05 - Deploying e Ridimensionamento

Data Science case study for a Microsoft 213x Project NYC Taxi BigData Analysis
Lorenzo Negri, April 2018
Applications used: Microsoft R Open, Visual Studio, RevoScaleR libraries

Scoring su SQL Server

Una volta che un modello è stato testato con successo e selezionato, 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.

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.