12
Case Study – Regression

We will now examine a set of `S3` classes designed to explore a specific data set. We will use `S3` classes to provide a longer example of an `S3` class. The context for the exploration of the data is regression. A heavier focus is given to the routines to read the data file and extract columns from the data set. The regression class is limited and serves as a placeholder for further development.

This chapter is roughly divided into the following parts:

- The data
- The relationship class
- The regression class
- Exploring the data

The data

We will first discuss the data set. The majority of code associated with the primary class is designed to read the data file and isolate certain aspects of the data. The data set is made available as part of Vancouver, British Columbia's Open Data Project. Being able to understand the data and simplifying the process of getting access to specific parts of the data is an important step. It is not uncommon that 70 percent of the effort for a project is to understand the data, 20 percent is to transform the data to a more usable form, and the remaining 10 percent is dedicated to the relevant mathematics.

The data set examined was located at `http://data.vancouver.ca/datacatalogue/streetTrees.htm`, and the information about specific trees is given for a variety of neighborhoods. Each geographic area within Vancouver has a CSV file with information about the trees in the area. There are 19 columns in each data file, but we are only interested in a few of the columns. In particular, the columns whose names are `DIAMETER`, `DATE`, and `STREET_SIDE_NAME` are explored. The unit for the `DIAMETER` column is inches. The `DATE` is given in the form YYYYMMDD. The `STREET_SIDE_NAME` is either `EVEN`, `ODD`, `MED` (for the median), or `TBD` if the position is not known.

The Relationship class

There are two classes used to explore the data. The first is the `Relationship` class that is used to read the data and make gaining access to specific columns easier. The second class is the `Regression` class that is briefly discussed in the next section. The complete details of the classes are not given to keep our discussion more focused, and you should closely examine the code provided on the website.

The members of the `Relationship` class include a full copy of the data read from the `CSV` file, vectors for the explanatory and response variables, a list of shortcuts to make it easier to get columns from the data file, and a member from one of the regression classes described in the following section. The methods include accessor functions to set and get copies of the data members. There are also methods to read the data file and to convert a character column into a date object.

The constructor for the class is given in the following code. One set of accessor functions included here is for the `getDataColumn` and `setDataColumn` methods. These methods are not simple and are kept here to emphasize their roles in the class. The idea is that the names often used in the title header of a file (especially those generated by government entities) can be difficult to remember and are not necessarily intuitive.

The shortcut element is a list, and in this context, it can be thought of as a set of key/value pairs. The name of an object within the list is treated as a keyword. The name is used to point to a character object that is the name of the column in the original `CSV` file. The keyword can be set to a word that is easier to remember than the original column title. A character object is passed to the `getDataColumn` or `setDataColumn` command. If that character object matches one of the names from within the list, then the corresponding value associated with the name is used to determine which column from the data file to use, otherwise it is assumed to be the name of the column.

A partial listing of the class including the `getDataColumn` and `setDataColumn` methods is as follows:

```
##################################
## Define the constructor for the
## relationship class
Relationship <- function()
    {

        ## Create the list used to represent an
        ## object for this class
        me = list(
            ## reserve the data associated with the class
            data = list(),
            explanatory = character(0),
            response = character(0),
            shortcut = list(
                id="TREE_ID",             # The id of the tree
                street="STD_STREET",      # Name of the street
                side="STREET_SIDE_NAME", # Which side of street
                dia="DIAMETER",           # Diameter of tree
                date="DATE_PLANTED",      # Date tree planted
                barrier="ROOT_BARRIER",   # Y/N ?
                curb="CURB",              # Y/N at curb?
                name="COMMON_NAME"        # Tree type's name
                ),

            theFit = NULL


            )

        class(me) <- append(class(me),"Relationship")
        return(me)
    }

getDataColumn <- function(theObject,columnName)
    {
        ## Return a copy of one of the columns from the data set
        UseMethod("getDataColumn",theObject)
    }

getDataColumn.default <- function(theObject,columnName)
    {
        return(NA)
    }

getDataColumn.Relationship <- function(theObject,columnName)
    {
        ## Return a copy of one of the columns from the data set
        ##print(columnName)

        if(columnName %in% names(theObject$shortcut)) {
            ## The name passed is defined in the shortcut vector
            ## use value in the shortcut to index into the data
            return(theObject$data[[ theObject$shortcut[[columnName]]
]])

        } else if (columnName %in% names(theObject$data)) {
            ## The name itself is a valid name. Use the
            ## column with the same name that is passed.
            return(theObject$data[[ columnName]])
        }

        ## The column name could not be found. Print out an
        ## error message.
        stop(paste("Column",theObject$columnName,"is not defined"))

    }

setDataColumn <- function(theObject,columnName,newColumn)
    {
        ## Set the specified column to the given data set
        UseMethod("setDataColumn",theObject)
    }

setDataColumn.default <- function(theObject,columnName,newColumn)
    {
        ## Set the specified column to the given data set
        return(theObject)
    }

setDataColumn.Relationship <- function(theObject,columnName,newColumn)
```

```
    {
        ## Set the specified column to the given data set

        if(columnName %in% names(theObject$shortcut)) {
            ## The name passed is defined in the shortcut vector
            ## use value in the shortcut to index into the data
            theObject$data[ theObject$shortcut[[columnName]] ] <-
newColumn
        } else {
            ## Use the column with the same name that is passed.
            theObject$data[columnName] <- newColumn
        }
        return(theObject)
    }
```

There are two additional methods. The first, `readFile`, is used to read a data file and assign the value of the data element within an object from the `Relationship` class. The second, `convertDate`, is a method to aid in converting a column that contains dates into vectors of the `Date` class. We first examine the `readFile` method. It includes the `...` reserved word to pass all optional arguments to the `read.csv` command. Note that there is an additional check for the `skip` option, and a warning is printed if it is not included. I personally like to add information at the start of a file that includes details about where and when the information was obtained.

The definition for the `readFile` method is as follows:

```
readFile <- function(theObject,file, ...)
    {
        UseMethod("readFile",theObject)
    }

readFile.Relationship <- function(theObject,file, ...)
    {
        print(noquote(paste("Reading file:",file)))
        arguments <- list(...)
        if(!('skip' %in% names(arguments))) {
          ## there is no skip argument
          warning("skip is not defined. Defaulting to skip=0")
        }
        theObject <- setData(theObject,read.csv(file,...))
    }
```

The second method, `convertDate`, is more straightforward. It obtains a copy of the requested column, converts it to a `Date` class, and then reassigns the result within the list that holds the data within the object. The code for this method is as follows:

```
convertDate <- function(theObject,columnName,timeFormat)
    {
        UseMethod("convertDate",theObject)
    }

convertDate.Relationship <- function(theObject,columnName,timeFormat)
    {

        timeColumn <- getDataColumn(theObject,columnName)
        theObject  <- setDataColumn(theObject,columnName,
                                    as.Date(as.character(timeColumn),
timeFormat))
    }
```

The codes for the `Relationship` class are assumed to be kept in a separate file, `relationship.R`. Prior to creating a member of this class, the file must be executed using the source command. One member of the class, the regression variable, has not been defined, and is described in the following section.

The Regression class

The `Regression` class is used to manage the details of the regression calculations. The details to make it a more useful regression operator are omitted as our focus is to discuss how to organize the classes and provide a framework to compare the relationships between variables. Regression is a complicated subject, and a whole book could probably be written on the topic of regression alone.

The `Regression` class has three data members. The first is a vector for data for the explanatory variable. The second is a vector for data for the response variable. Finally, the third is an object returned from the `glm` command that contains information about the regression relationship. One obvious improvement to the class would be to replace the data vectors with a single object from the `formula` class. (See the help page on the formula class, `help(formula)`, for more information.) Here, we assume the simplest relationship between the two variables, which is reflected in the method used to create the `glm` object.

In this example, we have three classes derived from the `Regression` class. The three classes are used for performing logit, Poisson, and linear regression depending on the data type of the response variable. First, the base class, `Regression`, is defined. As in the previous section, the accessors are ignored for the sake of brevity and allow us to keep our discussion more focused:

```
##################################
## Define the constructor for the
## regression class
Regression <- function(theResponse=numeric(0),theExplanatory=numer
ic(0))
    {

        ## reserve the data associated with the class

        ## Create the list used to represent an
        ## object for this class
        me = list(

            ## Define the environment where this list is defined so
            ## that you can refer to it later.
            explanatory = theExplanatory,
            response    = theResponse,
            fit         = NULL
            )

        class(me) <- append(class(me),"Regression")
        return(me)
    }
```

The three derived classes, `LogitRegression`, `CountRegression`, and `ContinuousRegression`, build on the `Regression` class. In this example, they are all simple classes, but each one can be easily expanded. Only one, `LogitRegression`, is used in the examples that follow in the next section, and it is defined in the following code:

```
##################################
## Define the constructor for the
## logit regression class
LogitRegression <- function(theResponse,theExplanatory)
    {

        ## Create the list used to represent an
```

```
            ## object for this class
            me = Regression(theResponse,theExplanatory)

            class(me) <- append(class(me),"LogitRegression")
            return(me)
        }
```

One more method is required for the `Regression` class. The method is used to call the `glm` function and perform the required regression calculations. The name of the method is called `regression`, and the code is as follows:

```
    regression <- function(theObject, ...)
        {
            #print("regression")
            UseMethod("regression",theObject)
        }

    regression.LogitRegression <- function(theObject,  ...)
        {
            theObject <- setFit(theObject,
                                glm(getResponse(theObject) ~
    getExplanatory(theObject),
                                    family=binomial(),...))
            return(theObject)
        }
```

Finally, there is one item missing from the `Relationship` class described in the previous section. We need a method to create an appropriate `Regression` object and call its regression method:

```
    calcRegression <- function(theObject,...)
        {
            UseMethod("calcRegression",theObject)
        }

    calcRegression.Relationship <- function(theObject,...)
        {
            ## Method to create a regression object and set it for
            ## this object.
            print("calcRegression.Relationship")
            response    <- getResponse(theObject)
            explanatory <- getExplanatory(theObject)

            if(is.logical(response)) {
                theFit <- LogitRegression(response,explanatory,...)
            } else if (is.integer(response)){
                theFit <- CountRegression(response,explanatory,...)
            } else {
                theFit <- ContinuousRegression(response,explanatory,...)
            }
            theObject <- setFit(theObject,regression(theFit))
            return(theObject)
        }
```

Exploring the data

We will now provide a brief example using the `Relationship` and `Regression` classes. The example demonstrates how to convert the `DATE_PLANTED` column into a `Date` object and grab a copy of the resulting column. The example is also a brief demonstration of how to perform logistic regression between the date the tree was planted and the side of the street the tree is planted. The first step in the example is to execute the code that includes the class definitions and to then create an object that is a `Relationship` object:

```
> source('relationship.R')
> tree <- Relationship()
```

The next step is to read in the data file and convert the column that has the dates with their corresponding date objects:

```
> tree <- readFile(tree,"trees/StreetTrees_KensingtonCedarCottage.
csv",skip=2)
> tree <- convertDate(tree,"date","%Y%m%d")
timeDefined <- !is.na(tm)
```

The next steps are to get a copy of the columns that have the information about which side the trees are planted and the date the trees were planted. The trees whose planting dates are unknown are censored:

```
theDate <- getDataColumn(tree,"date")
side    <- getDataColumn(tree,"side")
side    <- side[!is.na(theDate)]
theDate <- theDate[!is.na(theDate)]
```

We can now perform the logistic regression and get a copy of the results:

```
tree <- setExplanatory(tree,theDate)
tree <- setResponse(tree,side=="EVEN")
tree <- calcRegression(tree)

# Get the regression and find the details of the results.
```

```
fit <- getFit(tree)
fit
summary(fit)
```

A quick check of the output indicates that there is not a clear relationship between the date a tree is planted and which side of the road it was planted. The residuals, though, are relatively large, and a look at the residuals when plotting the `glm` object shows that the assumptions for the regression are not satisfied.

Summary

An example of a set of classes to aid in performing regression on a particular data set is examined in this chapter. The set of classes are roughly divided into two parts. The first part is a class that is used to aid in reading the file and getting access to certain columns within the data file. The second class is used to perform the regression between two variables, and the type of class used depends on the data types of the variables that are compared.

In the next chapter, another set of classes are developed. The classes in that chapter provide an example of a set of classes that can be used to generate the results from a stochastic process and manage the results from a large number of simulations. The classes can be used to generate results from either a discrete or continuous process, and the distribution of the results can be explored.