



Lab 01

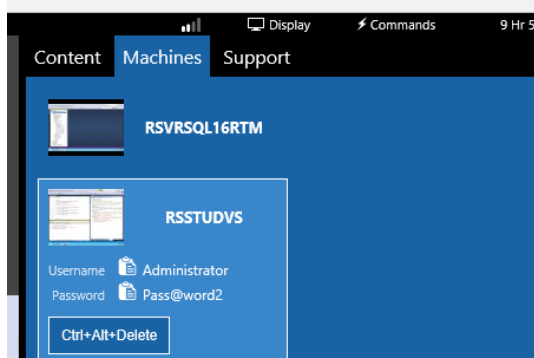
Exploring SQL Server 2016 R Services and Microsoft R Client with R Tools for Visual Studio

1 GETTING STARTED

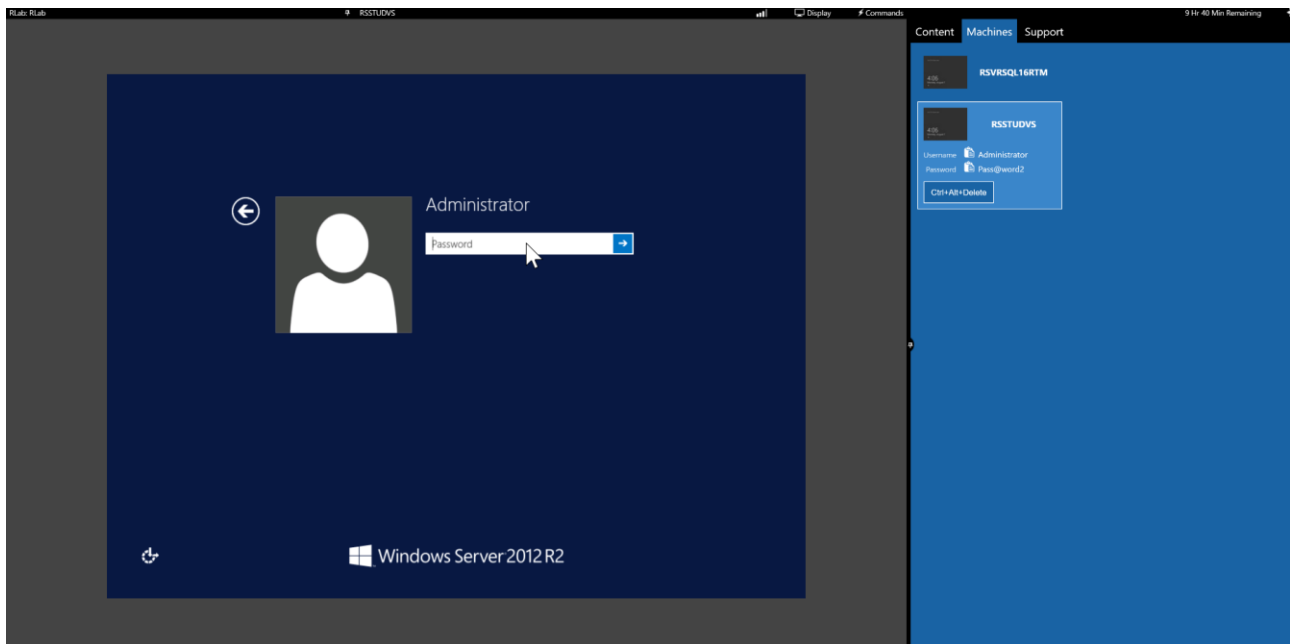
This lab includes two VMs, the Student VM and the Server VM, with all the necessary components and R code to optimize your learning experience. In this exercise you will login to both the Student and Server VM.

If you are undertaking this lab outside of the pre-provisioned virtual labs environment then you may need to install Visual Studio and the R Tools for Visual Studio.

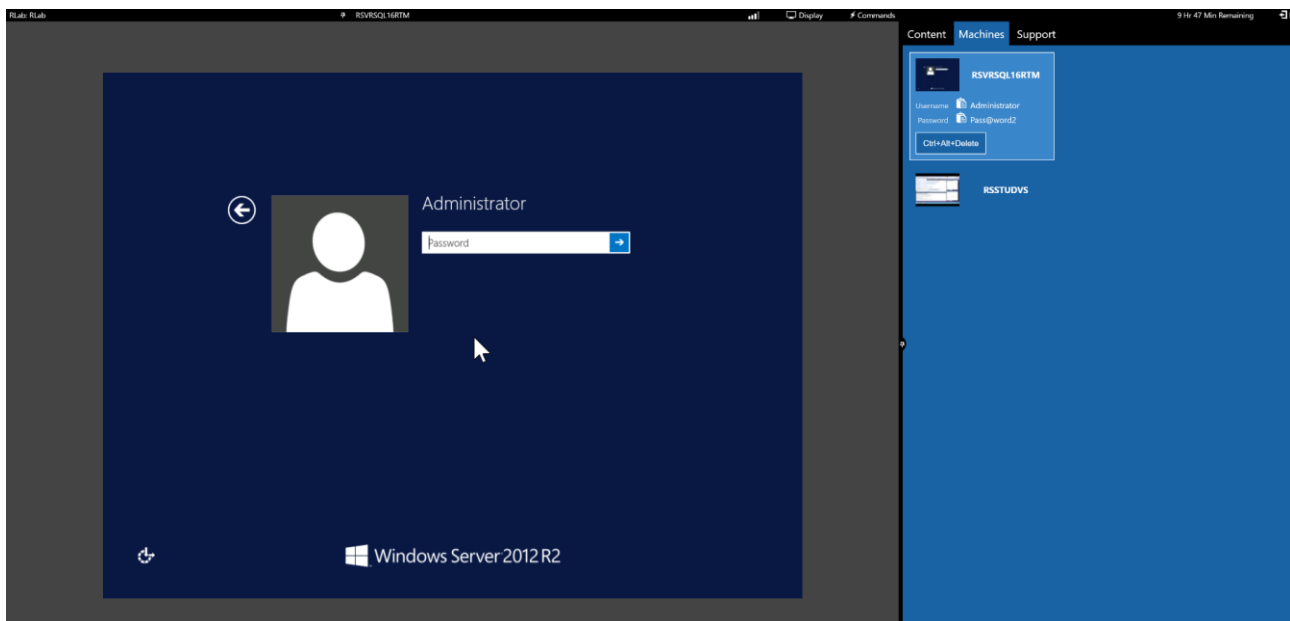
1. From within the virtual labs environment, **open** the "Machines" Tab



2. **Select** the RSSTUDVS "Student VM"
3. **Click** on the "Ctrl+Alt+Delete" button
4. **Type** the "Pass@word2" password



5. **Open** Visual Studio by clicking the taskbar icon
6. **Open** "myRProject.sln" from `c:\Lab_Content\myRProject`
7. **Launch** the Server VM in a similar way. Repeating steps 1 to 4 above



The Server VM contains:

- SQL Server – with R Services
- SQL Server Management Studio
- Example SQL scripts (all tables and stored procedures have been previously created for you)
- Taxi data (both in a csv file and pre-loaded in the database to maximize your learning experience)

The Student VM contains:

- Microsoft R Client
 - Microsoft R
 - Microsoft R ScaleR Library
 - Visual Studio
 - R Tools for Visual Studio
- R scripts – which you will use in the hands-on exercise for various common R services tasks, such as, logistic regression, RTVS, how to load data directly into the table.

2 LAB OVERVIEW

The estimated time to complete all 5 lessons of this lab is 3 hours.

The R language is widely used among statisticians and data miners for developing statistical and data analysis.

In this lab, you will use SQL Server 2016 R Services and R tools for Visual Studio to explore a large data set from the City of New York containing details of millions of taxi and Uber trips in the city.

Using this data, you will use SQL R Services, R Tools for Visual Studio and T-SQL to create and deploy a function that predicts the time, fare and tip for a taxi ride.

The lab consists of 5 lesson modules:

1. Loading NYC Taxi data and exploring data using R and SQL queries
2. Exploring R tools for Visual Studio
3. Preparing data using T-SQL and SQL user-defined functions
4. Building a script to predict the probability of a taxi driver receiving a tip
5. Operationalize the predictive model for use via T-SQL stored procedures

For additional background and introduction to R Services, refer to the Appendix (found in the last section of this document).

2.1 Lesson 1: Loading Packages and Exploring Data

The estimated time to complete this lesson is **60 minutes**

2.1.1 Preparing for Lesson 1

For this Lab you should be logged into the two VMs as set out above.

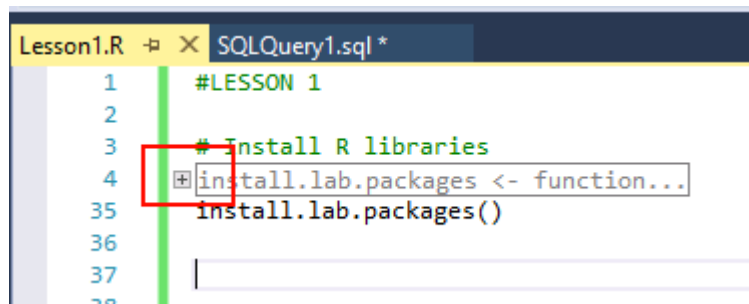
Optional: Install Microsoft R Client

In this hands-on lab, both the student's VM and server's VM have been pre-deployed for you, so you will not need to install R Services (in Database) or Microsoft R Client.

If you are undertaking this lab outside the pre-provisioned virtual environment, then you will need to install the Microsoft R Client on your workstation (student) machine. This can be found at <https://msdn.microsoft.com/en-us/microsoft-r/install-r-client-windows>.

2.1.2 A Note on Installing R Packages

During this lab you will use several Microsoft and 3rd party R packages. The lesson scripts all include a function *install.lab.packages* which will install any missing packages. This will allow each lab script to be executed on a stand alone basis. All of the packages have been preinstalled on the virtual lab environment. You may like to collapse the *install.lab.packages* function within the code editor.



```
Lesson1.R  SQLQuery1.sql *
1 #LESSON 1
2
3 # Install R libraries
4 install.lab.packages <- function...
35 install.lab.packages()
36
37
38
```

The screenshot shows an R script editor with two tabs: 'Lesson1.R' and 'SQLQuery1.sql *'. The script content is as follows: Line 1: '#LESSON 1', Line 2: blank, Line 3: '# Install R libraries', Line 4: 'install.lab.packages <- function...', Line 35: 'install.lab.packages()', Line 36: blank, Line 37: blank, Line 38: blank. A red box highlights a small square icon with a plus sign on line 4, which is used to collapse the function definition.

Packages are loaded into the environment as they are used during the scripts; look out for the use of the *library()* function.

2.1.3 Taxi Data

The data used is a representative sampling of the New York City taxi data set, which contains records of over 173 million individual trips in 2013, including origins, destinations, durations, fares and tip amounts paid (among other things) for each trip.

For this lab exercise, we've sampled to get about 1% of the database occupying about 350MB of raw storage before loading.

2.1.4 Reviewing the Scripts that Created the Taxi Database

Below is the name of the Script used to create the NYC Taxi Database on the Training server, and a short description of what it does.

SQL Script file name	What it does?
create-db-tb-upload-data.sql	<p>Creates NYC Taxi database and two tables:</p> <p>nyctaxi_sample: Creates the table that stored the training data - the one-percent sample of the NYC taxi data set.</p> <p>A clustered columnstore index is added to the table to improve storage and query performance.</p> <p>nyc_taxi_models: An empty table that you will use later to save the trained classification model.</p>

☒ Student activity

Take some time to look at create-db-tb-upload-data.sql SQL script provided in the Student's VM.

1. Open the *create-db-tb-upload-data.sql* file in Visual Studio.
 - a. Go to the Open File menu
 - b. Open the C:\Lab_content\dataScience folder
 - c. Select the *create-db-tb-upload-data.sql* file
2. Look at the section which creates the table and familiarize yourself with the name of the fields:

```

CREATE TABLE {tb_name}
(
    medallion varchar(50) not null,
    hack_license varchar(50) not null,
    vendor_id char(3),
    rate_code char(3),
    store_and_fwd_flag char(3),
    pickup_datetime datetime not null,
    dropoff_datetime datetime,
    passenger_count int,
    trip_time_in_secs bigint,
    trip_distance float,
    pickup_longitude varchar(30),
    pickup_latitude varchar(30),
    dropoff_longitude varchar(30),
    dropoff_latitude varchar(30),
    payment_type char(3),
    fare_amount float,
    surcharge float,
    mta_tax float,
    tolls_amount float,
    total_amount float,
    tip_amount float,
    tipped int,
    tip_class int
)

```

3. Look at the section which creates the columnstore index:

```
CREATE CLUSTERED COLUMNSTORE INDEX [nyc_cci] ON {tb_name} WITH  
(DROP_EXISTING = OFF)
```

Columnstore indexes provide an optimized storage mechanism for analytical processing in SQL Server. They are used by SQL Server column-based data storage query processing to achieve up to 10x SQL query performance gains over traditional row-oriented storage, and are another advantage of using R and SQL services together. For more on Columnstore indexes see: <https://msdn.microsoft.com/en-us/library/gg492088.aspx>

2.1.5 The BCP Utility

The BCP utility can be used to import large numbers of new rows into SQL Server tables or to export data out of tables into data files.

☒ Student activity

Explore BCP script used to load Taxi data to Training Server VM. **DO NOT RELOAD DATA**

1. Take a look at BCP schema file [here](#).
2. Take a look BCP script to load data into NYC Taxi Database [here](#).
3. For example, look at the section where the bcp utility is executed.

```
bcp $db_tb_s in $csvfilepaths -t ',' -S $servers -f taxiimportfmt.xml -F 2  
-C "RAW" -b 200000 -U $u -P $p
```

Where:

- `$db_tb_s` is the name of the SQL Server table. For example, NYCTaxiData
- `$csvfilepaths` is the name and path of the Taxi data file to be loaded
- `$servers` is the name of the SQL Server instance.

2.1.6 Exploratory Data Analysis using SQL

One of the advantages of using R Services, combined with Visual Studio and SQL Server, is that you can use familiar tools and T-SQL queries to connect to the SQL Server database to explore and modify the data.

In this part of the lesson, you will connect to the SQL Server NYC Taxi Database to run basic SQL queries to confirm the data has been loaded correctly.

This will serve a dual purpose of confirming that you can connect to the Training VM, and as preparation for applying Microsoft R Open and Microsoft ScaleR data analysis and statistical functions in the next lessons.

2.1.7 Use Visual Studio to Query the Taxi Database

Please take note of the following information, as you will need this information for the instructions below. These are the SQL Server Authentication credentials for the Server VM.

Server name: 192.168.1.10

User name: sa

Password: Pass@word2

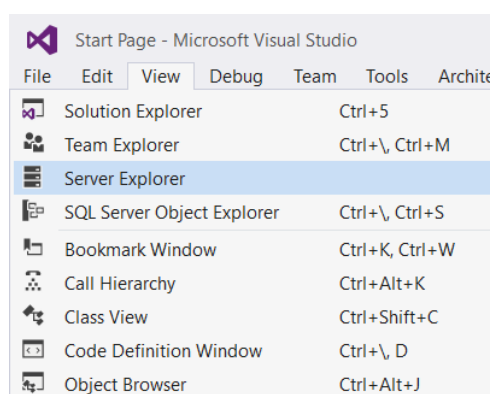
☑ Student activity

In this section you will use Visual Studio to connect to the SQL Server (running in the server VM) and execute SQL queries. There are other methods for submitting SQL queries to SQL Server, but, this is likely to be the most convenient for a data scientist working from within Visual Studio.

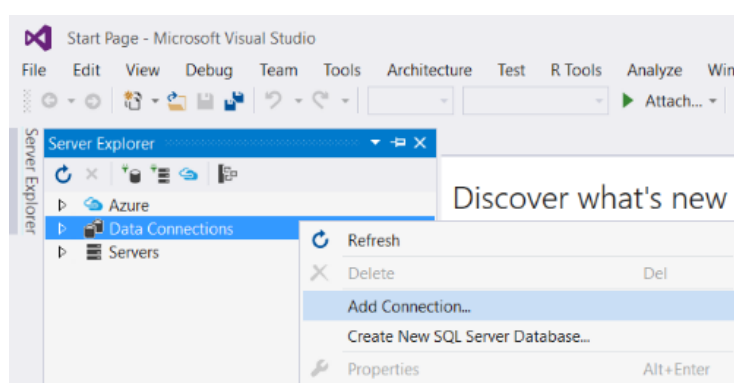
1. If it is not already open, **Open** Visual Studio from the taskbar icon.



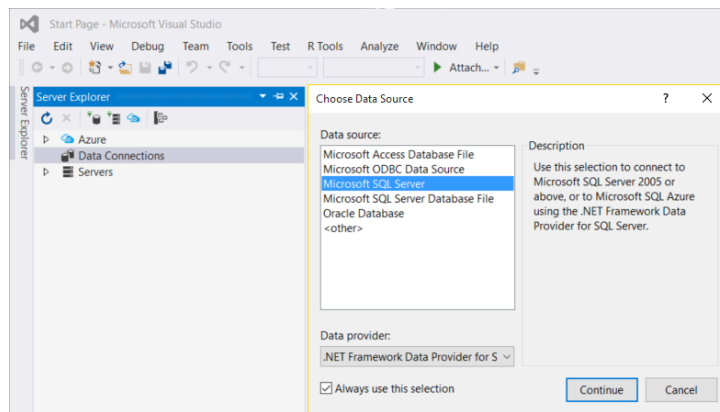
2. **Select** *View>Server Explorer* from the menu.



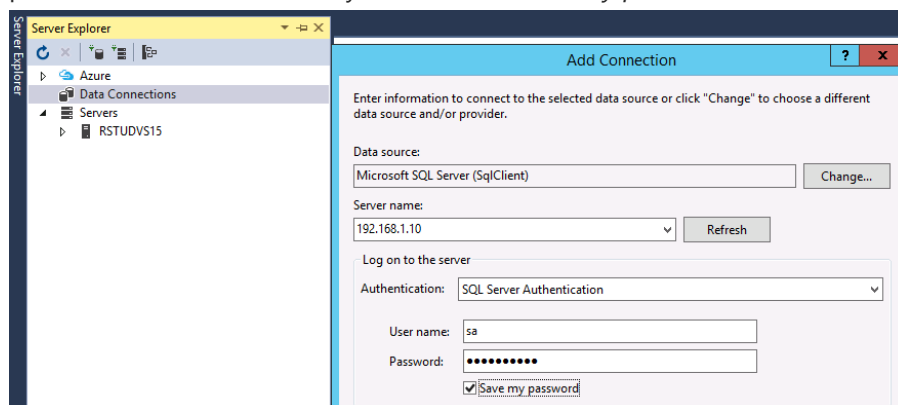
3. **Add** a new *Connection* using the right-click menu on *Data Connections*.



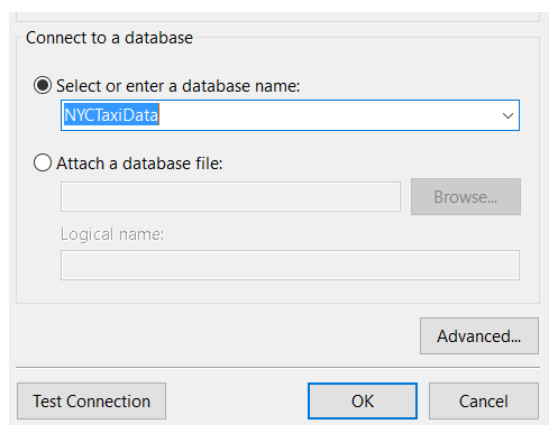
4. In the *Choose Data Source*, set the Data Source to **Microsoft SQL Server**.



5. Click **Continue**.
6. Enter **192.168.1.10** into the *Server Name*.
7. Connect to the server by using **SQL Server Authentication** and entering connection values as provided above. Ensure that you **check** the *Save my password* check-box.

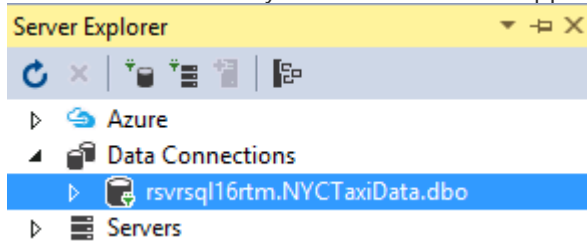


8. Under *Connect to a database* and select the **NYCTaxiData** database.

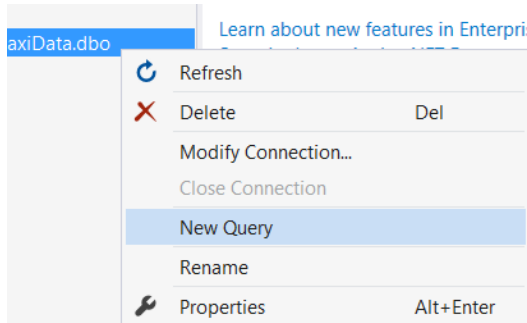


9. Click **Test Connection** to confirm that it is configured correctly.

10. Click **OK**. You will see your connection now appears in the Server Explorer.



11. In the **Server Explorer** pane, right click **rsrvsql16rtm.NYCTaxiData.dbo** and select **New Query**. This will open a new T-SQL query window where you will write and execute your SQL query.

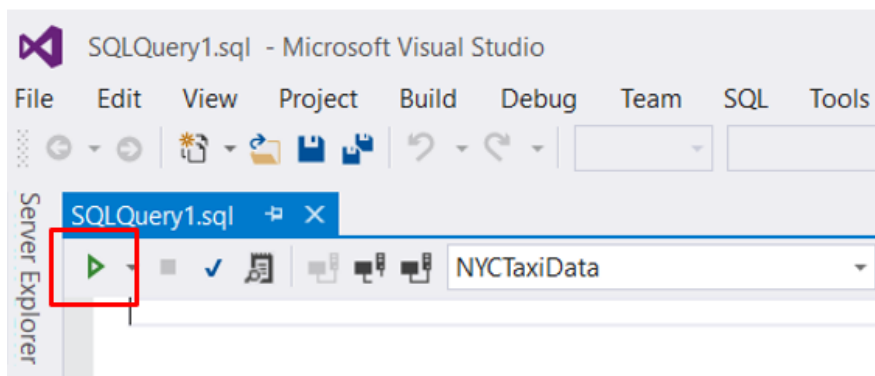


12. Write this SQL code in the query window that you just opened

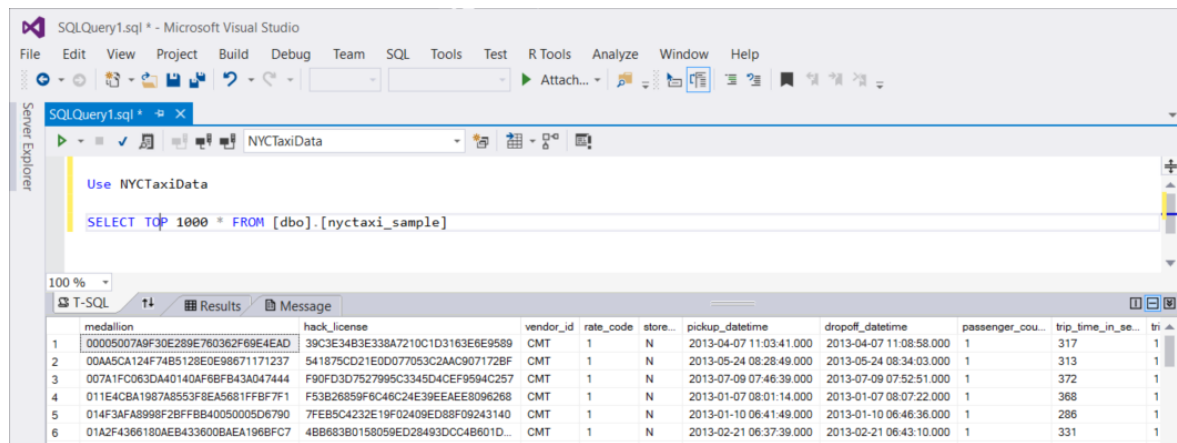
```
SELECT TOP 1000 * FROM [dbo].[nyctaxi_sample]
```

This query will select the top 1000 rows of the **nyctaxi_sample** table, and provide the results in the same window. You should find that Visual Studio provides *intellisense* to help resolve the database4 objects.

13. Execute this SQL query by clicking **Execute** or pressing **F5**



14. You will see the data outputted in the **Results** tab. **Explore** the data contained in each column. Additional information on this dataset can be found at http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml



2.1.8 Generate Summaries Using SQL

SQL Server has the ability to perform set-based calculations very quickly, which makes it a good companion to R, particularly in situations where the data set is too large to fit into memory. A key benefit of working with your data inside SQL Server is this ability to perform fast feature engineering tasks using SQL scripts. In this exercise, you will see how to create new features representing the total and average fares grouped by the passenger count, using T-SQL directly from Visual Studio.

☒ Student activity

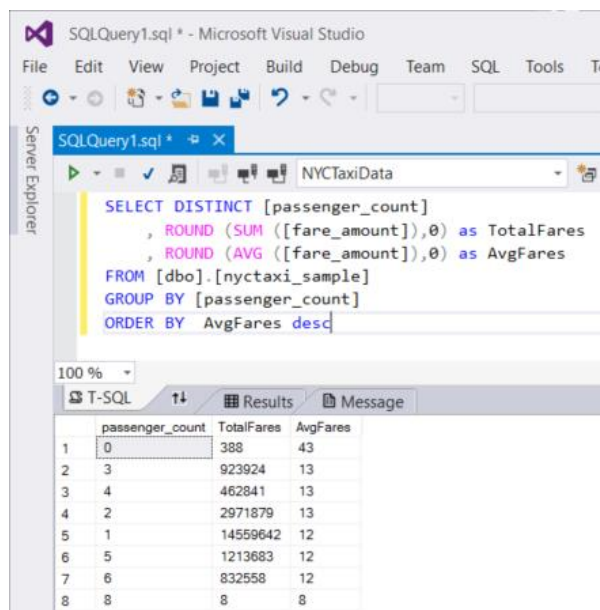
Run the SQL code below to get a quick overview of your data.

1. Write this T-SQL code in a Visual Studio query window. Ensure there are no errors if copy/pasted. This T-SQL is also contained in the file `c:\Lab_Content\dataScience\PassengerCount.sql` on the lab machine.

```
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
```

2. Click **Execute** (F5)

You will see the summary output of the total and average fares for different passenger counts under the **Results** tab.



2.1.9 Introduction to Microsoft R Services

As an experienced .Net Developer or SQL Server administrator, you might be familiar with various Microsoft tools for managing SQL Server, such as Management Studio. R Tools for Visual Studio (RTVS) is a new tool that extends Visual Studio functionality to allow Interactive R queries, which will be shown later in this course. RTVS will also be familiar to you if you've been used to working in the R-Studio IDE.

☒ Student activity

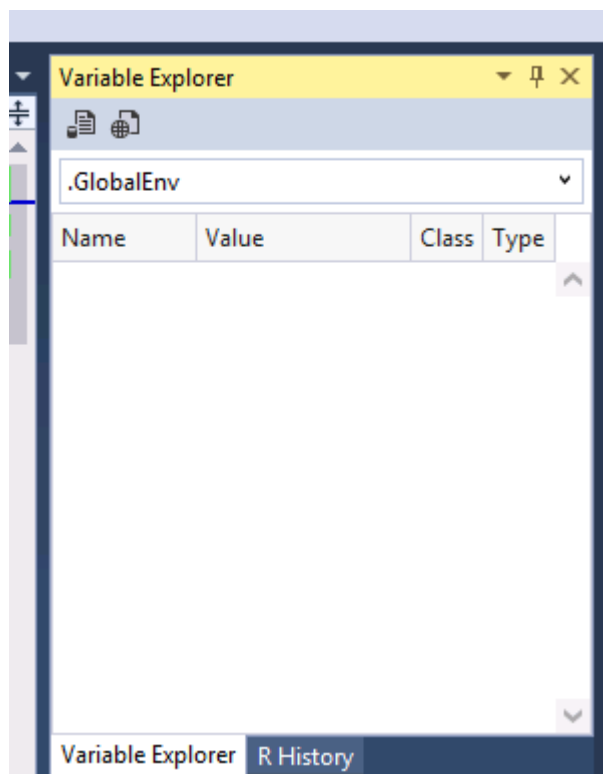
8. If you do not already have it open, go to C:\Lab_Content\myRProject and open the myRProject solution in Visual Studio.
9. Select the **Solution Explorer** pane in Visual Studio
As a result of opening *myRProject* in Visual Studio, you should be able to see the following R scripts in the solution explorer window.
 - a. Lesson1.R
 - b. Lesson3.R
 - c. Lesson4.R
 - d. Lesson5.R
2. Open **Lesson1.R**.

Throughout this lab we have provided all of the R scripts for you to execute. We recommend executing these interactively, a few statements at a time, as a good way to develop an understanding of the code.

1. **Run** the following lines of code by selecting it and pressing **Ctrl-Enter**.

```
install.lab.packages <- function() {  
  code emitted...  
}  
  
install.lab.packages();  
  
rm( list = ls())
```

You should see no errors in the interactive R console and a completely empty *Variable Explorer* pane as shown below. You can also reset the environment using the menu *Tools > Session > Reset*



2. **Run** this line

```
library(RevoScaleR)
```

This loads the *RevoScaleR* package, part of SQL Server R Services. We will be using functions from this package to work with R executing on the SQL Server.

2.1.10 Define an R Services Data Source

In R Services, a *data source* specifies a set of data or tables inside SQL Server that you wish to use for running various data analysis and exploration methods with the R language. Defining a *data source* will allow you to run Microsoft ScaleR library functions against that data.

In this section, you will define a SQL Server R Services data source using Microsoft's *RxSQLServerData* and *rxSetComputeContext* ScaleR functions.

- *RxSqlServerData* is a function provided in the RevoScaleR package to support improved data connectivity to SQL Server.
 - You use this function in your R code to define the data source. The data source object specifies the server and tables where the data resides and manages the task of reading data from and writing data within SQL Server.
- The *RxInSqlServer* function is used to specify an execution context or compute context. Once set, the compute context setting determines whether the R code that follows it will be executed on your local workstation, or on the remote server that hosts the SQL Server instance. Remote execution facilitates analysis of large data sets by reducing movement and increasing parallelism during execution. This is a key capability of SQL Server R Services.

2.1.11 Specify and Call the Compute Context

Typically, when you are using R, all operations run in memory on your local computer. However, in SQL Server R Services you can specify that R operations are executed in an R environment on the SQL Server server hosting the SQL Server instance.

To set the compute context to SQL Server, you will define three parameters:

- *sqlShareDir* - You need to specify a temporary directory when serializing R objects back and forth to the SQL Server. In the code below, you will be using your username.
- *sqlWait* - an R variable that you create to store the parameters passed to the SQL Server data object.
- *sqlConsoleOutput* - stores the values you pass to the constructor *RxSQLServerData* constructor when the object is created.

Also, a *colClasses* argument is used to specify the column class format in R.

Some important notes on datatypes in R and SQL Server:

- Whereas SQL Server supports several dozen data types, R has a limited number of scalar data types (numeric, integer, complex, logical, character, date/time and raw). Therefore, when you use data from SQL Server in R scripts, several different things can happen:
 - Data is implicitly converted to a compatible data type.
 - Data cannot be implicitly converted and an error is returned
- *colClasses* is not a replacement for CAST and CONVERT T-SQL function. In general, additional data “wrangling” and data engineering might be needed to ensure that the data type conversions are performed as intended before using the data in your R script – specially with publicly available data which may need to be adequately converted before loading to any database, as will be shown in one of the next lessons of this course.

Once the *RxSQLServerData* object is created, you will be able to use it to run queries and R code as in the example in the student activity below.

☒ Student activity

In this section you will create an R Services data source by running a SQL query from within R. All of the code to be executed is contained within the script file that you opened.

1. If you have not already done so, open your Solution and R Script as per the instructions previously provided and follow these instructions by running line or group of lines as indicated below.
2. **Run** these lines of code to create the SQL Server In Database Compute Context.

```
connStr <- "Driver=SQL
Server;192.168.1.10;Database=NYCTaxiData;Uid=sa;Pwd=Pass@word2"
sqlShareDir <- paste("C:\\AllShare\\", Sys.getenv("USERNAME"), sep="")
sqlWait <- TRUE
sqlConsoleOutput <- TRUE
cc <- RxInSqlServer(connectionString = connStr, shareDir = sqlShareDir,
                    wait = sqlWait, consoleOutput = sqlConsoleOutput)
rxSetComputeContext(cc)
```

You may like to view the documentation by running the command `?RxInSqlServer` but the key parameters to note are.

- `sqlShareDir` - A temporary directory when serializing R objects back and forth to the SQL Server. In the code below, you will be using your username to ensure this is unique
- `sqlWait` - a logical flag indicating whether the remote execution is blocking on the client.
- `sqlConsoleOutput` - a logical flag indicating that the *stdout* of the remote R process should be printed in the client R console. We have set this to true here for illustrative purposes.

3. Now we can check our compute context. **Run** this block of code.

```
rxGetComputeContext()
rxExec(R.Version)
```

You will note that the output indicates that we are using the SQL Server In Database Context. The second line executes the `R.Version` function in that remote context and you will be able to confirm that this is indeed executing remotely on the SQL Server.

4. Now that you have defined the SQL Server connection and remote context you can define a data source as the result from an SQL query. **Run** these lines of code.

```
sampleDataQuery <- "select top 100000 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"
inDataSource <- RxSqlServerData(sqlQuery = sampleDataQuery,
                                connectionString = connStr,
                                colClasses = c(pickup_longitude =
"numeric", pickup_latitude = "numeric", dropoff_longitude = "numeric",
dropoff_latitude = "numeric"),
                                rowsPerRead=500)
```

10. Finally **Run** this block of code to test the data source.

```
localData <- rxImport(inDataSource)
summary(localData)
```

This will retrieve the results of the query back into a local *data.frame* and then use the R *summary* function to show summary information.

2.1.12 View and Summarize Data Using R

The R language allows you to apply various statistical and data analysis methods to data in order to form a better understanding of the data and its properties. This 'exploratory data analysis' is often the first step in a data science process.

As an R developer, you might want to start your analysis by retrieving some key summary information about your data. For example, you might calculate the minimum and maximum values of your data their mean and other distributional characteristics such as standard deviation.

In this section we will apply R functions to a SQL data source to perform exploratory analysis of those data. We will be using the following Microsoft ScaleR functions:

- *rxGetVarInfo* – will be used to output a description of the variables and their data types in the *rxSqlServerData* object you have created.
- *rxSummary* – will be used to get more detailed statistics about individual variables.

In this section, you will generate data summaries using both R functions. The first function will return a summary of the data types of the source data set, and the second will return a summary of the fare amounts based on passenger count.

☒ Student activity

Execute the following R code in the script provided using CTRL+ENTER or copy pasting from the R Script into the Interactive R console.

1. **Run** the function *RxGetVarInfo*. This will return promptly as the information required can be determined from the SQL Server metadata (schema).

```
rxGetVarInfo(data = inDataSource)
```
2. Now **run** *RxSummary* function.

```
rxSummary( ~., data = inDataSource)
```
11. There may be a small processing delay. In order to calculate the summary information R Services for SQL Server needs to process the actual data stored within the database. If you are familiar with the R *summary* function then the *RxSummary* function is similar but optimized to execute in the context of Microsoft R Server. Unlike the call to *summary* when we tested the data source above, all of the execution here is being performed in the remote context. The first parameter to *RxSummary* uses the *rxFormula* notation; this is similar but not identical to the R 'formula' notation. See *?rxFormula* for more details.

2.1.13 Create Graphs and Plots Using R

As part of data exploration you will often want to visualize your data as this may provide more useful insights such as allowing identification of outliers or providing a better understanding of the data distribution.

SQL Server R services provides several remote context aware charting options. These support the Compute Context and allow the bulk of the processing to occur remotely without the need to move data. In this section you'll look at how to use the *rxHistogram* function (which supports the remote context) as well as how to use *rxImport* to pull data back into local memory for charting in other R frameworks such as *ggplot* or *lattice*.

2.1.13.1 Create a Histogram

A histogram can be used to show the distribution of fares across the entire dataset. You will be able to see the frequency of the different fare amounts in the data.

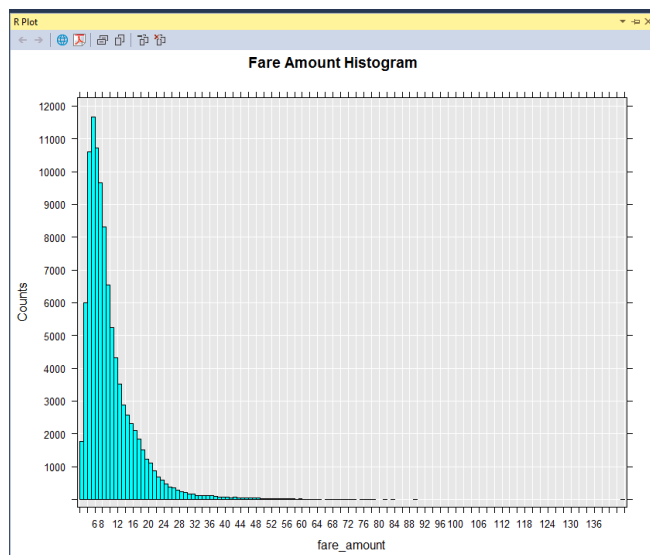
☒ Student activity

Execute the following R code in the script provided using CTRL+ENTER or copy pasting from the R Script into the Interactive R console.

1. **Run** the *RxHistogram* function.

```
rxHistogram(~fare_amount, data = inDataSource, title = "Fare Amount Histogram")
```

2. The image of the plot is returned in the **Plot Window**.



2.1.13.2 Create a Map Plot

In this example, you will generate a plot object using the *ggplot* library. In order to use *ggplot* we need the data in local memory, we use *rxImport* to move that data back to the local machine where it is rendered as a plot.

☑ Student activity

1. **Run** this code.

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

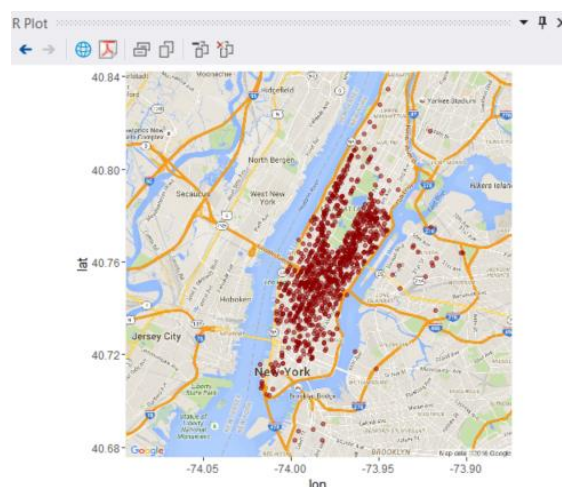
2. Define a custom function called `mapPlot` by **running** this code.

```
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))
}
```

3. **Call** the function

```
myplots <- mapPlot(inDataSource, googMap)
plot(myplots[[1]])
```

View the resulting image in the **Plot Window**. You may see a warning indicating that some of the data is missing the *geom_point* value. You can safely ignore this missing data.



2.2 Lesson 2: Exploring R Tools for Visual Studio

2.2.1 Preparing for Lesson 2

The purpose of this lesson is to provide a tour of R Tools for Visual Studio (RTVS). Once again, the pre-prepared virtual machine contains all of the components required for the lab already installed.

Total time to complete this lesson is 30 minutes

If you are undertaking this lab on your own machine then you will need:

1. Visual Studio 2015
2. R Tools for Visual Studio (RTVS).
3. A copy of the lab solution "ShowRTVS.sln" Summary and Context

R Tools for Visual Studio (RTVS) is a new tool that brings the power of the Visual Studio IDE experience to R users. RTVS extends Visual Studio functionality for R programming and is a free open source tool distributed under the MIT license.

In this lesson, you will have the opportunity to explore the latest version of R tools for Visual Studio. Following a short introduction to RTVS's general R programming functionality, this lesson will dive into three of the key areas of R development. Authoring and interactive execution of R code, debugging and plotting and data visualization.

This R Tools for Visual Studio lesson is prepared for two main audiences, by focusing on the core features that can differentiate and provide a better IDE experience to R users.

- As an experienced R user you might already be familiar with other R IDEs, such as R Studio; R Tools for Visual Studio provides a similar experience to R Studio but within a rich integrated development environment, Microsoft Visual Studio.
- As a seasoned SQL Server or .Net developer you may have already used Visual Studio's powerful and advanced features for building modern web, mobile and cloud applications. RTVS is a new tool to allow R programming inside of Visual Studio. If you are new to R then using RTVS will allow you to learn from within the comfort of an IDE you already know well.

As mentioned in previous sections, this course uses RTVS version 0.4 for Visual Studio. RTVS is continuously evolving, and both the Microsoft development team and the RTVS tool community will always welcome your feedback and contributions. The Github repo can be found here: <https://github.com/Microsoft/RTVS>

2.2.2 Introducing R Programming in Visual Studio

The R language is used by thousands of data scientists and programmers for statistical computing and data analysis. The R Tools for Visual Studio (RTVS) provide an extension to the Microsoft Visual Studio IDE to support R development.

The main capabilities of RTVS are;

1. **Interactive R** – R code can be authored and executed interactively in the context of an R environment. RTVS includes full intellisense support for both the code editor and interactive window.

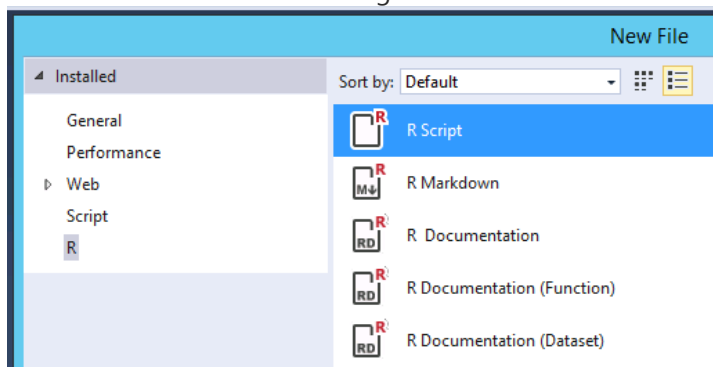
2. **Variable Explorer** – allows you to see the variables within your environment and within the scope of the packages that you have loaded. Includes a tabular view of R *data.frame* types.
3. R code **Debugging** – supports attaching the Visual Studio debugger to an R process.
4. Getting **R Help** – R packages typically provide comprehensive documentation; RTVS provides easy access to package documentation.
5. **R Plots** – RTVS provides rich support for the R graphics framework. This makes it easy to work with the various R libraries tools to produce graphs and plots.
6. **R Documentation** – including authoring of R markdown to generate professional data analytics reports including data used, executable code, charts and analytics content.

2.2.3 Exploring Interactive R

☑ Student activity

In this section you'll see the basics of working with the Interactive editor in RTVS. We'll use one of the sample datasets that ships with R, the *mtcars* dataset. See the help for *?mtcars* for more details on this data set.

1. Use the menu to select **R Tools > Data Science Settings**
This will configure Visual Studio with an optimal layout of windows for data science tasks. If you are an R-Studio user you may also like to explore the *R Tools > Editor Options* menu to change settings such as *Intellisense > Commit on Space/Enter Key*
2. **Open** the *ShowRTVS* solution from *c:\Lab_Content\ShowRTVS*
3. Use the menu to select **File > New > File**
The *New File* dialog will appear
4. From left hand side of the dialog select **R** and then choose **R Script**

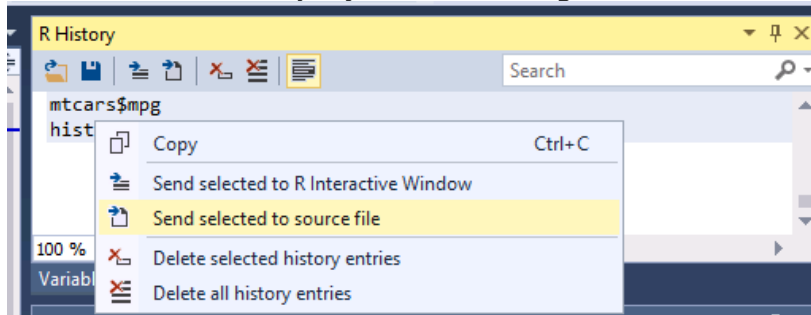


5. Click the **Open** button
You will see a new R Script open in the editor window. There are two options for us to enter R code into RTVS. We can enter it into the script editor or we can enter it directly into the R Interactive Window. We'll look at both of these options now.
6. Inside the *Script1.R* file in the editor **type** the following R code and then press **Ctrl + Enter**
Ctrl+Enter will submit the current line or the currently selected line(s) of code to R. You will see the command is sent to the Interactive window and is executed.

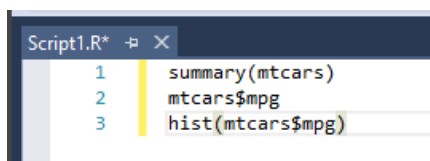
```
summary(mtcars)
```

7. Now select the *R Interactive* window and press the **Up Arrow** key. You will see that the command you just executed is retrieved from the history. Press **Enter** to re-execute that command.
8. You can enter R code directly into the *R Interactive* window. Type the following code into the interactive window and press **Enter**. You should see the mpg column vector output to the *R Interactive* window and a histogram of the *mpg* column of the data will be drawn in the *R Plot* window.

```
mtcars$mpg
hist(mtcars$mpg)
```
9. As well as using the up arrow key to scroll back through history you can view the command history in the *R History* window. Select the **R History** tab, **scroll** to the bottom of the history, **select** the last two lines (the command you just submitted), **right click** and choose **Send selected to source file**



The two lines of code will be inserted into the Script1.R source file that you have created.

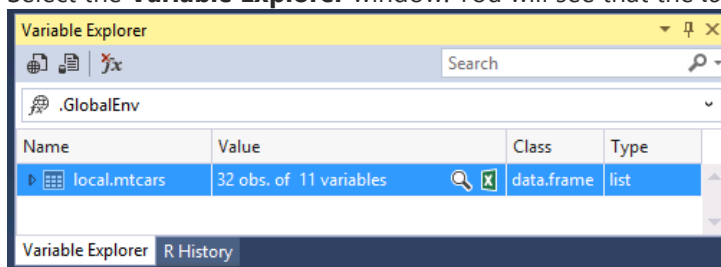


This is a very useful pattern to understand; you are able to use the Interactive window to prototype and test commands. Once you are comfortable you can send those commands into the source window to be saved as part of your script.

10. In the R Interactive window **execute** the following command to create a local *data.frame* variable containing the *mtcars* data

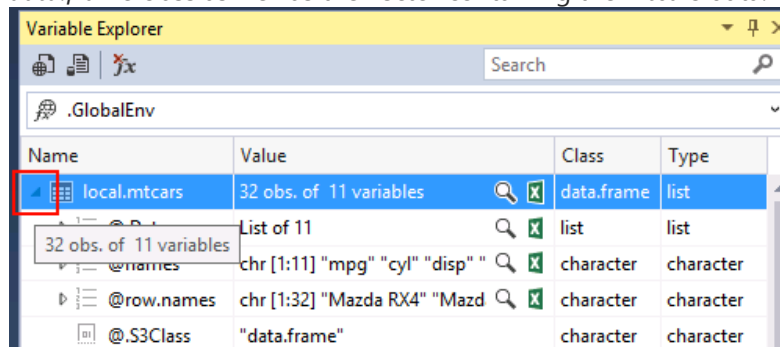
```
local.mtcars <- as.data.frame(mtcars)
```

11. Select the **Variable Explorer** window. You will see that the *local.mtcars* variable is shown

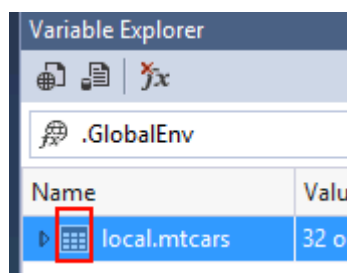


The *Variable Explorer* allows you to easily see the variables that are currently in various scopes within your R Environment.

- Click the expand glyph on the left hand side of the variable name. You will see the *attributes* of the *data.frame* class as well as the vector containing the *mtcars* data.



- The *data.frame* type is a very important construct in R, it effectively represents a tabular set of data. The variable explorer allows us to view this class in a special view whereby we can see and interact with the data in tabular form. **Double click** the small table glyph in the *Variable Explorer* window



You will see the *local.mtcars data.frame* is opened in the main Visual Studio window as a tabular view. **Experiment** by interacting with that window, for example scroll around the data, sort by clicking the column headings and so forth.

R Data: local.mtcars							
	mpg	cyl	disp	hp	drat	wt	qsec
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.99
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02
Datsun 710	22.8	4	108.0	93	3.85	2.320	16.46
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	16.86
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.05

- RTVS has one more neat trick for working with *data.frame* variables. **Double click** the Microsoft Excel icon in the *Variable Explorer* window. This will open the *data.frame* as a CSV in Microsoft EXcel or any other CSV or Text editor.

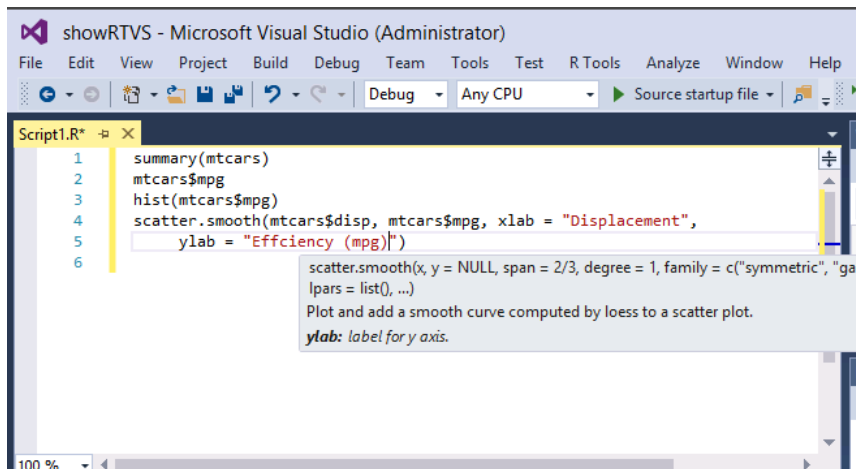
At this time both the internal tabular viewer and external CSV viewing mechanism are read-only. Any changes that you make to the CSV file will not be reflected back into your R environment variables.

- RTVS provides rich *Intellisense* code completion and inline documentation. In the Script1.R code editor **type** and **execute** the following command.

```
scatter.smooth(mtcars$disp,mtcars$mpg,
               xlab="Displacement",ylab="Efficiency (mpg)")
```

Note how Intellisense suggests completion for the command and for the parameters and also the

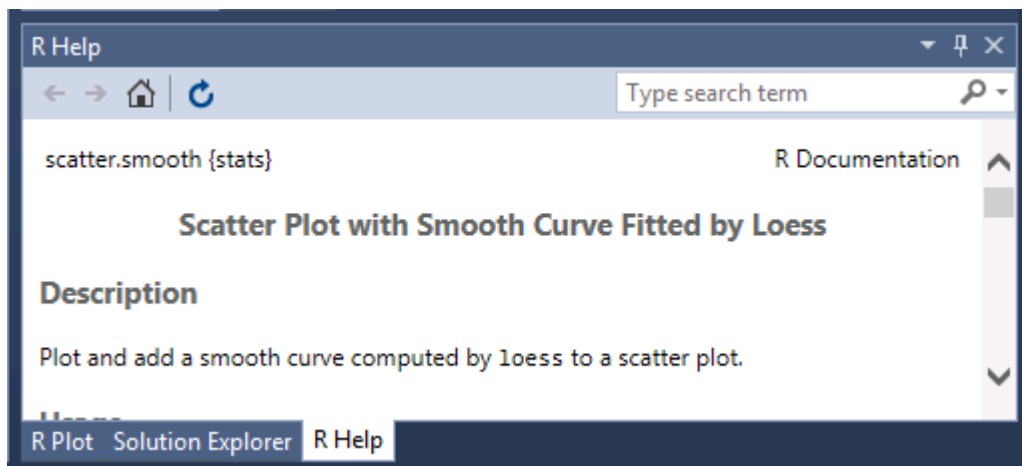
fact that documentation on parameters is presented as you type. The plot shows us that there is, unsurprisingly, a relationship between engine displacement and fuel efficiency.



You can access help for a function at any time by typing `?<function_name>` into the *R Interactive* window. So, for example, type the following to retrieve the help for the `scatter.smooth` function that we just executed

```
?scatter.smooth
```

You will see the detailed help displayed in the *R Help* window.



16. **Close** *Script1.R*. There is no need to save this file.

2.2.4 Debugging R

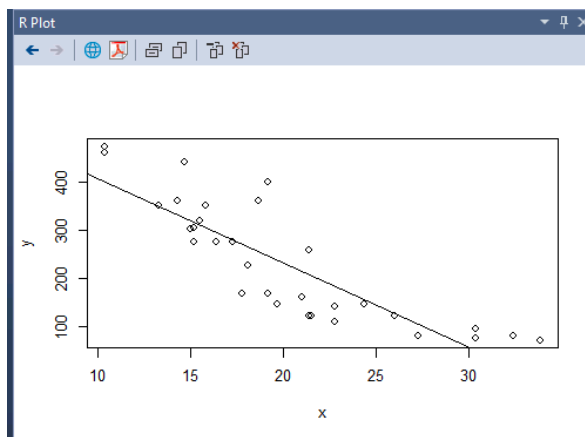
RTVS brings the power of Visual Studio debugging to R. Debugging is usually a key capability of an IDE and you may well be familiar with the Visual Studio debugger from working with other languages and platforms, the first thing you will do to be able to debug an R code is to attach the debugger to the R Runtime. There are a number of ways to do this. In this lesson, we will present a straightforward way of using the "Attach Debug" button in the Interactive R console, setting up a breakpoint, sourcing the code and then moving through the code using familiar F11 and F10 keys.

If you're an experienced Visual Studio user, you'll find the R debugging experience a bit different from what you may be used to. For example, if you're writing a C# console application, you're used to just pressing F5 to launch your console application under control of the debugger.

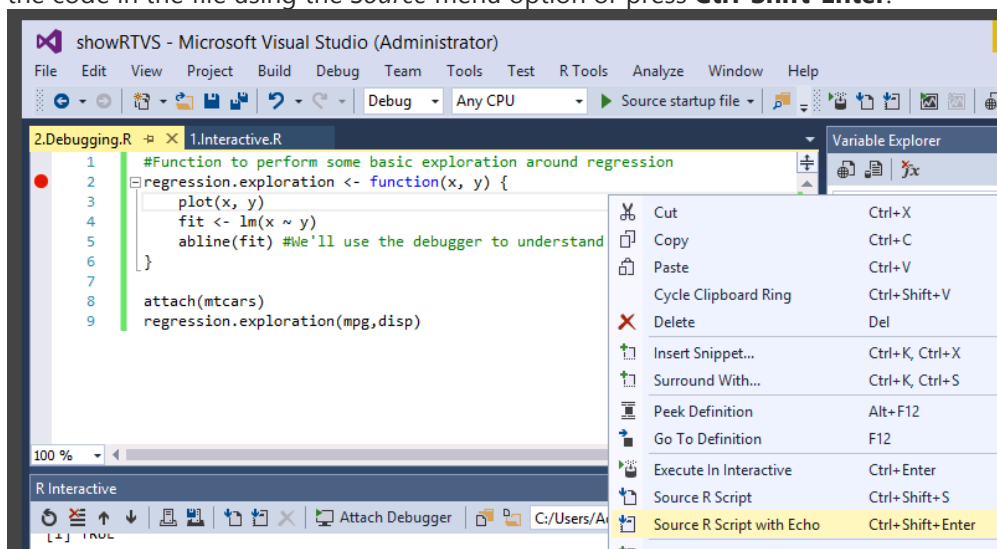
In R there isn't a `main()` function to start debugging at. Furthermore, since there isn't a compilation step either, you need to first tell RTVS what code you would like to debug. In this way, R is more like ASP.NET debugging, where you need to tell Visual Studio which page to start debugging at.

✓ Student activity

1. Reset your R environment using the menu **R Tools > Session > Reset**
2. **Open** the script file *2.Debugging.R* from the *Solution Explorer* Window.
This script contains a function which will plot two variables and then fit a simple linear regression and plot that as a line. Ideally it'll output something like the following.



3. At the moment the function is not working correctly. You can take a look for yourself. **Execute** all of the code in the file using the *Source* menu option or press **Ctrl-Shift-Enter**.

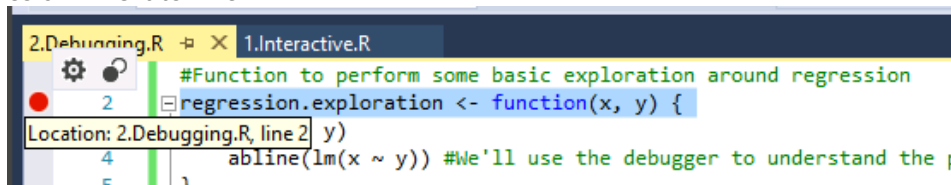


Notice that the regression line is not visible yet the code looks fine at first glance. Let's use the debugger to better understand what is going on by stepping through the code.

4. In the *Interactive R* window and click the **Attach Debugger** button



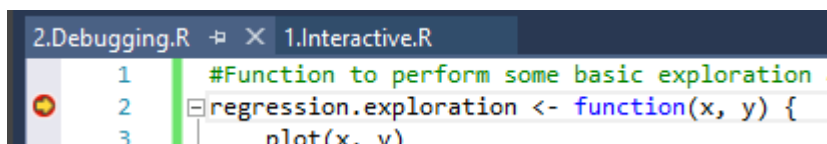
5. Set a breakpoint on the `regression.exploration` function by clicking in the left hand side column next to *Line 2*



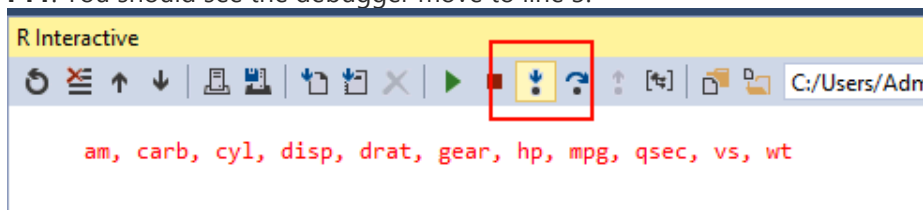
6. From the *R Interactive* window call the `regression.exploration` function by **executing** this command

```
regression.exploration(mpg, disp)
```

You should see the debugger break on line 2. This is denoted by the small yellow arrow on the left hand side.

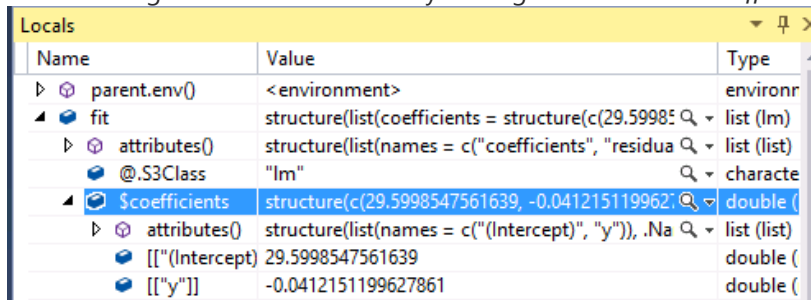


7. **Step-into** the function using either the button on the *R Interactive* window toolbar or by pressing **F11**. You should see the debugger move to line 3.



8. We know that the scatter-plot is being rendered successfully so we will **step-over** this line using **F10**. You will see the plot correctly drawn.
9. Now let's **step-over** the call to `lm` using **F10**. The debugger will move to line 4.
10. The Visual Studio debugger provides a variety of options for us to view variables in local scope. **View** the `fit` variable using these three approaches;

- a. **View** it using the *Locals* Window. Try drilling down into the `coefficients` attribute



- b. **Hover** over the `fit` variable name in the *Code Editor* window. You'll be able to drill down in a similar fashion

- Straight away you might be able to see the issue here. The *Intercept* for the regression line is 29.59 but, looking at the scatter plot and intuiting, it should be intercepting the y at a value of several hundred.

- ```
fit <- lm(y~x)
```

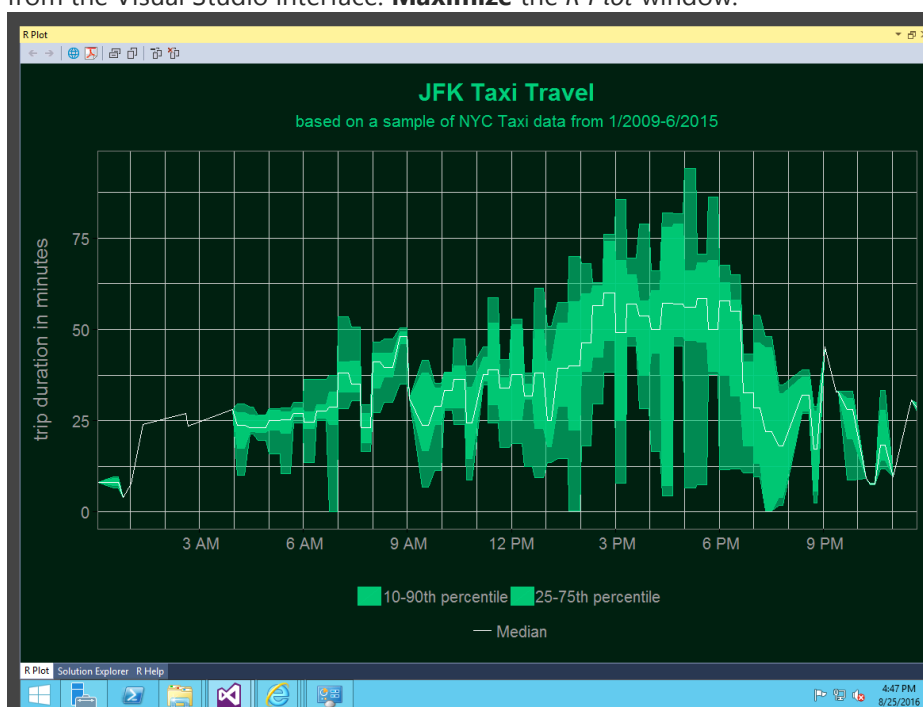
| Name            | Value                                                                                                                                                                                                                                  | Type        |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|
| parent.env()    | <environment>                                                                                                                                                                                                                          | environr    |
| fit             | structure(list(coefficients = structure(c(580.883822558502, -17.4291216703563), .Dim = c(2L)), .Names = c("Intercept", "x")), .Data = matrix(c(580.883822558502, -17.4291216703563), 2, 1), .Dimnames = list(c("Intercept", "x"), .Nai | list (lm)   |
| attributes()    | structure(list(names = c("coefficients", "residuals"), .Names = c("coefficients", "residuals")), .Data = matrix(c(580.883822558502, -17.4291216703563), 2, 1), .Dimnames = list(c("coefficients", "residuals"), .Nai                   | list (list) |
| @S3Class        | "lm"                                                                                                                                                                                                                                   | character   |
| \$coefficients  | structure(c(580.883822558502, -17.4291216703563), .Dim = c(2L)), .Names = c("Intercept", "x"))                                                                                                                                         | double (    |
| attributes()    | structure(list(names = c("Intercept", "x")), .Names = c("Intercept", "x"))                                                                                                                                                             | list (list) |
| [["Intercept"]] | 580.883822558502                                                                                                                                                                                                                       | double (    |
| [["x"]]         | -17.4291216703563                                                                                                                                                                                                                      | double (    |
| \$residuals     | structure(c(-54.8722674810208, -54.8722674810208), .Dim = c(2L)), .Names = c("Intercept", "x"))                                                                                                                                        | double (    |
| \$effects       | structure(c(-1305.1600190446, 584.86252752782), .Dim = c(2L)), .Names = c("Intercept", "x"))                                                                                                                                           | double (    |

- ### 2.2.5 Plots in R

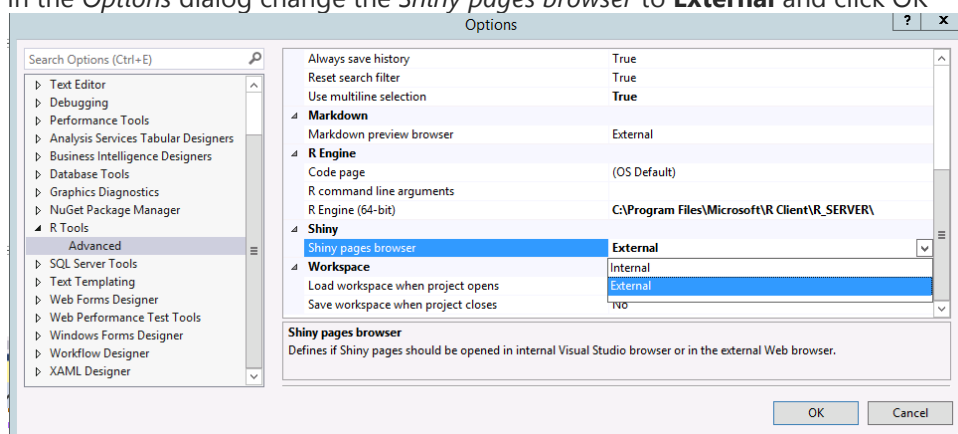
The actual plotting code itself is not the focus of this exercise; we'll simply source and then execute the scripts, but, feel free to explore the code if you'd like to understand it better.

1. Open the **Plot.R** Script.
2. **Source** the script by pressing **Ctrl-Shift-Enter**
3. **Execute** the `do.qqplot` function by calling it from the *R Interactive* window using `do.qqplot()`

- This plot is designed to be rendered at full screensize. **Drag** the *R Plot* window to detach/undock it from the Visual Studio interface. **Maximize** the *R Plot* window.

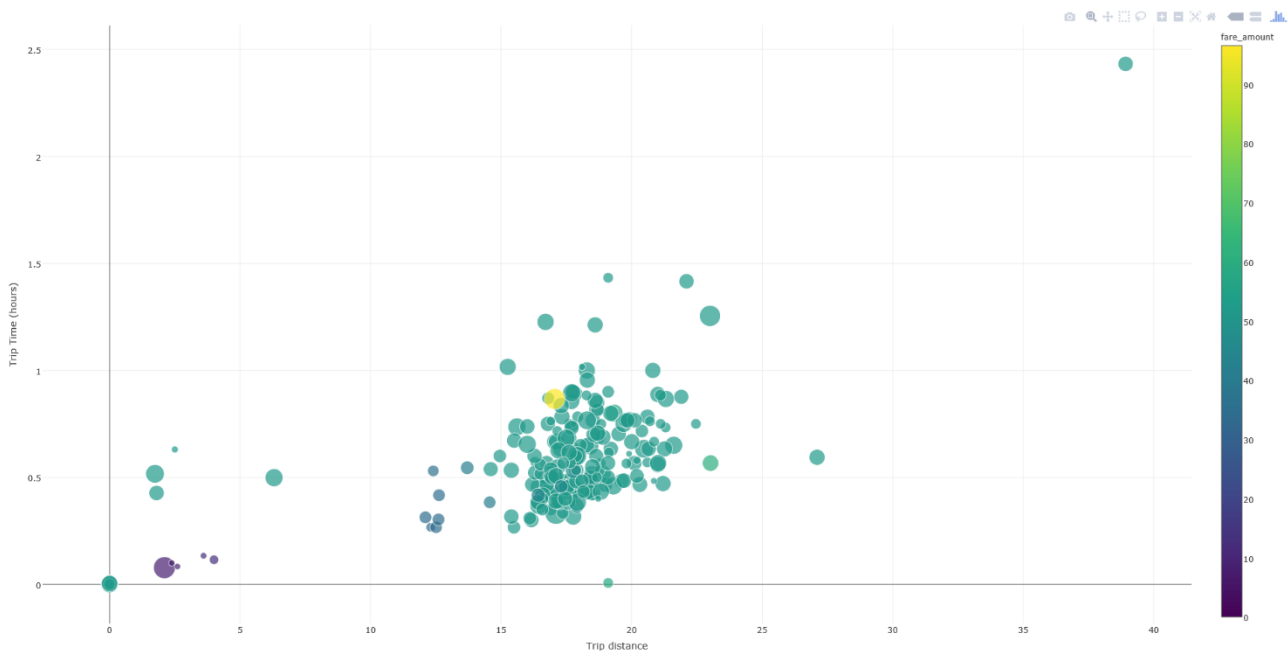


- The toolbar in the *R Plot* window within RTVS provides a number of helper functions for exporting data. Try using the toolbar to **Export** the chart as a PDF.
- Re-dock** the *R Plot* window into Visual Studio.
- The next plot uses a library called *plotly* and a framework called *shiny* which allows R to generate rich HTML content. To view this correctly it is best to open the result in an external browser. Use the menu to open **R Tools > Options**
- In the *Options* dialog change the *Shiny pages browser* to **External** and click OK



- Return to the *R Interactive* window and **execute** the `do.plotly` function using `do.plotly()`. The *plotly* library will generate an html output. You may need to click the security warning to *Allow Blocked Content* to show the page.

You should see the following plot open in your browser. The plot is interactive, try hovering your mouse pointer over portions of the plot and observe how tool-tips appear.



## 2.3 Lesson 3: Preparing Data using T-SQL and R Services

In many data science situations, the effort involved in loading, cleansing and transforming data actually exceeds the time spent actually building models. Relational Database Management Systems are well equipped for many of these data manipulation tasks. The R language support provided by SQL Server makes it much easier to include as part of an overall data science pipeline.

**Total time to complete this lesson is 30 minutes**

### 2.3.1 Feature Engineering with Transformations in R Services

Most machine learning models are extremely sensitive to the way in which data is presented to them. Feature engineering is the process of using domain knowledge of the data to create features that make machine learning algorithms work<sup>1</sup>. Feature engineering typically involves some set of informed transformations of the original source data, often it also requires combination of multiple sources of data. So, for example, the taxi data set might contain a *'pickup\_datetime'* value indicating the time that a particular taxi journey took place, a data scientist might choose to transform that date value into many new features.

- *IsWeekend*. There is likely to be value in creating a feature that indicates whether that particular date is a weekend day as taxi journeys may well be different between week days and weekends
- *IsPublicHoliday*. The nature of taxi journeys are also likely to differ on public holidays as opposed to normal working days.
- *CountOfJourneysThisDay*. On particularly busy days it is likely that journeys will be different to those on relatively quiet days.

We could go on here, and this is just looking at potential derivations from the *pickup\_datetime* feature. The process of brainstorming features is a mixture of intuition, art and statistical knowledge as well as an understanding of the machine learning model to which the features will be presented. The process of actually creating those envisioned features is fundamentally a challenge of data manipulation. One key capability of SQL Server R Services is that it allows data scientists to more easily perform the manipulation mechanics of feature engineering.

You have already seen feature engineering in action in Lesson 1. We used the geo-spatial query capability within SQL Server to take the starting and ending co-ordinates of a journey and calculated the distance of the journey. Say that we are endeavoring to build a predictive model for trip-time; knowing where the journey started may be somewhat useful, maybe trips originating in Manhattan tend to be shorter than trips originating in other areas of New York. But, it's a near certainty that the 'as-the-crow-flies' distance between the start and end of the journey will have strong predictive power. The spatial features of T-SQL were extremely well suited to engineering this feature.

### 2.3.2 Feature Engineering using T-SQL

In the previous lessons, the Taxi data you loaded created a NYC Taxi data table, which contained latitude and longitude values for the pickup and drop-off locations, but did not include a direct distance column. It is not

---

<sup>1</sup> [https://en.wikipedia.org/wiki/Feature\\_engineering](https://en.wikipedia.org/wiki/Feature_engineering)

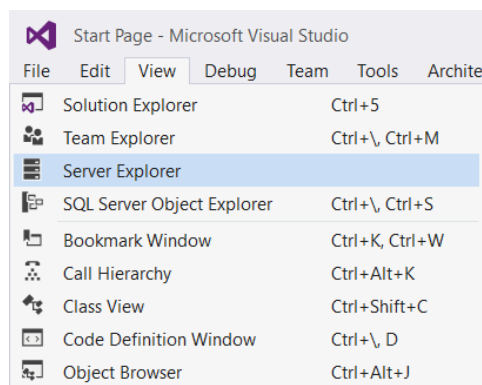
difficult to imagine that the direct distance in miles has a strong correlation to the time and expense of the trip, so we will add a direct distance column. You will synthesize this direct distance column using the Haversine formula, which calculates the *great-circle distance* between two points on a sphere.

You will deploy this formula using a user-defined function in T-SQL, which works in a similar way to other programming languages routines and subroutines. It receives input parameters, encapsulates logic and return results but it does this closer to the database, extending SQL database functionality and leveraging the SQL Server database engine.

To facilitate the flow of this training, a user-defined SQL function has been created for you. In this lesson, you will look at the SQL code that calculates the great circle distance from *s* in two pairs of latitude and longitude values and outputs a new column that tells you the calculated distance in miles. You will then run this function in Visual Studio as part of a SQL feature-engineering query used to select the dataset for training the model. You will then verify the new column is created both in SQL and in R, running an updated R summary of the dataset with a newly created feature set.

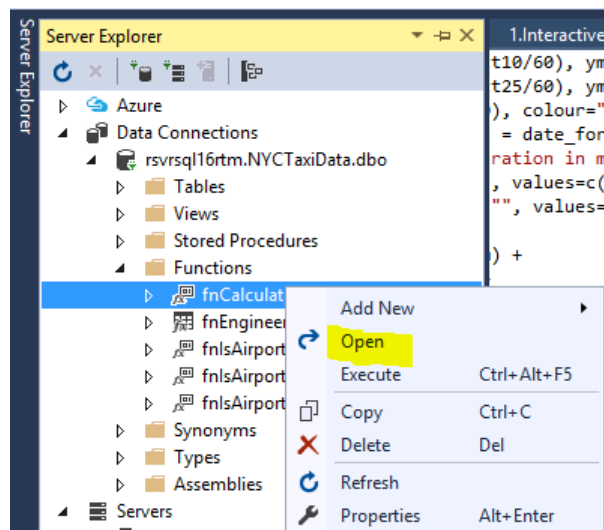
### ✓ Student activity

1. If you do not already have it open, **open** the solution `c:\Lab_Content\MyRProject\myRProject.sln`
2. Use the menu to open **View > Server Explorer**.



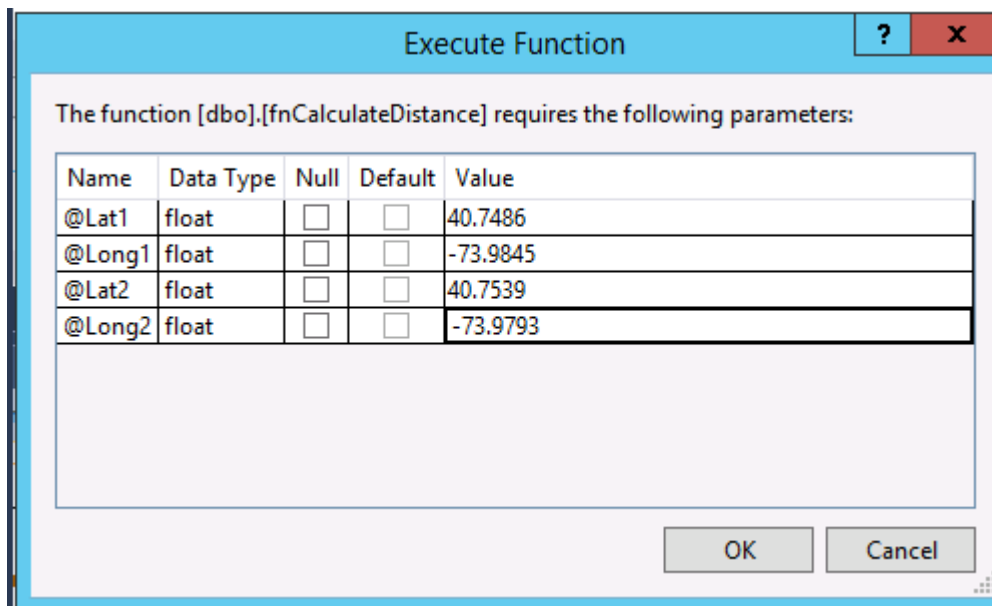
3. From within Server Explorer **expand** *Data Connections > rsrvsql16rtm.NYCTaxiData.dbo*. Return to *Lesson 1* if you do not have a data connection setup
4. **Expand** the *Functions* folder.

5. **Right click** the *fnCalculateDistance* function and select **Open**.

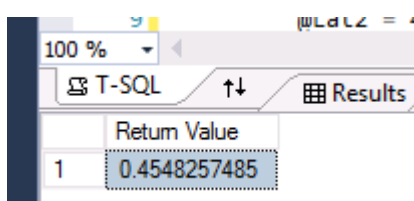


The *fnCalculateDistance* calculates the straight line distance between two points in miles.

6. In the *Server Explorer* pane, **right click** the *fnCalculateDistance* function again and select **Execute**. The *Execute Function* dialog will open.
7. **Enter** the parameters shown below. Click **OK**



The results panel should show a distance of approximately *.454 miles*. Note that this execute dialog generates the appropriate T-SQL to call the user-defined function. We can also call this UDF inline from within a SQL query.



8. Open a **SQL query** window and **Execute** the following:

```
SELECT dbo.fnCalculateDistance(40.748653, -73.98452, 40.753902, -73.979355)
as direct_distance
```

As you can see below, the function returns the calculated distance of 0.451 miles between these two points.

9. **Copy & paste** this query into the query window and **Execute the query**

```
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
```

Note the use of the *fnCalculateDistance* function within the query.

10. Open **Lesson3.R** from the *Solution Explorer*. If you have closed this project browse to C:\Lab\_Content\myRProject. If you cleared variables run `source("Lesson1.R")`.
11. Open your R Script and execute the following R code by highlighting and pressing CTRL + ENTER or copy pasting from the R Script into the Interactive R Console (for instructions how to open your R Script go to Lesson 1)

```
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 sample
tablesample (1 percent) repeatable (98052) "
```

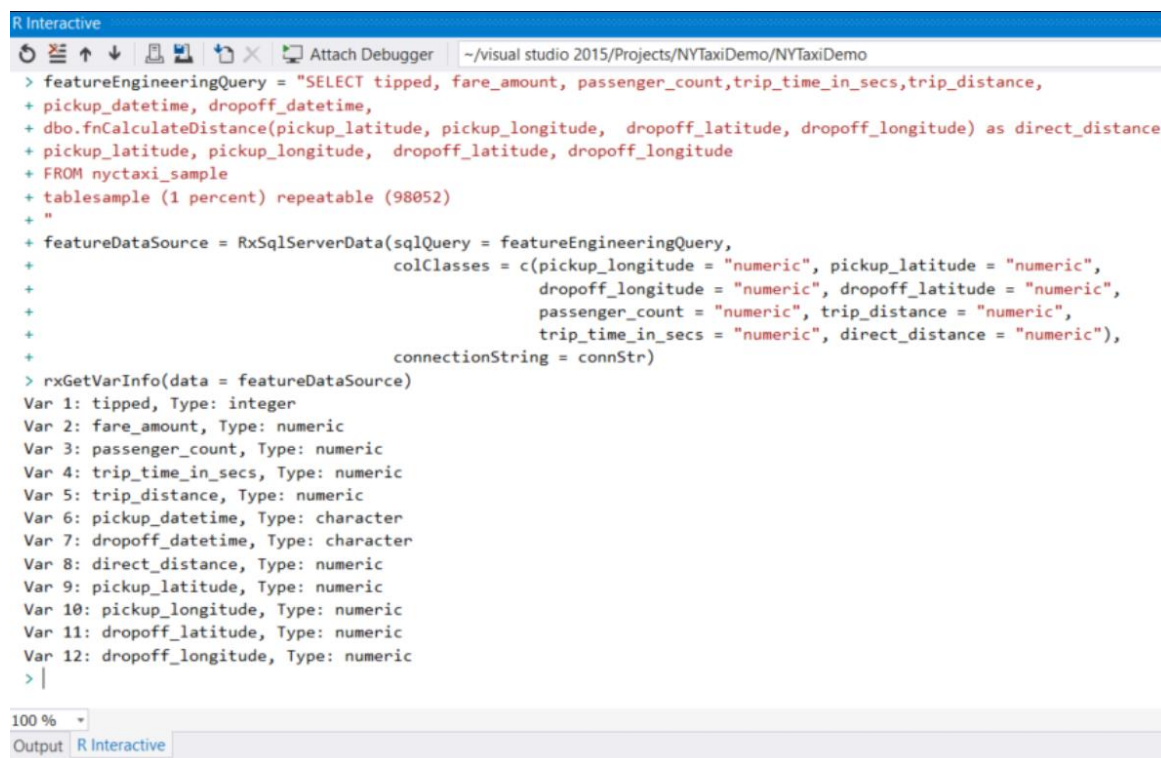
```
featureDataSource = RxSqlServerData(sqlQuery = featureEngineeringQuery,
colClasses = c(pickup_longitude = "numeric", pickup_latitude =
"numeric",
dropoff_longitude = "numeric", dropoff_latitude = "numeric",
passenger_count = "numeric", trip_distance = "numeric",
trip time in secs = "numeric", direct_distance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

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

Once the query is defined in R, you can use it for your R analysis and call on the previously defined custom T-SQL function *fnCalculateDistance* directly in the database to create the additional column, which in this case is labelled as direct distance in the same query. The *RxSqlServerData* object uses the query previously defined including the new column, as `sqlQuery` parameter input, allowing the *rxGetVarInfo* instruction to run directly on the query defined.



As a result of running the above R code, you should see a list of the variables in your feature engineering query in the Interactive R console, including the direct distance column you have created with the custom SQL function, as shown in the screenshot below.



```
R Interactive
~/visual studio 2015/Projects/NYTaxiDemo/NYTaxiDemo

> 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_sample
+ tablesample (1 percent) repeatable (98052)
+ "
+ 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)
> rxGetVarInfo(data = featureDataSource)
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: direct_distance, Type: numeric
Var 9: pickup_latitude, Type: numeric
Var 10: pickup_longitude, Type: numeric
Var 11: dropoff_latitude, Type: numeric
Var 12: dropoff_longitude, Type: numeric
> |
```

As shown above, the output of the `rxGetVarInfo` function includes a list of the variables with the different data types of your variables in R. This information can be very useful for your model training and data preparation analysis. Take note of the different data types as you will use this information later in this course.

Note that for this exercise the `SELECT` statement uses a table sample to reduce the number of rows in the query. This will make the training more efficient.

### 2.3.3 Preparing Data Using R

R is known for its rich and diverse set of statistical libraries, but, once they've been working with it for a while most data scientists will find that they come across a couple of key challenges when working with larger data sets. The first is that R works with data in memory; often you will find yourself in a situation where the data set does not fit into the Ram on the machine. The second related challenge is that R was designed from the outset as a single threaded execution engine; once again, with a large data set, it can be very frustrating to wait for some data manipulation to finish while the machine is only using one of the CPU 8 cores available. The solution is to chunk the dataset up and process it in batches, this solves for the memory problem and in most cases it's possible to parallelize these batches over multiple cores.

Implementing an approach for dealing with large data sets in R is very challenging. One of the key capabilities of SQL Server R Services (and Microsoft R Services more generally) is that it provides a set of libraries for automating this parallelized batch processing of data.

In this lesson you will learn how to use the `RxDataStep`.

You will start by defining a custom function in R. The R language provides various ways to extend its functionality with user defined functions. In fact, many of the libraries in R started as user-defined functions. After completing this training, perhaps you can contribute to the R community by writing your own R libraries and publishing through various forums or archives.

R Services provides a clear way to link user defined functions as R environment variables, both in the local (workstation) context and in the SQL Server context. In this section, you will define your user defined function using an R Services environment variable. Once defined, this will become a function that you can submit to the SQL Server R Services in-database engine to process large sets of data, very near the database to maximize the performance and scale of the data that can be transformed.

Once the function is defined, you will define an R Services data source using the Microsoft ScaleR *SqlServerData*, which you previously learned how to use in Lessons 1 and 2 of this course.

Once the data source and the R function are defined, you will use the Microsoft ScaleR *RxData* function to apply your function to the data in your SQL query and store the results in an SQL Server table.

As discovered in earlier lessons, the first challenge in analyzing a large data set is loading and exploring it to see what variables it contains, identify its data types, compute basic summary statistics and get an estimate of data outliers, missing values and possible errors. This has historically been a challenge for R users because the open source R engine has limited scalability and memory limitations. It is also a challenge for SQL developers because it can be difficult to predict what requirements specific to the R data analysis are needed to help the R specialist clean and prepare for predictive modelling. The exploration phase of predictive modelling in R is generally an iterative process, which does not make things any easier for traditional data warehouse and ETL tools.

Microsoft R Services achieves far greater data scale than R by overcoming R's inherent memory limits. It does this by acting on chunks of data in parallel, and can do this work inside of SQL Server. In this way, R Server users can tackle far larger data sets, more quickly, than users of open source R, but still use the R skills they have developed.

In this lesson, we will use the *RxDataStep* function from the *RevoScaleR* package to process and transform the source data. *RxDataStep* allows us to define a set of transformations to be applied to rows of data and then manages the parallel execution of those transformations across batches of data.

We will perform a very simple transformation and calculate the speed over ground from the distance that we have already calculated in T-SQL. *RxDataStep* allows us to provide transformations inline, but, in our case we'll go ahead and define our own reusable function. This ability to call functions allows *RxDataStep* to be used with most R functions from the rich corpus of R packages available.

You will see the following parameters in the call to *RxDataStep* function:

- *inData*: Is an R Services data source. In this example, created using an SQL query and *RxSqlServerData*
- *outFile*: Is an R Services data target. In this example, a table in SQL Server named "features"
- *varsToKeep*: a vector of variable names to keep in the data set (which columns to keep from the query into the table)

- *transforms*: a list of expressions for creating the new columns. In this example, a call to the user defined function in R that will calculate the additional column.
- *transformEnvir*: The R environment where the function or transformation will be executed. In this example, in R environment in the SQL Server compute context.
- *rowsPerRead*: The number of rows of the datasource to be read or transformed. In this example, equal to 500k.

For more details on using Microsoft R Server to transform large data sets see this article <https://msdn.microsoft.com/en-us/microsoft-r/scaler-user-guide-data-transform>.

## ☑ Student activity

1. If it is not already open, **open** the *myRproject.sln* solution.
2. **Open** the *Lesson3.R* script file.
3. **Execute** this line of R code to *source* the Lesson1.R script file.  
This will re-run the Lesson1.R script and re-initialize various R variables that we'll need in this lab such as the *Compute Context* and so forth. It may take a minute or so to execute.

```
source("Lesson1.R")
```

4. **Execute** the R script code that creates the new SQL query, crates the new data source and then runs *RxGetVarinfo* on that data source. *Approximately lines 18 to 36* of the *Lesson3.R* file. You may like to review the code comments to refresh your memory on how the *RxDataSource* works.
5. Now we'll create our custom *computeSpeed* function.  
**Execute** the following R code to define a new environment variable.

```
env <- new.env()
```

In R, an environment is a construct for defining a boundary for variable scoping. It is a bit different to the *Runtime Environment* concept that you may be familiar with from .NET or Java. For more on Environments see this excellent blog post from Hadley Wickham <http://adv-r.had.co.nz/Environments.html>

6. **Execute** the following R code to define the transformation function within the new environment.

```
env$computeSpeed <- function(trip_distance, trip_time_in_secs) {
 trip_time_in_hr <- trip_time_in_secs / 3600
 speed <- trip_distance / trip_time_in_hr
 return(speed)
}
```

Functions in R are themselves just variables. This is very important because it is going to allow Microsoft R Server to pass our Environment up to Microsoft SQL Server at execution time. You can see for yourself by typing *env\$computeSpeed* and noting that the function definition is returned to the *R Interactive* window.

7. **Execute** the following code to define the R Services data source you will use to store the results of your data preparation. In this case an SQL table called "features."

```
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)
```

8. You will now apply the transformation function to the data using the *rxDataStep*.  
**Execute** this code

```
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(speed = computeSpeed(trip_distance,
trip_time_in_secs)),
 transformEnvir = env, rowsPerRead = 500, reportProgress = 3)
```

You will see detailed information about the execution of the *rxDataStep* operation output to the *R Interactive* window. Note how multiple threads were used for execution and how the execution has been broken into multiple 'Blocks'.

```
RSVRSQ16RTM [11256]: ReadNum=199, CurrentBlockNum=199, CurrentNumRows=500, TotalRowsProcessed=99500, ReadTi
RSVRSQ16RTM [11256]: ReadNum=200, CurrentBlockNum=200, CurrentNumRows=500, TotalRowsProcessed=100000, ReadTi

Overall compute summaries time (RSVRSQ16RTM [11256]): 33.171 secs.
Total loop time: 33.109
Total read time for 200 reads: 6.54
Total transform data time: 13.06
Total process data time: Less than .001 seconds
Average read time per read: 0.0327
Average data transform time per read: 0.0653
Average process data time per read: Less than .001 seconds
Number of threads used: 4
===== RSVRSQ16RTM (process 1) has completed run at 2016-08-30 12:58:55.56 =====
>
```

9. **Execute** *rxGetVarInfo* to get a the metadata of the table that was just populated.

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

The result should look like the screenshot below:

```
> rxGetVarInfo(data = featureDataSource)
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: speed, Type: numeric
```

10. Verify the feature has been created by exploring the *Tables* folder in *Server Explorer*. **Right click** the *features* table and choose **Show Table Data**.

## 2.3.4 Creating a new SQL Server Table from Local R context

As a data scientist you'll often encounter situations where you need to quickly pull in data from another source. In this exercise you'll see how to use *RxDataStep* to import and transform a file into SQL Server. The same approach of chunking and processing files that we saw used above can be useful for importing very large files.

For this lesson, you will use a new data set, the Airline delay dataset. This dataset is publicly available and is used in various product samples for SQL Server R Services published with SQL Server 2016. The data files are available in the same directory as the other R Scripts used in this course.

The data is in XDF format; this is the eXternal Data Frame format that is native to RevoScaleR. You will see below that it has several benefits over the common CSV format.

### ☑ Student activity

1. If it is not already open, **open** the *myRProject* solution from *C:\Lab\_Content\MyRProject* and *Lesson3.R* script
2. **Execute** this code to set the compute context to the local workstation. No output will be produced in the Interactive R window.

```
rxSetComputeContext("local")
```

3. **Execute** this code to call the *RxXdfData* function and specify the path to the data file, assigning it to an R variable. This creates a pointer to the file without actually reading it into memory. You can confirm this by running *object.size(xdfAirDemo)*.

```
xdfAirDemo <- RxXdfData(file.path(rxGetOption("sampleDataDir"),
"AirlineDemoSmall.xdf"))
```

4. **Execute** the *rxGetVarInfo* function to view the metadata of this dataset. You should see this output in the Interactive R Window..

```
rxGetVarInfo(xdfAirDemo)
```

```

R Interactive
~/visual studio 2015/Projects/rproject2/rproject2

> rxSetComputeContext("local")
+ xdfAirDemo <- RxXdfData(file.path(rxGetOption("sampleDataDir"), "AirlineDemoSmall.xdf"))
> rxGetVarInfo(xdfAirDemo)
Var 1: ArrDelay, Type: integer, Low/High: (-86, 1490)
Var 2: CRSDepTime, Type: numeric, Storage: float32, Low/High: (0.0167, 23.9833)
Var 3: DayOfWeek
 7 factor levels: Monday Tuesday Wednesday Thursday Friday Saturday Sunday
> |

```

Note that this is returning metadata about the XDF file; that metadata is stored in the file itself; i.e. it R does not load the data. Contrast this with running `rxSummary(~.,xdfAirDemo)` which you will see does parse the whole file. You can also run `rxImport(xdfAirDemo)` to read and output all of the rows to the *R interactive* window. The key benefit of using a pointer based approach is that we will be able to read the eXternal Data Frame format in chunks thereby allowing us to read and process files that are too large to fit into memory.

- Now let's move this data into a SQL Server table. **Execute** this code to define the destination by calling the `RxSqlServerData` function. No output will be produced.

```
sqlServerAirDemo <- RxSqlServerData(table = "AirDemoSmallTest",
connectionString = connStr)
```

- Execute** this code to check and drop the table if it already exists.

```
if (rxSqlServerTableExists("AirDemoSmallTest", connectionString =
connStr))

 rxSqlServerDropTable("AirDemoSmallTest", connectionString = connStr)
```

- Execute** the `rxDataStep` function to read and copy the data, chunk by chunk into SQL Server.

```
rxDataStep(inData = xdfAirDemo, outFile = sqlServerAirDemo,
overwrite = TRUE, transformEnvir = env)
```

You'll see an output something like this.

```

R Interactive
Total Rows written: 200000, Total time: 8.007
Total Rows written: 300000, Total time: 11.632
Total Rows written: 400000, Total time: 15.397
Total Rows written: 500000, Total time: 19.132
Total Rows written: 600000, Total time: 22.855
, Total Chunk Time: 22.948 seconds
>

```

- Use **Server Explorer** in Visual Studio to check that the table has been created. You may also like to execute a `SELECT COUNT(*)` T-SQL query

If you'd like an extra challenge to test your learning, try reversing the process and reading data from SQL Server out to a local \*.xdf file using `rxDataStep`.

## 2.4 Lesson 4: Building and Saving the Model to SQL Server

In this lesson you will use a common predictive model algorithm known as logistic regression to predict whether a taxi driver in NYC is likely to get a tip on a particular journey. You will do this using the NYC Taxi Dataset loaded and prepared in the previous lessons.

**Total time to complete this lesson is 30 minutes**

One of the key advantages of SQL R Services is the ability to scale for large data, overcome R's inherent performance and scale limitations, and keep analytics close to the data. The RevoScaleR package that is included with SQL Server R Services includes several machine learning algorithms implemented as Parallel External Memory Algorithms (PEMAs) which are able to deal with very large datasets in a similar fashion to the use of *rxDataStep* in the previous lesson.

For the sake of time in the lab we are using a very simple (logistic regression) model and we are not showing techniques such as hold-out or cross validation. In a real predictive modelling scenario, you may want to do additional exploratory analysis to find relevant features, divide your data set into both training and testing data, and test a couple of different methods for evaluating the results of your model before deploying to production.

After "fitting" your logistic regression model, you will evaluate your model using the *rxRocCurve* function to plot the Receiver Operating Characteristic curve. The use of the ROC curve is a common way of visualizing the sensitivity and recall of a predictive model. The metric often referred to when assessing the predictive power of a model is the AUC or area under the ROC curve. For more on this we'd suggest taking a look at [https://en.wikipedia.org/wiki/Receiver\\_operating\\_characteristic](https://en.wikipedia.org/wiki/Receiver_operating_characteristic)

Once you have tested your model, you will deploy the model using SQL Server R Services so that other apps can invoke the model by calling a T-SQL stored procedure. Operationalizing models by deploying into SQL Server has several advantages:

- Data need not be moved outside the database server, speeding both model training and execution.
- Modeling on data in the database enables R and SQL users to act on the same datasets.
- SQL can be used to run R models once they are deployed, providing access via means other than R. Most application developers are very familiar with calling SQL Server procedures.

Part of making models callable from external applications requires us to save the model to the database. In this lesson, we'll serialize the model object to a hexadecimal text string, and save that string to the database in a *varbinary(max)* column. Once the model is saved to the database, it can be called retrieved by the stored procedure and then used to predict novel data.

### 2.4.1 Create a Classification Model using rxLogit

In this section, you will "fit" a logistic regression model. This model predicts whether the taxi driver is likely to receive a tip on a particular journey. A logistic regression model predicts this from one or more metrics (passenger count, trip distance, etc.). We won't delve into the specific details of logistic regression except to say that it is one of the simpler models available in the RevoScaleR package and therefore is relatively fast to fit/train. See this link for more information [https://en.wikipedia.org/wiki/Logistic\\_regression](https://en.wikipedia.org/wiki/Logistic_regression)



Here are the columns you will use in the model:

- passenger\_count
- trip\_distance
- trip\_time\_in\_secs
- direct\_distance

From these, we will use the parallelized logistic regression model function in SQL Server R Services called *rxLogit* to fit a model to predict probability of a tip.

## ☑ Student activity

1. If it is not already open, **open** the *Lesson4.R* script from the *myRProject* solution located at *C:\Lab\_Content\MyRProject*
2. This script is dependent on *Lesson1.R* and *Lesson3.R*. If you need to re-run these you can **execute** the two source lines.

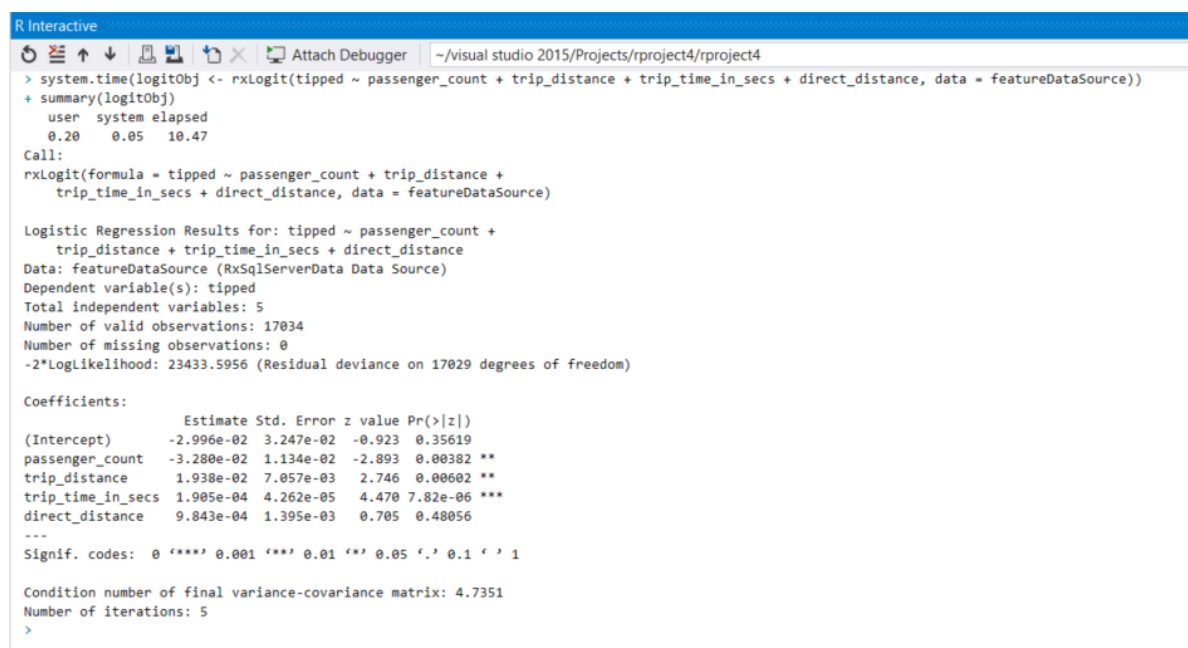
```
source("Lesson1.R")
source("Lesson3.R")
```

3. **Execute** the lines of code to reset the compute context to the remote SQL Server
4. Now we'll fit the model., **Execute** the call to *rxLogit* and then print the *summary* of the fitted model.

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

summary(modelFit)
```

As a result of executing the summary on the logistic regression model *logitObj*, you should see a summary of your model in the R Interactive window as below. Again., if you'd like to understand some of these details we'd suggest taking a look at the Wikipedia article noted above.



```
R Interactive
~/visual studio 2015/Projects/rproject4/rproject4
> system.time(logitObj <- rxLogit(tipped ~ passenger_count + trip_distance + trip_time_in_secs + direct_distance, data = featureDataSource))
+ summary(logitObj)
 user system elapsed
 0.20 0.05 10.47

Call:
rxLogit(formula = tipped ~ passenger_count + trip_distance +
 trip_time_in_secs + direct_distance, data = featureDataSource)

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: 17034
Number of missing observations: 0
-2*LogLikelihood: 23433.5956 (Residual deviance on 17029 degrees of freedom)

Coefficients:
 Estimate Std. Error z value Pr(>|z|)
(Intercept) -2.996e-02 3.247e-02 -0.923 0.35619
passenger_count -3.280e-02 1.134e-02 -2.893 0.00382 **
trip_distance 1.938e-02 7.057e-03 2.746 0.00602 **
trip_time_in_secs 1.905e-04 4.262e-05 4.470 7.82e-06 ***
direct_distance 9.843e-04 1.395e-03 0.705 0.48056

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Condition number of final variance-covariance matrix: 4.7351
Number of iterations: 5
>
```



That is all you need to train and inspect this Model in R. Just two lines of code.

If you're curious how what is influencing the tip, have a look at the "Pr(>|z|)" or P-values in the last column of the results returned. P-values indicate the effect each input variable or feature contributes to predicting whether the driver was tipped or not. (Note: a smaller value (closer to 0) will have more statistical significance and a larger p-value shows less significance.) As you can see, passenger count and trip time have a good level of statistical significance, that is, can help better predict if the driver will be tipped or not.

## 2.4.2 Use the Model and Store the Results

Now that the model is built, you can use it to predict whether the driver is likely to get a tip on a particular drive. To run your model, you'll use the model object created with a second ScaleR function called *rxPredict*. *rxPredict* is similar to the R predict function but scores data using models built with other functions in the ScaleR library shipped with R Services and as such can be parallelized.

When our script runs *rxPredict* to execute a model object, we'll save the predictions to the table *taxiScoreOutput*. Notice that the schema for this table is not defined *a priori* when you reference it using *rxSqlServerData* but instead is implicitly created from the *scoredOutput* object output from *rxPredict*.

To create the table that stores the predicted values, the SQL login running the *rxSqlServerData* function must have DDL privileges in the SQL Server database.

R Services uses external processes to run R Scripts on behalf of the database engine. This architecture enables isolation of the database engine from the R engine to assure stability. By isolating R and SQL, R users can be given more latitude in which scripts and packages they run without putting the database engine at risk of instability. However, these additional processes may generate error conditions. To provide additional debugging capability, the events generated in the external processes can be captured by following the steps described [here](#).

### ☒ Student activity

1. Execute this code to define an R Services data object for storing the prediction results, and assign it to an R variable..

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

17. **Execute** this code to use the model in R to predict whether a driver will be tipped or not, and write the prediction results back to a SQL Server table.

```
If (rxSqlServerTableExists("taxiScoreOutput", connectionString = connStr))
 rxSqlServerDropTable("taxiScoreOutput", connectionString = connStr)

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

18. *rxPredict* does not output the predictions as they are made. These have been saved into the *taxiScoreOutput* table. Use *Server Explorer* to **Query** the contents of the *taxiScoreOutput* table..

The screenshot shows the SQL Server Enterprise Manager interface. On the left, the 'Server Explorer' pane displays the 'Data Connections' tree, with 'rsvrsql16rtm.NYCTaxiData.dl' expanded to show 'Tables'. The 'taxiScoreOutput' table is selected. The main pane shows a SQL query: 'Select \* from taxiScoreOutput'. Below the query, the 'Results' tab displays the data in a table format.

|   | Score             | tipped | passenger_count | trip_distance | trip_time_in_secs | direct_distance |
|---|-------------------|--------|-----------------|---------------|-------------------|-----------------|
| 1 | 0.506159512942795 | 0      | 1               | 1             | 352               | 0.9729186983    |
| 2 | 0.506530323954642 | 0      | 1               | 1             | 360               | 0.9317790954    |
| 3 | 0.509075065305553 | 0      | 1               | 1             | 414               | 0.8247131972    |
| 4 | 0.506545544191084 | 0      | 1               | 1             | 360               | 0.9936383744    |

### 2.4.3 Generate Plots of Model Accuracy

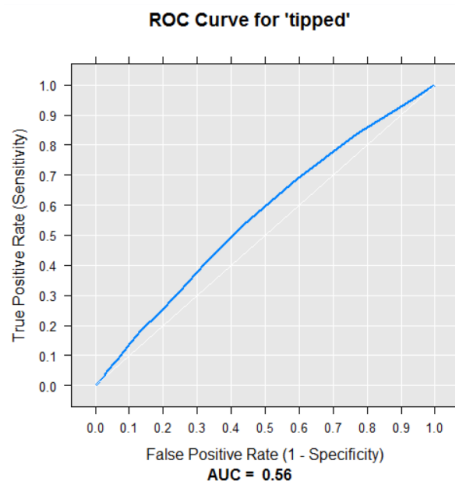
Now we will use an analysis of the Receiver Operating Curve to determine how well our logistic regression model was able to fit to the data. This is a common approach for assessing the predictive performance of binary classification model such as the one that we are using. Microsoft R Server provides native support for creation of ROC curves via the *rtxRocCurve* function.

#### ☒ Student activity

1. **Execute** the *rxRocCurve* function, using the *scoredOutput* database object as input.

```
plot ROC curve
rxRocCurve("tipped", "Score", scoredOutput)
```

2. In RTVS, you'll be able to **view** the plot in the *R Plot Window* as shown below:



The AUC (area under the curve) value indicates that the model is not very effective at being able to predict the tipped value yet. A value of .50 indicates some predictive power. Again, for the convenience of our lab environment this is OK, but, you may like to experiment with;

- a) Including additional features into the model. For example, do you think that things such as the time of day or day of the week will have an influence on whether the passenger tips?
- b) Fitting an alternative model. The *rxLogit* model is a linear model, you by like to try some non-linear models, for example try fitting an *rxBTree* model to the data.

## 2.4.4 Save the Model to the Database for Use in Scoring

Now that you have built and evaluated a model, you must save it for subsequent use in the database. To do so, you must first convert the model object to a text format that can be saved to the database easily.

In order to persist our model for future use we will;

- Serialize the model into a hexadecimal string – this function is included in the R base package.
- Transmit the serialized object to the database.
- Save the model in a *varbinary(max)* column.

### ☒ Student activity

1. **Execute** this code to serialize the model into a hexadecimal string. Then, use paste as shown to remove spaces.

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

2. **Execute** this code to open an ODBC connection and call a stored procedure called *PersistModel* to store the binary representation of the model in the database. Note that we are using the *RODBC* package to call the stored procedure 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 the database is simply INSERTing the serialized model into a table. For convenience we've already created a stored procedure to wrap this insert. You may like to view the code for the *PersistModel* stored procedure using **Server Explorer**

## 2.4.5 Summary

We've now "trained" a logistic regression model that predicts the likelihood of a tip, we've characterized it using the receiver operating characteristic curve to see that it is producing improvements over random, and we've stored a serialized version of the predictive model to the database for use in lesson 5.

This concludes Lesson 4. Proceed to lesson 5 to use your predictive model to score trips individually.

## 2.5 Lesson 5: Using the Predictive Model to Score Data in SQL Server

### 2.5.1 Overview and Context

Deploying machine learning models into production can be challenging. By using SQL Server R Services we are able to expose our newly built machine learning model to users via T-SQL. In this lesson you will use R and SQL to operationalize your predictive model for use via T\*SQL stored procedures.

**Total time to complete this lesson is 30 minutes**

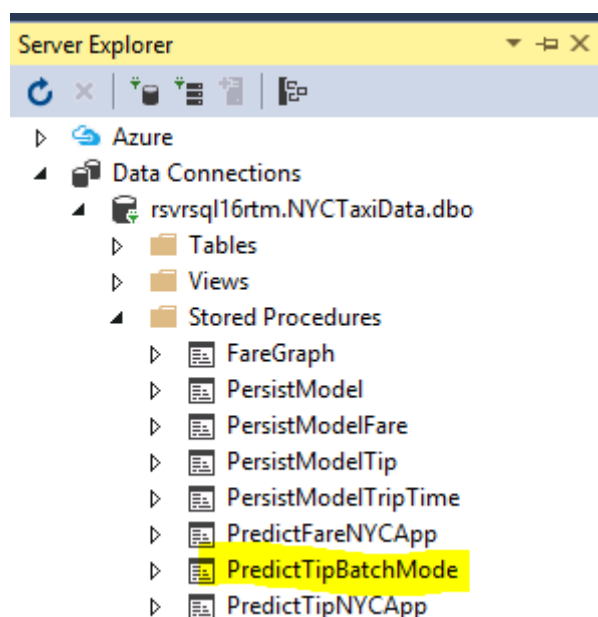
### 2.5.2 How to Deploy R Models in SQL Server

Once serialized and persisted to the database, your predictive model built in lesson 4 can be used to generate predictions by running a stored procedure that uses that saved model with new data inputs. SQL Server provides a system stored procedure, `sp_execute_external_script`, that allows us to execute an R Script and we'll use that to predict new data with the model.

When using `sp_execute_external_script` it is important to understand how data type mapping between T-SQL and R occurs. There is no automatic data type conversion in the parameter call to the system-stored procedure. Any query that contains valid data will be executed. You must explicitly CAST or CONVERT your data before submitting it to the R Script; the lab Appendix includes information on data type differences between R and SQL.

#### ☒ Student activity

1. **Open** Visual Studio Server Explorer and the *NYCTaxiData* database. **Expand** *Stored Procedures*, **find** the *PredictTipBatchMode*, **right click** and **open** it.



You will see the *PredictTipBatchMode* looks something like the following.

```
CREATE PROCEDURE [dbo].[PredictTipBatchMode] @inquiry 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 = @inquiry,
 @params = N'@model varbinary(max)',
 @model = @lmodel2

 WITH RESULT SETS ((Score float));
END
```

2. **Identify** the following in the *PredictTipBatchMode* code above:

- The *SELECT* statement that retrieves the trained model from the database
- The call to *EXEC sp\_execute\_external\_script* and the various parameters
  - The *@language* is R. At this time R is the only supported language.
  - The *@script* value is passed in inline; you may well want to retrieve the script code from a database table in a production use case.
  - The *@input\_data\_1* dataset. This is a T-SQL query and is exposed with the R execution environment as an R *data.frame*. For very large input datasets the external script mechanism supports streaming of batches of rows.
  - The *@params* parameter allows us to pass in one or more parameters, in this case we're passing in the model.
- The *OutputDataSet* which is returned to T-SQL on completion of the procedure

The full details on the *sp\_execute\_external\_script* procedure can be found here.

<https://msdn.microsoft.com/en-US/library/mt604368.aspx>

## 2.5.3 Score Multiple Rows as a Batch using R Directly

### ☒ Student activity

Execute the following R code in Interactive R console in order to predict the tip with stored procedure in a batch of rows.

1. If it is not already open, **open** the *Lesson5.R* script from the *myRProject* solution
2. This script is dependent on *Lesson1.R*, *Lesson3.R* and *Lesson4.R* scripts. If you need to re-run these you can **execute** the three source lines.

```
source("Lesson1.R")
source("Lesson3.R")
source("Lesson4.R")
```

1. **Execute** the code below by highlighting and pressing **CTL+ENTER**

```
predict with 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 @inquiry = ", input, sep="")
sqlQuery (conn, q)
```

The screenshot displays the Microsoft Visual Studio interface with the RStudio package installed. The main editor window shows an R script titled 'RSQL\_R\_Walkthrough.R'. The script contains comments and a SQL query. The comments explain that two stored procedures are available for prediction and that the following query selects the top 10 observations not in the training set. The SQL query is a complex join between a table 'nyctaxi\_sample' and a function 'dbo.fnCalculateDistance'. The R Interactive window at the bottom shows the execution of the query, resulting in a table with 10 rows and two columns: 'id' and 'Score'.

```
We have already provided and installed two stored procs to call for prediction on this model - PredictTipBatchMode
predict with stored procedure in batch mode. Take a few records that are not part of training data
NOTE: You need to generate the distance feature when you extract the records to send for prediction in
The following query selects the top 10 observations that are not in training set.
This query is parsed as an input parameter to a stored procedure PredictTipBatchMode to make prediction
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 dir
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
)"

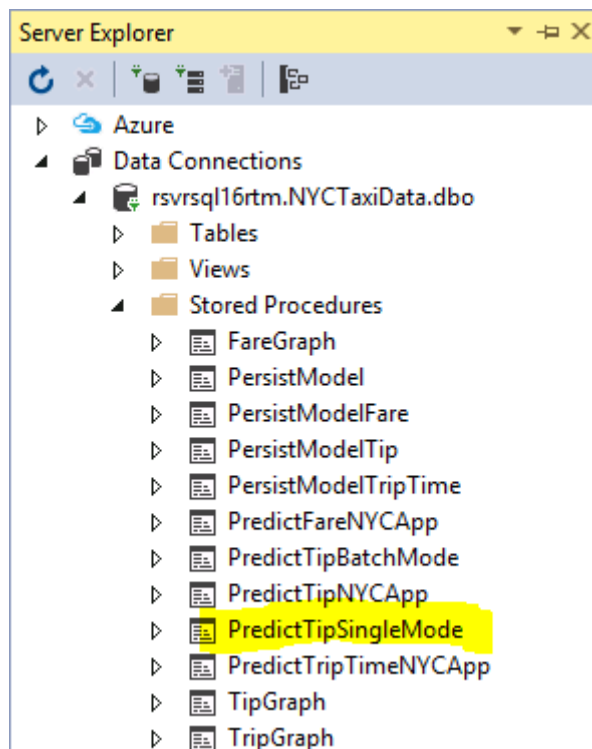
> q<-paste("EXEC PredictTipBatchMode @inquiry = ", input, sep="")
> sqlQuery (conn, q)
 Score
1 0.5043645
2 0.5042583
3 0.5070562
4 0.5068501
5 0.5030014
6 0.5051313
7 0.5032077
8 0.5076051
9 0.5050473
10 0.5079361
```

As explained in the previous section, the input data query used for scoring is passed into the stored procedure. You can define any query that contains valid input data for this particular model.

## 2.5.4 Score Single Rows by Building a Stored Procedure

### ✓ Student activity

1. **Open** the Stored Procedure *PredictTipSingleMode*. Review the code as you did for the batch procedure.



**Note: Do NOT create this function. Reference Only**

```
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)
```

2. **Open** *Lesson5.R* and **Execute** the code below by highlighting and pressing **CTL+ENTER**.

```
predict with stored procedure in single mode
q = "EXEC PredictTipSingleMode 1, 2.5, 631, 40.763958,-73.973373,
40.782139,-73.977303 "
sqlQuery (conn, q)
```

The screenshot shows the Microsoft Visual Studio interface with the R console open. The code in the editor includes a SQL query to predict a tip based on medallion, hack\_license, and pickup\_datetime. The R console output shows the execution of the code, including the SQL query and the resulting tip score for a single observation.

```

)
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 @inquiry = ", input, sep="")
sqlQuery (conn, q)

Call predict on a single observation
q = "EXEC PredictTipSingleMode 1, 2.5, 631, 40.763958,-73.973373, 40.782139,-73.977303 "
sqlQuery (conn, q)

```

R Interactive

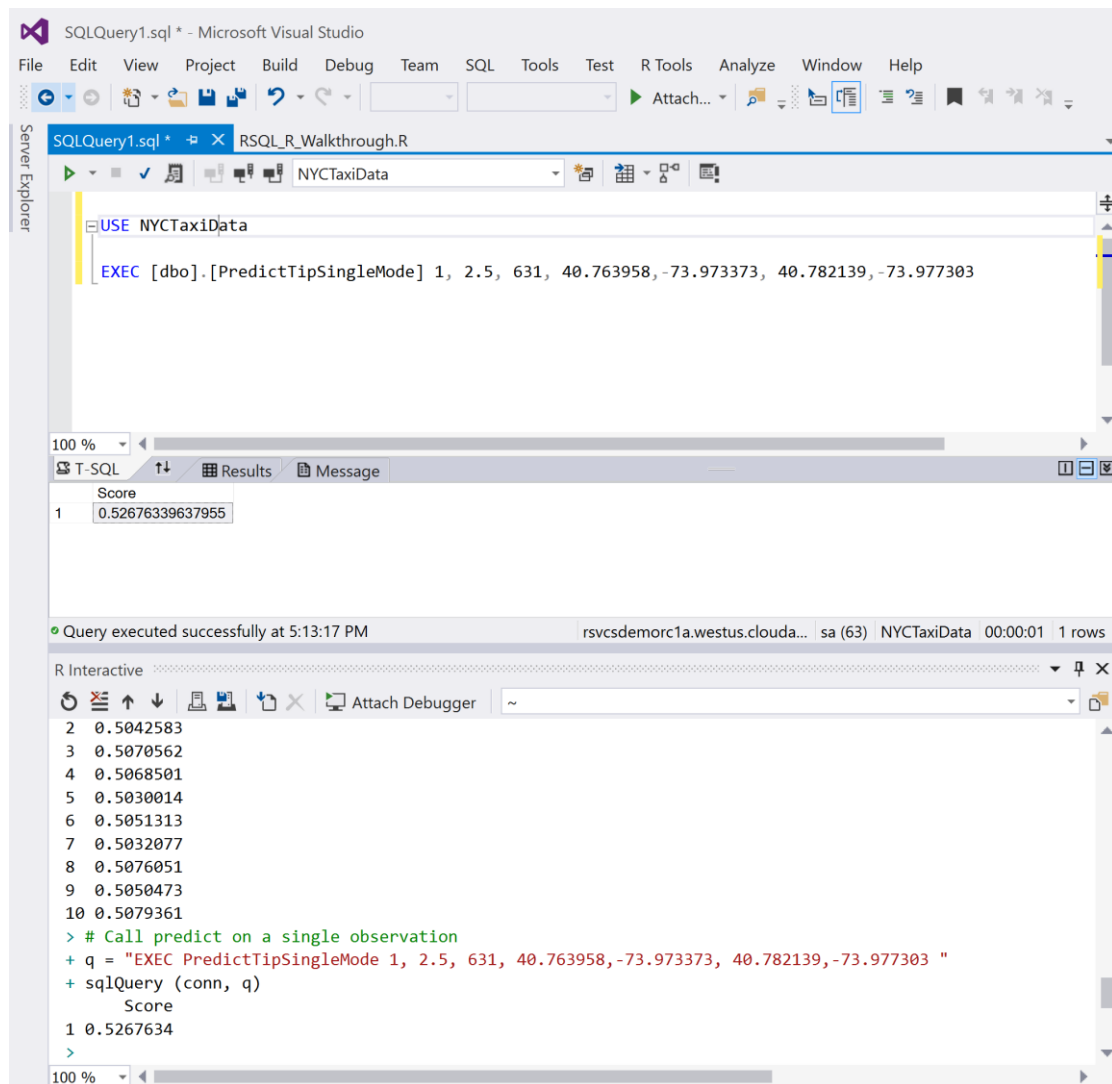
```

2 0.5042583
3 0.5070562
4 0.5068501
5 0.5030014
6 0.5051313
7 0.5032077
8 0.5076051
9 0.5050473
10 0.5079361
> # Call predict on a single observation
+ q = "EXEC PredictTipSingleMode 1, 2.5, 631, 40.763958,-73.973373, 40.782139,-73.977303 "
+ sqlQuery (conn, q)
 Score
1 0.5267634
>

```

3. The stored procedure can also be executed with T-SQL. For example, **open** a new *SQL Query* in Visual Studio and run the following T-SQL script.

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



## 2.5.5 Using SQL Server R Services with other Microsoft R products

In this Lab you have worked with Microsoft SQL Server R Services in the context of SQL Server 2016. R Services is part of a family of products that also includes Microsoft R Server. R Server provides many of the same capabilities for large-scale data analytics on platforms including Linux, Hadoop, Spark, and Teradata database EDWs.

R Server can be used to build models for use elsewhere, and can be used to score models that were created elsewhere, greatly expanding the range of options available to systems designers.

This completes Lesson 5 of the lab. You should now have the basis for combining R and T-SQL to build predictive models and use them in your own applications.

## 2.5.6 Appendix

---

For additional information about the products and services used in this Lab, please see the references provided below.

### **Lab 1: Lesson 1: Loading Packages and Exploring Data**

Bulk import and export in SQL Server

<https://msdn.microsoft.com/en-us/library/ms175937.aspx>

Packages installed with Microsoft R Open

<https://mran.microsoft.com/rro/installed/>

How to install additional packages in SQL Server

<https://msdn.microsoft.com/en-us/library/mt591989.aspx>

Data collection for this lab and how to access the full data set

[http://chriswhong.com/open-data/foil\\_nyc\\_taxi/](http://chriswhong.com/open-data/foil_nyc_taxi/)

How to create workflows using R and SQL Server

<https://msdn.microsoft.com/en-us/library/mt590871.aspx>

Complete reference of BCP utility

<https://msdn.microsoft.com/en-us/library/ms162802.aspx>

FAQ for Microsoft R open to further assist with troubleshooting R Package installations

<https://mran.microsoft.com/faq/>

### **Lab 1: Lesson 2: Exploring R Tools for Visual Studio**

Installing Microsoft R Client

<https://msdn.microsoft.com/en-us/microsoft-r/install-r-client-windows>

RTVS

<https://www.visualstudio.com/en-us/features/rtvs-vs.aspx>

R and T-SQL Server data type conversion

<https://msdn.microsoft.com/en-us/library/mt590948.aspx>

Visual Studio

<https://www.visualstudio.com/en-us/visual-studio-homepage-vs.aspx>

R Tools for Visual Studio and a FAQ on the new versions

<http://microsoft.github.io/RTVS-docs/>

How to attach a debugger to RTVS

<https://microsoft.github.io/RTVS-docs/debugging.html>

ggplot2

<https://mran.microsoft.com/package/ggplot2/>

plotly

<https://plot.ly/>

### **Lab 1: Lesson 3: Preparing data using T-SQL and R Services**

Haversine formula

[https://en.wikipedia.org/wiki/Haversine\\_formula](https://en.wikipedia.org/wiki/Haversine_formula)

Custom Transact-SQL functions

<https://msdn.microsoft.com/en-NZ/library/ms186755.aspx>

*RxDataStep*

<http://www.rdocumentation.org/search?q=RevoScaleR>

### **Lab 1: Lesson 4: Building and Saving the Model to SQL Server**

Model testing and validation

<https://books.google.com>

Interpreting the model parameters

<http://blog.minitab.com/blog/adventures-in-statistics/how-to-interpret-regression-analysis-results-p-values-and-coefficients>

rxPredict

<http://www.rdocumentation.org/search?q=RevoScaleR>

How to interpret a ROC Curve

<https://www.r-bloggers.com/illustrated-guide-to-roc-and-auc/>

### **Lab 1: Lesson 5: Using the Predictive Model to Score Data in SQL Server**

T-SQL supported data types

<https://msdn.microsoft.com/en-us/library/mt590948.aspx>

Microsoft R Open documentation and user forum

<https://mran.microsoft.com/open/>

R packages

<https://cran.r-project.org/web/packages/>

Microsoft R Server Scale R

[http://download.microsoft.com/download/Microsoft\\_R\\_Server\\_ScaleR](http://download.microsoft.com/download/Microsoft_R_Server_ScaleR)

How to set up R Services (In-Database) on SQL Server 2016

<https://msdn.microsoft.com/en-us/library/mt696069.aspx>

Fixing CRAN snapshots issues

<https://mran.microsoft.com/documents/rro/reproducibility/#reproducibility>

CRAN data import/export packages

<https://cran.r-project.org/doc/manuals/r-release/R-data.pdf>

Visual Studio

<https://www.visualstudio.com/en-us/visual-studio-homepage-vs.aspx>

Data Exploration and Predictive Modelling in R Services

<https://msdn.microsoft.com/en-us/library/mt590947.aspx>