Microsoft

# SQL Server R Services

# Contents

# Prerequisites

To complete this walkthrough on your computer, you will need an R development environment, or any other command line tool that can run R commands.

- **R Tools for Visual Studio** is a free plug-in that provides Intellisense, debugging, and support for Microsoft R Server and SQL Server R Services. To download, see R Tools for Visual Studio.

- **Microsoft R Client** is a lightweight development tool that supports development in R using the ScaleR packages. To get it, see Get Started with Microsoft R Client.

- **RStudio** is one of the more popular environments for R development. For more information, see https://www.rstudio.com/products/RStudio/.

  However, you cannot complete this tutorial using a generic installation of RStudio or other environment; you must also install the R packages and connectivity libraries for Microsoft R Open. For more information, see Set Up a Data Science Client.

- R tools (R.exe, RTerm.exe, RScripts.exe) are installed by default when you install Microsoft R Open. If you do not wish to install an IDE you can use these tools.

Alternatively, you may consider using remote cloud-based environment. All necessary components are provided as part of Data Science Virtual Machine available in Microsoft Azure Marketplace.

# Prepare the data

To prepare for this walkthrough, you will need to do the following:

1. Download the data and all R scripts used in the walkthrough. A PowerShell script is provided to simplify download from GitHub.

2. Install some additional R packages, both on the server and on your R workstation.

3. Prepare the environment, including the database and data used for modeling and scoring.

   For this, you'll use a second PowerShell script, RunSQL_R_Walkthrough.ps1. The script configures the database and uploads the data into the table you specify. It also creates some SQL functions and stored procedures that simplify data science tasks.

## Download data and scripts

All the code that you will need to complete this walkthrough has been provided in a GitHub repository. You can use a PowerShell script to make a local copy of the files.

1. Open a Windows PowerShell command prompt as administrator.

2. If you have not run PowerShell before on this instance, or you do not have permission to run scripts, you might encounter an error. If so, run this command before running the script, to temporarily allow scripts without changing system defaults.

```
Set-ExecutionPolicy Unrestricted -Scope Process -Force
```

3. Run the following command to download script files to a local directory. If you do not specify a different directory, by default the folder C:\tempR is created and all files saved there.

```
$source = 'https://raw.githubusercontent.com/Azure/Azure-MachineLearning-
DataScience/master/Misc/RSQL/Download_Scripts_R_Walkthrough.ps1'
$ps1_dest = "$pwd\Download_Scripts_R_Walkthrough.ps1"
$wc = New-Object System.Net.WebClient
$wc.DownloadFile($source, $ps1_dest)
.\Download_Scripts_R_Walkthrough.ps1 –DestDir 'C:\tempR'
```

If you want to save the files in a different directory, edit the values of the parameter *DestDir* and specify a different folder on your computer. If you type a folder name that does not exist, the PowerShell script will create the folder for you.

### Install required packages

This walkthrough requires some R libraries that are not installed by default as part of R Services (In-Database). You must install the packages both on the client where you will be developing the solution, and on the SQL Server computer where you will deploy the solution.

The R script that you downloaded includes the commands to download and install these packages.

1. In your R environment, open the script file, RSQL_R_Walkthrough.R.

2. Highlight and execute these lines.

```
# Install required R libraries for this walkthrough if they are not installed.

if (!('ggmap' %in% rownames(installed.packages()))){
  install.packages('ggmap')
}
if (!('mapproj' %in% rownames(installed.packages()))){
  install.packages('mapproj')
}
if (!('ROCR' %in% rownames(installed.packages()))){
  install.packages('ROCR')
}
if (!('RODBC' %in% rownames(installed.packages()))){
  install.packages('RODBC')
}
```

3. On the SQL Server computer, open RGui.exe as administrator. If you have installed SQL Server R Services using the defaults, RGui.exe can be found in C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\R_SERVICES\bin\x64 . Or, if you have installed another R environment on the SQL Server computer (such as RStudio) you can use the R console to run the commands.

4. At an R prompt, run the following R commands:

```
install.packages("ggmap", lib=grep("Program Files", .libPaths(), value=TRUE)[1])
install.packages("mapproj", lib=grep("Program Files", .libPaths(),
value=TRUE)[1])
install.packages("ROCR", lib=grep("Program Files", .libPaths(), value=TRUE)[1])
install.packages("RODBC", lib=grep("Program Files", .libPaths(), value=TRUE)[1])
```

Notes:

- If you think the packages are already installed, check the list of installed packages by using the R function `installed.packages()`.

- On the client, you can install to a user library if you cannot write to the main library in Program Files. However, when installing packages to the SQL Server computer, you must install them in the default library used by SQL Server R Services. Do not use a user library. For more information, see Installing New R Packages on SQL Server.

### Run Poweshell script

> ! *If you are using Data Science Virtual Machine you may safely skip this step as all necessary database components have been configured already.*

You can run this PowerShell script on the computer where you will be building the solution, for example, the computer develop and test your R code. This computer must be able to connect to the SQL Server computer using the Named Pipes protocol.

The script performs these actions:

- Checks whether the SQL Native Client and command-line utilities for SQL Server are installed. The command-line tool are needed to run the bcp Utility, which is used for fast bulk loading of data into SQL tables.
- Connects to the specified instance of SQL Server and runs some Transact-SQL scripts that configure the database and create the tables for the model and data.
- Runs a SQL script to create several stored procedures.
- Loads the data you downloaded previously into the table nyctaxi_sample.
- Rewrites the arguments in the R script file to use the database name that you specify.

1. Open a PowerShell command line as administrator.
2. Navigate to the folder where you downloaded the scripts, and type the name of the script as shown. Press ENTER.

```
.\RunSQL_R_Walkthrough.ps1
```

3. You will be prompted for each of the following parameters:

- The name of the database you want to create. For example, you might type **Tutorial** or **Taxi**
- Credentials under which to run the script. There are two options:
    - Type the name of a SQL login that has CREATE DATABASE privileges, and provide the SQL password on a successive prompt.

- Press ENTER without typing any name to use your own Windows identity, and at the secured prompt, type your Windows password. PowerShell does not support entering a different Windows user name.

  If you fail to specify a valid user, the script will default to using integrated Windows authentication.

- The full path to the csv file you want to upload to the database

The script should download the file and load the data into the database automatically, but if this fails, you can always upload the data manually.

4. Press ENTER to run the script.

# View and explore data

Data exploration is an important part of modeling data, and involves reviewing summaries of data objects to be used in the analyses, as well as data visualization. In this lesson, you'll explore the data objects and generate plots, using both Transact-SQL and R functions included in R Services (In-Database).

Then you will generate plots to visualize the data, using new functions provided by packages installed with R Services (In-Database).

## View downloaded data using SQL

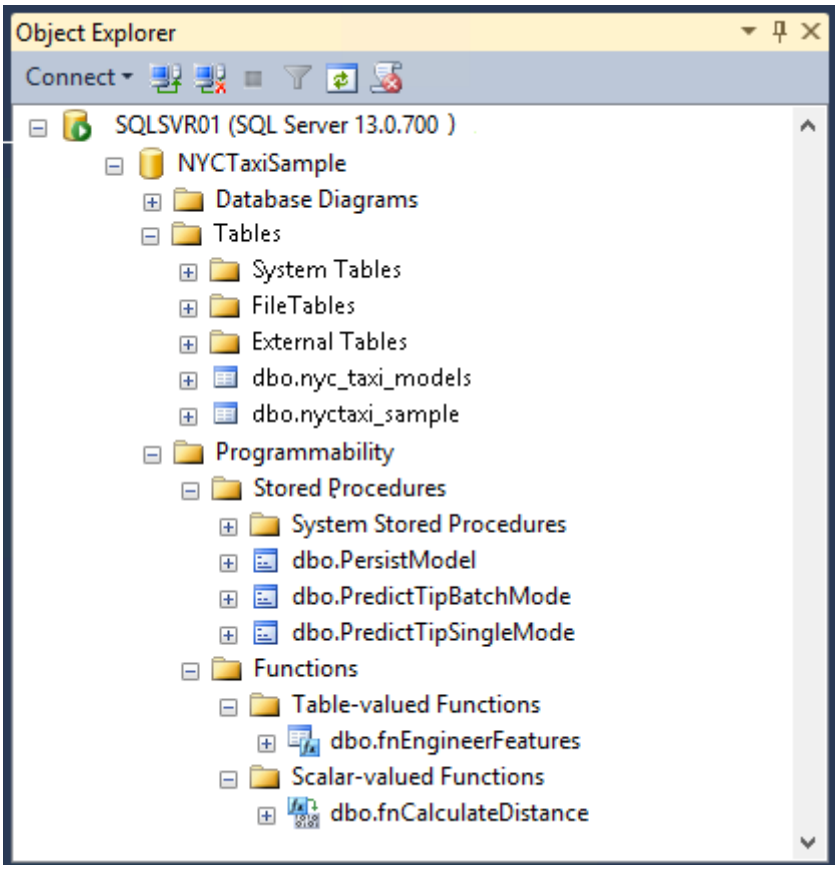
First, take a minute to ascertain that your data was loaded correctly.

1. Connect to your SQL Server instance.

   You can use a variety of tools to connect and view SQL Server databases.

   - SQL Server Management Studio
   - Server Explorer in Visual Studio

2. Expand the database you created. The following image shows the new database, tables, and functions in **Server Explorer.**




3. You can also run simple queries on the data. For example, in Management Studio, right-click the table and select **Select top 1000 rows** to generate and run this query:

```
SELECT TOP 1000 * FROM [dbo].[nyctaxi_sample]
```

4. To view the schema and data types of the data, you can use the system management views in SQL Server.

```
SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, COLUMN_DEFAULT
FROM [TaxiSample].INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = N'nyctaxi_sample';
```

### Generate Summaries using SQL

One of the strengths of SQL Server that makes it a good companion for R is the ability to perform set-based calculations very fast. The code that created the table for this walkthrough also applied a columnstore index to make calculations even faster.

```
SELECT DISTINCT [passenger_count]
    , ROUND (SUM ([fare_amount]),0) as TotalFares
    , ROUND (AVG ([fare_amount]),0) as AvgFares
FROM [dbo].[nyctaxi_sample]
GROUP BY [passenger_count]
ORDER BY  AvgFares desc
```

In the next step, you'll use R to generate some more sophisticated summaries and plots, using the data in SQL Server.

### View and Summarize Data using R

! *To make the rest of your lab run correctly please execute following SQL command in Management Studio first:*

```
CREATE LOGIN [%COMPUTERNAME%\SQLRUserGroup] FROM WINDOWS
```

*(don't forget to replace %COMPUTERNAME% with your actual VM name).*

Now you'll work with the same data using R code. You'll also learn how to use the functions in the **RevoScaleR** package included with R Services (In-Database) to generate data summaries in the context of the SQL Server server, and send the results back to your R environment.

An R script is provided with this walkthrough that includes all the code needed to create the data object, generate summaries, and build models. The R script file, **RSQL_RWalkthrough.R**, can be found in the location where you installed the script files.

- If you are experienced with R, you can run the script all at once.
- For people learning to use RevoScaleR, this tutorial goes through the script line by line
- To run individual lines from the script, you can highlight a line or lines in the file and press Ctrl + ENTER.

### Defining SQL Server Data Sources and Compute Contexts

To get data from SQL Server to use in your R code, you need to:

- Create a connection to a SQL Server instance
- Define a query that has the data you need, or specify a table or view
- Define one or more compute contexts to use when running R code
- Optionally, define functions that can be applied to the data source

### Load the RevoScaleR library

If the RevoScaleR package is not already loaded, run:

```
library("RevoScaleR")
```

If you get an error, make sure that your R development environment is using the library that includes the RevoScaleR package. Use a command like `.libPaths())` to view the current path:

If this is the first time using the **RevoScaleR** package, get the online help in the R environment by typing `help("RevoScaleR")` or `help("RxSqlServerData")`.

### Create connection strings

Define connection strings. For this walkthrough, we've provided examples of both SQL logins and Windows integrated authentication. We recommend that you use Windows authentication where possible, to avoid saving paswords in your R code.

The account that you use must have permissions to read data and to create new tables in the specified database. For information about how to add users to the SQL database and give them the correct permissions, see Post-Installation Server Configuration (SQL Server R Services).

Be sure to edit the server name, database name, user name and password as needed.

```
# SQL authentication
connStrSql <- "Driver=SQL
Server;Server=<SQL_instance_name>;Database=<database_name>;
Uid=<user_name>;Pwd=<user password>"

# Windows authentication
connStrWin <- "Driver=SQL
Server;Server=<SQL_instance_name>;Database=<database_name>;Trusted_Connection=Yes"

# Use one of the connection strings
connStr <- connStrWin
```

### Define and set a compute context

Next, you'll define a *compute context* that enable the R code to run on the SQL Server computer. Typically, when you are using R, all operations run in memory on your computer. However, by indicating that R operations should be run on the SQL Server instance, you can run some tasks in parallel, and tae advantage of server resources.

By default, the compute context is local, so you'll need to explicitly set the compute context, depending on the operation.

1. Begin by defining some variables used to create the compute context.

```
sqlShareDir <- paste("C:\\AllShare\\",Sys.getenv("USERNAME"),sep="")
sqlWait <- TRUE
sqlConsoleOutput <- FALSE
```

   - R uses a temporary directory when serializing R objects back and forth between your workstation and the SQL Server computer. You can specify the local directory that is used as *sqlShareDir*, or accept the default.
   - Use *sqlWait* to indicate whether you want R to wait for results or not. For a discussion of waiting vs. non-waiting jobs, see ScaleR Distributed Computing.
   - Use the argument *sqlConsoleOutput* to indicate that you dont want to see output from the R console.

2. Create the compute context object with the variables and connection strings already defined, and save it in the R variable *sqlcc*.

```
sqlcc <- RxInSqlServer(connectionString = connStr, shareDir = sqlShareDir,
                       wait = sqlWait, consoleOutput = sqlConsoleOutput)
```

3. Set the compute context.

```
rxSetComputeContext(sqlcc)
```

- `rxSetComputeContext` returns the previously active compute context invisibly so that you can use it
- `rxGetComputeContext` returns the active compute context

Note that setting a compute context only affects operations that use functions in the **RevoScaleR** package; the compute context does not affect the way that open source R operations are performed.

**Create an RxSqlServer data object**

Having defined the SQL Server connection to work with, you can use that data connection object as the basis to define different data sources. A *data source* specifies some set of data that you want to use for a task, such as training, exploration, scoring, or generating features.

You define sets of SQL Server data by using the *RxSqlServer* function, and passing a connection string and the definition of the data to get.

1. Save the SQL statement as a string variable. This query defines the data you'll use to train the model.

```
sampleDataQuery <- "SELECT TOP 1000 tipped, fare_amount,
       passenger_count,trip_time_in_secs,trip_distance,
       pickup_datetime, dropoff_datetime, pickup_longitude,
       pickup_latitude, dropoff_longitude,
       dropoff_latitude
       FROM nyctaxi_sample"
```

2. Pass the query definition as an argument to the SQL Server data source. The colClasses argument specifies the schema of the data to return, by mapping the SQL

```
inDataSource <- RxSqlServerData(sqlQuery = sampleDataQuery,
     connectionString = connStr,
     colClasses = c(pickup_longitude = "numeric", pickup_latitude = "numeric",
     dropoff_longitude = "numeric", dropoff_latitude = "numeric"),
     rowsPerRead=500)
```

- The argument *colClasses* specifies the column types to use when moving the data between SQL Server and R. This is important because SQL Server uses different data types than R, and more data types. For more information, see Working with R Data Types.

- The argument *rowsPerRead* is important for handling memory usage and efficient computations. Most of the enhanced analytical functions inR Services (In-Database) process data in chunks and accumulate intermediate results, returning the final computations after all of the data has been read. By adding the `rowsPerRead` parameter, you can control how many rows of data are read into each chunk for processing. If the value of this parameter is too large, data access might be slow because you don't have enough memory to efficiently process such a large chunk of data. On some systems, setting `rowsPerRead` to too small a value can also provide slower performance.

After the inDataSource object has been created, you can reuse this object as many times as you need, to get basic information about the data and the variables used, to manipulate and transform the data, or to use it for training a model. However, the inDataSource object itself doesn't contain any data from the SQL query yet. The data is not pulled into the local environment until you run a function such as rxImport or rxSummary.

### Using the SQL Server Data in R

You can now apply R functions to the data source, to explore, summarize, and chart the SQL Server data. In this section, you 'll try out several of the functions provided in R Services (In-Database) that support remote compute contexts.

- `rxGetVarInfo`: Use this function with any data frame or set of data in a remote data object (also with some lists and matrices) to get information such as the maximum and minimum values, the data type, and the number of levels in factor columns.

  Consider running this function after any kind of data input, feature transformation, or feature engineering. By doing so you can ensure that all the features you want to use in your model are of the expected data type and avoid errors.

- `rxSummary`: Use this function to get more detailed statistics about individual variables. You can also transform values, compute summaries using factor levels, and save the summaries for re-use.

**Inspect variables in the data source**

Call the function rxGetVarInfo, using the data source inDataSource as an argument, to get a list of the variables in the data source and their data types.

```
rxGetVarInfo(data = inDataSource)
```

```
Results:
Var 1: tipped, Type: integer
Var 2: fare_amount, Type: numeric
Var 3: passenger_count, Type: integer
Var 4: trip_time_in_secs, Type: numeric,
Storage: int64
Var 5: trip_distance, Type: numeric
Var 6: pickup_datetime, Type: character
Var 7: dropoff_datetime, Type: character
Var 8: pickup_longitude, Type: numeric
Var 9: pickup_latitude, Type: numeric
Var 10: dropoff_longitude, Type: numeric
```

**Create a summary using R**

Call the RevoScaleR function `rxSummary` to summarize the fare amount, based on the number of passengers.

```
start.time <- proc.time()
rxSummary(~fare_amount:F(passenger_count), data = inDataSource)
used.time <- proc.time() - start.time
print(paste("It takes CPU Time=", round(used.time[1]+used.time[2],2)," seconds,
  Elapsed Time=", round(used.time[3],2),
  " seconds to summarize the inDataSource.", sep=""))
```

- The first argument to `rxSummary` specifies the formula or term to summarize by. Here, the `F()` function is used to convert the values in passenger_count into factors before summarizing.
- If you do not specify the statistics to output, by default `rxSummary` outputs Mean, StDev, Min, Max, and the number of valid and missing observations.
- This example also includes some code to track the time the function starts and completes, so that you can compare performance.

## Create Graphs and plots using R

In this lesson, you'll learn techniques for generating plots and maps using R with SQL Server data. You'll see how easy it is to view plots that are created on the server, and how you can pass graphics objects to the server.

**Creating graphics**

In R Services (In-Database), graphics objects as well as models and results are serialized between your computer and the SQL Server compute context.

When using a SQL Server compute context, you might not be able to download the map representation, because most

production database servers completely block Internet access. Therefore, to create the second set of plots, you will generate the map representation in the client and then overlay on the map the points that are stored as attributes in the *nyctaxi_sample* table.

To do this, you first create the map representation by calling into Google maps, and then pass the map representation to the SQL context.

This is a pattern that you might find useful when developing your own applications.

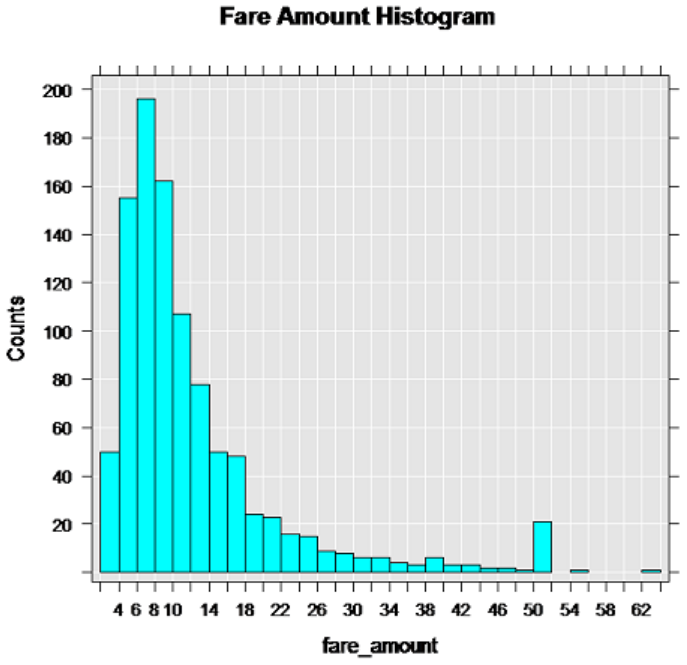
**Creating histogram**

To create a histogram, you'll use the SQL Server data source you created earlier, together with the `rxHistogram` function provided in R Services.

1. Generate the first plot, using the *rxHistogram* function. The *rxHistogram* function provides functionality similar to that in open source R packages, but can run in a remote execution context.

```
#Plot fare amount on SQL Server and return the plot
start.time <- proc.time()
rxHistogram(~fare_amount, data = inDataSource, title = "Fare Amount Histogram")
used.time <- proc.time() - start.time
print(paste("It takes CPU Time=", round(used.time[1]+used.time[2],2), " seconds,
Elapsed Time=", round(used.time[3],2), " seconds to generate features.", sep=""))
```

2. The image is returned in the R graphics device for your development environment. For example, in RStudio, click the **Plot** window. In R Tools for Visual Studio, a separate graphics window is opened.



Because the ordering of rows using TOP is non-deterministic without an ORDER BY clause, you might see very different results. We recommend that you experiment with different numbers of rows to get different graphs, and note how long it takes to return the results in your environment. This particular image was generated using ~10,000 rows of data.

**Create map plot**

In this example, you'll generate a plot object using the SQL Server instance as the compute context, and then return the plot object to the local compute context for rendering.

1. First, define the function that creates the plot object.

```
rxSetComputeContext(RxLocalSeq())
mapPlot <- function(inDataSource, googMap){
    library(ggmap)
    library(mapproj)
    ds <- rxImport(inDataSource)
    p <- ggmap(googMap)+
    geom_point(aes(x = pickup_longitude, y =pickup_latitude ), data=ds, alpha
=.5,
color="darkred", size = 1.5)
    return(list(myplot=p))
}
```

- The custom R function *mapPlot* creates a scatter plot that uses the taxi pickup locations to plot the number of rides that started from each location. It uses the **ggplot2** and **ggmap** packages, which should already be installed and loaded.

- The *mapPlot* function takes two arguments: an existing data object, which you defined earlier using *RxSqlServerData*, and the map representation passed from the client.

- Note the use of the *ds* variable to load data from the previously created data source, *inDataSource*. Whenever you use open source R functions, data must be loaded into data frames in memory. You can do this by using the *rxImport* function in the **RevoScaleR** package. However, this function runs in memory in the SQL Server context defined earlier. That is, the function is not using the memory of your local workstation.

2. Next, load the libraries required for creating the maps in your local R environment.

```
library(ggmap)
library(mapproj)
gc <- geocode("Manhattan", source = "google")
googMap <- get_googlemap(center = as.numeric(gc), zoom = 12, maptype = 'roadmap',
color = 'color';
```

- This code is run on the R client. Note the repeated call to the libraries **ggmap** and **mapproj**. The reason is that the previous function definition ran in the server context and the libraries were never loaded locally; now you are bringing the plotting operation back to your workstation.

- The *gc* variable stores a set of coordinates for Times Square, NY.

- The line beginning with *googmap* generates a map with the specified coordinates at the center.

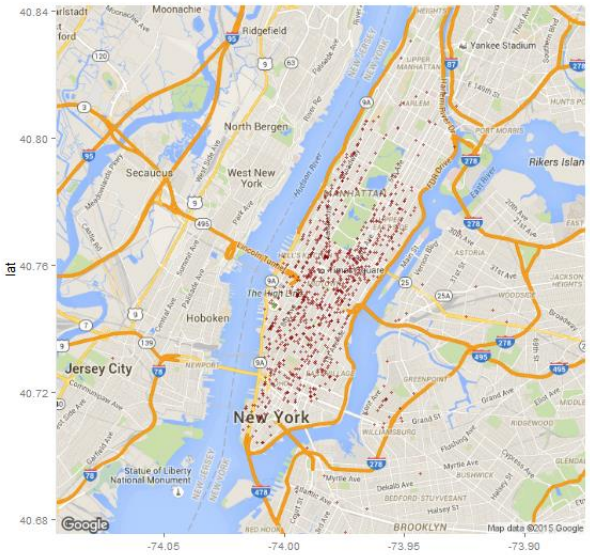
3. Execute the plotting function and render the results in your local R environment. To do this, wrap the plot function in *rxExec* as shown here. The *rxExec* function is included in the **RevoScaleR** package, and supports execution of arbitrary R functions in a remote compute context.

```
myplots <- rxExec(mapPlot, inDataSource, googMap, timesToRun = 1)
plot(myplots[[1]][["myplot"]]);
rxSetComputeContext(cc)
```

- In the first line, you can see that the map data is passed as an argument (*googMap*) to the remotely executed function *mapPlot*. That is because the maps were generated in your local environment, and must be passed to the function in order to create the plot in the context of SQL Server.

  The rendered data is then serialized back to the local R environment so that you can view it, using the **Plot** window in RStudio or other R graphics device.

4. Here is the output plot, showing the pickup locations on the map as red dots.



# Create data features

Data engineering is another important part of machine learning. Often data needs to be transformed before you can use it for

need, you can engineer them from existing values.

For this modeling task, rather than using the raw latitude and longitude values of the pickup and drop-off location, you'd like to have the distance in miles between the two locations. To create this feature, you'll compute the direct linear distance between two points, by using the haversine formula.
You'll compare two different methods for creating a feature from data:

- Using R and the `rxDataStep` function
- Using a custom function in Transact-SQL

For both methods, the result of the code is a SQL Server data source object, `featureDataSource`, that includes the new numeric feature, `direct_distance`.

### Creating Features Using R

The R language is well-known for its rich and varied statistical libraries, but you still might need to create custom data transformations.

- You'll create a new R function, `ComputeDist`, to calculate the linear distance between two points specified by latitude and longitude values.
- You'll call the function to transform the data in the SQL Server data object you created earlier, an save it in a new data source, `featureDataSource`.

**Create transformation function**

Create a custom R function, *ComputeDist*. It takes in two pairs of latitude and longitude values, and calculates the linear distance between them. The function returns a distance in miles.

```
env <- new.env()
env$ComputeDist <- function(pickup_long, pickup_lat, dropoff_long, dropoff_lat){
  R <- 6371/1.609344 #radius in mile
  delta_lat <- dropoff_lat - pickup_lat
  delta_long <- dropoff_long - pickup_long
  degrees_to_radians = pi/180.0
  a1 <- sin(delta_lat/2*degrees_to_radians)
  a2 <- as.numeric(a1)^2
  a3 <- cos(pickup_lat*degrees_to_radians)
  a4 <- cos(dropoff_lat*degrees_to_radians)
  a5 <- sin(delta_long/2*degrees_to_radians)
  a6 <- as.numeric(a5)^2
  a <- a2+a3*a4*a6
  c <- 2*atan2(sqrt(a),sqrt(1-a))
  d <- R*c
  return (d)
}
```

- The first line defines a new environment. In R, an environment can be used to encapsulate name spaces in packages and such.

- You can use the `search()` function to view the environments in your workspace. To view the objects in a specific environment, type `ls(<envname>)`.

- The lines beginning with `$env.ComputeDistance` contain the code that defines the haversine formula, which calculates the *great-circle distance* between two points on a sphere.

**Apply the transformation function to data**

Having defined the function, you will apply it to the data to create a new feature column, *direct_distance*.

1. Create a data source to work with by using the `RxSqlServerData` constructor.

```
featureDataSource = RxSqlServerData(table = "features",
   colClasses = c(pickup_longitude = "numeric",
   pickup_latitude = "numeric",
   dropoff_longitude = "numeric",
   dropoff_latitude = "numeric",
   passenger_count  = "numeric",
   trip_distance  = "numeric",
   trip_time_in_secs  = "numeric",
   direct_distance  = "numeric"),
  connectionString = connStr)
```

2. Call the `rxDataStep` function to apply the `env$ComputeDist` function to the specified data.

```
start.time <- proc.time()

rxDataStep(inData = inDataSource, outFile = featureDataSource,
     overwrite = TRUE,
     varsToKeep=c("tipped", "fare_amount", passenger_count", "trip_time_in_secs",
        "trip_distance", "pickup_datetime", "dropoff_datetime",
"pickup_longitude",
        "pickup_latitude", "dropoff_longitude", "dropoff_latitude")
     , transforms = list(direct_distance=ComputeDist(pickup_longitude,
        pickup_latitude, dropoff_longitude, dropoff_latitude)),
        transformEnvir = env, rowsPerRead=500, reportProgress = 3)

used.time <- proc.time() - start.time
print(paste("It takes CPU Time=", round(used.time[1]+used.time[2],2)," seconds,
Elapsed Time=", round(used.time[3],2), " seconds to generate features.", sep=""))
```

- The `rxDataStep` function can modify data in place. The arguments include a character vector of columns to pass through (*varsToKeep*), and a list that defines transformations.

- Any columns that are transformed are automatically output and therefore do not need to be included in the *varsToKeep* argument.

- Alternatively, you can specify that all columns in the source be included except for the specified variables, by using the *varsToDrop* argument.

3. Finally, call rxGetVarInfo to inspect the schema of the new data source:

```
rxGetVarInfo(data = featureDataSource)
```

```
Results
"It takes CPU Time=0.74 seconds, Elapsed
Time=35.75 seconds to generate features."
Var 1: tipped, Type: integer
Var 2: fare_amount, Type: numeric
Var 3: passenger_count, Type: numeric
Var 4: trip_time_in_secs, Type: numeric
Var 5: trip_distance, Type: numeric
Var 6: pickup_datetime, Type: character
Var 7: dropoff_datetime, Type: character
Var 8: pickup_longitude, Type: numeric
Var 9: pickup_latitude, Type: numeric
Var 10: dropoff_longitude, Type: numeric
Var 11: dropoff_latitude, Type: numeric
Var 12: direct_distance, Type: numeric
```

### Creating features using Transact-SQL

Now that you've seen how to create a feature using an R function, you'll create a custom SQL function, `ComputeDist`, to do the same thing. The SQL function `ComputeDist` operates on an existing `RxSqlServerData` data object to create the new distance features from the existing latitude and longitude values.

You'll save the results of the transformation to a SQL Server data object, `featureDataSource`, just as you did using R.

Very often, feature engineering using Transact-SQL will be faster than R. Choose the most efficient method based on your data and task.

**Define the T-SQL custom function**

1. Define a new SQL function, fnCalculateDistance

```
CREATE FUNCTION [dbo].[fnCalculateDistance] (@Lat1 float, @Long1 float, @Lat2
float, @Long2 float)
-- User-defined function calculates the direct distance between two geographical
coordinates.
RETURNS float
AS
BEGIN
  DECLARE @distance decimal(28, 10)
  -- Convert to radians
  SET @Lat1 = @Lat1 / 57.2958
  SET @Long1 = @Long1 / 57.2958
  SET @Lat2 = @Lat2 / 57.2958
  SET @Long2 = @Long2 / 57.2958
  -- Calculate distance
  SET @distance = (SIN(@Lat1) * SIN(@Lat2)) + (COS(@Lat1) * COS(@Lat2) *
COS(@Long2 - @Long1))
  --Convert to miles
  IF @distance <> 0
  BEGIN
    SET @distance = 3958.75 * ATAN(SQRT(1 - POWER(@distance, 2)) / @distance);
  END
  RETURN @distance
END
```

- The code for this SQL *user-defined function* was provided as part of the PowerShell script you ran to create and configure the database. The function should already exist in your database. If it doesn't exist, use SQL Server Management Studio to generate the function in the same database where the taxi data is stored.

2. To see how the function works, the following Transact-SQL statement from any application that supports Transact-SQL.

```
SELECT tipped, fare_amount, passenger_count,trip_time_in_secs,trip_distance,
pickup_datetime, dropoff_datetime,
dbo.fnCalculateDistance(pickup_latitude, pickup_longitude,  dropoff_latitude,
dropoff_longitude) as direct_distance,
pickup_latitude, pickup_longitude,  dropoff_latitude, dropoff_longitude
FROM nyctaxi_sample
```

### Call the function from R

1. Save the SQL statement that calls the custom function in an R variable.

```
featureEngineeringQuery = "SELECT tipped, fare_amount, passenger_count,
    trip_time_in_secs,trip_distance, pickup_datetime, dropoff_datetime,
    dbo.fnCalculateDistance(pickup_latitude, pickup_longitude,  dropoff_latitude,
dropoff_longitude) as direct_distance,
    pickup_latitude, pickup_longitude,  dropoff_latitude, dropoff_longitude
    FROM nyctaxi_joined_1_percent
    tablesample (1 percent) repeatable (98052)"
```

This query is slightly different from the Transact-SQL query used earlier. It has been modified to get a smaller sample of data, to make this walkthrough faster.

2. Now, use the following lines of code to call the Transact-SQL function from your R environment and apply it to the data defined in `featureEngineeringQuery`.

```
featureDataSource = RxSqlServerData(sqlQuery = featureEngineeringQuery,
  colClasses = c(pickup_longitude = "numeric", pickup_latitude = "numeric",
         dropoff_longitude = "numeric", dropoff_latitude = "numeric",
         passenger_count  = "numeric", trip_distance  = "numeric",
          trip_time_in_secs  = "numeric", direct_distance  = "numeric"),
  connectionString = connStr)
```

3. Now that the new feature is created, call `rxGetVarsInfo` to create a summary of the feature table.

```
rxGetVarInfo(data = featureDataSource)
```

### Comparing R Functions and SQL Functions

As it turns out, for this particular task, the Transact-SQL function approach is faster than the custom R function. Therefore, you'll use the Transact-SQL function for these calculations in subsequent steps.

Proceed to the next lesson to learn how to build a predictive model using this data and save the model to a SQL Server table.

# Build and save model

## Creating a Classification model

The model you'll build is a binary classifier that predicts whether the taxi driver is likely to get a tip on a particular ride or not. You'll use the data source you created in the previous lesson, `featureDataSource`, to train the tip classifier, using logistic regression.

Here are the features you'll use in the model:

- passenger_count
- trip_distance
- trip_time_in_secs
- direct_distance

**Create the model using rxLogit**

1. Call the *rxLogit* function, included in the **RevoScaleR** package, to create a logistic regression model.

```
system.time(logitObj <- rxLogit(tipped ~ passenger_count + trip_distance +
trip_time_in_secs + direct_distance, data = featureDataSource))
```

2. After you build the model, you'll want to inspect it using the summary function, and view the coefficients.

```
summary(logitObj)
```

```
Results
Logistic Regression Results for: tipped ~
passenger_count + trip_distance + trip_time_in_secs +
direct_distance
Data: featureDataSource (RxSqlServerData Data Source)
Dependent variable(s): tipped
Total independent variables: 5
Number of valid observations: 17068
Number of missing observations: 0
-2*LogLikelihood: 23540.0602 (Residual deviance on
17063 degrees of freedom)
Coefficients:
Estimate Std. Error z value Pr(>|z|)
(Intercept) -2.509e-03 3.223e-02 -0.078 0.93793
passenger_count -5.753e-02 1.088e-02 -5.289 1.23e-07
***
trip_distance -3.896e-02 1.466e-02 -2.658 0.00786 **
trip_time_in_secs 2.115e-04 4.336e-05 4.878 1.07e-06
***
direct_distance 6.156e-02 2.076e-02 2.966 0.00302 **
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.'
0.1 ' ' 1
Condition number of final variance-covariance matrix:
48.3933
Number of iterations: 4
```

### Use the logistic regression model for scoring

Now that the model is built, you can use to predict whether the driver is likely to get a tip on a particular drive or not.

1. First, define the data object to use for storing the scoring results

```
scoredOutput <- RxSqlServerData(
  connectionString = connStr,
  table = "taxiScoreOutput"  )
```

- To make this example simpler, the input to the logistic regression model is the same `featureDataSource` that you used to train the model. More typically, you might have some new data to score with, or you might have set aside some data for testing vs. training.

- The prediction results are saved in the table, *taxiscoreOutput*. Notice that the schema for this table is not defined when you create it using `rxSqlServerData`, but is obtained from the *scoredOutput* object output from `rxPredict`.

- To create the table that stores the predicted values, the SQL login running the `rxSqlServer` data function must have DDL privileges in the database. If the login cannot create tables, the statement will fail.

2. Call the rxPredict function to generate results.

```
rxPredict(modelObject = logitObj, data = featureDataSource, outData =
scoredOutput, predVarNames = "Score", type = "response", writeModelVars = TRUE,
overwrite = TRUE)
```

### Plotting model accuracy

To get an idea of the accuracy of the model, you can use the *rxRocCurve* function to plot the Receiver Operating Curve. Because *rxRocCurve* is one of the new functions provided by the RevoScaleR package that supports remote compute contexts, you have two options:

- You can use the `rxRocCurve` function to execute the plot in the remote computer context and then return the plot to your local client.

- You can also import the data to your R client computer, and use other R plotting functions to create the performance graph.

In this section, you'll use both techniques.

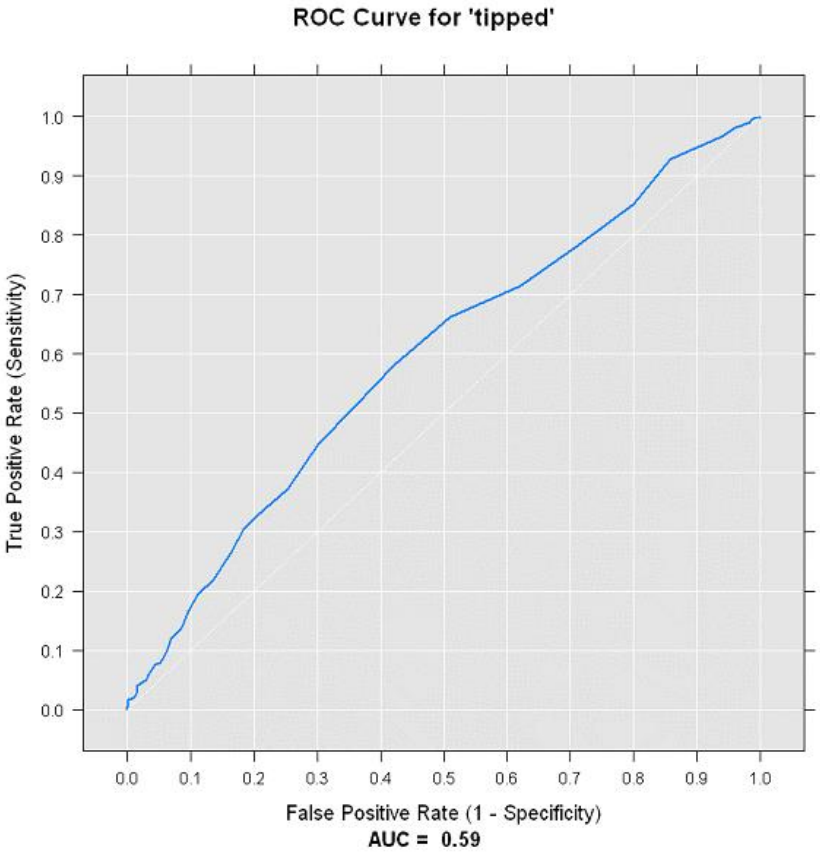
**Execute a plot in the remote (SQL Server) compute context**

Call the function rxRocCurve and provide the data defined earlier as input.

```
rxRocCurve( "tipped", "Score", scoredOutput)
```

Note that you must also specify the label column tipped (the variable you are trying to predict) and the name of the column that stores the prediction (*Score*).

View the graph that is generated by opening the R graphics device, or by clicking the **Plot** window in RStudio.




The graph is created on the remote compute context, and then returned to your R environment.

**Create plots in the local compute context using data from SQL Server**

1. Use the rxImport function to bring the specified data to your local R environment.

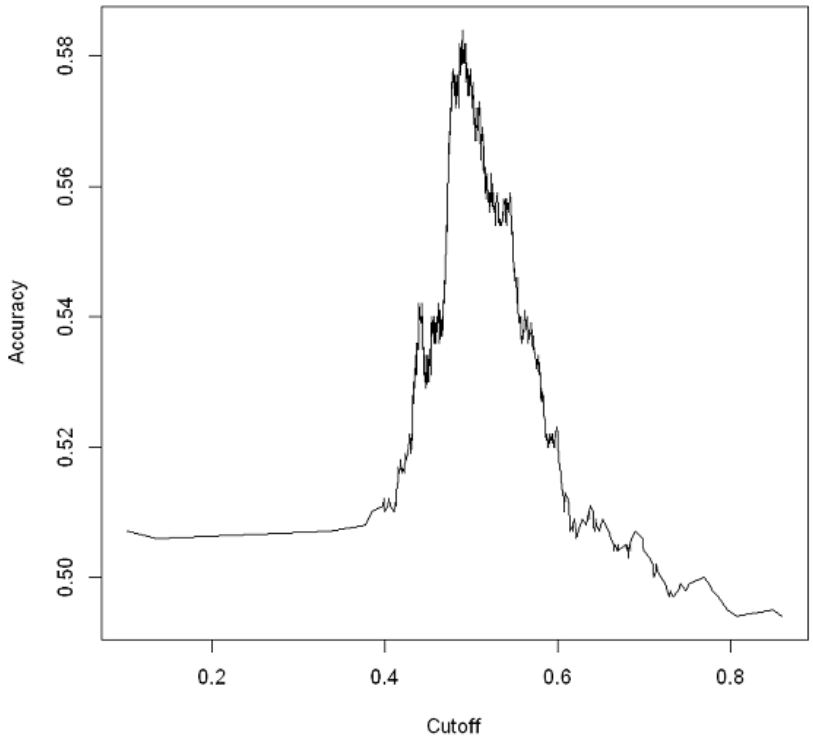
```
scoredOutput = rxImport(scoredOutput)
```

2. Having loaded the data into local memory, you can then call the ROCR library to create some predictions, and generate the plot.

```
library('ROCR')
pred <- prediction(scoredOutput$Score, scoredOutput$tipped)

acc.perf = performance(pred, measure = 'acc')
plot(acc.perf)
ind = which.max( slot(acc.perf, 'y.values')[[1]] )
acc = slot(acc.perf, 'y.values')[[1]][ind]
cutoff = slot(acc.perf, 'x.values')[[1]][ind]
```

3. The following plot is generated in both cases.



## Deploying a model

After you have built a model and ascertained that it is performing well, you might want to *operationalize* the model. Because R Services (In-Database) lets you invoke an R model using a Transact-SQL stored procedure, it is extremely easy to use R in a client application.

However, before you can call the model from an external application, you must save the model to the database used for production. In R Services (In-Database), trained models are stored in binary form, in a single column of type **varbinary(max)**.

Therefore, moving a trained model from R to SQL Server includes these steps:

- Serializing the model into a hexadecimal string
- Transmitting the serialized object to the database
- Saving the model in a varbinary(max) column

In this section, you will learn how to persist the model, and how to call it to make predictions.

### Serialize the model

In your local R environment, serialize the model and save it in a variable.

```
modelbin <- serialize(logitObj, NULL)
modelbinstr=paste(modelbin, collapse="")
```

The serialize function is included in the R base package, and provides a simple low-level interface for serializing to connections. For more information, see http://www.inside-r.org/r-doc/base/serialize.

### Move the model to SQL Server

Open an ODBC connection, and call a stored procedure to store the binary representation of the model in a column in the database.

```
library(RODBC)
conn <- odbcDriverConnect(connStr )

# persist model by calling a stored procedure from SQL
q<-paste("EXEC PersistModel @m='", modelbinstr,"'", sep="")
sqlQuery (conn, q)
```

Saving a model to a table requires only an INSERT statement. However, to make it easier, here we have used the *PersistModel* stored procedure is used.

For reference, here is the complete code of the stored procedure:

```
CREATE PROCEDURE [dbo].[PersistModel]  @m nvarchar(max)
AS
BEGIN
        -- SET NOCOUNT ON added to prevent extra result sets from
        -- interfering with SELECT statements.
        SET NOCOUNT ON;
        insert into nyc_taxi_models (model) values (convert(varbinary(max),@m,2))
END
```

We recommend creating helper functions such as this stored procedure to make it easier to manage and update your R models in SQL Server.

**Invoke the saved model**

After you have saved the model to the database, you can call it directly from Transact-SQL code, using the system stored procedure, sp_execute_external_script (Transact-SQL).

For example, to generate predictions, you simply connect to the database and run a stored procedure that uses the saved model as an input, together with some input data.

However, if you have a model you use often, it's easier to wrap the input query and the call to the model, as well as any other parameters, in a custom stored procedure.

In the next lesson, you'll learn how to perform scoring against the saved model using Transact-SQL.

# Deploy and use model

In this lesson, you will use your R models in a production environment, by wrapping the persisted model in a stored procedure. You can then invoke the stored procedure from R or any application programming language that supports Transact-SQL (such as C#, Java, Python, etc), to use the model to make predictions on new observations.

There are two different ways that you can call a model for scoring:

- **Batch scoring mode** lets you create multiple predictions based on input from a SELECT query.
- **Individual scoring mode** lets you create predictions one at a time, by passing a set of feature values for an individual case to the stored procedure, which returns a single prediction or other value as the result.

You'll learn how to create predictions using both the individual scoring and batch scoring methods.

## Batch scoring

For convenience, you can use a stored procedure that was created when you initially ran the PowerShell script in Lesson 1. This stored procedure does the following:

- Gets a set of input data as a SQL query
- Calls the trained logistic regression model that you saved in the previous lesson
- Predicts the probability that the driver will get a tip

**Use the stored procedure PredictTipBatchMode**

1. Take a minute to look over the script that defines the stored procedure, *PredictTipBatchMode*. It illustrates several aspects of how a model can be operationalized using R Services (In-Database).

```
CREATE PROCEDURE [dbo].[PredictTipBatchMode]
@input nvarchar(max)
AS
BEGIN

  DECLARE @lmodel2 varbinary(max) = (SELECT TOP 1 model  FROM nyc_taxi_models);
  EXEC sp_execute_external_script @language = N'R',
     @script = N'
       mod <- unserialize(as.raw(model));
       print(summary(mod))
       OutputDataSet<-rxPredict(modelObject = mod,
           data = InputDataSet,
           outData = NULL,
           predVarNames = "Score", type = "response",
           writeModelVars = FALSE, overwrite = TRUE);
       str(OutputDataSet)
       print(OutputDataSet)',
  @input_data_1 = @input,
  @params = N'@model varbinary(max)',
  @model = @lmodel2
  WITH RESULT SETS ((Score float));
END
```

- Note the SELECT statement that calls the stored model. You can store any trained model in a SQL table, by using a column of type **varbinary(max)**. In this code, the model is retrieved from the table, stored in the SQL variable *@lmodel2*, and passed as the parameter *mod* to the system stored procedure sp_execute_external_script (Transact-SQL).

- The input data used for scoring is passed as a string to the stored procedure.

To define input data for this particular model, create a query that returns valid data. As data is retrieved from the database, it is stored in a data frame called *InputDataSet*. All the rows in this data frame are used for batch scoring.

  - *InputDataSet* is the default name for input data to the sp_execute_external_script (Transact-SQL) procedure; you can define another variable name if needed.

  - To generate the scores, the stored procedure calls the *rxPredict* function from the **RevoScaleR** library.

  - The return value for the stored procedure, *Score*, is a predicted probability that the driver will be given a tip.

2. Optionally, you could easily apply some kind of filter to the returned values to categorize the return values into "yes - tip " or "no tip" groups. For example, a probability of less than 0.5 would mean no tip is likely.

3. Call the stored procedure in batch mode:

```
input = "N'
  SELECT TOP 10
      a.passenger_count AS passenger_count,
      a.trip_time_in_secs AS trip_time_in_secs,
      a.trip_distance AS trip_distance,
      a.dropoff_datetime AS dropoff_datetime,
      dbo.fnCalculateDistance(
        pickup_latitude,
        pickup_longitude,
        dropoff_latitude,
        dropoff_longitude) AS direct_distance
  FROM

  (SELECT medallion, hack_license, pickup_datetime,
  passenger_count,trip_time_in_secs,trip_distance,
  dropoff_datetime, pickup_latitude, pickup_longitude,
  dropoff_latitude, dropoff_longitude from nyctaxi_sample)a

  LEFT OUTER JOIN

  ( SELECT medallion, hack_license, pickup_datetime
    FROM nyctaxi_sample  tablesample (1 percent) repeatable (98052)  )b

  ON a.medallion=b.medallion
    AND a.hack_license=b.hack_license
    AND a.pickup_datetime=b.pickup_datetime

  WHERE b.medallion is null
'"
q<-paste("EXEC PredictTipBatchMode @inquery = ", input, sep="")
sqlQuery (conn, q)
```

## Single Row Scoring

Instead of using a query to pass the input values to the saved R model, you might want to provide the features as arguments to the stored procedure.

**Use the stored procedure PredictTipSingleMode**

1. Take a minute to review the following code is for the stored procedure, PredictTipSingleMode, which should already be created in your database.

```
CREATE PROCEDURE [dbo].[PredictTipSingleMode] @passenger_count int = 0,
@trip_distance float = 0,
@trip_time_in_secs int = 0,
@pickup_latitude float = 0,
@pickup_longitude float = 0,
@dropoff_latitude float = 0,
@dropoff_longitude float = 0
AS
BEGIN
  DECLARE @inquery nvarchar(max) = N'
    SELECT * FROM [dbo].[fnEngineerFeatures](
        @passenger_count,
        @trip_distance,
        @trip_time_in_secs,
        @pickup_latitude,
        @pickup_longitude,
        @dropoff_latitude,
        @dropoff_longitude)
    '
  DECLARE @lmodel2 varbinary(max) = (SELECT TOP 1 model FROM nyc_taxi_models);

  EXEC sp_execute_external_script @language = N'R',  @script = N'
        mod <- unserialize(as.raw(model));
        print(summary(mod))
        OutputDataSet<-rxPredict(
                  modelObject = mod,
                  data = InputDataSet,
                  outData = NULL,
                  predVarNames = "Score",
                  type = "response",
                  writeModelVars = FALSE,
                  overwrite = TRUE);
        str(OutputDataSet)
        print(OutputDataSet)
        ',
  @input_data_1 = @inquery,
  @params = N'@model varbinary(max),
            @passenger_count int,
            @trip_distance float,
            @trip_time_in_secs int ,
            @pickup_latitude float ,
            @pickup_longitude float ,
            @dropoff_latitude float ,
            @dropoff_longitude float',
            @model = @lmodel2,
            @passenger_count =@passenger_count ,
            @trip_distance=@trip_distance,
            @trip_time_in_secs=@trip_time_in_secs,
            @pickup_latitude=@pickup_latitude,
            @pickup_longitude=@pickup_longitude,
            @dropoff_latitude=@dropoff_latitude,
            @dropoff_longitude=@dropoff_longitude
  WITH RESULT SETS ((Score float));
END
```

This stored procedure takes feature values as input, such as passenger count and trip distance, scores these features using the stored R model, and outputs a score.

### Call the stored procedure and pass parameters

1. In SQL Server Management Studio, you can use the Transact-SQL EXEC to call the stored procedure, and pass it the required inputs.

```
EXEC [dbo].[PredictTipSingleMode] 1, 2.5, 631, 40.763958,-73.973373, 40.782139,-
73.977303
```

The values passed in here are, respectively, for the variables passenger_count, trip_distance, trip_time_in_secs, pickup_latitude, pickup_longitude, dropoff_latitude, and dropoff_longitude.

2. To run this same call from R code, you simply define an R variable that contains the entire stored procedure call.

```
q = "EXEC PredictTipSingleMode 1, 2.5, 631, 40.763958,-73.973373, 40.782139,-
73.977303 "
```

### Generate scores

Call the sqlQuery function of the RODBC package, and pass the connection string and the string variable containing the the stored procedure call.

```
# predict with stored procedure in single mode
sqlQuery (conn, q)
```

For more information about RODBC, see http://www.inside-r.org/packages/cran/RODBC/docs/sqlQuery.

# Conclusion and next steps

Now that you have learned how to work with SQL Server data and persist trained R models to SQL Server, it should be relatively easy for you to create some additional models based on this data set. For example, you might try creating models like these:

- A regression model that predicts the tip amount
- A multiclass classification model that predicts whether the tip will be big, medium, or small.

# Terms of Use


By using this Hands-on Lab, you agree to the following terms:

The technology/functionality described in this Hands-on Lab is provided by Microsoft Corporation in a "sandbox" testing environment for purposes of obtaining your feedback and to provide you with a learning experience. You may only use the Hands-on Lab to evaluate such technology features and functionality and provide feedback to Microsoft. You may not use it for any other purpose. You may not modify copy, distribute, transmit, display, perform, reproduce, publish, license, create derivative works from, transfer, or sell this Hands-on Lab or any portion thereof.

COPYING OR REPRODUCTION OF THE HANDS-ON LAB (OR ANY PORTION OF IT) TO ANY OTHER SERVER OR LOCATION FOR FURTHER REPRODUCTION OR REDISTRIBUTION IS EXPRESSLY PROHIBITED.

THIS HANDS-ON LAB PROVIDES CERTAIN SOFTWARE TECHNOLOGY/PRODUCT FEATURES AND FUNCTIONALITY, INCLUDING POTENTIAL NEW FEATURES AND CONCEPTS, IN A SIMULATED ENVIRONMENT WITHOUT COMPLEX SET-UP OR INSTALLATION FOR THE PURPOSE DESCRIBED ABOVE. THE TECHNOLOGY/CONCEPTS REPRESENTED IN THIS HANDS-ON LAB MAY NOT REPRESENT FULL FEATURE FUNCTIONALITY AND MAY NOT WORK THE WAY A FINAL VERSION MAY WORK. WE ALSO MAY NOT RELEASE A FINAL VERSION OF SUCH FEATURES OR CONCEPTS. YOUR EXPERIENCE WITH USING SUCH FEATURES AND FUNCITONALITY IN A PHYSICAL ENVIRONMENT MAY ALSO BE DIFFERENT.

**FEEDBACK**. If you give feedback about the technology features, functionality and/or concepts described in this Hands-on Lab to Microsoft, you give to Microsoft, without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft software or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its software or documentation to third parties because we include your feedback in them. These rights survive this agreement.

MICROSOFT CORPORATION HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THE HANDS-ON LAB, INCLUDING ALL WARRANTIES AND CONDITIONS OF MERCHANTABILITY, WHETHER EXPRESS, IMPLIED OR STATUTORY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. MICROSOFT DOES NOT MAKE ANY ASSURANCES OR REPRESENTATIONS WITH REGARD TO THE ACCURACY OF THE RESULTS, OUTPUT THAT DERIVES FROM USE OF THE VIRTUAL LAB, OR SUITABILITY OF THE INFORMATION CONTAINED IN THE VIRTUAL LAB FOR ANY PURPOSE.

DISCLAIMER

This lab contains only a portion of the features and enhancements in Microsoft Azure. Some of the features might change in future releases of the product.