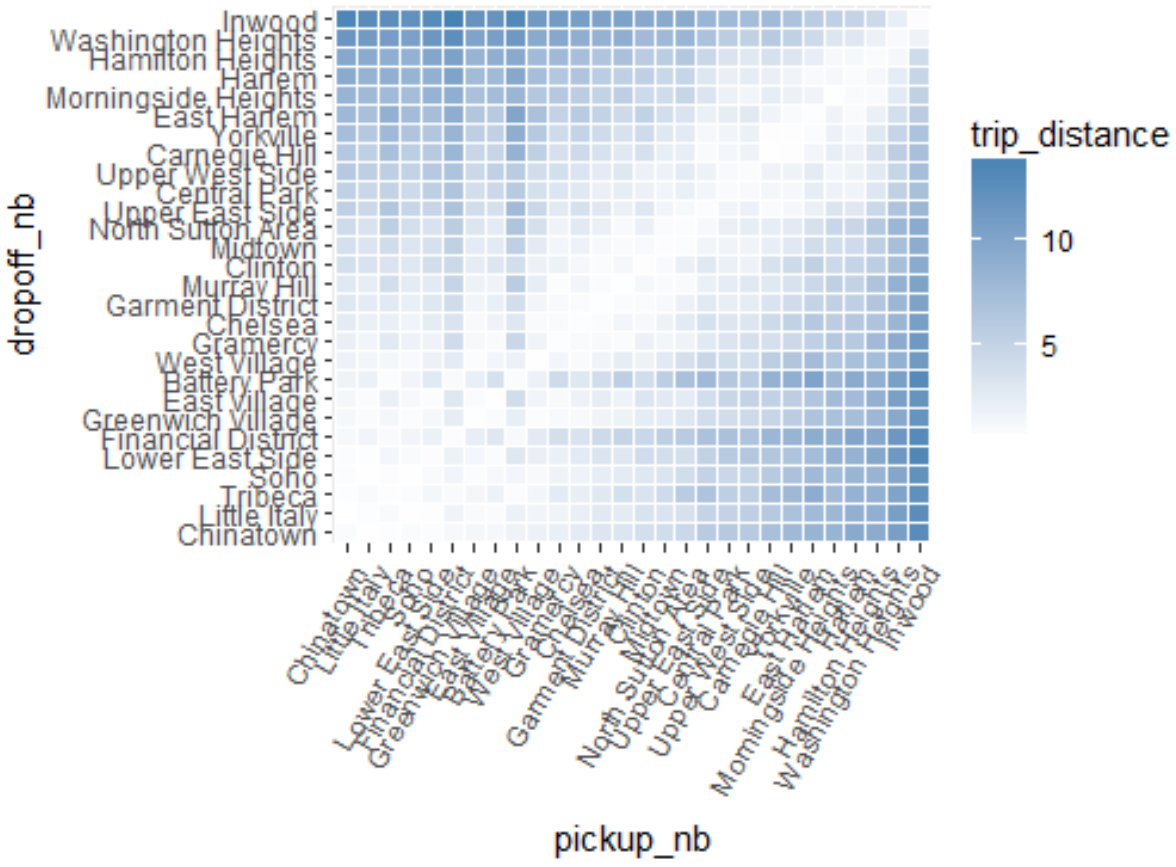
Ora uso `rxCube` per ottenere invece un `data.frame`, dal momento che intendiamo usarlo per il plottaggio con `ggplot2`, che è più facile da codificare utilizzando un `data.frame` lungo come input rispetto a un ampia matrice.

```
rxc1 <- rxCube(trip_distance ~ pickup_nb:dropoff_nb, mht_xdf)
rxc2 <- rxCube(minutes_per_mile ~ pickup_nb:dropoff_nb,
               mht_xdf,
               transforms=list(minutes_per_mile=
                               (trip_duration/60)/trip_distance))
rxc3 <- rxCube(tip_percent ~ pickup_nb:dropoff_nb, mht_xdf)
res <- bind_cols(list(rxc1, rxc2, rxc3))
res <- res[ , c('pickup_nb',
                'dropoff_nb',
                'trip_distance',
                'minutes_per_mile', 'tip_percent')]
head(res)
```

```
      pickup_nb   dropoff_nb trip_distance minutes_per_mile tip_percent
         <fctr>       <fctr>         <dbl>            <dbl>       <dbl>
1  Battery Park Battery Park      1.015857        11.579629   11.394900
2 Carnegie Hill Battery Park      8.570623         3.944350   12.391030
```

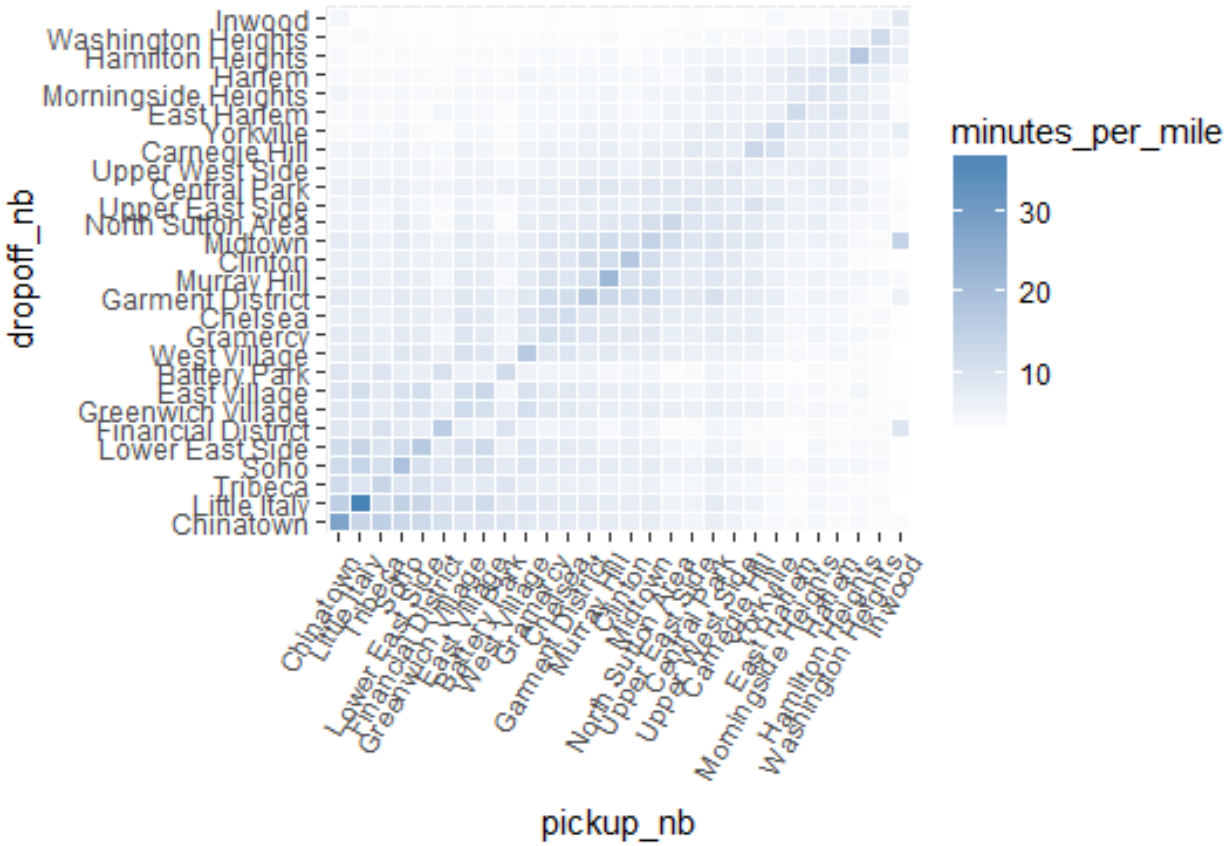
```
3  Central Park Battery Park      6.277666          5.243241   10.326531
4       Chelsea Battery Park      2.995946          5.169887   11.992151
5     Chinatown Battery Park      1.771597          9.001305   10.292683
6       Clinton Battery Park      3.993806          4.839858    9.794098
```

```r
# ordino il grafico in maniera più leggibile
newlevs <- levels(res$pickup_nb)[unlist(nb_order)]
res$pickup_nb <- factor(res$pickup_nb, levels = unique(newlevs))
res$dropoff_nb <- factor(res$dropoff_nb, levels = unique(newlevs))

# visualizzo il grafico a matrice per mostrare alcune tendenze interessanti.
library(ggplot2)
ggplot(res, aes(pickup_nb, dropoff_nb)) +
  geom_tile(aes(fill = trip_distance), colour = "white") +
  theme(axis.text.x = element_text(angle = 60, hjust = 1)) +
  scale_fill_gradient(low = "white", high = "steelblue") +
  coord_fixed(ratio = .9)
```



Poiché le distanze di viaggio rimangono fisse, ma le durate dipendono in gran parte dalla quantità di traffico, possiamo tracciare un grafico per la colonna *minutes_per_mile*, che ci darà un'idea di quali quartieri hanno il maggior traffico tra di loro.

```
ggplot(res, aes(pickup_nb, dropoff_nb)) +
  geom_tile(aes(fill = minutes_per_mile), colour = "white") +
  theme(axis.text.x = element_text(angle = 60, hjust = 1)) +
  scale_fill_gradient(low = "white", high = "steelblue") +
  coord_fixed(ratio = .9)
```

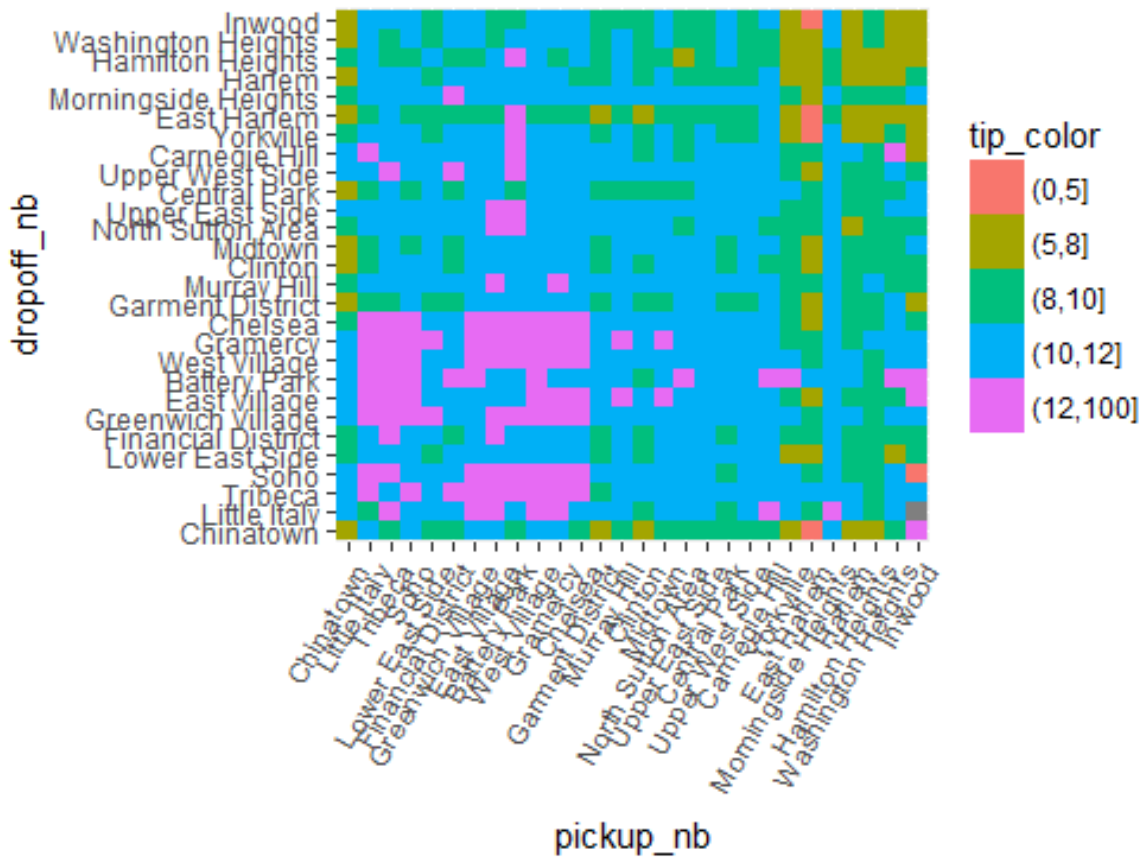


Come mostrano i grafici, molto traffico avviene tra quartieri vicini l'uno all'altro. Questo non è molto sorprendente dato che i viaggi tra quartieri lontani possono essere fatti utilizzando percorsi periferici che bypassano la maggior parte del traffico del centro città.

Un'altra questione interessante da prendere in considerazione è la relazione tra l'importo della corsa e quanta mancia i passeggeri lasciano in base tra quali quartieri viaggiano. Creiamo un altro grafico simile a quelli sopra, mostrando l'importo della tariffa su una

scala di colori di sfondo grigio e mostrando quanta mancia in media lasciano per il viaggio. Per rendere più semplice la visualizzazione, codifico con colori la mancia media in base al fatto che sia superiore al 12%, inferiore al 12%, inferiore al 10%, inferiore all'8% e inferiore al 5%.

```
res %>%
  mutate(tip_color = cut(tip_percent, c(0, 5, 8, 10, 12, 100))) %>%
  ggplot(aes(pickup_nb, dropoff_nb)) +
  geom_tile(aes(fill = tip_color)) +
  theme(axis.text.x = element_text(angle = 60, hjust = 1)) +
  coord_fixed(ratio = .9)
```



Alcuni considerazioni interessanti:

- I viaggi che partono da Battery Park o il quartiere finanziario che va a Midtown o nei quartieri periferici, sembrano costare (in mancia) un po' più di quanto

sembra giustificato, e vale lo stesso per i viaggi che lasciano il Greenwich Village andando a Chinatown.

- I viaggi dentro e fuori Chinatown sono costantemente bassi (sotto il 10%), specialmente se si viaggia o viene da quartieri alti.
- I più generosi (circa il 12%) sono quelli che viaggiano tra i quartieri del centro (ad eccezione di Chinatown). I successivi più generosi (circa l'11%) sono quelli che viaggiano tra quartieri centrali e quartieri del centro in entrambe le direzioni. I peggiori (quelle che lasciano meno valore di mancia) sono quelli che viaggiano tra quartieri alti.

Abbiamo modificato (solo una parte del *dataset*) l'ordine dei livelli dei fattori per *pickup_nb* e *dropoff_nb* per disegnare i grafici sopra. Tuttavia, questo cambiamento parziale non giova all'analisi stessa, perché ogni volta che tracciamo qualcosa che coinvolge *pickup_nb* o *dropoff_nb* avremo bisogno di cambiare l'ordine dei livelli dei fattori. Quindi vado a ora modificare l'intero *dataset*.

```
rxDataStep(inData = mht_xdf, outFile = mht_xdf,
           transforms = list(pickup_nb = factor(pickup_nb, levels = newlevels),
                             dropoff_nb    =    factor(dropoff_nb,    levels    =
newlevels)),
           transformObjects = list(newlevels = unique(newlevs)),
           overwrite = TRUE)
```

Ora vorrei analizzare tra quali quartieri si verificano maggiormente i viaggi dei taxi. Per farlo devo trovare la distribuzione (o la proporzione) di viaggi tra due quartieri, prima come percentuale del numero totale di viaggi, poi come percentuale di viaggi in partenza da un determinato quartiere e infine come percentuale di viaggi diretti a un determinato quartiere.

```
rxc <- rxCube( ~ pickup_nb:dropoff_nb, mht_xdf)
rxc <- as.data.frame(rxc)

library(dplyr)
rxc %>%
  filter(Counts > 0) %>%
  mutate(pct_all = Counts/sum(Counts) * 100) %>%
  group_by(pickup_nb) %>%
  mutate(pct_by_pickup_nb = Counts/sum(Counts) * 100) %>%
  group_by(dropoff_nb) %>%
  mutate(pct_by_dropoff_nb = Counts/sum(Counts) * 100) %>%
  group_by() %>%
  arrange(desc(Counts)) -> rxcs
```

```
head(rxcs)
```

```
         pickup_nb      dropoff_nb  Counts  pct_all pct_by_pickup_nb
            <fctr>          <fctr>   <dbl>    <dbl>            <dbl>
1  Upper East Side Upper East Side 3299324 5.738650         36.88840
2          Midtown         Midtown 2216184 3.854700         21.84268
3  Upper West Side Upper West Side 1924205 3.346849         35.14494
4          Midtown Upper East Side 1646843 2.864422         16.23127
5  Upper East Side         Midtown 1607925 2.796730         17.97756
6 Garment District         Midtown 1072732 1.865847         28.94205
  pct_by_dropoff_nb
              <dbl>
1          38.28066
2          22.41298
3          35.15770
4          19.10762
5          16.26146
6          10.84888
```

Sulla base della prima riga, possiamo vedere che i viaggi dall'Upper East Side all'Upper East Side costituiscono circa il 5% di tutti i viaggi a Manhattan. Tra tutti i viaggi che sono partiti dall'Upper East Side, circa il 36% arriva nell'Upper East Side. Di tutti i viaggi che sono arrivati nell'Upper East Side, il 37% sono anche partiti dall'Upper East Side.

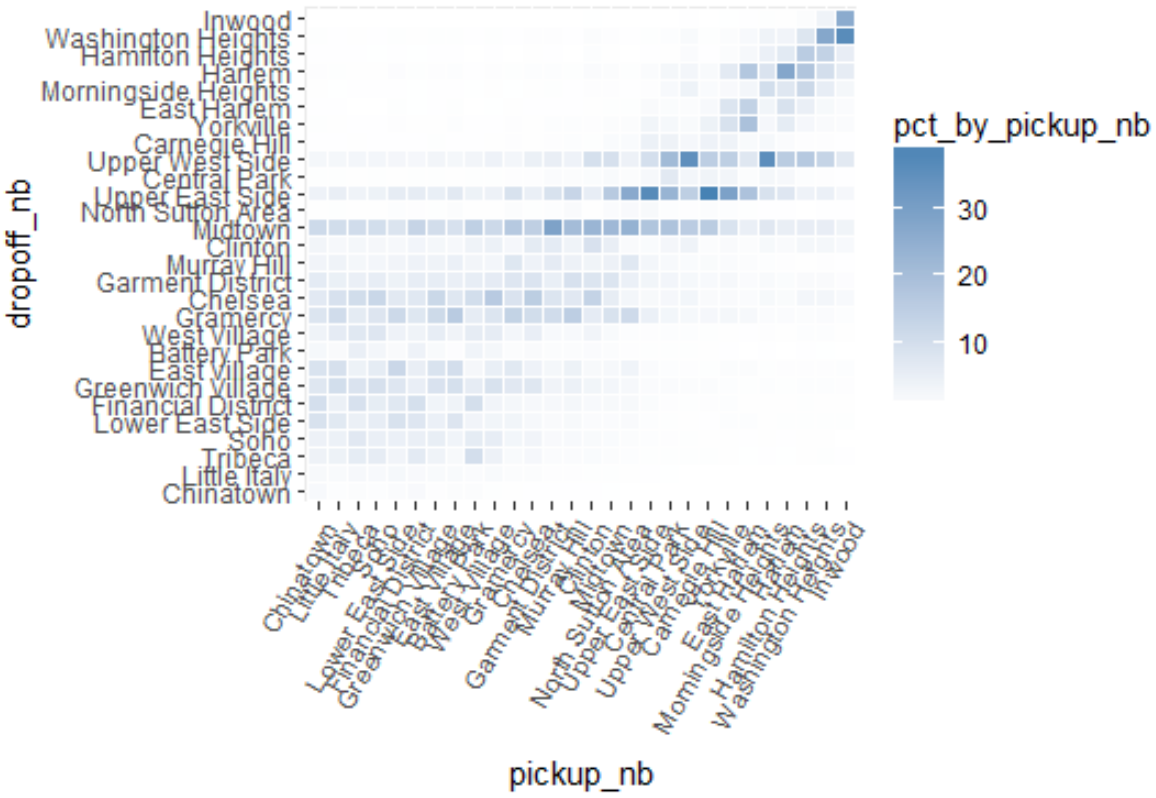
Possiamo ora utilizzare i dati sopra e visualizzarli in grafico a matrice per rendere più facile la lettura che mostra come vengono distribuiti i viaggi in taxi tra ogni coppia di quartieri.

```
ggplot(rxcs, aes(pickup_nb, dropoff_nb)) +
  geom_tile(aes(fill = pct_all), colour = "white") +
  theme(axis.text.x = element_text(angle = 60, hjust = 1)) +
  scale_fill_gradient(low = "white", high = "black") +
  coord_fixed(ratio = .9)
```

Il grafico mostra che da e per l'Upper East Side costituiscono la maggior parte dei viaggi, un risultato un po 'inaspettato. Inoltre, in generale, la maggior parte dei viaggi è da e per l'Upper East Side e l'Upper West Side e i quartieri di midtown (con molti viaggi di questa categoria che hanno Midtown come origine o destinazione). Un altro fatto sorprendente dei dati visualizzati nel grafico, è la simmetria di vicinato, il che suggerisce che forse la maggior parte dei passeggeri usa i taxi per un "viaggio di andata e ritorno", cioè che prendono un taxi per raggiungere la loro destinazione e lo stesso per il viaggio di ritorno. Questo punto merita un'ulteriore indagine (forse coinvolgendo l'ora del giorno nell'analisi) ma per ora non approfondiamo ulteriormente.

Poi analizziamo come i viaggi lasciano un determinato quartiere (un punto sull'asse x nella grafico seguente), "si riversano" poi in altri quartieri (mostrato dal gradiente di colore verticale lungo l'asse y in ogni punto dell'asse x).

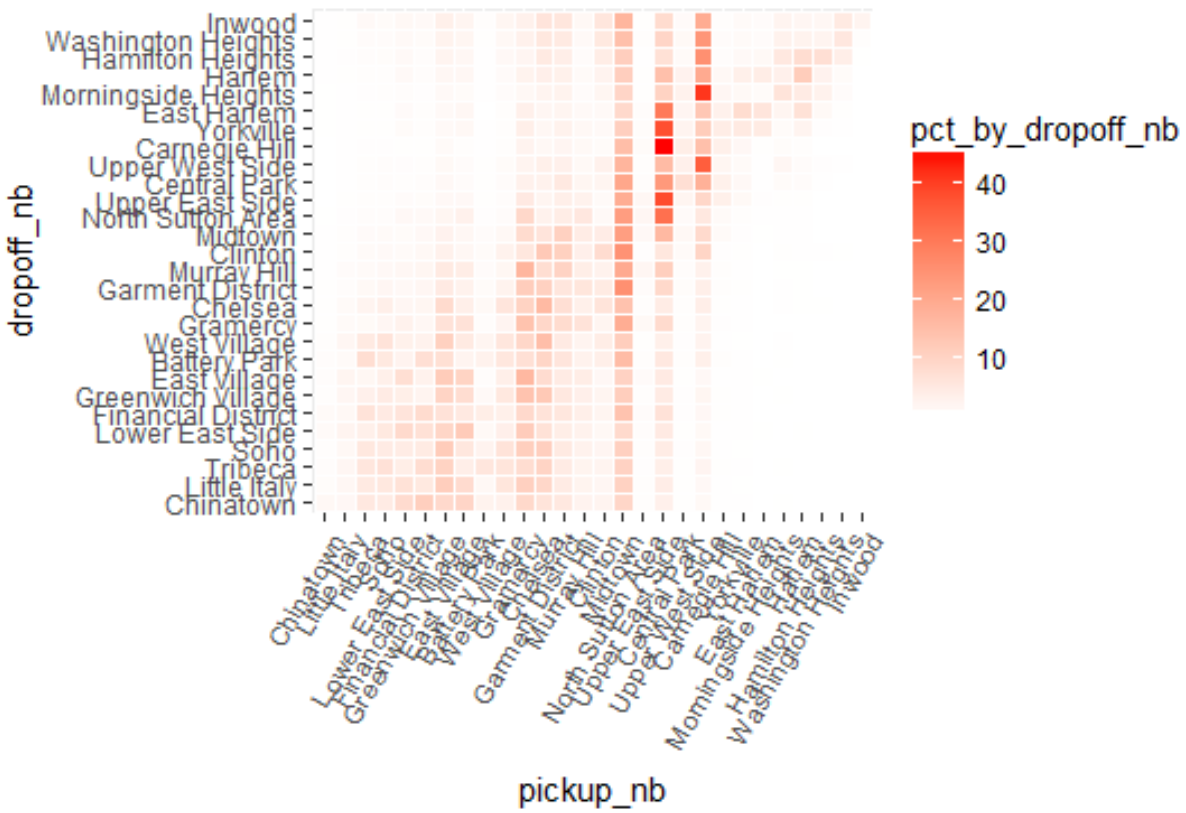
```
ggplot(rxcs, aes(pickup_nb, dropoff_nb)) +
  geom_tile(aes(fill = pct_by_pickup_nb), colour = "white") +
  theme(axis.text.x = element_text(angle = 60, hjust = 1)) +
  scale_fill_gradient(low = "white", high = "steelblue") +
  coord_fixed(ratio = .9)
```



Possiamo vedere come la maggior parte dei viaggi dal centro sono verso altri quartieri del centro o ai quartieri vicino midtown (in particolare Midtown). Midtown e Upper East Side sono destinazioni comuni da qualsiasi quartiere, e l'Upper West Side è una destinazione comune per la maggior parte dei quartieri residenziali.

Per una corsa che termina in un determinato quartiere (rappresentato da un punto sull'asse y) ora guardiamo alla distribuzione da cui il viaggio ha avuto origine (il gradiente di colore orizzontale lungo l'asse x per ogni punto sull'asse y).

```
ggplot(rxcs, aes(pickup_nb, dropoff_nb)) +
  geom_tile(aes(fill = pct_by_dropoff_nb), colour = "white") +
  theme(axis.text.x = element_text(angle = 60, hjust = 1)) +
  scale_fill_gradient(low = "white", high = "red") +
  coord_fixed(ratio = .9)
```

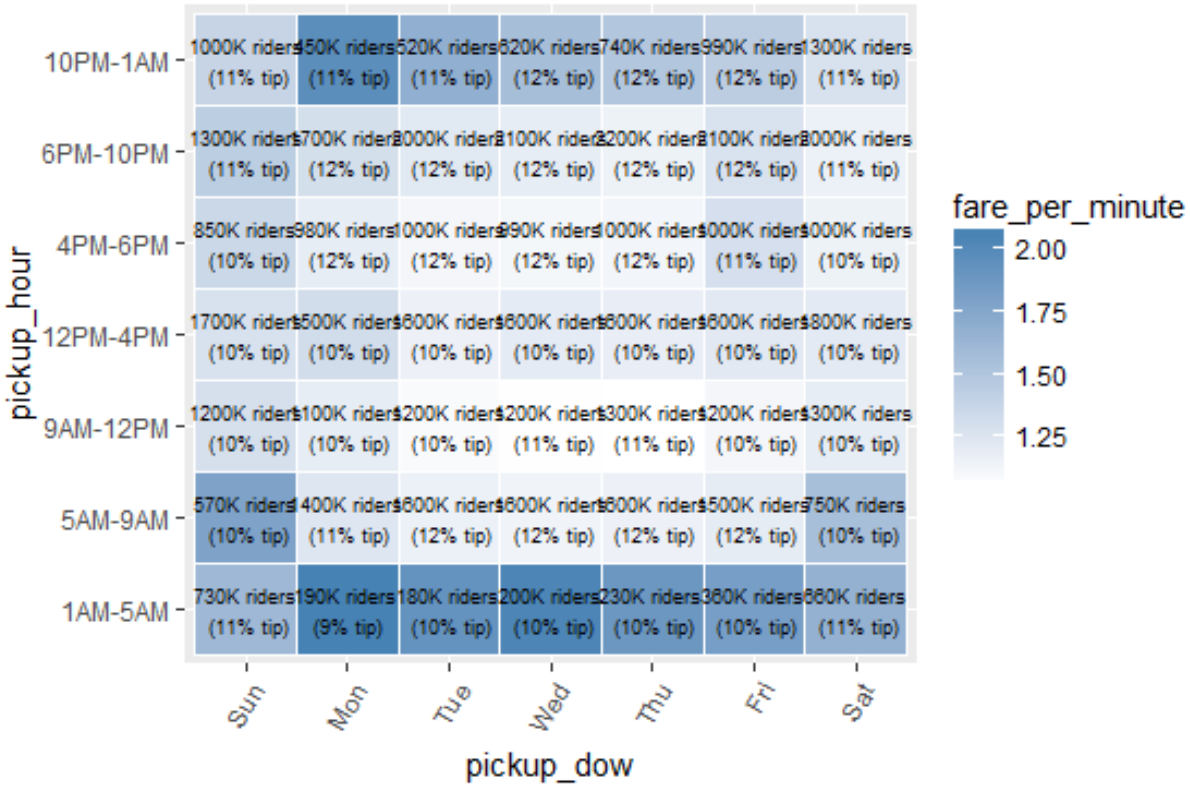


Come possiamo vedere, molte corse richiamano Midtown indipendentemente da dove sono partite. L'Upper East Side e Upper West Side sono anche loro origini comunemente utilizzate per i viaggi che scendono in uno dei quartieri alti.

Vediamo ora informazioni possono essere ricavate dalle colonne temporali che abbiamo estratto dai dati, vale a dire il giorno della settimana e l'ora in cui il passeggero è stato prelevato.

```
res1 <- rxCube(tip_percent ~ pickup_dow:pickup_hour, mht_xdf)
res2 <- rxCube(fare_amount/(trip_duration/60) ~ pickup_dow:pickup_hour, mht_xdf)
names(res2)[3] <- 'fare_per_minute'
res <- bind_cols(list(res1, res2))
res <- res[ , c('pickup_dow',
                'pickup_hour',
                'fare_per_minute',
                'tip_percent',
                'Counts')]

library(ggplot2)
ggplot(res, aes(pickup_dow, pickup_hour)) +
  geom_tile(aes(fill = fare_per_minute), colour = "white") +
  theme(axis.text.x = element_text(angle = 60, hjust = 1)) +
```

```
  scale_fill_gradient(low = "white", high = "steelblue") +
  geom_text(aes(label = sprintf('%dK riders\n (%d%% tip)',
                                signif(Counts/1000, 2),
                                 round(tip_percent, 0))),
                                 size = 2.5) +
  coord_fixed(ratio = .9)
```



Dalla grafico a matrice sopra possiamo vedere che una corsa in taxi costa di più al fine settimana che in un giorno feriale se è presa tra le 5:00 e le 22:00 e viceversa dalle 22:00 alle 5:00. Il grafico suggerisce anche che i passeggeri lascino più mancia nei giorni feriali e soprattutto subito dopo l'orario di ufficio. La questione delle mance dovrebbe essere guardata più da vicino, soprattutto dal momento che la percentuale di persone è influenzata dal fatto che le persone usano contanti o carta, che finora non ho preso in considerazione.

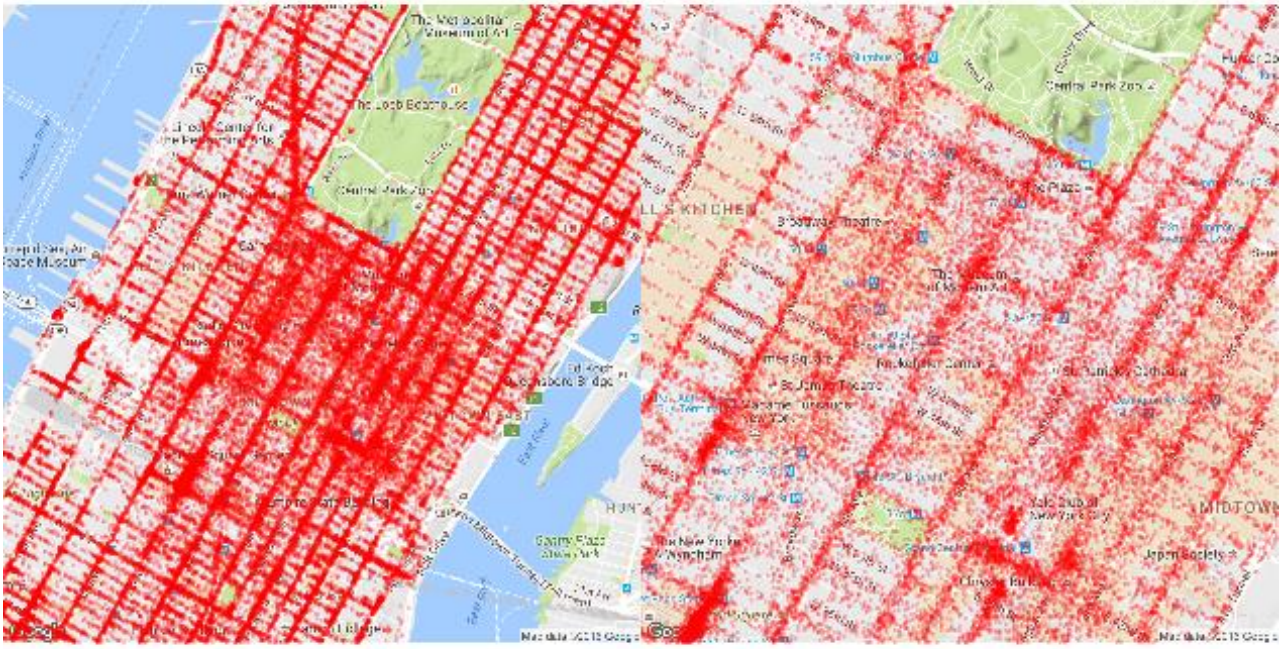
Clustering and Modeling

Bisogna ora cominciare a preparare i dati per l'analisi raggruppata e la creazione di modelli predittivi.

Clustering

La prima cosa da fare è usare il pacchetto `ggmap` che utilizza l'API di Google per ottenere un paio di mappe. In questo caso, le mappe avranno diversi livelli di zoom. Quindi una di loro sarà leggermente ingrandita, e l'altra sarà in zoom ad su di un particolare zona di Manhattan.

```
library(ggmap)
map_13 <- get_map(location =  c(lon = -73.98, lat = 40.76), zoom = 13)
map_14 <- get_map(location =  c(lon = -73.98, lat = 40.76), zoom = 14)
map_15 <- get_map(location =  c(lon = -73.98, lat = 40.76), zoom = 15)

q1 <- ggmap(map_14) +
  geom_point(aes(x = dropoff_longitude, y = dropoff_latitude),
             data = mht_sample_df,
             alpha = 0.15,
             na.rm = TRUE, col = "red", size = .5) +
  theme_nothing(legend = TRUE)

q2 <- ggmap(map_15) +
  geom_point(aes(x = dropoff_longitude, y = dropoff_latitude),
             data = mht_sample_df,
             alpha = 0.15,
             na.rm = TRUE, col = "red", size = .5) +
  theme_nothing(legend = TRUE)

require(gridExtra)
grid.arrange(q1, q2, ncol = 2)
```



Dal grafico è possibile individuare sulla mappa dove avvengono con più frequenza le partenze e gli arrivi dei taxi.

Posso ora usare k-means clustering per raggruppare i dati in base alla longitudine e alla latitudine, e da ridimensionare in modo da avere la stessa influenza sui cluster (un modo semplice per ridimensionarli è dividere longitudine per -74 e latitudine per 40). Una volta ottenuti i cluster, possiamo tracciare i centroidi del cluster sulla mappa anziché i singoli punti dati che comprendono ciascun cluster.

```
xydata <- transmute(mht_sample_df,
                    long_std = dropoff_longitude / -74,
                    lat_std = dropoff_latitude / 40)

start_time <- Sys.time()
rxkm_sample <- kmeans(xydata, centers = 300, iter.max = 2000, nstart = 50)
Sys.time() - start_time

# bisogna rimettere i centroidi nella scala originale
centroids_sample <- rxkm_sample$centers %>%
  as.data.frame %>%
  transmute(long = long_std*(-74), lat = lat_std*40, size = rxkm_sample$size)

head(centroids_sample)
```

```
Time difference of 2.017159 mins

       long      lat size
1 -74.01542 40.71149  277
2 -74.00814 40.71107  443
3 -73.99687 40.72133  335
4 -74.00465 40.75183  475
5 -73.96324 40.77466  589
6 -73.98444 40.73826  424
```

Nella porzione di codice sopra riportata ho usato la funzione `kmeans` per raggruppare il set di dati campione `mht_sample_df`. In `RevoScaleR` c'è una controparte della funzione `kmeans` chiamata `rxKmeans`, che oltre a lavorare con i `data.frame`, funziona anche con i file XDF. Possiamo quindi utilizzare `rxKmeans` per creare cluster da tutti i dati anziché dal campione rappresentato da `mht_sample_df`.

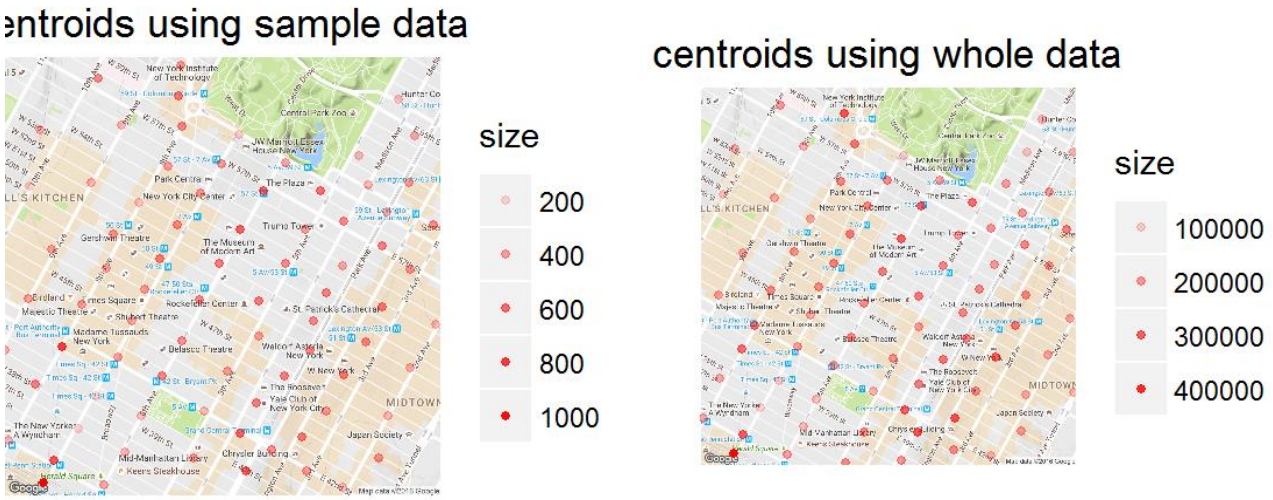
```
start_time <- Sys.time()
rxkm <- rxKmeans( ~ long_std + lat_std, data = mht_xdf,
                  outFile = mht_xdf,
                  outColName = "dropoff_cluster",
                  centers = rxkm_sample$centers,
                  transforms = list(long_std = dropoff_longitude / -74,
                  lat_std = dropoff_latitude / 40),
                  blocksPerRead = 1, overwrite = TRUE,
                  maxIterations = 100, reportProgress = -1)
Sys.time() - start_time

clsdf <- cbind(
  transmute(as.data.frame(rxkm$centers),
            long = long_std*(-74),
            lat = lat_std*40),
  size = rxkm$size, withinss = rxkm$withinss)

head(clsdf)
```

```
Time difference of 2.529844 hours

       long      lat   size      withinss
1 -73.96431 40.80540 301784 0.00059328668
2 -73.99275 40.73042 171080 0.00007597645
3 -73.98032 40.76031 198077 0.00005138354
```

```
4 -73.98828 40.77187 134539 0.00011077493
5 -73.96651 40.75752 133927 0.00004789548
6 -73.98446 40.74836 186906 0.00005435595
```

Il mio sistema ha impiegato poco più di due ore e mezza per eseguire tutti i calcoli. Possiamo ora estrarre i centroidi del cluster dall'oggetto risultante e tracciarli su una mappa e confrontare i centroidi calcolati sul sample data con quelli dei dati completi.

```
centroids_whole <- cbind(
  transmute(as.data.frame(rxkm$centers), long = long_std*(-74), lat = lat_std*40),
  size = rxkm$size, withinss = rxkm$withinss)

q1 <- ggmap(map_15) +
  geom_point(data = centroids_sample, aes(x = long, y = lat, alpha = size),
             na.rm = TRUE, size = 1, col = 'red') +
  theme_nothing(legend = TRUE) +
  labs(title = "centroids using sample data")

q2 <- ggmap(map_15) +
  geom_point(data = centroids_whole, aes(x = long, y = lat, alpha = size),
             na.rm = TRUE, size = 1, col = 'red') +
  theme_nothing(legend = TRUE) +
  labs(title = "centroids using whole data")

require(gridExtra)
grid.arrange(q1, q2, ncol = 2)
```



Come possiamo vedere, i risultati non sono molto diversi, tuttavia esistono differenze e, a seconda del caso d'uso, tali piccole differenze possono avere un grande significato pratico. Se, ad esempio, volessimo scoprire quali sono i luoghi in cui i taxi sono più propensi a lasciare i passeggeri e vietare ai venditori ambulanti di operare in quei punti (per non creare troppo traffico), possiamo fare un lavoro molto più preciso utilizzando i cluster creati utilizzando tutti i dati.

Elaborazione modello per prevedere la variabile tip_percent

Inizo creando un modello lineare che coinvolge due variabili interattive: uno tra *pickup_nb* e *dropoff_nb* e un altro tra *pickup_dow* e *pickup_hour*. L'idea è che vorrei dimostrare che la percentuale di mancia non sia solo influenzata da quale quartiere è stato prelevato il passeggero o a quale quartiere siano stati portati solamente, ma anche da quale quartiere sono stati prelevati e a quale quartiere siano stati portati in correlazione tra loro. Allo stesso modo, vorrei capire se il giorno della settimana e l'ora del giorno possono influenzano la mance nella previsione del modello. Ad esempio, solo perché le persone lasciano molte mance la domenica tra 9 e 12 post meridiane, non è detto che lascino mance alte ogni giorno della settimana tra le 9 e le 12 PM, o in qualsiasi altro momento della giornata di domenica. Questa intuizione è codificata nell'argomento formula del modello che passiamo alla funzione `rxLinMod`: `tip_percent ~ pickup_nb: dropoff_nb + pickup_dow: pickup_hour` dove usiamo `:` per separare i termini interattivi e `+` per separare termini additivi.

```
form_1 <- as.formula(tip_percent ~ pickup_nb:dropoff_nb + pickup_dow:pickup_hour)
rxlm_1 <- rxLinMod(form_1, data = mht_xdf, dropFirst = TRUE, covCoef = TRUE)
```

Esaminare i coefficienti del modello individualmente è un compito noioso a causa della quantità. Inoltre, quando si lavora con Big datasets, molti coefficienti vengono rilevati come statisticamente significativi in virtù di una grande dimensione del campione, senza necessariamente essere praticamente significativi. Invece per ora guarderò solo come stanno le nostre previsioni. Inizio estraendo i livelli di ciascuna variabile categorica in una lista che possiamo passare a `expand.grid` per creare un set di dati con tutte le possibili combinazioni dei livelli. Quindi usiamo `rxPredict` per prevedere `tip_percent` usando il modello sopra.

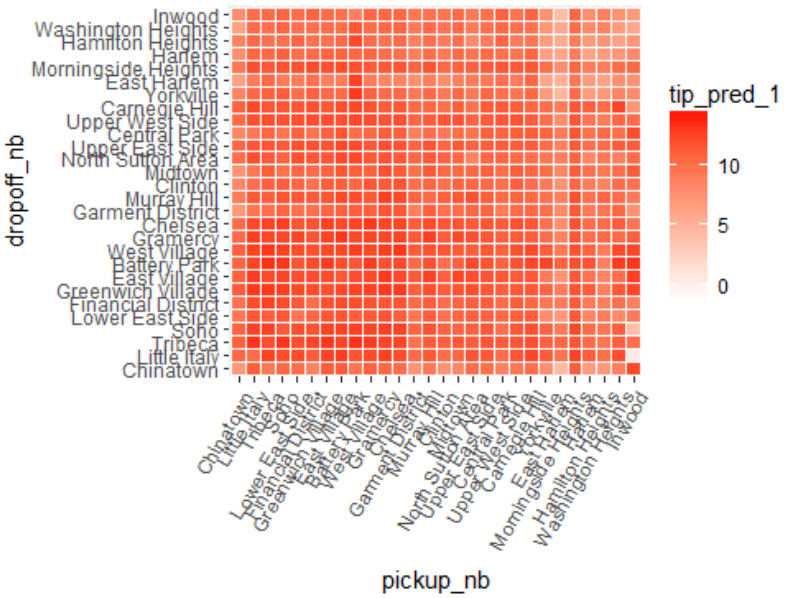
```
rxs <- rxSummary( ~ pickup_nb + dropoff_nb + pickup_hour + pickup_dow, mht_xdf)
ll <- lapply(rxs$categorical, function(x) x[ , 1])
names(ll) <- c('pickup_nb', 'dropoff_nb', 'pickup_hour', 'pickup_dow')
pred_df_1 <- expand.grid(ll)
pred_df_1 <- rxPredict(rxlm_1,
```

```
                        data = pred_df_1,
                        computeStdErrors = TRUE,
                        writeModelVars = TRUE)
names(pred_df_1)[1:2] <- paste(c('tip_pred', 'tip_stderr'), 1, sep = "_")
head(pred_df_1, 10)
```

	tip_pred_1	tip_stderr_1	pickup_nb	dropoff_nb	pickup_dow	pickup_hour
1	6.796323	0.16432197	Chinatown	Chinatown	Sun	1AM-5AM
2	10.741766	0.15853956	Little Italy	Chinatown	Sun	1AM-5AM
3	9.150114	0.09162002	Tribeca	Chinatown	Sun	1AM-5AM
4	10.174307	0.09819651	Soho	Chinatown	Sun	1AM-5AM
5	9.706202	0.07365164	Lower East Side	Chinatown	Sun	1AM-5AM
6	8.475197	0.06354026	Financial District	Chinatown	Sun	1AM-5AM
7	10.866035	0.07150005	Greenwich Village	Chinatown	Sun	1AM-5AM
8	10.997276	0.06831955	East Village	Chinatown	Sun	1AM-5AM
9	9.313165	0.12507373	Battery Park	Chinatown	Sun	1AM-5AM
10	10.613802	0.11624956	West Village	Chinatown	Sun	1AM-5AM

Ora possiamo visualizzare le previsioni del modello tracciando le previsioni medie per tutte le combinazioni delle variabili interattive.

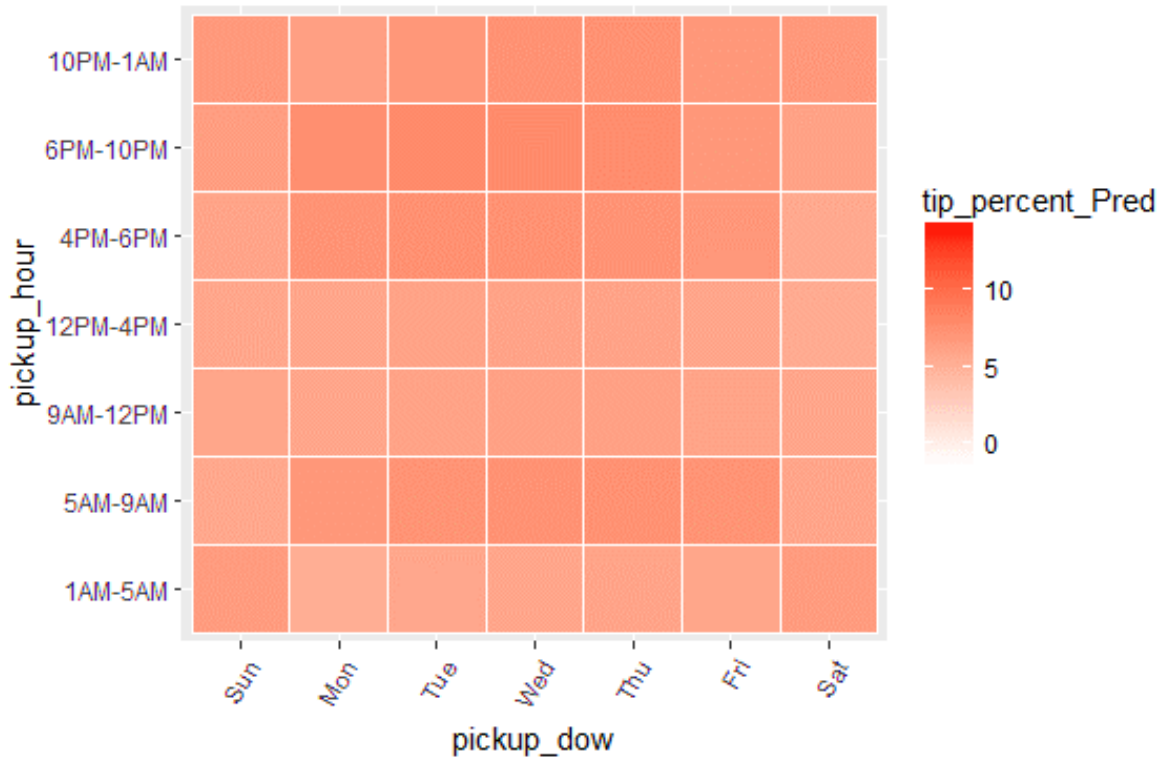
```
ggplot(pred_df_1, aes(x = pickup_nb, y = dropoff_nb)) +
  geom_tile(aes(fill = tip_pred_1), colour = "white") +
  theme(axis.text.x = element_text(angle = 60, hjust = 1)) +
  scale_fill_gradient(low = "white", high = "red") +
  coord_fixed(ratio = .9)
```



Il grafico mostra le percentuali di mancia predette per i quartieri. Fornisce un'idea generale di come il modello distribuisce le mance. Ma non abbiamo una buona visualizzazione di come si comporta.

Il prossimo sarà in combinazione con il giorno della settimana e ora del giorno.

```
ggplot(pred_df_1, aes(x = pickup_dow, y = pickup_hour)) +
  geom_tile(aes(fill = tip_pred_1), colour = "white") +
  theme(axis.text.x = element_text(angle = 60, hjust = 1)) +
  scale_fill_gradient(low = "white", high = "red") +
  coord_fixed(ratio = .9)
```



E ancora una volta possiamo notare alcune tendenze che abbiamo visto anche quando stavamo guardando i dati in precedenza. Vale a dire, i clienti che vengono prelevati durante l'ora di punta del mattino in un giorno feriale o nell'ora di punta del pomeriggio in un giorno della settimana, tendono ad essere generalmente più generosi rispetto alle persone che viaggiano in qualsiasi altro momento della giornata. Quindi alcune di queste tendenze sembrano le stesse ed è un buon segno sapere che il modello è stato capace di catturare parte della variabilità che abbiamo notato nei dati.

Una domanda che potremmo porci adesso sarebbe; quanto è importante l'interazione tra *pickup_dow* e *pickup_hour* nelle previsioni? Quanto peggiorerebbero le previsioni se avessimo mantenuto l'interazione tra *pickup_nb* e *dropoff_nb* e abbandonato la seconda variabile interattiva? Per rispondere a queste domande, possiamo costruire un modello più semplice con `rxLinMod` in cui includiamo solo `pickup_nb:dropoff_nb`.

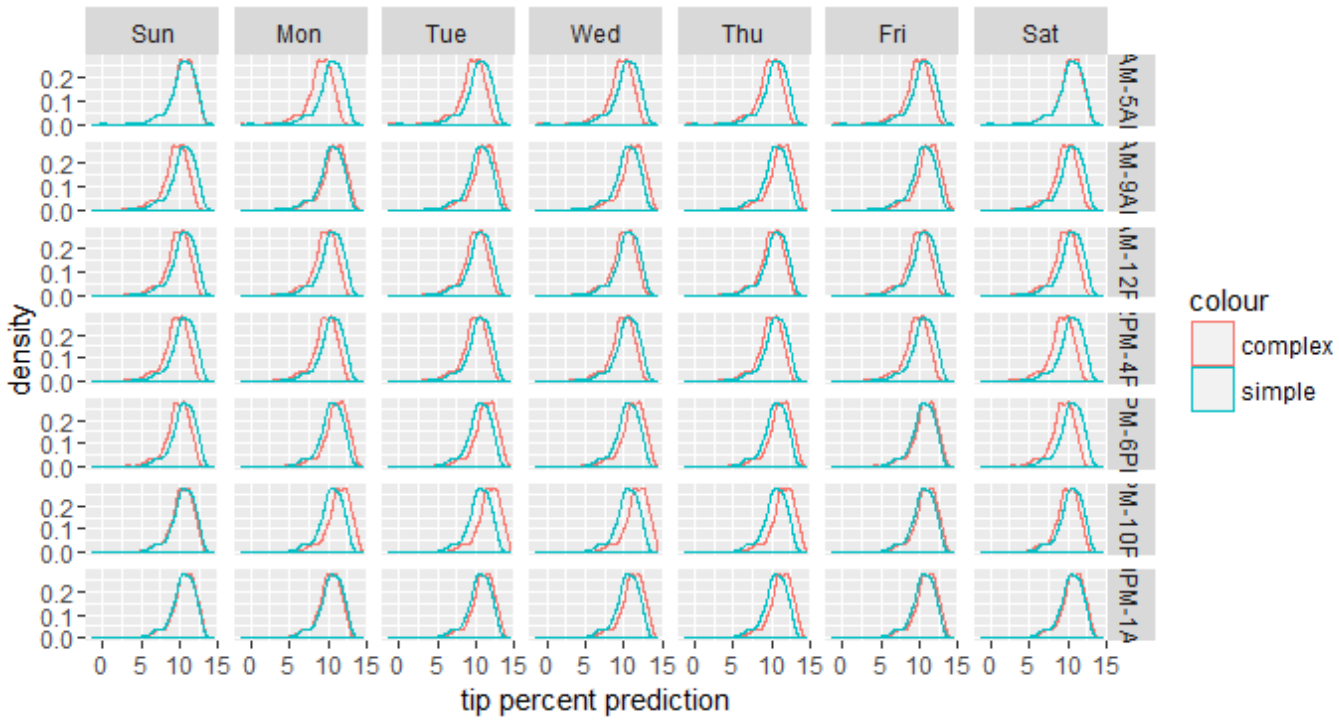
```
form_2 <- as.formula(tip_percent ~ pickup_nb:dropoff_nb)
rxlm_2 <- rxLinMod(form_2, data = mht_xdf, dropFirst = TRUE, covCoef = TRUE)
pred_df_2 <- rxPredict(rxlm_2,
                       data = pred_df_1,
                       computeStdErrors = TRUE,
                       writeModelVars = TRUE)
names(pred_df_2)[1:2] <- paste(c('tip_pred', 'tip_stderr'), 2, sep = "_")

pred_df <- pred_df_2 %>%
  select(starts_with('tip_')) %>%
  cbind(pred_df_1) %>%
  arrange(pickup_nb, dropoff_nb, pickup_dow, pickup_hour) %>%
  select(pickup_dow,
         pickup_hour,
         pickup_nb,
         dropoff_nb,
         starts_with('tip_pred_'))

head(pred_df)
```

```
  pickup_dow pickup_hour pickup_nb dropoff_nb tip_pred_2 tip_pred_1
1        Sun     1AM-5AM Chinatown  Chinatown   6.782043   6.796323
2        Sun     5AM-9AM Chinatown  Chinatown   6.782043   5.880284
3        Sun    9AM-12PM Chinatown  Chinatown   6.782043   6.103625
4        Sun    12PM-4PM Chinatown  Chinatown   6.782043   5.913130
5        Sun     4PM-6PM Chinatown  Chinatown   6.782043   6.121957
6        Sun    6PM-10PM Chinatown  Chinatown   6.782043   6.642192
```

Possiamo vedere dai risultati sopra che le previsioni con il modello più semplice sono identiche per tutti i giorni della settimana e tutte le ore per la stessa combinazione pick-up e drop-off. Mentre le previsioni del modello più complesso sono uniche per ogni combinazione di tutte e quattro le variabili. In altre parole, `aggiungendo pickup_dow:pickup_hour` al modello, si aggiunge una variazione extra alle previsioni e ciò che vorremmo sapere è se questa variazione contiene segnali importanti o se si comporta più o meno come semplice rumore.

Per ottenere la risposta, confrontiamo la distribuzione delle due previsioni suddividendole per *pickup_dow* e *pickup_hour*.

```
ggplot(data = pred_df) +
  geom_density(aes(x = tip_pred_1, col = "complex")) +
  geom_density(aes(x = tip_pred_2, col = "simple")) +
  facet_grid(pickup_hour ~ pickup_dow)
```



Il modello più semplice mostra la stessa distribuzione per tutto il grafico, perché queste due variabili non hanno alcun effetto sulle sue previsioni, ma il modello più complesso mostra una distribuzione leggermente diversa per ciascuna combinazione di *pickup_dow* e *pickup_hour*, sotto forma di un leggero scostamento nella distribuzione (simile ad un effetto stereoscopico). Quel effetto è dato da *pickup_dow* e *pickup_hour* ad ogni combinazione delle due variabili. Poiché lo scostamento direzionale (non casuale), cattura un qualche tipo di segnale importante (anche se il suo significato pratico è ancora poco chiaro). Possiamo semplificare il grafico se applichiamo qualche logica di business ad esso.

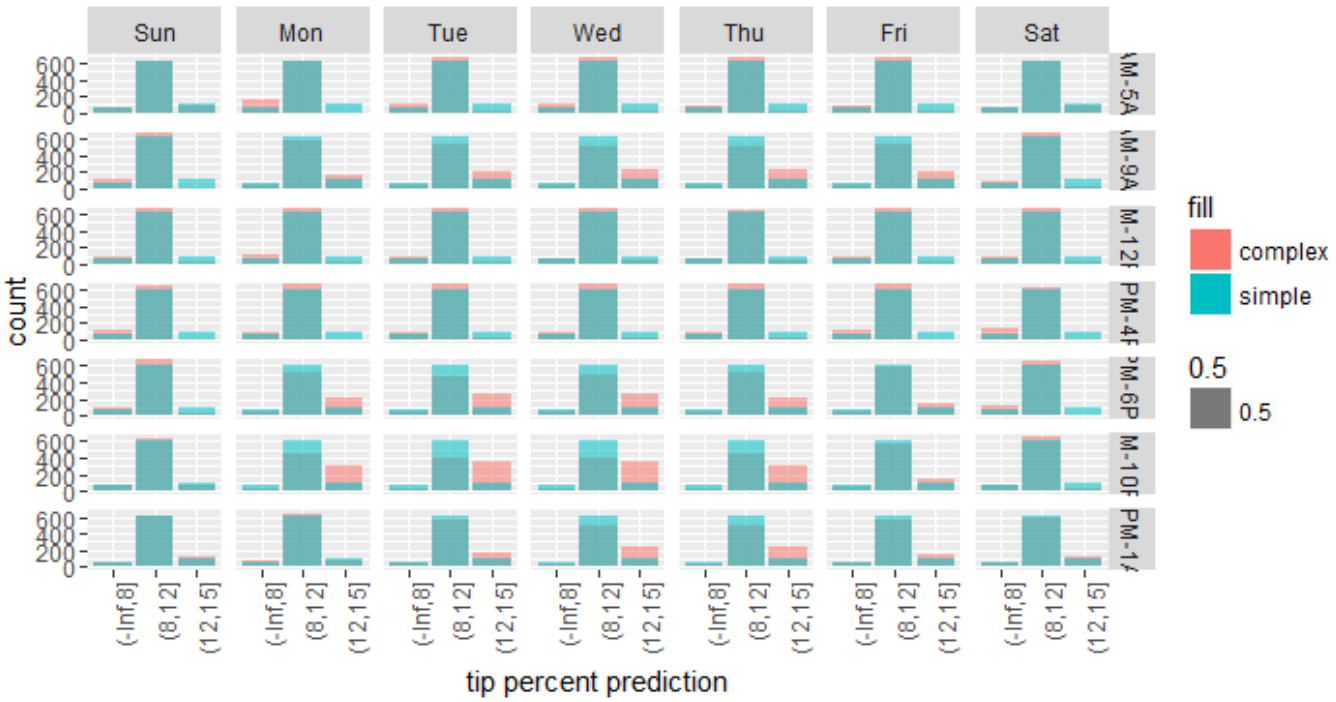
Utilizzo `cut` per raggruppare le previsioni. Per scegliere i *cut-off*, posso usare la funzione `rxQuantile e vedere le distribuzioni`.

```
rxQuantile("tip_percent", data = mht_xdf, probs = seq(0, 1, by = .05))
```

```
 0%    5%   10%   15%   20%   25%   30%   35%   40%   45%   50%   55%   60%   65%   70%   75%
80%
  -1     0     0     0     0     0     0     0     9    12    15    17    17    17    18    18
19
 85%   90%   95% 100%
  20    21    23    99
```

In base ai risultati sopra riportati, possiamo raggruppare *tip_percent* quando; è inferiore all'8%, tra l'8% e il 12%, tra il 12% e il 15%, tra il 15% e il 18% o il 18% o più. Possiamo quindi tracciare un grafico a barre che mostri le stesse informazioni di prima, ma leggermente più facile da interpretare.

```
pred_df %>%
  mutate_at(vars(tip_pred_1, tip_pred_2), funs(cut(.,
                                                   c(-Inf, 8, 12, 15, 18, Inf)))) %>%
  ggplot() +
  geom_bar(aes(x = tip_pred_1, fill = "complex", alpha = .5)) +
  geom_bar(aes(x = tip_pred_2, fill = "simple", alpha = .5)) +
  facet_grid(pickup_hour ~ pickup_dow) +
  xlab('tip percent prediction') +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



Sulla base del grafico sopra, possiamo vedere che rispetto al modello semplice, il modello complesso tende a prevedere più clienti taxi generosi di mance e meno quelli di medio livello durante determinate combinazioni di giorno e di tempo (come dal lunedì al giovedì durante le ore di punta).

Finora abbiamo analizzato i modelli senza preoccuparci della loro efficacia di previsione. Per verificarlo abbiamo bisogno di suddividere il dataset in due parti una per il test delle previsione e un altro per l'apprendimento.

Per dividere i dati, prima userò `rxDataStep` per creare una nuova colonna chiamata *split* Per dividere i dati in parti di *training* e *testing*, la nuova colonna avrà ogni riga una dicitura come "*train*" o "*test*" in modo tale che una determinata proporzione dei dati (qui sceglierò il 75 percento) venga utilizzata per l'apprendimento del modello e il resto utilizzato per testare la potenza predittiva. Utilizzo quindi la funzione `rxSplit` per dividere i dati. La funzione `rx_split_xdf` che verrà creata combina i due passaggi in uno e imposta alcuni argomenti sui valori predefiniti.

```
dir.create('output', showWarnings = FALSE)
rx_split_xdf <- function(xdf = mht_xdf,
                         split_perc = 0.75,
                         output_path = "output/split",
                         ...) {

  # prima creo la colonna per la suddivisione
  outFile <- tempfile(fileext = 'xdf')
  rxDataStep(inData = xdf,
             outFile = xdf,
             transforms = list(
               split = factor(ifelse(rbinom(.rxNumRows,
                                            size = 1,

                                            prob = splitperc),
                                     "train", "test"))),
             transformObjects = list(splitperc = split_perc),
             overwrite = TRUE, ...)

  # quindi suddivido i dati in due in base alla colonna che abbiamo appena creato
  splitDS <- rxSplit(inData = xdf,
                     outFilesBase = file.path(output_path, "train"),
                     splitByFactor = "split",
                     overwrite = TRUE)
```

```
  return(splitDS)
}

# we can now split to data in two
mht_split <- rx_split_xdf(xdf = mht_xdf, varsToKeep = c('payment_type',
                                                        'fare_amount',
                                                        'tip_amount',
                                                        'tip_percent',
                                                        'pickup_hour',
                                                        'pickup_dow',
                                                        'pickup_nb',
                                                        'dropoff_nb'))
names(mht_split) <- c("train", "test")
```

Ora eseguiamo tre diversi algoritmi:

- `rxLinMod`, il modello lineare precedente con i termini `tip_percent ~ pickup_nb:dropoff_nb + pickup_dow:pickup_hour`
- `rxDTree`, l'algoritmo ad albero decisionale con i termini `tip_percent ~ pickup_nb + dropoff_nb + pickup_dow + pickup_hour` (gli alberi decisionali non hanno bisogno di fattori interattivi perché le interazioni sono incorporate nell'algoritmo stesso)
- `rxDForest`, l'algoritmo di foresta casuale con gli stessi termini sopra.

```
system.time(linmod <- rxLinMod(tip_percent ~ pickup_nb:dropoff_nb
                               + pickup_dow:pickup_hour,
                               data = mht_split$train,
                               reportProgress = 0))
system.time(dtree <- rxDTree(tip_percent ~ pickup_nb
                             + dropoff_nb + pickup_dow + pickup_hour,
                             data = mht_split$train,
                             pruneCp = "auto", reportProgress = 0))
system.time(dforest <- rxDForest(tip_percent ~ pickup_nb
                                 + dropoff_nb + pickup_dow + pickup_hour,
                                  mht_split$train, nTree = 10, importance = TRUE,
                                  useSparseCube = TRUE, reportProgress = 0))
```

```
   user  system elapsed
```

```
   0.00    0.00    1.62

   user  system elapsed
   0.03    0.00  778.00

   user  system elapsed
   0.02    0.00  644.17
```

Poiché l'esecuzione degli algoritmi possono richiedere tanto tempo, può valere la pena di salvare i modelli così nel caso perdessimo la nostra sessione R o in cui dobbiamo riavviare, non è necessario eseguire i modelli una seconda volta. Possiamo semplicemente recuperarli dalla nostra directory e usarli per fare previsioni. Questo è anche molto utile se creiamo i modelli localmente sui nostri laptop. E poi vogliamo usarli in Hadoop o su di un server SQL o un server più capiente con più dati in esso.

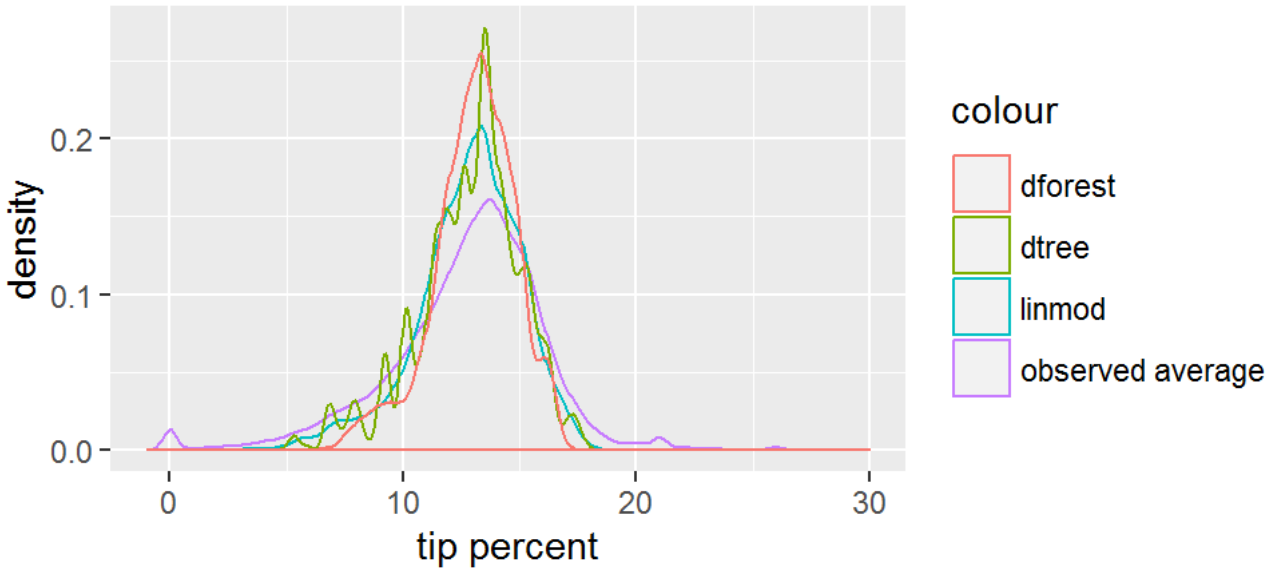
```
trained.models <- list(linmod = linmod, dtree = dtree, dforest = dforest)
save(trained.models, file = 'trained_models.Rdata')
```

Prima di applicare l'algoritmo a tutto il dataset, applico il modello al sample dataset con tutte le combinazioni di variabili categoriali e visualizzo i risultati. Questo potrebbe aiutarci a sviluppare nuove intuizioni migliorative su ciascun algoritmo.

```
pred_df <- expand.grid(ll)
pred_df_1 <- rxPredict(trained.models$linmod,
                       data = pred_df, predVarNames = "pred_linmod")
pred_df_2 <- rxPredict(trained.models$dtree,
                       data = pred_df, predVarNames = "pred_dtree")
pred_df_3 <- rxPredict(trained.models$dforest,
                       data = pred_df, predVarNames = "pred_dforest")
pred_df <- do.call(cbind, list(pred_df, pred_df_1, pred_df_2, pred_df_3))
head(pred_df)
```

```
           pickup_nb dropoff_nb pickup_hour pickup_dow pred_linmod pred_dtree pred_dforest
1          Chinatown  Chinatown     1AM-5AM        Sun 6.869645    5.772054   9.008643
2       Little Italy  Chinatown     1AM-5AM        Sun 10.627190   9.221250   10.634590
3            Tribeca  Chinatown     1AM-5AM        Sun 9.063741    9.221250   10.099731
4               Soho  Chinatown     1AM-5AM        Sun 10.107815   8.313437   10.162946
5    Lower East Side  Chinatown     1AM-5AM        Sun 9.728399    9.221250   10.525242
6 Financial District  Chinatown     1AM-5AM        Sun 8.248997    6.937500   8.674807
```

```
observed_df<-rxSummary(tip_percent~pickup_nb:dropoff_nb:pickup_dow:pickup_hour,
                       mht_xdf)
observed_df <- observed_df$categorical[[1]][ , c(2:6)]
pred_df <- inner_join(pred_df, observed_df, by = names(pred_df)[1:4])

ggplot(data = pred_df) +
  geom_density(aes(x = Means, col = "observed average")) +
  geom_density(aes(x = pred_linmod, col = "linmod")) +
  geom_density(aes(x = pred_dtree, col = "dtree")) +
  geom_density(aes(x = pred_dforest, col = "dforest")) +
  xlim(-1, 30) +
  xlab("tip percent")
```



Sia il modello lineare che la foresta casuale ci forniscono previsioni fluide. Possiamo vedere che le previsioni *dforest* sono le più concentrate. Le previsioni per *dtree* seguono una distribuzione frastagliata, probabilmente a causa del sovradattamento (overfitting), ma non lo sappiamo fino a quando non controlleremo le prestazioni rispetto al set di test. Nel complesso, le previsioni sono più ristrette rispetto alla media reale.

Ora applichiamo il modello ai dati del test in modo da poter confrontare il potere predittivo di ciascun modello. Se abbiamo ragione sul fatto che l'algoritmo *decision tree* si adatta in modo eccessivo, dovremmo vederlo preformare peggio nei dati del test rispetto agli altri due modelli. Se crediamo che la foresta casuale catturi alcuni segnali intrinseci nei dati che mancano al modello lineare, dovremmo vederlo funzionare meglio del modello lineare sui dati del test.

La prima metrica che osserviamo è la media dei residui al quadrato, che ci dà un'idea di quanto siano vicine le previsioni ai valori osservati. Dato che prevediamo la percentuale di mancia (*tip percent*), che di solito scende in un intervallo ristretto compreso tra lo 0 e il 20 percento, dovremmo aspettarci in media che i residui per un buon modello non siano più di 2 o 3 punti percentuale.

```
rxPredict(trained.models$linmod,
          data = mht_split$test,
          outData = mht_split$test,
          predVarNames = "tip_percent_pred_linmod",
          overwrite = TRUE)
rxPredict(trained.models$dtree,
          data = mht_split$test,
          outData = mht_split$test,
          predVarNames = "tip_percent_pred_dtree",
          overwrite = TRUE)
rxPredict(trained.models$dforest,
          data = mht_split$test,
          outData = mht_split$test,
          predVarNames = "tip_percent_pred_dforest",
          overwrite = TRUE)

rxSummary(~ SE_linmod + SE_dtree + SE_dforest, data = mht_split$test,
          transforms = list(SE_linmod = (tip_percent - tip_percent_pred_linmod)^2,
                            SE_dtree = (tip_percent - tip_percent_pred_dtree)^2,
                            SE_dforest = (tip_percent
                                          - tip_percent_pred_dforest)^2))
```

```
Call:
rxSummary(formula = ~ SE_linmod + SE_dtree + SE_dforest, data = mht_split$test,
    transforms = list(SE_linmod = (tip_percent - tip_percent_pred_linmod)^2,
        SE_dtree = (tip_percent - tip_percent_pred_dtree)^2,
        SE_dforest = (tip_percent - tip_percent_pred_dforest)^2))

Summary Statistics Results for: ~SSE_linmod + SSE_dtree + SSE_dforest
Data: mht_split$test (RxXdfData Data Source)
File name: C:\Data\NYC_taxi\output\split\train.split.train.xdf
Number of valid observations: 43118543

 Name       Mean     StdDev   Min                 Max     ValidObs MissingObs
 SE_linmod  82.66458 108.9904 0.00000000005739206 9034.665 43118542 1
 SE_dtree   82.40040 109.1038 0.00000251589457986 8940.693 43118542 1
 SE_dforest 82.47107 108.0416 0.00000000001590368 8606.201 43118542 1
```

Un'altra metrica che vale la pena di osservare è una matrice di correlazione. Questo può aiutarci a determinare in quale misura le previsioni dei diversi modelli siano vicine l'una all'altra e in che misura ciascuna si avvicina alla percentuale di mancia effettiva o osservata.

```
rxc <- rxCor( ~ tip_percent
              + tip_percent_pred_linmod
              + tip_percent_pred_dtree
              + tip_percent_pred_dforest, data = mht_split$test)
print(rxc)
```

```
                         tip_percent pred_linmod pred_dtree pred_dforest
tip_percent              1.0000000   0.1391751   0.1500126  0.1499031
tip_percent_pred_linmod  0.1391751   1.0000000   0.8580617  0.9084119
tip_percent_pred_dtree   0.1500126   0.8580617   1.0000000  0.9404640
tip_percent_pred_dforest 0.1499031   0.9084119   0.9404640  1.0000000
```

Possiamo vedere che c'è solo una leggera differenza di correlazione tra i modelli e *tip-percent*. La predizione dal decision tree risulta la migliore. Possiamo anche vedere che la correlazione tra le previsioni sono abbastanza vicine tra loro. La foresta casuale fa previsioni che sono simili al modello lineare e piuttosto vicine a quelle dell'albero decisionale. Quindi sembra che ci troviamo di fronte a una situazione dove abbiamo tre diversi modelli, tutti e tre non stanno facendo un buon lavoro di previsione. Ma tutti e tre comunque stanno facendo predizioni che sono ragionevoli e vicine l'una all'altra. Tutti e tre i modelli non sono ancora elaborati in maniera completa. Possiamo renderli più ricchi aggiungendo altre variabili, aggiungendo altri input.

Dopo alcuni esperimenti i risultati ottenuti con il modello lineare sono:

Linear Model Formula Predictors	*R-sqd (adj)*
tip_percent~trip_duration + pickup_dow : pickup_hour	**< 0.01**
tip_percent~trip_duration + payment_type_desc + pickup_dow : pickup_hour	**> 0.70**

Potete vedere che l'R2 corretto è aumentato parecchio rispetto al modello precedente. Come abbiamo esaminato in precedenza, quasi tutti i pagamenti in contanti mostrano una percentuale zero di mancia, rendendo *payment_type_desc* una caratteristica fondamentale per la previsione.

Deploying e Ridimensionamento

Una volta che un modello è stato testato con successo e scelto, di solito si è interessati ad usarlo per fare previsioni su dati futuri (*scoring*). La procedura non è molto diversa da come è stato usato il modello nell'ultima sezione per fare previsioni sui dati di test con la funzione `rxPredict`. Ma i dati futuri potrebbero risiedere in un ambiente differente da quello utilizzato per sviluppare il modello.

Ad esempio, supponiamo di voler predire regolarmente nuovi dati quando arrivano in un server SQL. Senza `RevoScaleR`, potrebbe essere necessario portare i nuovi dati fuori da SQL Server (ad esempio su file o utilizzando una connessione ODBC) in una macchina con R installato. Dovremmo quindi caricare i dati in R per segnarlo e inviare i punteggi ad SQL Server. Lo spostamento dei dati è inefficiente e spesso può uscire dai protocolli di sicurezza sulla governance dei dati. Inoltre, se i dati che stiamo cercando di predire sono troppo grandi per rientrare nella sessione R, dovremmo anche eseguire questo processo usando una porzione di dati alla volta. Microsoft R invece utilizza l'efficiente funzione `rxPredict` che può fare uno *scoring* dei dati senza doverli caricare in una sessione R nella sua interezza.

Poiché `RevoScaleR` è dotato di algoritmi di modellazione parallela e di apprendimento automatico, possiamo anche utilizzarli per sviluppare modelli su dataset molto più grandi che si trovano in SQL Server o su HDFS. Ciò riduce considerevolmente il divario tra lo sviluppo e l'ambiente di produzione.

Scoring su SQL Server

Come prima cosa indichiamo una tabella SQL contenente una copia del set di dati NYC Taxi. Bisogna impostare una stringa di connessione SQL Server, che contiene le credenziali di accesso SQL. Poiché la stringa di connessione contiene informazioni sensibili, di solito viene memorizzata in un file in una posizione limitata e letta da R, oppure può essere codificato tramite stringa di connessione e memorizzata in `sqlConnString`. Supponiamo che il set di dati di NYC Taxi sia memorizzato in una tabella chiamata NYCTaxiBig all'interno del database RDB a cui punta la stringa di connessione. L'ultima cosa che resta da fare è puntare alla tabella, cosa che facciamo con la funzione `RxSqlServerData`. Questo è l'equivalente di `RxXdfData` quando si punta a un file XDF memorizzato su disco.

```
sqlConnString <- "Driver=SQL Server;
                  Server=SETHMOTTDSVM;
                  Database=RDB;
                  Uid=ruser;Pwd=ruser"
sqlRowsPerRead <- 100000
sqlTable <- "NYCTaxiBig"

nyc_sql <- RxSqlServerData(connectionString = sqlConnString,
                           rowsPerRead = sqlRowsPerRead, table = sqlTable)
```

Ora scarichiamo il contenuto di *nyc_xdf* nella tabella SQL rappresentata da *nyc_sql* (che si chiama `NYCTaxiBig` nel database SQL). Se il file XDF in questione è molto grande, questo può richiedere molto tempo.

```
system.time(
  rxDataStep(nyc_xdf, nyc_sql, overwrite = TRUE)
)
```

Ora possiamo usare `nyc_sql` nello stesso modo in cui abbiamo usato `nyc_xdf` in precedenza. Non ho però specificato quali fossero i tipi di colonna. In questo caso, `RxSqlServerData` cercherà al meglio possibile di convertire il tipo di colonna di SQL Server in un tipo di colonna R. Questo può causare problemi però. Innanzitutto, SQL Server presenta una varietà più ricca di tipi di colonne rispetto a R. In secondo luogo, alcuni tipi di colonne di SQL Server come ad esempio *datetime* non vengono sempre trasferiti correttamente al tipo di colonna R corrispondente. In terzo luogo, il fattore di tipo di colonna R non ha un buon equivalente in SQL Server, quindi per inserire una colonna come fattore dobbiamo specificarlo manualmente. Ciò tuttavia ci dà il vantaggio che possiamo anche specificare i livelli e le etichette per esso e, come abbiamo visto, non sempre devono essere i livelli esatti che vediamo nei dati. Ad esempio, se *payment_type* è rappresentato dagli interi da 1 a 5 nei dati, ma ci interessa solo 1 e 2 e li vogliamo rispettivamente con carta e contanti, possiamo farlo qui senza che sia necessario farlo in seguito come trasformazione separata. Per gestire i tipi di colonna, creiamo un oggetto che memorizza le informazioni sulle colonne e le passa all'argomento `colInfo` in `RxSqlServerData`.

```
ccColInfo <- list(
  tpep_pickup_datetime = list(type = "character"),
  tpep_dropoff_datetime = list(type = "character"),
  passenger_count = list(type = "integer"),
  trip_distance = list(type = "numeric"),
  pickup_longitude = list(type = "numeric"),
  pickup_latitude = list(type = "numeric"),
```

```
  dropoff_longitude = list(type = "numeric"),
  dropoff_latitude = list(type = "numeric"),
  RateCodeID = list(type = "factor",
                    levels = as.character(1:6),
                    newLevels = c("standard",
                                  "JFK",
                                  "Newark",
                                  "Nassau or Westchester",
                                  "negotiated",
                                  "group ride")),
  store_and_fwd_flag = list(type = "factor",
                            levels = c("Y", "N")),
                      payment_type = list(type = "factor",
                                          levels = as.character(1:2),
                                          newLevels = c("card", "cash")),
                                          fare_amount = list(type = "numeric"),
                                          tip_amount = list(type = "numeric"),
                                          total_amount = list(type = "numeric")
)
```

Da notare che non è necessario specificare i tipi per ciascuna colonna nei dati. Possiamo limitarlo solo alle colonne di interesse, e anche solo quelle che devono essere esplicitamente sovrascritte. Tuttavia, poiché tendiamo a essere più prudenti in un ambiente di produzione, è meglio essere più espliciti. Dopo tutto, alcune colonne numeriche in SQL Server potrebbero essere archiviate come qualcosa di diverso (ad esempio VARCHAR) che si trasformerebbe in una colonna di caratteri in R.

Oltre alle colonne presenti nei dati originali, dobbiamo anche specificare i tipi di colonna per le colonne che abbiamo aggiunto ai dati durante l'analisi.

```
weekday_labels <- c('Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat')
hour_labels <- c('1AM-5AM', '5AM-9AM', '9AM-12PM',
                 '12PM-4PM', '4PM-6PM', '6PM-10PM', '10PM-1AM')

ccColInfo$pickup_dow <- list(type = "factor", levels = weekday_labels)
ccColInfo$pickup_hour <- list(type = "factor", levels = hour_labels)
ccColInfo$dropoff_dow <- list(type = "factor", levels = weekday_labels)
ccColInfo$dropoff_hour <- list(type = "factor", levels = hour_labels)
```

Quando si distribuisce il codice in produzione è una buona norma fare un ulteriore check al codice e semplificare le cose ovunque sia necessario. Ad esempio, quando si lavora con il file XDF, abbiamo prima scritto una funzione per estrarre le colonne *pickup_nhood* e *dropoff_nhood* dalle coordinate *pick-up* e *drop-off*. Abbiamo poi notato

che quelle colonne contengono quartieri al di fuori dei limiti di Manhattan (la nostra area di interesse), quindi abbiamo fatto un secondo passaggio attraverso i dati per rimuovere i livelli dei quartieri irrilevanti. Con `nyc_sql`, potremmo adottare un approccio simile: leggere tali colonne con livelli così come sono e quindi utilizzare `rxDataStep` per eseguire una trasformazione che rimuova i livelli indesiderati. L'approccio migliore consiste nel trovare tutti i livelli rilevanti (i quartieri di Manhattan, che possiamo ottenere direttamente dallo *shapefile*) e nell'oggetto `ccColInfo` specificare solo quelli per noi interessanti.

```
yc_shapefile <- readShapePoly('ZillowNeighborhoods-NY/ZillowNeighborhoods-NY.shp')
mht_shapefile <- subset(nyc_shapefile, str_detect(CITY, 'New York City-Manhattan'))
manhattan_nhoods <- as.character(mht_shapefile@data$NAME)

ccColInfo$pickup_nhood <- list(type = "factor", levels = manhattan_nhoods)
ccColInfo$dropoff_nhood <- list(type = "factor", levels = manhattan_nhoods)
```

Ora siamo pronti a puntare alla tabella SQL una seconda volta, ma questa volta specifichiamo come le colonne dovrebbero essere trattate in R usando l'argomento `colInfo`.

```
nyc_sql <- RxSqlServerData(connectionString = sqlConnString,
                           table = sqlTable,
                           rowsPerRead = sqlRowsPerRead,
                           colInfo = ccColInfo)
```

A questo punto, il resto dell'analisi non è diverso da quello che era con il file XDF, quindi possiamo modificare *nyc_xdf* in *nyc_sql* ed eseguire il codice rimanente come prima. Ad esempio, possiamo iniziare con `rxGetInfo` e `rxSummary` per ricontrollare i tipi di colonna e ottenere alcune statistiche di riepilogo.

```
rxGetInfo(nyc_sql, getVarInfo = TRUE, numRows = 3)
# mostra tipi di Colonna e prime 10 righe

system.time(
  rxsum_sql <- rxSummary( ~ ., nyc_sql)
# statistical summaries per tutte le colonne
)
rxsum_sql
```

Eseguiamo ora lo stesso modello lineare utilizzato sul file XDF usando ora la tabella SQL. Costruiremo il modello sul 75 percento dei dati (di apprendimento) creando una

colonna *u* di numeri uniformi casuali e usando `rowSelection` selezionando solo righe in cui *u* <0.75.

```
system.time(linmod <- rxLinMod(tip_percent ~ pickup_nhood:dropoff_nhood
                               + pickup_dow:pickup_hour,
                               data = nyc_sql, reportProgress = 0,
                               rowSelection = (u < .75)))
```

Ora puntiamo a una nuova tabella SQL chiamata `NYCTaxiScore` (il nostro puntatore in R sarà chiamato *nyc_score*). Quindi usiamo `rxPredict` per fare lo *scoring* dei dati di test (righe in *nyc_sql* dove *u*> = .75) e scaricare i risultati.

```
sqlTable <- "NYCTaxiScore"
nyc_score <- RxSqlServerData(connectionString = sqlConnString,
                             rowsPerRead = sqlRowsPerRead,
                             table = sqlTable)

rxPredict(trained.models$linmod,
          data = nyc_sql,
          outData = nyc_score,
          predVarNames = "tip_percent_pred_linmod",
          overwrite = TRUE,
          rowSelection = (u >= .74))
```

Tutto il calcolo fino ad ora è avvenuto nella macchina client (quella che ospita la nostra attuale sessione R). Di solito, la macchina client (che potrebbe essere il nostro laptop) non è la stessa della macchina che ospita SQL Server. Ciò significa che per eseguire il calcolo, abbiamo dovuto trasferire i dati sulla macchina client (usando quella che viene chiamata una connessione ODBC). Se la tabella SQL in questione è molto grande, il trasferimento dei dati richiederà molto tempo (la durata dipende dall'infrastruttura di rete sottostante). Inoltre, il trasferimento di dati come questo, può mettere a rischio l'integrità del *dataset*.

L'analisi in-database significa che facciamo il contrario: anziché portare i dati all'analisi, portiamo l'analisi ai dati. In altre parole, eseguiamo l'analisi non la sessione R corrente sul nostro computer client, ma avviando una nuova sessione R (remota) sulla macchina che ospita SQL Server. In questo modo, eseguiamo l'analisi in remoto (e senza dover trasferire i dati dall'host di SQL Server) e trasferiamo solo i risultati alla sessione R del client. Ciò ovviamente richiede che l'installazione R sull'host di SQL Server sia sincronizzata con l'installazione R sul computer client e che in particolare vengano installati anche tutti i pacchetti utilizzati nel codice.

Tutto quanto sopra viene generalmente gestito dall'amministratore di SQL Server, non da singoli utenti. Ma per eseguire analisi in-database, l'utente client R deve impostare il contesto di calcolo come server SQL / R remoto.