

Quickstart tutorial for the R programming language for Azure Machine Learning

Stephen F Elston, Ph.D.

Introduction

This quickstart tutorial helps you quickly start extending Azure Machine Learning by using the R programming language. Follow this R programming tutorial to create, test and execute R code within Azure Machine Learning. As you work through tutorial, you will create a complete forecasting solution by using the R language in Azure Machine Learning.

Microsoft Azure Machine Learning contains many powerful machine learning and data manipulation modules. The powerful R language has been described as the lingua franca of analytics. Happily, analytics and data manipulation in Azure Machine Learning can be extended by using R. This combination provides the scalability and ease of deployment of Azure Machine Learning with the flexibility and deep analytics of R.

Try Azure Machine Learning for free

No credit card or Azure subscription needed. [Get started now >](#)

Forecasting and the dataset

Forecasting is a widely employed and quite useful analytical method. Common uses range from predicting sales of seasonal items, determining optimal inventory levels, to predicting macroeconomic variables. Forecasting is typically done with time series models.

Time series data is data in which the values have a time index. The time index can be regular, e.g. every month or every minute, or irregular. A time series model is based on time series data. The R programming language contains a flexible framework and extensive analytics for time series data.

In this quickstart guide we will be working with California dairy production and pricing data. This data includes monthly information on the production of several dairy products and the price of milk fat, a benchmark commodity.

The data used in this article, along with R scripts, can be [downloaded here](#). This data was originally synthesized from information available from the University of Wisconsin at <http://future.aae.wisc.edu/tab/production.html>.

Organization

We will progress through several steps as you learn how to create, test and execute analytics and

data manipulation R code in the Azure Machine Learning environment.

- First we will explore the basics of using the R language in the Azure Machine Learning Studio environment.
- Then we progress to discussing various aspects of I/O for data, R code and graphics in the Azure Machine Learning environment.
- We will then construct the first part of our forecasting solution by creating code for data cleaning and transformation.
- With our data prepared we will perform an analysis of the correlations between several of the variables in our dataset.
- Finally, we will create a seasonal time series forecasting model for milk production.

Interact with R language in Machine Learning Studio

This section takes you through some basics of interacting with the R programming language in the Machine Learning Studio environment. The R language provides a powerful tool to create customized analytics and data manipulation modules within the Azure Machine Learning environment.

I will use RStudio to develop, test and debug R code on a small scale. This code is then cut and paste into an [Execute R Script](#) module in Machine Learning Studio ready to run.

The Execute R Script module

Within Machine Learning Studio, R scripts are run within the [Execute R Script](#) module. An example of the [Execute R Script](#) module in Machine Learning Studio is shown in Figure 1.

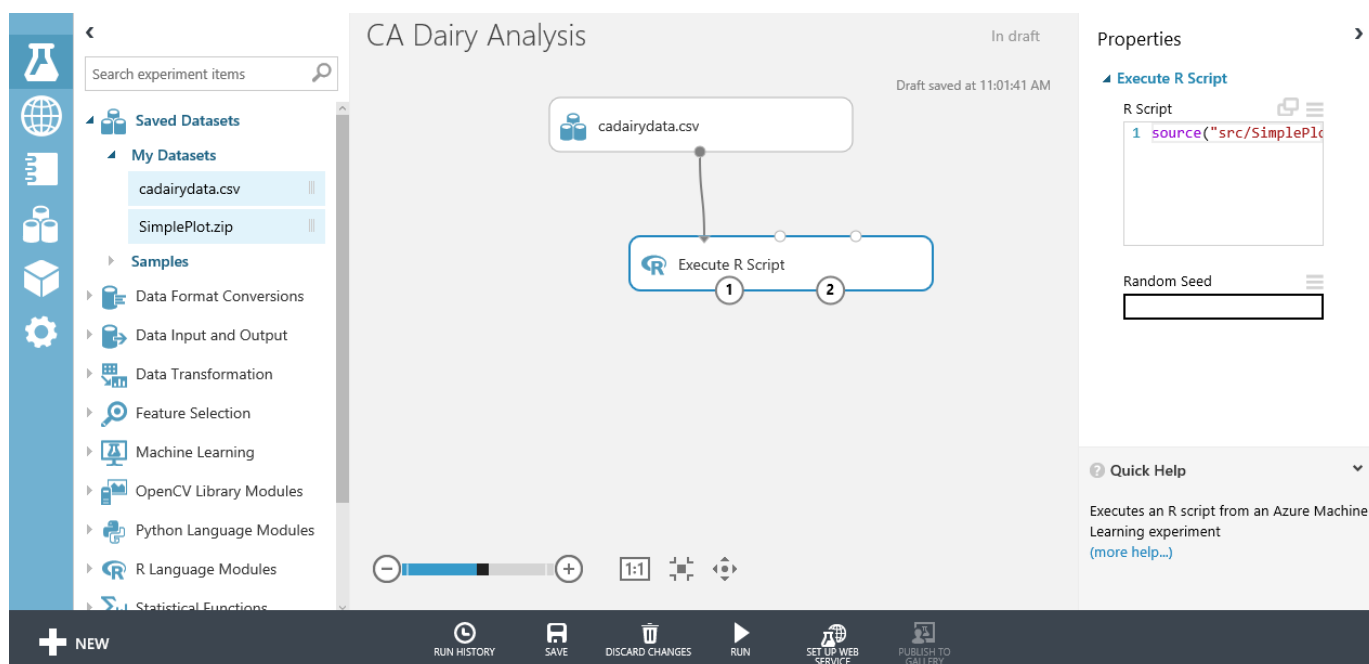


Figure 1. The Machine Learning Studio environment showing the Execute R Script module selected.

Referring to Figure 1, let's look at some of the key parts of the Machine Learning Studio environment for working with the [Execute R Script](#) module.

- The modules in the experiment are shown in the center pane.
- The upper part of the right pane contains a window to view and edit your R scripts.
- The lower part of right pane shows some properties of the [Execute R Script](#). You can view the error and output logs by clicking on the appropriate spots of this pane.

We will, of course, be discussing the [Execute R Script](#) in greater detail in the rest of this document.

When working with complex R functions, I recommend that you edit, test and debug in RStudio. As with any software development, extend your code incrementally and test it on small simple test cases. Then cut and paste your functions into the R script window of the [Execute R Script](#) module. This approach allows you to harness both the RStudio integrated development environment (IDE) and the power of Azure Machine Learning.

Execute R code

Any R code in the [Execute R Script](#) module will execute when you run the experiment by clicking on the **Run** button. When execution has completed, a check mark will appear on the [Execute R Script](#) icon.

Defensive R coding for Azure Machine Learning

If you are developing R code for, say, a web service by using Azure Machine Learning, you should definitely plan how your code will deal with an unexpected data input and exceptions. To maintain clarity, I have not included much in the way of checking or exception handling in most of the code examples shown. However, as we proceed I will give you several examples of functions by using R's exception handling capability.

If you need a more complete treatment of R exception handling, I recommend you read the applicable sections of the book by Wickham listed in [Appendix B - Further Reading](#).

Debug and test R in Machine Learning Studio

To reiterate, I recommend you test and debug your R code on a small scale in RStudio. However, there are cases where you will need to track down R code problems in the [Execute R Script](#) itself. In addition, it is good practice to check your results in Machine Learning Studio.

Output from the execution of your R code and on the Azure Machine Learning platform is found primarily in output.log. Some additional information will be seen in error.log.

If an error occurs in Machine Learning Studio while running your R code, your first course of action should be to look at error.log. This file can contain useful error messages to help you understand and

correct your error. To view error.log, click on **View error log** on the **properties pane** for the [Execute R Script](#) containing the error.

For example, I ran the following R code, with an undefined variable y, in an [Execute R Script](#) module:

Copy

```
x <- 1.0  
z <- x + y
```

This code fails to execute, resulting in an error condition. Clicking on **View error log** on the **properties pane** produces the display shown in Figure 2.

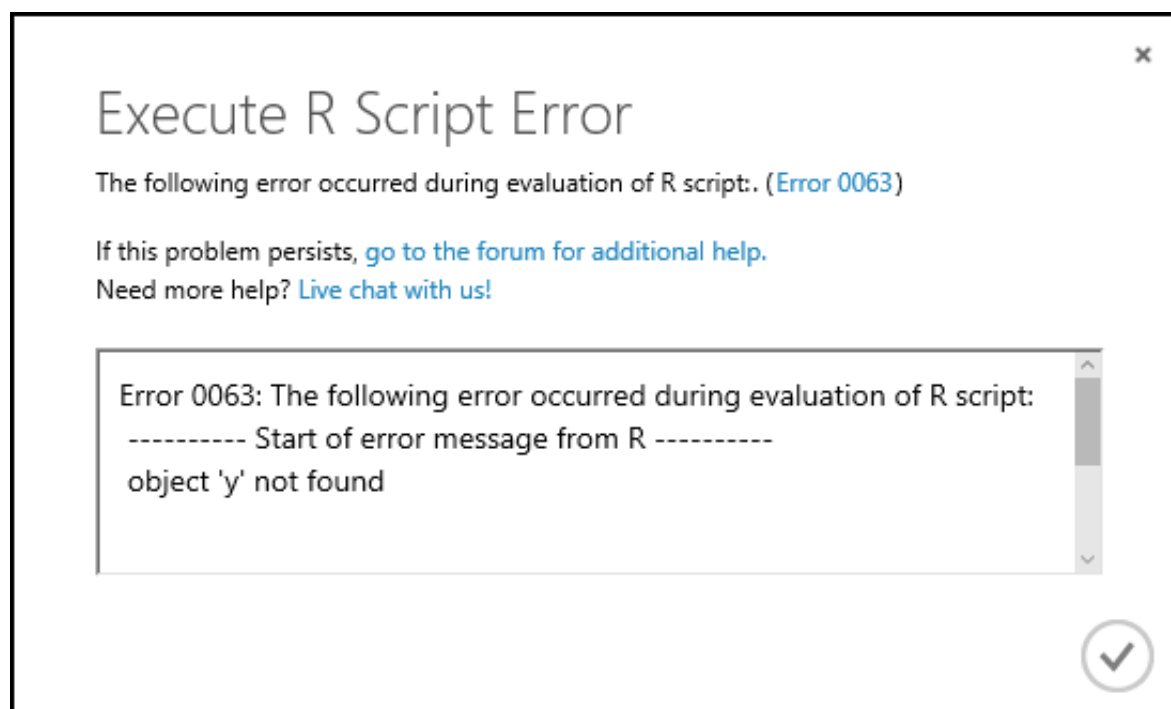


Figure 2. Error message pop-up.

It looks like we need to look in output.log to see the R error message. Click on the [Execute R Script](#) and then click on the **View output.log** item on the **properties pane** to the right. A new browser window opens, and I see the following.

Copy

```
[Critical]      Error: Error 0063: The following error occurred during evaluation  
of R script:  
----- Start of error message from R -----  
object 'y' not found  
  
object 'y' not found  
----- End of error message from R -----
```

This error message contains no surprises and clearly identifies the problem.

To inspect the value of any object in R, you can print these values to the output.log file. The rules for examining object values are essentially the same as in an interactive R session. For example, if you type a variable name on a line, the value of the object will be printed to the output.log file.

Packages in Machine Learning Studio

Azure Machine Learning comes with over 350 preinstalled R language packages. You can use the following code in the [Execute R Script](#) module to retrieve a list of the preinstalled packages.

Copy to clipboardCopy

```
data.set <- data.frame(installed.packages())  
maml.mapOutputPort("data.set")
```

If you don't understand the last line of this code at the moment, read on. In the rest of this document we will extensively discuss using R in the Azure Machine Learning environment.

Introduction to RStudio

RStudio is a widely used IDE for R. I will use RStudio for editing, testing and debugging some of the R code used in this quick start guide. Once R code is tested and ready, you simply cut and paste from the RStudio editor into a Machine Learning Studio [Execute R Script](#) module.

If you do not have the R programming language installed on your desktop machine, I recommend you do so now. Free downloads of open source R language are available at the Comprehensive R Archive Network (CRAN) at <http://www.r-project.org/>. There are downloads available for Windows, Mac OS, and Linux/UNIX. Choose a nearby mirror and follow the download directions. In addition, CRAN contains a wealth of useful analytics and data manipulation packages.

If you are new to RStudio, you should download and install the desktop version. You can find the RStudio downloads for Windows, Mac OS, and Linux/UNIX at <http://www.rstudio.com/products/RStudio/>. Follow the directions provided to install RStudio on your desktop machine.

A tutorial introduction to RStudio is available at <https://support.rstudio.com/hc/sections/200107586-Using-RStudio>.

I provide some additional information on using RStudio in [Appendix A](#).

Get data in and out of the Execute R Script module

In this section we will discuss how you get data into and out of the [Execute R Script](#) module. We will review how to handle various data types read into and out of the [Execute R Script](#) module.

The complete code for this section is in the zip file you downloaded earlier.

Load and check data in Machine Learning Studio

Load the dataset

We will start by loading the **csdairydata.csv** file into Azure Machine Learning Studio.

- Start your Azure Machine Learning Studio environment.
- Click on **+ NEW** at the lower left of your screen and select **Dataset**.
- Select **From Local File**, and then **Browse** to select the file.
- Make sure you have selected **Generic CSV file with header (.csv)** as the type for the dataset.
- Click the check mark.
- After the dataset has been uploaded, you should see the new dataset by clicking on the **Datasets** tab.

Create an experiment

Now that we have some data in Machine Learning Studio, we need to create an experiment to do the analysis.

- Click on **+ NEW** at the lower left and select **Experiment**, then **Blank Experiment**.
- You can name your experiment by selecting, and modifying, the **Experiment created on ...** title at the top of the page. For example, changing it to **CA Dairy Analysis**.
- On the left of the experiment page, expand **Saved Datasets**, and then **My Datasets**. You should see the **csdairydata.csv** that you uploaded earlier.
- Drag and drop the **csdairydata.csv dataset** onto the experiment.
- In the **Search experiment items** box on the top of the left pane, type [Execute R Script](#). You will see the module appear in the search list.
- Drag and drop the [Execute R Script](#) module onto your pallet.
- Connect the output of the **csdairydata.csv dataset** to the leftmost input (**Dataset1**) of the [Execute R Script](#).
- **Don't forget to click on 'Save'!**

At this point your experiment should look something like Figure 3.

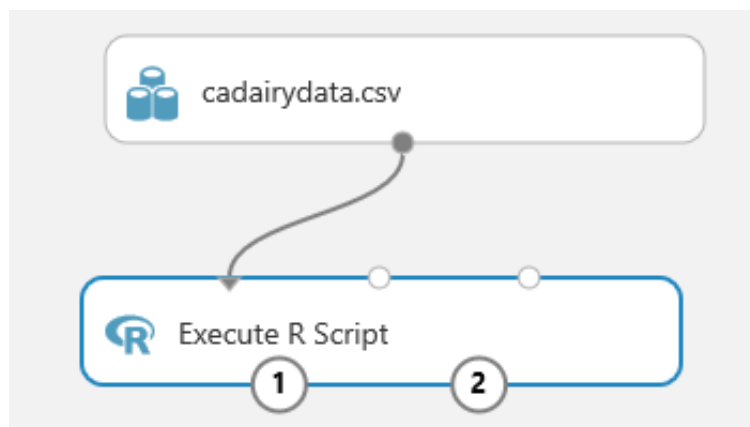


Figure 3. The CA Dairy Analysis experiment with dataset and Execute R Script module.

Check on the data

Let's have a look at the data we have loaded into our experiment. In the experiment, click on the output of the **cadairydata.csv dataset** and select **visualize**. You should see something like Figure 4.

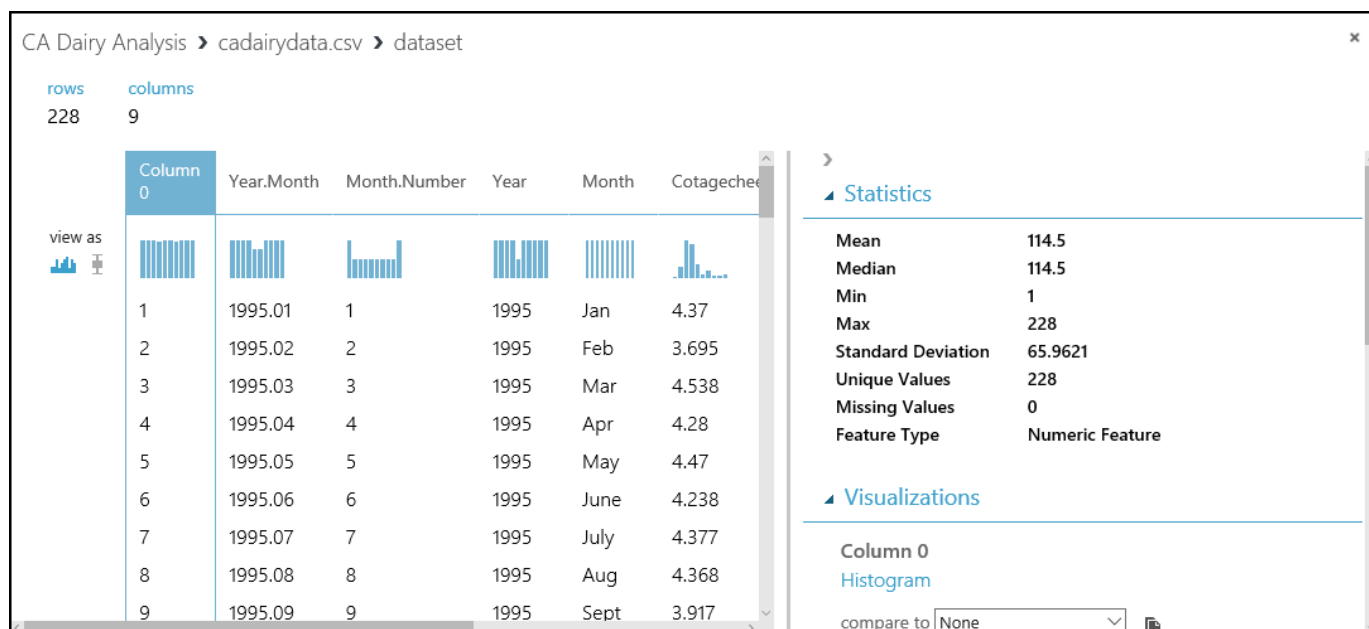


Figure 4. Summary of the cadairydata.csv dataset.

In this view we see a lot of useful information. We can see the first several rows of that dataset. If we select a column, the Statistics section shows more information about the column. For example, the Feature Type row shows us what data types Azure Machine Learning Studio assigned to the column. Having a quick look like this is a good sanity check before we start to do any serious work.

First R script

Let's create a simple first R script to experiment with in Azure Machine Learning Studio. I have created and tested the following script in RStudio.

Copy

```
## Only one of the following two lines should be used
## If running in Machine Learning Studio, use the first line with
maml.mapInputPort()
```

```
## If in RStudio, use the second line with read.csv()
cadairydata <- maml.mapInputPort(1)
# cadairydata <- read.csv("cadairydata.csv", header = TRUE, stringsAsFactors =
FALSE)
str(cadairydata)
pairs(~ Cotagecheese.Prod + Icecream.Prod + Milk.Prod + N.CA.Fat.Price, data =
cadairydata)
## The following line should be executed only when running in
## Azure Machine Learning Studio
maml.mapOutputPort('cadairydata')
```

Now I need to transfer this script to Azure Machine Learning Studio. I could simply cut and paste. However, in this case, I will transfer my R script via a zip file.

Data input to the Execute R Script module

Let's have a look at the inputs to the [Execute R Script](#) module. In this example we will read the California dairy data into the [Execute R Script](#) module.

There are three possible inputs for the [Execute R Script](#) module. You may use any one or all of these inputs, depending on your application. It is also perfectly reasonable to use an R script that takes no input at all.

Let's look at each of these inputs, going from left to right. You can see the names of each of the inputs by placing your cursor over the input and reading the tooltip.

Script Bundle

The Script Bundle input allows you to pass the contents of a zip file into [Execute R Script](#) module. You can use one of the following commands to read the contents of the zip file into your R code.

Copy

```
source("src/yourfile.R") # Reads a zipped R script
load("src/yourData.rdata") # Reads a zipped R data file
```

NOTE:

Azure Machine Learning treats files in the zip as if they are in the `src/` directory, so you need to prefix your file names with this directory name. For example, if the zip contains the files `yourfile.R` and `yourData.rdata` in the root of the zip, you would address these as `src/yourfile.R` and `src/yourData.rdata` when using `source` and `load`.

We already discussed loading datasets in [Loading the dataset](#). Once you have created and tested the R script shown in the previous section, do the following:

1. Save the R script into a .R file. I call my script file "simpleplot.R". Here's the contents.

Copy

```
## Only one of the following two lines should be used
## If running in Machine Learning Studio, use the first line with
maml.mapInputPort()
## If in RStudio, use the second line with read.csv()
cadairydata <- maml.mapInputPort(1)
# cadairydata <- read.csv("cadairydata.csv", header = TRUE, stringsAsFactors
= FALSE)
str(cadairydata)
pairs(~ Cotagecheese.Prod + Icecream.Prod + Milk.Prod + N.CA.Fat.Price, data =
cadairydata)
## The following line should be executed only when running in
## Azure Machine Learning Studio
maml.mapOutputPort('cadairydata')
```

2. Create a zip file and copy your script into this zip file. On Windows, you can right-click on the file and select **Send to**, and then **Compressed folder**. This will create a new zip file containing the "simpleplot.R" file.
3. Add your file to the **datasets** in Machine Learning Studio, specifying the type as **zip**. You should now see the zip file in your datasets.
4. Drag and drop the zip file from **datasets** onto the **ML Studio canvas**.
5. Connect the output of the **zip data** icon to the **Script Bundle** input of the [Execute R Script](#) module.
6. Type the `source()` function with your zip file name into the code window for the [Execute R Script](#) module. In my case I typed `source("src/simpleplot.R")`.
7. Make sure you click **Save**.

Once these steps are complete, the [Execute R Script](#) module will execute the R script in the zip file when the experiment is run. At this point your experiment should look something like Figure 5.

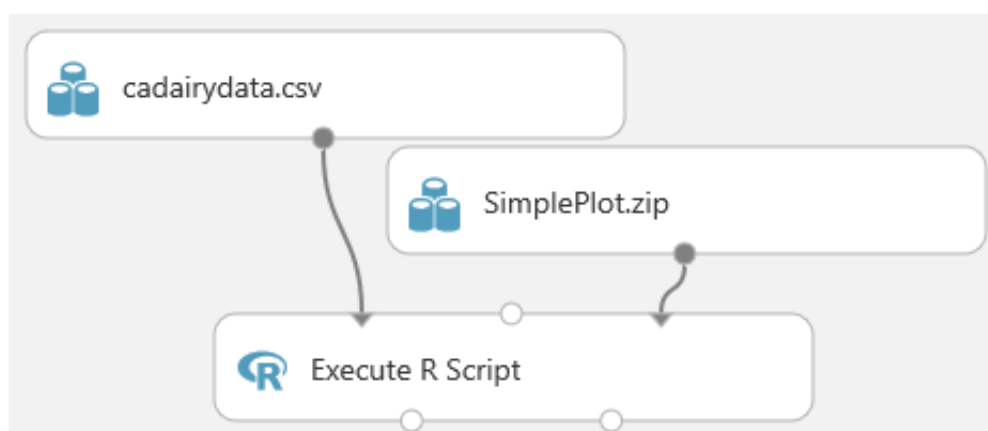


Figure 5. Experiment using zipped R script.

Dataset1

You can pass a rectangular table of data to your R code by using the Dataset1 input. In our simple script the `maml.mapInputPort(1)` function reads the data from port 1. This data is then assigned to a dataframe variable name in your code. In our simple script the first line of code performs the assignment.

Copy

```
cadairydata <- maml.mapInputPort(1)
```

Execute your experiment by clicking on the **Run** button. When the execution finishes, click on the [Execute R Script](#) module and then click **View output log** on the properties pane. A new page should appear in your browser showing the contents of the output.log file. When you scroll down you should see something like the following.

Copy

```
[ModuleOutput] InputDataStructure
[ModuleOutput]
[ModuleOutput] {
[ModuleOutput]   "InputName":Dataset1
[ModuleOutput]   "Rows":228
[ModuleOutput]   "Cols":9
[ModuleOutput]   "ColumnTypes":System.Int32,3,System.Double,5,System.String,1
[ModuleOutput] }
```

Farther down the page is more detailed information on the columns, which will look something like the following.

Copy

```
[ModuleOutput] [1] "Loading variable port1..."
[ModuleOutput]
[ModuleOutput] 'data.frame':   228 obs. of  9 variables:
[ModuleOutput]
[ModuleOutput] $ Column 0      : int  1 2 3 4 5 6 7 8 9 10 ...
[ModuleOutput]
[ModuleOutput] $ Year.Month    : num  1995 1995 1995 1995 1995 ...
[ModuleOutput]
[ModuleOutput] $ Month.Number  : int  1 2 3 4 5 6 7 8 9 10 ...
[ModuleOutput]
[ModuleOutput] $ Year          : int  1995 1995 1995 1995 1995 1995 1995 1995
1995 1995 ...
[ModuleOutput]
[ModuleOutput] $ Month         : chr  "Jan" "Feb" "Mar" "Apr" ...
[ModuleOutput]
```

```
[ModuleOutput] $ Cotagecheese.Prod: num  4.37 3.69 4.54 4.28 4.47 ...  
[ModuleOutput]  
[ModuleOutput] $ Icecream.Prod    : num  51.6 56.1 68.5 65.7 73.7 ...  
[ModuleOutput]  
[ModuleOutput] $ Milk.Prod        : num   2.11 1.93 2.16 2.13 2.23 ...  
[ModuleOutput]  
[ModuleOutput] $ N.CA.Fat.Price   : num   0.98 0.892 0.892 0.897 0.897 ...
```

These results are mostly as expected, with 228 observations and 9 columns in the dataframe. We can see the column names, the R data type and a sample of each column.

NOTE:

This same printed output is conveniently available from the R Device output of the [Execute R Script](#) module. We will discuss the outputs of the [Execute R Script](#) module in the next section.

Dataset2

The behavior of the Dataset2 input is identical to that of Dataset1. Using this input you can pass a second rectangular table of data into your R code. The function `maml.mapInputPort(2)`, with the argument 2, is used to pass this data.

Execute R Script outputs

Output a dataframe

You can output the contents of an R dataframe as a rectangular table through the Result Dataset1 port by using the `maml.mapOutputPort()` function. In our simple R script this is performed by the following line.

Copy

```
maml.mapOutputPort('cadairydata')
```

After running the experiment, click on the Result Dataset1 output port and then click on **Visualize**. You should see something like Figure 6.

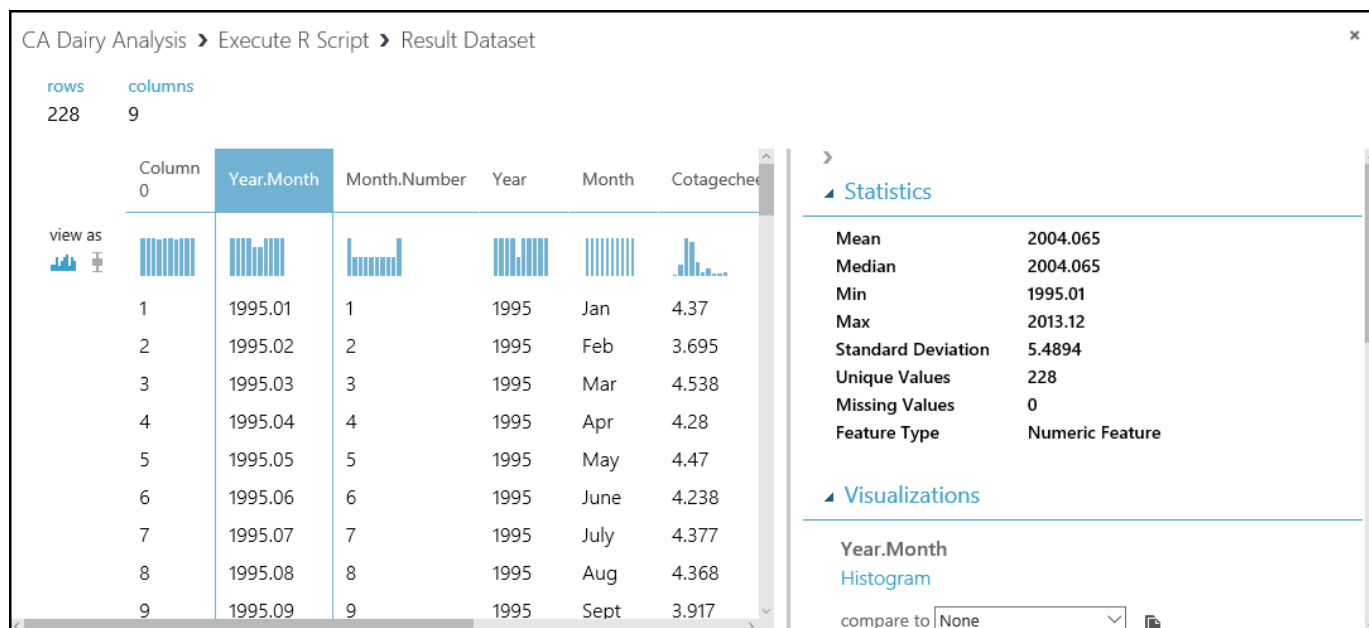


Figure 6. The visualization of the output of the California dairy data.

This output looks identical to the input, exactly as we expected.

R Device output

The Device output of the [Execute R Script](#) module contains messages and graphics output. Both standard output and standard error messages from R are sent to the R Device output port.

To view the R Device output, click on the port and then on **Visualize**. We see the standard output and standard error from the R script in Figure 7.

CA Dairy Analysis > Execute R Script > R Device

Standard Output

RWorker pushed "port1" to R workspace.
Beginning R Execute Script

Standard Error

Graphics Device

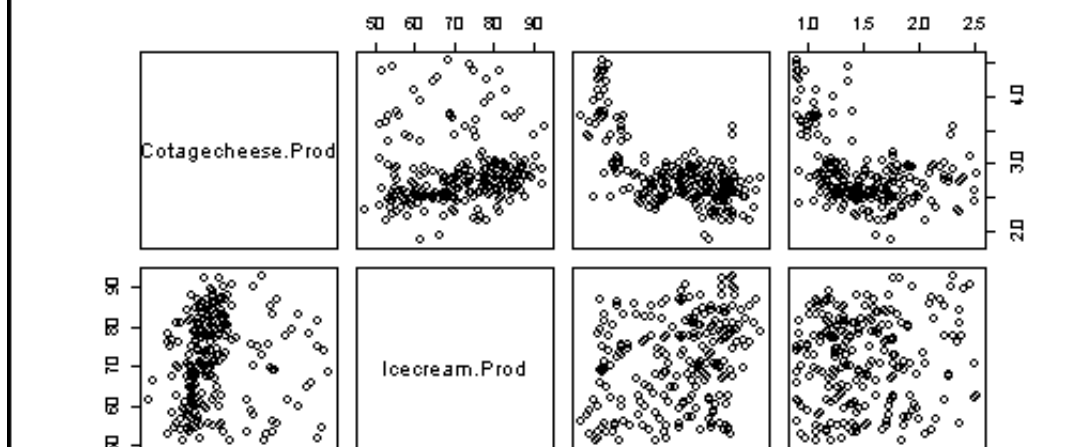


Figure 7. Standard output and standard error from the R Device port.

Scrolling down we see the graphics output from our R script in Figure 8.

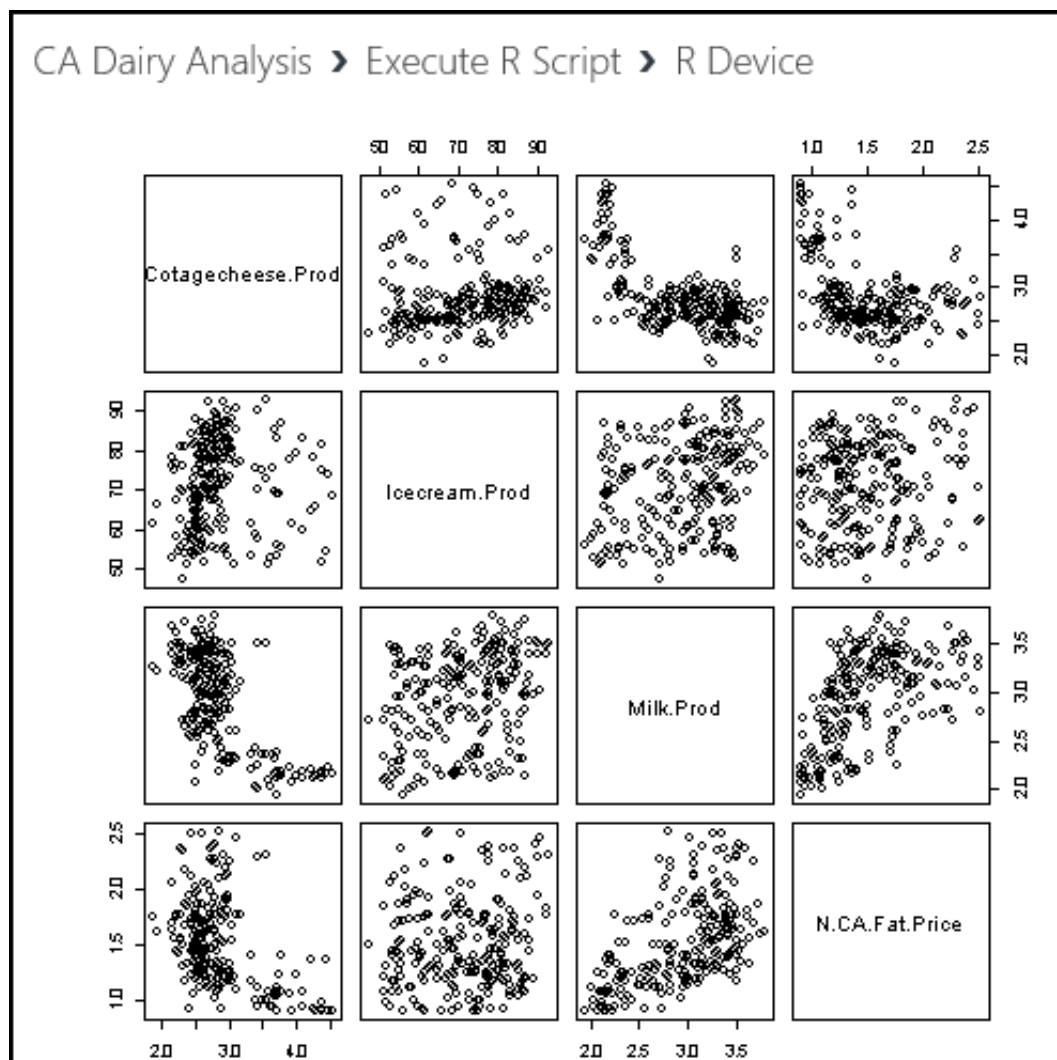


Figure 8. Graphics output from the R Device port.

Data filtering and transformation

In this section we will perform some basic data filtering and transformation operations on the California dairy data. By the end of this section we will have data in a format suitable for building an analytic model.

More specifically, in this section we will perform several common data cleaning and transformation tasks: type transformation, filtering on dataframes, adding new computed columns, and value transformations. This background should help you deal with the many variations encountered in real-world problems.

The complete R code for this section is available in the zip file you downloaded earlier.

Type transformations

Now that we can read the California dairy data into the R code in the [Execute R Script](#) module, we need to ensure that the data in the columns has the intended type and format.

R is a dynamically typed language, which means that data types are coerced from one to another as required. The atomic data types in R include numeric, logical and character. The factor type is used

to compactly store categorical data. You can find much more information on data types in the references in [Appendix B - Further reading](#).

When tabular data is read into R from an external source, it is always a good idea to check the resulting types in the columns. You may want a column of type character, but in many cases this will show up as factor or vice versa. In other cases a column you think should be numeric is represented by character data, e.g. '1.23' rather than 1.23 as a floating point number.

Fortunately, it is easy to convert one type to another, as long as mapping is possible. For example, you cannot convert 'Nevada' into a numeric value, but you can convert it to a factor (categorical variable). As another example, you can convert a numeric 1 into a character '1' or a factor.

The syntax for any of these conversions is simple: `as.datatype()`. These type conversion functions include the following.

- `as.numeric()`
- `as.character()`
- `as.logical()`
- `as.factor()`

Looking at the data types of the columns we input in the previous section: all columns are of type numeric, except for the column labeled 'Month', which is of type character. Let's convert this to a factor and test the results.

I have deleted the line that created the scatterplot matrix and added a line converting the 'Month' column to a factor. In my experiment I will just cut and paste the R code into the code window of the [Execute R Script](#) Module. You could also update the zip file and upload it to Azure Machine Learning Studio, but this takes several steps.

Copy

```
## Only one of the following two lines should be used
## If running in Machine Learning Studio, use the first line with
maml.mapInputPort()
## If in RStudio, use the second line with read.csv()
cadairydata <- maml.mapInputPort(1)
# cadairydata <- read.csv("cadairydata.csv", header = TRUE, stringsAsFactors =
FALSE)
## Ensure the coding is consistent and convert column to a factor
cadairydata$Month <- as.factor(cadairydata$Month)
str(cadairydata) # Check the result
## The following line should be executed only when running in
## Azure Machine Learning Studio
maml.mapOutputPort('cadairydata')
```

Let's execute this code and look at the output log for the R script. The relevant data from the log is shown in Figure 9.

Copy

```
[ModuleOutput] [1] "Loading variable port1..."
[ModuleOutput]
[ModuleOutput] 'data.frame':    228 obs. of  9 variables:
[ModuleOutput]
[ModuleOutput] $ Column 0      : int  1 2 3 4 5 6 7 8 9 10 ...
[ModuleOutput]
[ModuleOutput] $ Year.Month    : num  1995 1995 1995 1995 1995 ...
[ModuleOutput]
[ModuleOutput] $ Month.Number  : int  1 2 3 4 5 6 7 8 9 10 ...
[ModuleOutput]
[ModuleOutput] $ Year          : int  1995 1995 1995 1995 1995 1995 1995 1995
1995 1995 ...
[ModuleOutput]
[ModuleOutput] $ Month         : Factor w/ 14 levels "Apr","April",...: 6 5 9
1 11 8 7 3 14 13 ...
[ModuleOutput]
[ModuleOutput] $ Cotagecheese.Prod: num  4.37 3.69 4.54 4.28 4.47 ...
[ModuleOutput]
[ModuleOutput] $ Icecream.Prod    : num  51.6 56.1 68.5 65.7 73.7 ...
[ModuleOutput]
[ModuleOutput] $ Milk.Prod        : num  2.11 1.93 2.16 2.13 2.23 ...
[ModuleOutput]
[ModuleOutput] $ N.CA.Fat.Price   : num  0.98 0.892 0.892 0.897 0.897 ...
[ModuleOutput]
[ModuleOutput] [1] "Saving variable  cadairydata  ..."
[ModuleOutput]
[ModuleOutput] [1] "Saving the following item(s):  .mam1.oport1"
```

Figure 9. Summary of the dataframe with a factor variable.

The type for Month should now say '**Factor w/ 14 levels**'. This is a problem since there are only 12 months in the year. You can also check to see that the type in **Visualize** of the Result Dataset port is '**Categorical**'.

The problem is that the 'Month' column has not been coded systematically. In some cases a month is called April and in others it is abbreviated as Apr. We can solve this problem by trimming the string to 3 characters. The line of code now looks like the following:

Copy

```
## Ensure the coding is consistent and convert column to a factor
cadairydata$Month <- as.factor(substr(cadairydata$Month, 1, 3))
```


Rerun the experiment and view the output log. The expected results are shown in Figure 10.

Copy

```
[ModuleOutput] [1] "Loading variable port1..."
[ModuleOutput]
[ModuleOutput] 'data.frame':    228 obs. of  9 variables:
[ModuleOutput]
[ModuleOutput] $ Column 0      : int  1 2 3 4 5 6 7 8 9 10 ...
[ModuleOutput]
[ModuleOutput] $ Year.Month    : num  1995 1995 1995 1995 1995 ...
[ModuleOutput]
[ModuleOutput] $ Month.Number  : int  1 2 3 4 5 6 7 8 9 10 ...
[ModuleOutput]
[ModuleOutput] $ Year          : int  1995 1995 1995 1995 1995 1995 1995 1995
1995 1995 ...
[ModuleOutput]
[ModuleOutput] $ Month         : Factor w/ 12 levels "Apr","Aug","Dec",...: 5
4 8 1 9 7 6 2 12 11 ...
[ModuleOutput]
[ModuleOutput] $ Cotagecheese.Prod: num  4.37 3.69 4.54 4.28 4.47 ...
[ModuleOutput]
[ModuleOutput] $ Icecream.Prod    : num  51.6 56.1 68.5 65.7 73.7 ...
[ModuleOutput]
[ModuleOutput] $ Milk.Prod        : num  2.11 1.93 2.16 2.13 2.23 ...
[ModuleOutput]
[ModuleOutput] $ N.CA.Fat.Price   : num  0.98 0.892 0.892 0.897 0.897 ...
[ModuleOutput]
[ModuleOutput] [1] "Saving variable  cadairydata  ..."
[ModuleOutput]
[ModuleOutput] [1] "Saving the following item(s):  .mam1.oport1"
```

Figure 10. Summary of the dataframe with correct number of factor levels.

Our factor variable now has the desired 12 levels.

Basic data frame filtering

R dataframes support powerful filtering capabilities. Datasets can be subsetted by using logical filters on either rows or columns. In many cases, complex filter criteria will be required. The references in [Appendix B - Further reading](#) contain extensive examples of filtering dataframes.

There is one bit of filtering we should do on our dataset. If you look at the columns in the `cadairydata` dataframe, you will see two unnecessary columns. The first column just holds a row number, which is not very useful. The second column, `Year.Month`, contains redundant information. We can easily

exclude these columns by using the following R code.

NOTE:

From now on in this section, I will just show you the additional code I am adding in the [Execute R Script](#) module. I will add each new line **before** the `str()` function. I use this function to verify my results in Azure Machine Learning Studio.

I add the following line to my R code in the [Execute R Script](#) module.

Copy

```
# Remove two columns we do not need
cadairydata <- cadairydata[, c(-1, -2)]
```

Run this code in your experiment and check the result from the output log. These results are shown in Figure 11.

Copy

```
[ModuleOutput] [1] "Loading variable port1..."
[ModuleOutput]
[ModuleOutput] 'data.frame':    228 obs. of  7 variables:
[ModuleOutput]
[ModuleOutput] $ Month.Number      : int  1 2 3 4 5 6 7 8 9 10 ...
[ModuleOutput]
[ModuleOutput] $ Year              : int  1995 1995 1995 1995 1995 1995 1995 1995
1995 1995 ...
[ModuleOutput]
[ModuleOutput] $ Month             : Factor w/ 12 levels "Apr","Aug","Dec",...: 5
4 8 1 9 7 6 2 12 11 ...
[ModuleOutput]
[ModuleOutput] $ Cotagecheese.Prod: num  4.37 3.69 4.54 4.28 4.47 ...
[ModuleOutput]
[ModuleOutput] $ Icecream.Prod    : num  51.6 56.1 68.5 65.7 73.7 ...
[ModuleOutput]
[ModuleOutput] $ Milk.Prod        : num  2.11 1.93 2.16 2.13 2.23 ...
[ModuleOutput]
[ModuleOutput] $ N.CA.Fat.Price   : num  0.98 0.892 0.892 0.897 0.897 ...
[ModuleOutput]
[ModuleOutput] [1] "Saving variable  cadairydata  ..."
[ModuleOutput]
[ModuleOutput] [1] "Saving the following item(s):  .maml.oport1"
```

Figure 11. Summary of the dataframe with two columns removed.

Good news! We get the expected results.

Add a new column

To create time series models it will be convenient to have a column containing the months since the start of the time series. We will create a new column 'Month.Count'.

To help organize the code we will create our first simple function, `num.month()`. We will then apply this function to create a new column in the dataframe. The new code is as follows.

Copy

```
## Create a new column with the month count
## Function to find the number of months from the first
## month of the time series
num.month <- function(Year, Month) {
  ## Find the starting year
  min.year <- min(Year)

  ## Compute the number of months from the start of the time series
  12 * (Year - min.year) + Month - 1
}

## Compute the new column for the dataframe
cadairydata$Month.Count <- num.month(cadairydata$Year, cadairydata$Month.Number)
```

Now run the updated experiment and use the output log to view the results. These results are shown in Figure 12.

Copy

```
[ModuleOutput] [1] "Loading variable port1..."
[ModuleOutput]
[ModuleOutput] 'data.frame':    228 obs. of  8 variables:
[ModuleOutput]
[ModuleOutput] $ Month.Number      : int  1 2 3 4 5 6 7 8 9 10 ...
[ModuleOutput]
[ModuleOutput] $ Year              : int  1995 1995 1995 1995 1995 1995 1995 1995 1995
1995 1995 ...
[ModuleOutput]
[ModuleOutput] $ Month             : Factor w/ 12 levels "Apr","Aug","Dec",...: 5
4 8 1 9 7 6 2 12 11 ...
[ModuleOutput]
[ModuleOutput] $ Cottagecheese.Prod: num  4.37 3.69 4.54 4.28 4.47 ...
[ModuleOutput]
[ModuleOutput] $ Icecream.Prod     : num  51.6 56.1 68.5 65.7 73.7 ...
[ModuleOutput]
[ModuleOutput] $ Milk.Prod         : num  2.11 1.93 2.16 2.13 2.23 ...
```

```

[ModuleOutput]
[ModuleOutput] $ N.CA.Fat.Price      : num  0.98 0.892 0.892 0.897 0.897 ...
[ModuleOutput]
[ModuleOutput] $ Month.Count          : num   0 1 2 3 4 5 6 7 8 9 ...
[ModuleOutput]
[ModuleOutput] [1] "Saving variable  cadairydata  ..."
[ModuleOutput]
[ModuleOutput] [1] "Saving the following item(s):  .mam1.oport1"

```

Figure 12. Summary of the dataframe with the additional column.

It looks like everything is working. We have the new column with the expected values in our dataframe.

Value transformations

In this section we will perform some simple transformations on the values in some of the columns of our dataframe. The R language supports nearly arbitrary value transformations. The references in [Appendix B - Further Reading](#) contain extensive examples.

If you look at the values in the summaries of our dataframe you should see something odd here. Is more ice cream than milk produced in California? No, of course not, as this makes no sense, sad as this fact may be to some of us ice cream lovers. The units are different. The price is in units of US pounds, milk is in units of 1 M US pounds, ice cream is in units of 1,000 US gallons, and cottage cheese is in units of 1,000 US pounds. Assuming ice cream weighs about 6.5 pounds per gallon, we can easily do the multiplication to convert these values so they are all in equal units of 1,000 pounds.

For our forecasting model we use a multiplicative model for trend and seasonal adjustment of this data. A log transformation allows us to use a linear model, simplifying this process. We can apply the log transformation in the same function where the multiplier is applied.

In the following code, I define a new function, `log.transform()`, and apply it to the rows containing the numerical values. The R `Map()` function is used to apply the `log.transform()` function to the selected columns of the dataframe. `Map()` is similar to `apply()` but allows for more than one list of arguments to the function. Note that a list of multipliers supplies the second argument to the `log.transform()` function. The `na.omit()` function is used as a bit of cleanup to ensure we do not have missing or undefined values in the dataframe.

Copy

```

log.transform <- function(invec, multiplier = 1) {
  ## Function for the transformation, which is the log
  ## of the input value times a multiplier

  warningmessages <- c("ERROR: Non-numeric argument encountered in function
log.transform",

```

```

        "ERROR: Arguments to function log.transform must be greater than zero",
        "ERROR: Argument multiplier to function log.transform must be a scalar",
        "ERROR: Invalid time series value encountered in function log.transform"
    )

    ## Check the input arguments
    if(!is.numeric(invec) | !is.numeric(multiplier)) {warning(warningmessages[1]); return(NA)}
    if(any(invec < 0.0) | any(multiplier < 0.0)) {warning(warningmessages[2]); return(NA)}
    if(length(multiplier) != 1) {{warning(warningmessages[3]); return(NA)}}

    ## Wrap the multiplication in tryCatch
    ## If there is an exception, print the warningmessage to standard error and return NA
    tryCatch(log(multiplier * invec),
            error = function(e){warning(warningmessages[4]); NA})
}

## Apply the transformation function to the 4 columns
## of the dataframe with production data
multipliers <- list(1.0, 6.5, 1000.0, 1000.0)
cadairydata[, 4:7] <- Map(log.transform, cadairydata[, 4:7], multipliers)

## Get rid of any rows with NA values
cadairydata <- na.omit(cadairydata)

```

There is quite a bit happening in the `log.transform()` function. Most of this code is checking for potential problems with the arguments or dealing with exceptions, which can still arise during the computations. Only a few lines of this code actually do the computations.

The goal of the defensive programming is to prevent the failure of a single function that prevents processing from continuing. An abrupt failure of a long-running analysis can be quite frustrating for users. To avoid this situation, default return values must be chosen that will limit damage to downstream processing. A message is also produced to alert users that something has gone wrong.

If you are not used to defensive programming in R, all this code may seem a bit overwhelming. I will walk you through the major steps:

1. A vector of four messages is defined. These messages are used to communicate information

about some of the possible errors and exceptions that can occur with this code.

2. I return a value of NA for each case. There are many other possibilities that might have fewer side effects. I could return a vector of zeroes, or the original input vector, for example.
3. Checks are run on the arguments to the function. In each case, if an error is detected, a default value is returned and a message is produced by the `warning()` function. I am using `warning()` rather than `stop()` as the latter will terminate execution, exactly what I am trying to avoid. Note that I have written this code in a procedural style, as in this case a functional approach seemed complex and obscure.
4. The log computations are wrapped in `tryCatch()` so that exceptions will not cause an abrupt halt to processing. Without `tryCatch()` most errors raised by R functions result in a stop signal, which does just that.

Execute this R code in your experiment and have a look at the printed output in the output.log file. You will now see the transformed values of the four columns in the log, as shown in Figure 13.

Copy

```
[ModuleOutput] [1] "Loading variable port1..."
[ModuleOutput]
[ModuleOutput] 'data.frame':    228 obs. of  8 variables:
[ModuleOutput]
[ModuleOutput]  $ Month.Number      : int  1 2 3 4 5 6 7 8 9 10 ...
[ModuleOutput]
[ModuleOutput]  $ Year              : int  1995 1995 1995 1995 1995 1995 1995 1995
1995 1995 ...
[ModuleOutput]
[ModuleOutput]  $ Month             : Factor w/ 12 levels "Apr","Aug","Dec",...: 5
4 8 1 9 7 6 2 12 11 ...
[ModuleOutput]
[ModuleOutput]  $ Cotagecheese.Prod: num  1.47 1.31 1.51 1.45 1.5 ...
[ModuleOutput]
[ModuleOutput]  $ Icecream.Prod    : num  5.82 5.9 6.1 6.06 6.17 ...
[ModuleOutput]
[ModuleOutput]  $ Milk.Prod        : num  7.66 7.57 7.68 7.66 7.71 ...
[ModuleOutput]
[ModuleOutput]  $ N.CA.Fat.Price   : num  6.89 6.79 6.79 6.8 6.8 ...
[ModuleOutput]
[ModuleOutput]  $ Month.Count      : num  0 1 2 3 4 5 6 7 8 9 ...
[ModuleOutput]
[ModuleOutput] [1] "Saving variable  cadairydata  ..."
[ModuleOutput]
[ModuleOutput] [1] "Saving the following item(s):  .maml.oport1"
```

Figure 13. Summary of the transformed values in the dataframe.

We see the values have been transformed. Milk production now greatly exceeds all other dairy product production, recalling that we are now looking at a log scale.

At this point our data is cleaned up and we are ready for some modeling. Looking at the visualization summary for the Result Dataset output of our [Execute R Script](#) module, you will see the 'Month' column is 'Categorical' with 12 unique values, again, just as we wanted.

Time series objects and correlation analysis

In this section we will explore a few basic R time series objects and analyze the correlations between some of the variables. Our goal is to output a dataframe containing the pairwise correlation information at several lags.

The complete R code for this section is in the zip file you downloaded earlier.

Time series objects in R

As already mentioned, time series are a series of data values indexed by time. R time series objects are used to create and manage the time index. There are several advantages to using time series objects. Time series objects free you from the many details of managing the time series index values that are encapsulated in the object. In addition, time series objects allow you to use the many time series methods for plotting, printing, modeling, etc.

The POSIXct time series class is commonly used and is relatively simple. This time series class measures time from the start of the epoch, January 1, 1970. We will use POSIXct time series objects in this example. Other widely used R time series object classes include zoo and xts, extensible time series.

Time series object example

Let's get started with our example. Drag and drop a **new** [Execute R Script](#) module into your experiment. Connect the Result Dataset1 output port of the existing [Execute R Script](#) module to the Dataset1 input port of the new [Execute R Script](#) module.

As I did for the first examples, as we progress through the example, at some points I will show only the incremental additional lines of R code at each step.

Reading the dataframe

As a first step, let's read in a dataframe and make sure we get the expected results. The following code should do the job.

Copy

```
# Comment the following if using RStudio
```

```
cadairydata <- maml.mapInputPort(1)
str(cadairydata) # Check the results
```

Now, run the experiment. The log of the new Execute R Script shape should look like Figure 14.

Copy

```
[ModuleOutput] [1] "Loading variable port1..."
[ModuleOutput]
[ModuleOutput] 'data.frame':    228 obs. of  8 variables:
[ModuleOutput]
[ModuleOutput] $ Month.Number      : int  1 2 3 4 5 6 7 8 9 10 ...
[ModuleOutput]
[ModuleOutput] $ Year              : int  1995 1995 1995 1995 1995 1995 1995 1995
1995 1995 ...
[ModuleOutput]
[ModuleOutput] $ Month             : Factor w/ 12 levels "Apr","Aug","Dec",...: 5
4 8 1 9 7 6 2 12 11 ...
[ModuleOutput]
[ModuleOutput] $ Cotagecheese.Prod: num  1.47 1.31 1.51 1.45 1.5 ...
[ModuleOutput]
[ModuleOutput] $ Icecream.Prod    : num  5.82 5.9 6.1 6.06 6.17 ...
[ModuleOutput]
[ModuleOutput] $ Milk.Prod        : num  7.66 7.57 7.68 7.66 7.71 ...
[ModuleOutput]
[ModuleOutput] $ N.CA.Fat.Price   : num  6.89 6.79 6.79 6.8 6.8 ...
[ModuleOutput]
[ModuleOutput] $ Month.Count      : num  0 1 2 3 4 5 6 7 8 9 ...
```

Figure 14. Summary of the dataframe in the Execute R Script module.

This data is of the expected types and format. Note that the 'Month' column is of type factor and has the expected number of levels.

Creating a time series object

We need to add a time series object to our dataframe. Replace the current code with the following, which adds a new column of class POSIXct.

Copy

```
# Comment the following if using RStudio
cadairydata <- maml.mapInputPort(1)

## Create a new column as a POSIXct object
Sys.setenv(TZ = "PST8PDT")
cadairydata$Time <- as.POSIXct(strptime(paste(as.character(cadairydata$Year), "-
```



```

", as.character(cadairydata$Month.Number), "-01 00:00:00", sep = ""), "%Y-%m-%d
%H:%M:%S"))

str(cadairydata) # Check the results

```

Now, check the log. It should look like Figure 15.

Copy

```

[ModuleOutput] [1] "Loading variable port1..."
[ModuleOutput]
[ModuleOutput] 'data.frame':    228 obs. of  9 variables:
[ModuleOutput]
[ModuleOutput] $ Month.Number      : int  1 2 3 4 5 6 7 8 9 10 ...
[ModuleOutput]
[ModuleOutput] $ Year              : int  1995 1995 1995 1995 1995 1995 1995 1995
1995 1995 ...
[ModuleOutput]
[ModuleOutput] $ Month             : Factor w/ 12 levels "Apr","Aug","Dec",...: 5
4 8 1 9 7 6 2 12 11 ...
[ModuleOutput]
[ModuleOutput] $ Cotagecheese.Prod: num  1.47 1.31 1.51 1.45 1.5 ...
[ModuleOutput]
[ModuleOutput] $ Icecream.Prod    : num  5.82 5.9 6.1 6.06 6.17 ...
[ModuleOutput]
[ModuleOutput] $ Milk.Prod        : num  7.66 7.57 7.68 7.66 7.71 ...
[ModuleOutput]
[ModuleOutput] $ N.CA.Fat.Price   : num  6.89 6.79 6.79 6.8 6.8 ...
[ModuleOutput]
[ModuleOutput] $ Month.Count      : num  0 1 2 3 4 5 6 7 8 9 ...
[ModuleOutput]
[ModuleOutput] $ Time             : POSIXct, format: "1995-01-01" "1995-02-01"
...

```

Figure 15. Summary of the dataframe with a time series object.

We can see from the summary that the new column is in fact of class POSIXct.

Exploring and transforming the data

Let's explore some of the variables in this dataset. A scatterplot matrix is a good way to produce a quick look. I am replacing the `str()` function in the previous R code with the following line.

Copy

```

pairs(~ Cotagecheese.Prod + Icecream.Prod + Milk.Prod + N.CA.Fat.Price, data =
cadairydata, main = "Pairwise Scatterplots of dairy time series")

```

Run this code and see what happens. The plot produced at the R Device port should look like Figure 16.

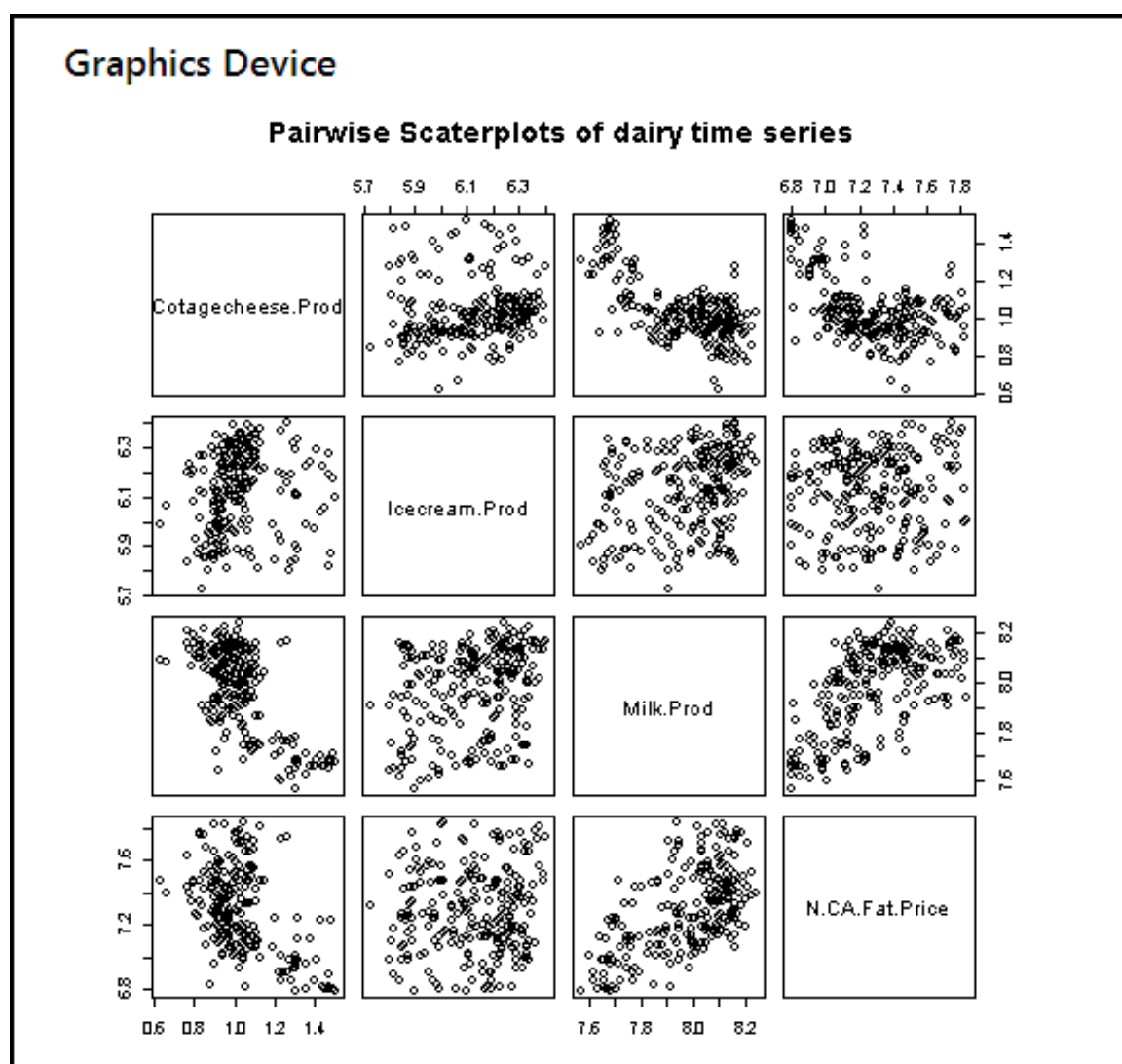


Figure 16. Scatterplot matrix of selected variables.

There is some odd-looking structure in the relationships between these variables. Perhaps this arises from trends in the data and from the fact that we have not standardized the variables.

Correlation analysis

To perform correlation analysis we need to both de-trend and standardize the variables. We could simply use the R `scale()` function, which both centers and scales variables. This function might well run faster. However, I want to show you an example of defensive programming in R.

The `ts.detrend()` function shown below performs both of these operations. The following two lines of code de-trend the data and then standardize the values.

Copy

```
ts.detrend <- function(ts, Time, min.length = 3){
  ## Function to de-trend and standardize a time series
```

```

## Define some messages if they are NULL
messages <- c('ERROR: ts.detrrend requires arguments ts and Time to have the
same length',
              'ERROR: ts.detrrend requires argument ts to be of type numeric',
              paste('WARNING: ts.detrrend has encountered a time series with
length less than', as.character(min.length)),
              'ERROR: ts.detrrend has encountered a Time argument not of class
POSIXct',
              'ERROR: Detrend regression has failed in ts.detrrend',
              'ERROR: Exception occurred in ts.detrrend while standardizing time
series in function ts.detrrend'
)

# Create a vector of zeros to return as a default in some cases
zerovec <- rep(length(ts), 0.0)

# The input arguments are not of the same length, return ts and quit
if(length(Time) != length(ts)) {warning(messages[1]); return(ts)}

# If the ts is not numeric, just return a zero vector and quit
if(!is.numeric(ts)) {warning(messages[2]); return(zerovec)}

# If the ts is too short, just return it and quit
if((ts.length <- length(ts)) < min.length) {warning(messages[3]); return(ts)}

## Check that the Time variable is of class POSIXct
if(class(cadairrydata$Time)[[1]] != "POSIXct") {warning(messages[4]);
return(ts)}

## De-trend the time series by using a linear model
ts.frame <- data.frame(ts = ts, Time = Time)
tryCatch({ts <- ts - fitted(lm(ts ~ Time, data = ts.frame))},
         error = function(e){warning(messages[5]); zerovec})

tryCatch( {stdev <- sqrt(sum((ts - mean(ts))^2)/(ts.length - 1)
          ts <- ts/stdev},
          error = function(e){warning(messages[6]); zerovec})

  ts
}

## Apply the detrend.ts function to the variables of interest
df.detrrend <- data.frame(lapply(cadairrydata[, 4:7], ts.detrrend,
cadairrydata$Time))

## Plot the results to look at the relationships

```

```
pairs(~ Cotagecheese.Prod + Icecream.Prod + Milk.Prod + N.CA.Fat.Price, data =
df.detrend, main = "Pairwise Scatterplots of detrended standardized time series")
```

There is quite a bit happening in the `ts.detrend()` function. Most of this code is checking for potential problems with the arguments or dealing with exceptions, which can still arise during the computations. Only a few lines of this code actually do the computations.

We have already discussed an example of defensive programming in [Value transformations](#). Both computation blocks are wrapped in `tryCatch()`. For some errors it makes sense to return the original input vector, and in other cases, I return a vector of zeros.

Note that the linear regression used for de-trending is a time series regression. The predictor variable is a time series object.

Once `ts.detrend()` is defined we apply it to the variables of interest in our dataframe. We must coerce the resulting list created by `lapply()` to data dataframe by using `as.data.frame()`. Because of defensive aspects of `ts.detrend()`, failure to process one of the variables will not prevent correct processing of the others.

The final line of code creates a pairwise scatterplot. After running the R code, the results of the scatterplot are shown in Figure 17.

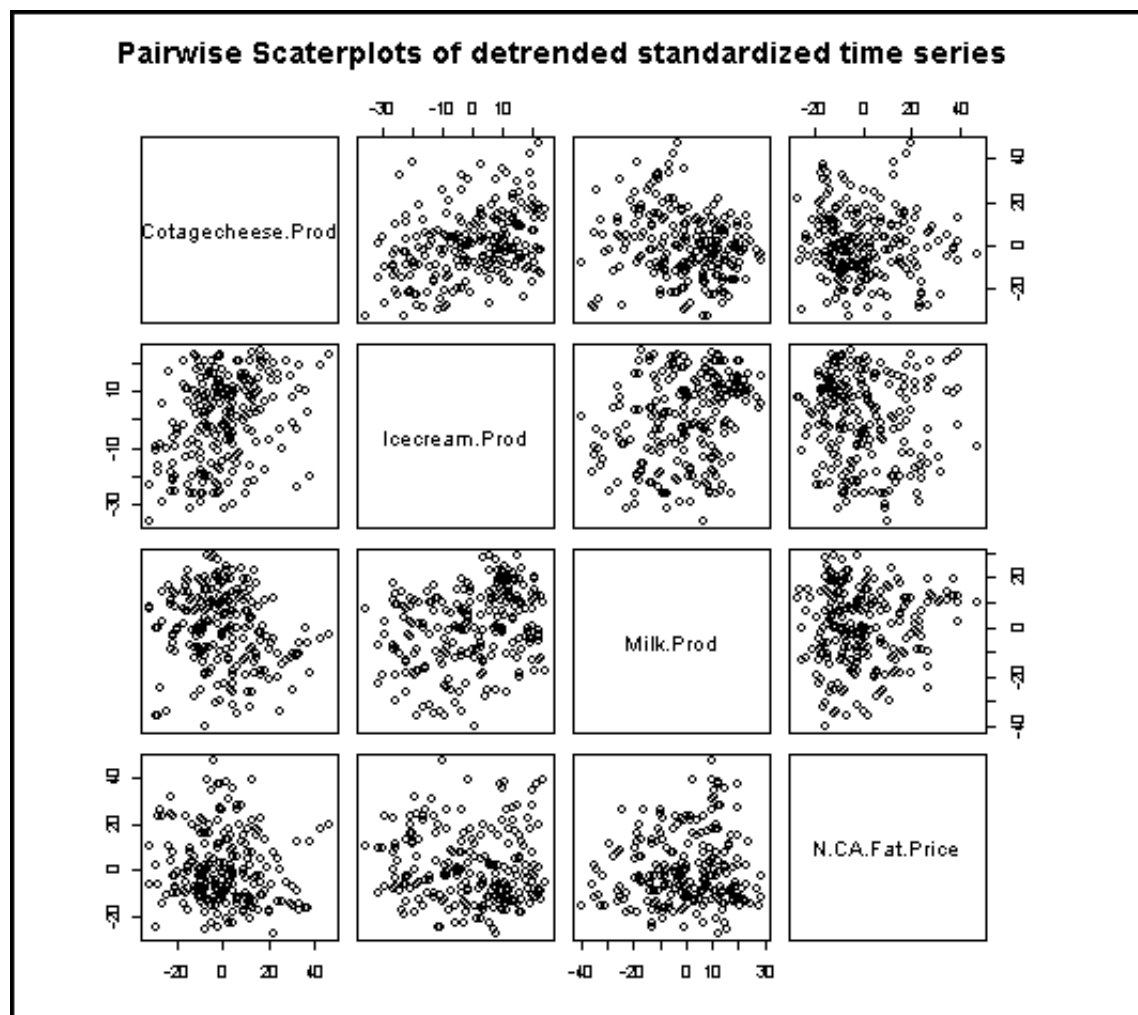


Figure 17. Pairwise scatterplot of de-trended and standardized time series.

You can compare these results to those shown in Figure 17. With the trend removed and the variables standardized, we see a lot less structure in the relationships between these variables.

The code to compute the correlations as R ccf objects is as follows.

Copy

```
## A function to compute pairwise correlations from a
## list of time series value vectors
pair.cor <- function(pair.ind, ts.list, lag.max = 1, plot = FALSE){
  ccf(ts.list[[pair.ind[1]]], ts.list[[pair.ind[2]]], lag.max = lag.max, plot =
plot)
}

## A list of the pairwise indices
corpairs <- list(c(1,2), c(1,3), c(1,4), c(2,3), c(2,4), c(3,4))

## Compute the list of ccf objects
cadairycorrelations <- lapply(corpairs, pair.cor, df.detrend)

cadairycorrelations
```

Running this code produces the log shown in Figure 18.

Copy

```
[ModuleOutput] Loading objects:
[ModuleOutput]   port1
[ModuleOutput] [1] "Loading variable port1..."
[ModuleOutput] [[1]]
[ModuleOutput]
[ModuleOutput]
[ModuleOutput] Autocorrelations of series 'X', by lag
[ModuleOutput]
[ModuleOutput]
[ModuleOutput]   -1    0    1
[ModuleOutput] 0.148 0.358 0.317
[ModuleOutput]
[ModuleOutput]
[ModuleOutput] [[2]]
[ModuleOutput]
[ModuleOutput]
[ModuleOutput] Autocorrelations of series 'X', by lag
[ModuleOutput]
[ModuleOutput]
[ModuleOutput]   -1    0    1
```

```

[ModuleOutput] -0.395 -0.186 -0.238
[ModuleOutput]
[ModuleOutput]
[ModuleOutput] [[3]]
[ModuleOutput]
[ModuleOutput]
[ModuleOutput] Autocorrelations of series 'X', by lag
[ModuleOutput]
[ModuleOutput]
[ModuleOutput]
[ModuleOutput]      -1      0      1
[ModuleOutput] -0.059 -0.089 -0.127
[ModuleOutput]
[ModuleOutput]
[ModuleOutput] [[4]]
[ModuleOutput]
[ModuleOutput]
[ModuleOutput] Autocorrelations of series 'X', by lag
[ModuleOutput]
[ModuleOutput]
[ModuleOutput]
[ModuleOutput]      -1      0      1
[ModuleOutput] 0.140 0.294 0.293
[ModuleOutput]
[ModuleOutput]
[ModuleOutput] [[5]]
[ModuleOutput]
[ModuleOutput]
[ModuleOutput] Autocorrelations of series 'X', by lag
[ModuleOutput]
[ModuleOutput]
[ModuleOutput]
[ModuleOutput]      -1      0      1
[ModuleOutput] -0.002 -0.074 -0.124

```

Figure 18. List of ccf objects from the pairwise correlation analysis.

There is a correlation value for each lag. None of these correlation values is large enough to be significant. We can therefore conclude that we can model each variable independently.

Output a dataframe

We have computed the pairwise correlations as a list of R ccf objects. This presents a bit of a problem as the Result Dataset output port really requires a dataframe. Further, the ccf object is itself a list and we want only the values in the first element of this list, the correlations at the various lags.

The following code extracts the lag values from the list of ccf objects, which are themselves lists.

Copy

```
df.correlations <- data.frame(do.call(rbind, lapply(cadairycorrelations, '[[',
1)))

c.names <- c("-1 lag", "0 lag", "+1 lag")
r.names <- c("Corr Cot Cheese - Ice Cream",
            "Corr Cot Cheese - Milk Prod",
            "Corr Cot Cheese - Fat Price",
            "Corr Ice Cream - Mik Prod",
            "Corr Ice Cream - Fat Price",
            "Corr Milk Prod - Fat Price")

## Build a dataframe with the row names column and the
## correlation data frame and assign the column names
outframe <- cbind(r.names, df.correlations)
colnames(outframe) <- c.names
outframe

## WARNING!
## The following line works only in Azure Machine Learning
## When running in RStudio, this code will result in an error
#maml.mapOutputPort('outframe')
```

The first line of code is a bit tricky, and some explanation may help you understand it. Working from the inside out we have the following:

1. The '[[operator with the argument '1' selects the vector of correlations at the lags from the first element of the ccf object list.
2. The `do.call()` function applies the `rbind()` function over the elements of the list returns by `lapply()`.
3. The `data.frame()` function coerces the result produced by `do.call()` to a dataframe.

Note that the row names are in a column of the dataframe. Doing so preserves the row names when they are output from the [Execute R Script](#).

Running the code produces the output shown in Figure 19 when I **Visualize** the output at the Result Dataset port. The row names are in the first column, as intended.

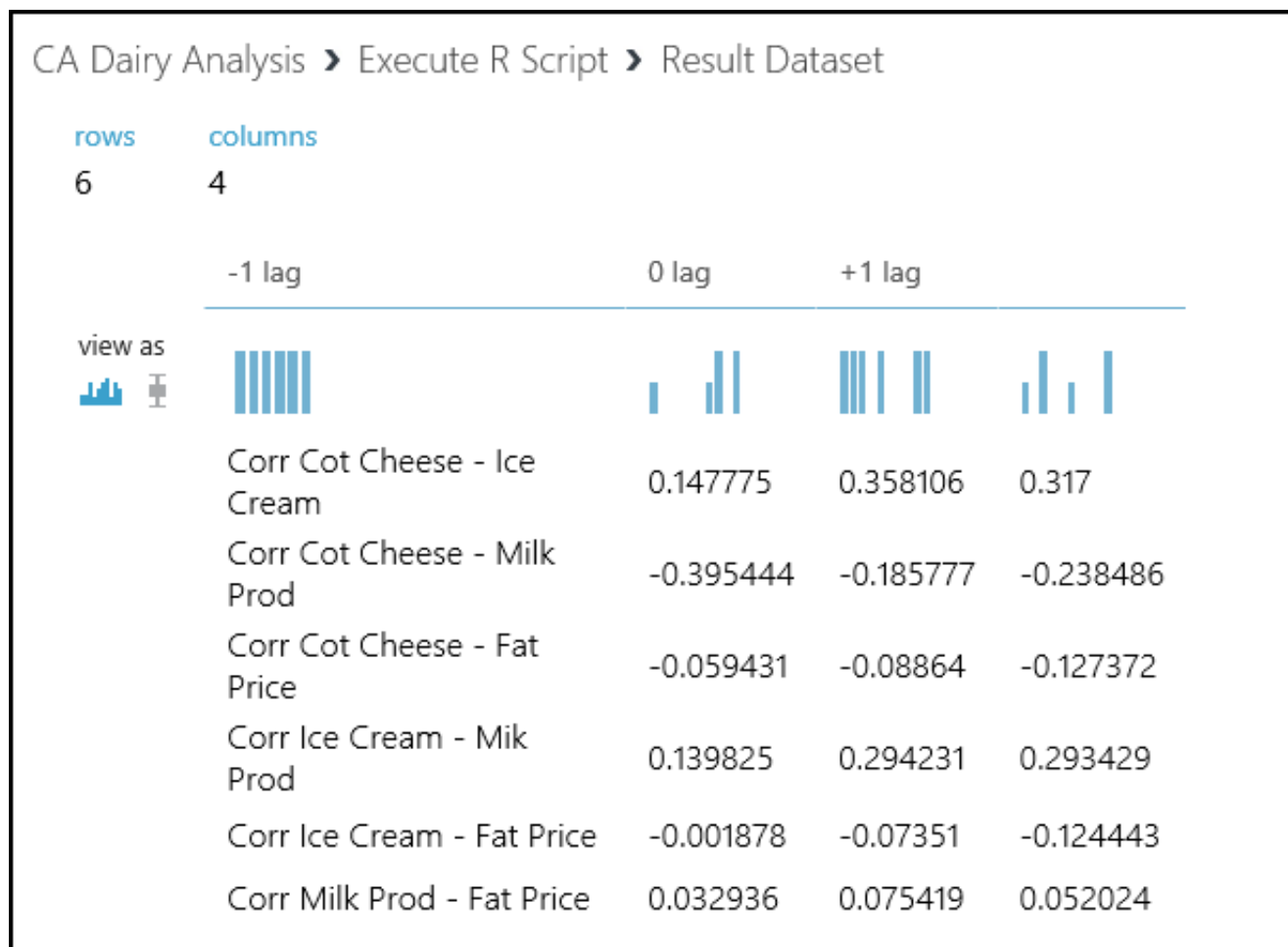


Figure 19. Results output from the correlation analysis.

Time series example: seasonal forecasting

Our data is now in a form suitable for analysis, and we have determined there are no significant correlations between the variables. Let's move on and create a time series forecasting model. Using this model we will forecast California milk production for the 12 months of 2013.

Our forecasting model will have two components, a trend component and a seasonal component. The complete forecast is the product of these two components. This type of model is known as a multiplicative model. The alternative is an additive model. We have already applied a log transformation to the variables of interest, which makes this analysis tractable.

The complete R code for this section is in the zip file you downloaded earlier.

Creating the dataframe for analysis

Start by adding a **new** [Execute R Script](#) module to your experiment. Connect the **Result Dataset** output of the existing [Execute R Script](#) module to the **Dataset1** input of the new module. The result should look something like Figure 20.

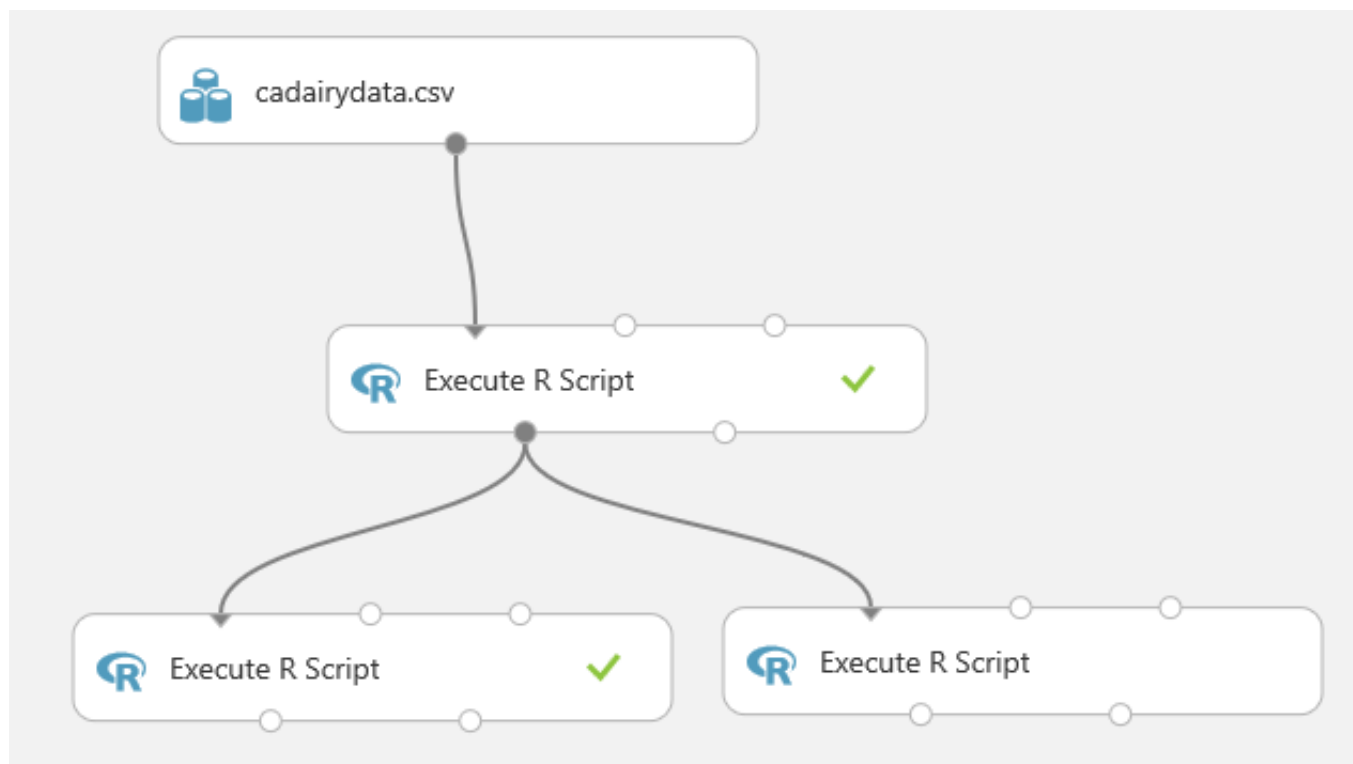


Figure 20. The experiment with the new Execute R Script module added.

As with the correlation analysis we just completed, we need to add a column with a POSIXct time series object. The following code will do just this.

Copy

```
# If running in Machine Learning Studio, uncomment the first line with
maml.mapInputPort()
cadairydata <- maml.mapInputPort(1)

## Create a new column as a POSIXct object
Sys.setenv(TZ = "PST8PDT")
cadairydata$Time <- as.POSIXct(strptime(paste(as.character(cadairydata$Year), "-
", as.character(cadairydata$Month.Number), "-01 00:00:00", sep = ""), "%Y-%m-%d
%H:%M:%S"))

str(cadairydata)
```

Run this code and look at the log. The result should look like Figure 21.

Copy

```
[ModuleOutput] [1] "Loading variable port1..."
[ModuleOutput]
[ModuleOutput] 'data.frame':    228 obs. of  9 variables:
[ModuleOutput]
[ModuleOutput] $ Month.Number      : int  1 2 3 4 5 6 7 8 9 10 ...
[ModuleOutput]
[ModuleOutput] $ Year              : int  1995 1995 1995 1995 1995 1995 1995 1995 1995
```

```

1995 1995 ...
[ModuleOutput]
[ModuleOutput] $ Month           : Factor w/ 12 levels "Apr","Aug","Dec",...: 5
4 8 1 9 7 6 2 12 11 ...
[ModuleOutput]
[ModuleOutput] $ Cotagecheese.Prod: num  1.47 1.31 1.51 1.45 1.5 ...
[ModuleOutput]
[ModuleOutput] $ Icecream.Prod   : num  5.82 5.9 6.1 6.06 6.17 ...
[ModuleOutput]
[ModuleOutput] $ Milk.Prod        : num  7.66 7.57 7.68 7.66 7.71 ...
[ModuleOutput]
[ModuleOutput] $ N.CA.Fat.Price   : num  6.89 6.79 6.79 6.8 6.8 ...
[ModuleOutput]
[ModuleOutput] $ Month.Count      : num  0 1 2 3 4 5 6 7 8 9 ...
[ModuleOutput]
[ModuleOutput] $ Time             : POSIXct, format: "1995-01-01" "1995-02-01"
...

```

Figure 21. A summary of the dataframe.

With this result, we are ready to start our analysis.

Create a training dataset

With the dataframe constructed we need to create a training dataset. This data will include all of the observations except the last 12, of the year 2013, which is our test dataset. The following code subsets the dataframe and creates plots of the dairy production and price variables. I then create plots of the four production and price variables. An anonymous function is used to define some augments for plot, and then iterate over the list of the other two arguments with `Map()`. If you are thinking that a for loop would have worked fine here, you are correct. But, since R is a functional language I am showing you a functional approach.

Copy

```

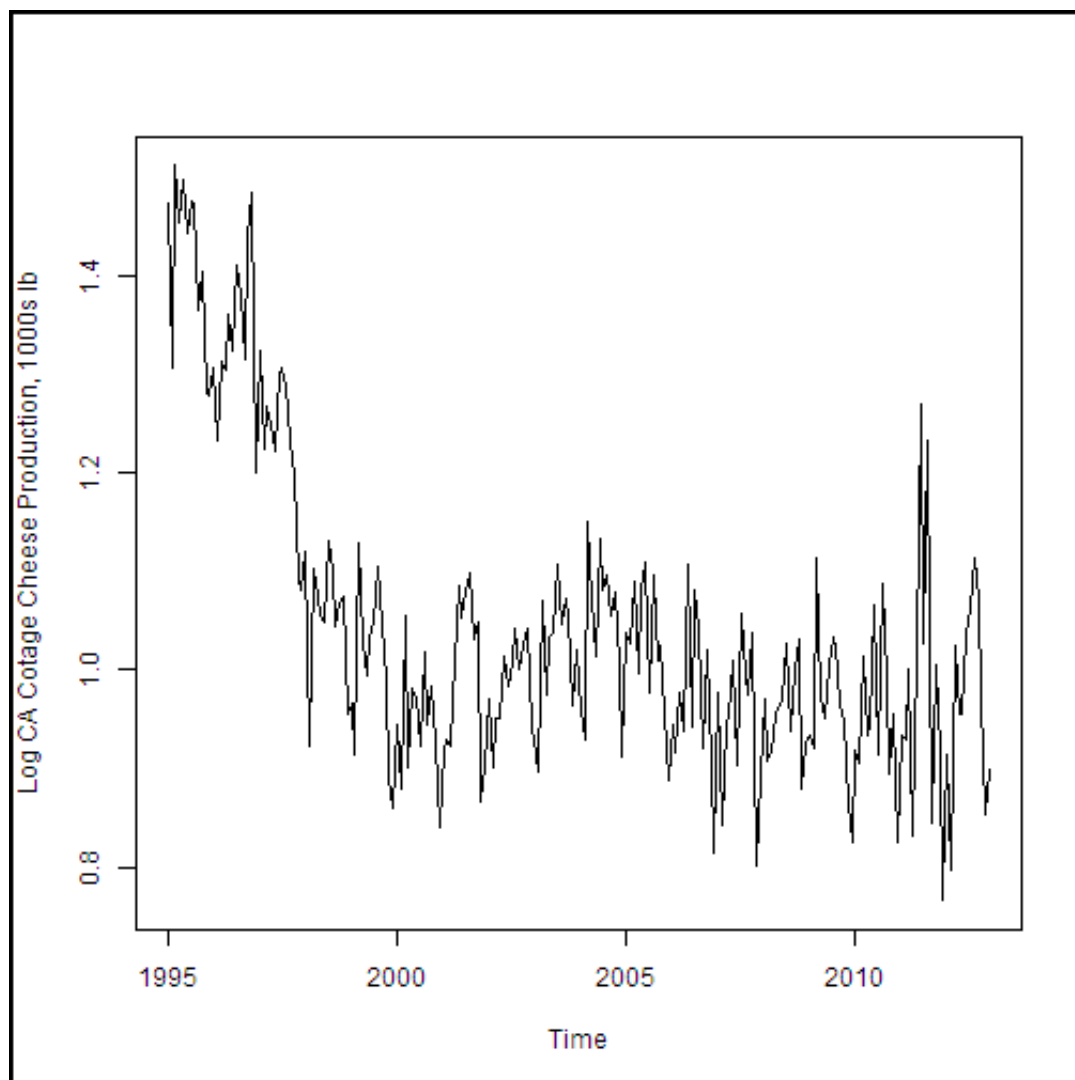
cadairytrain <- cadairydata[1:216, ]

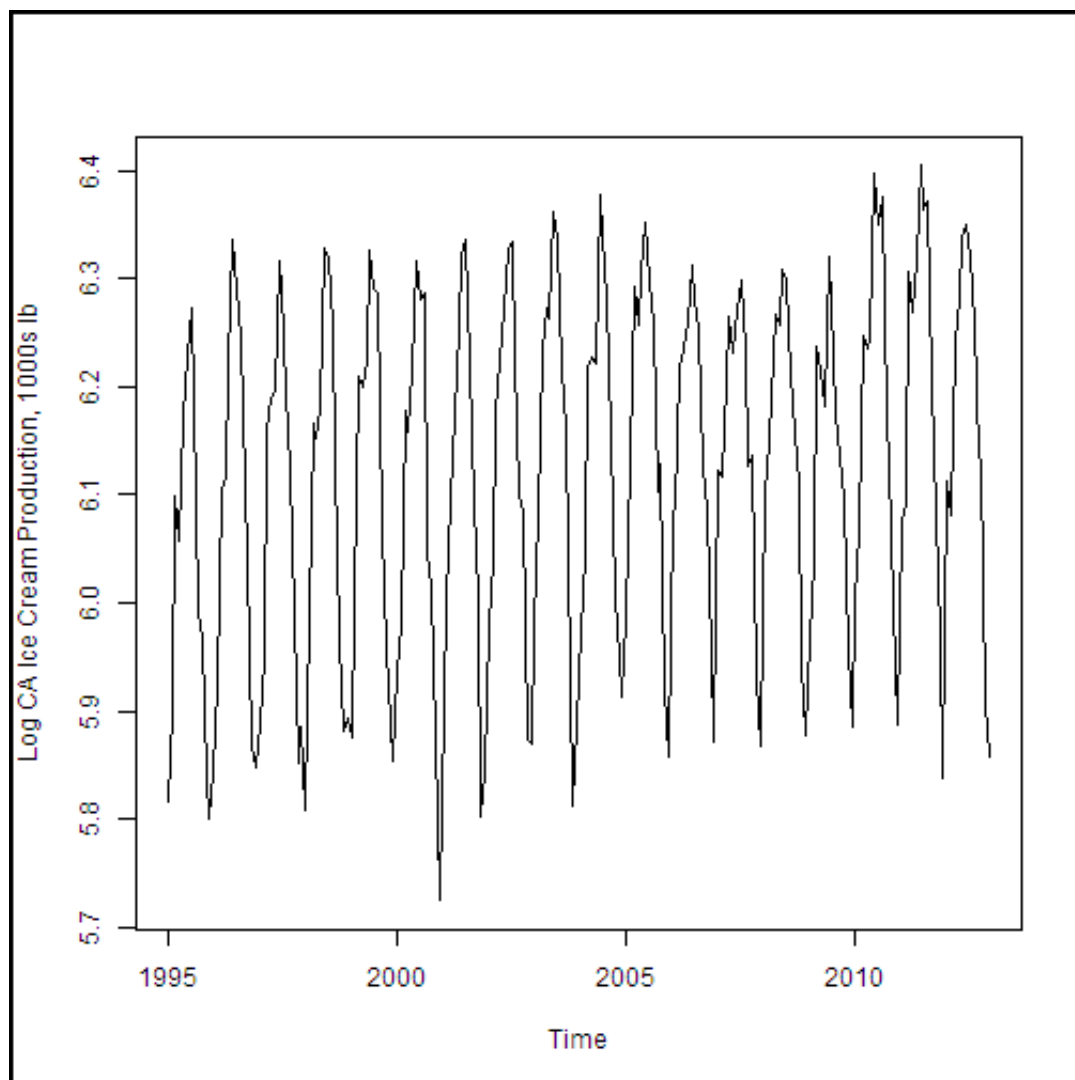
Ylabs <- list("Log CA Cotage Cheese Production, 1000s lb",
             "Log CA Ice Cream Production, 1000s lb",
             "Log CA Milk Production 1000s lb",
             "Log North CA Milk Milk Fat Price per 1000 lb")

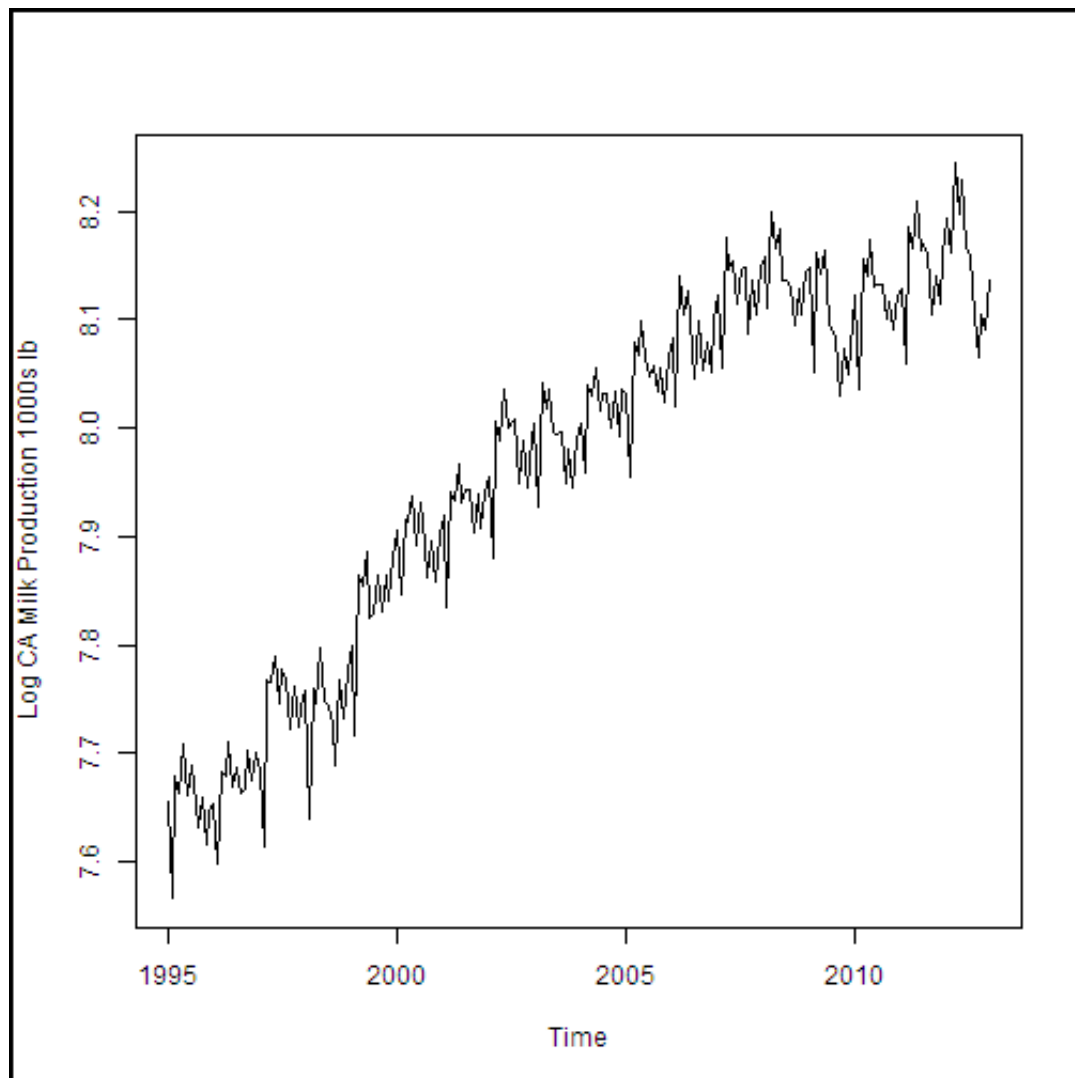
Map(function(y, Ylabs){plot(cadairytrain$Time, y, xlab = "Time", ylab = Ylabs,
type = "l")}, cadairytrain[, 4:7], Ylabs)

```

Running the code produces the series of time series plots from the R Device output shown in Figure 22. Note that the time axis is in units of dates, a nice benefit of the time series plot method.







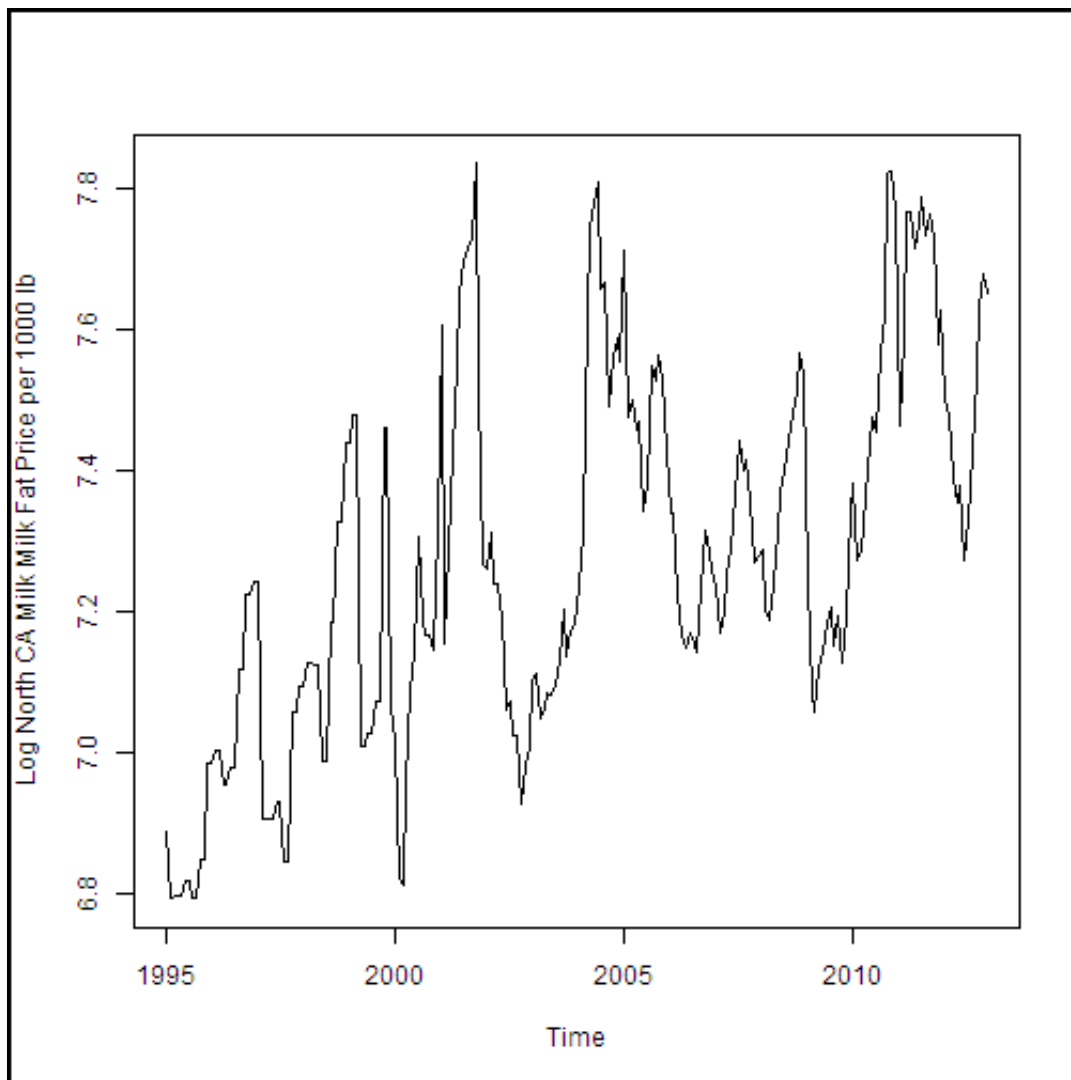


Figure 22. Time series plots of California dairy production and price data.

A trend model

Having created a time series object and having had a look at the data, let's start to construct a trend model for the California milk production data. We can do this with a time series regression. However, it is clear from the plot that we will need more than a slope and intercept to accurately model the observed trend in the training data.

Given the small scale of the data, I will build the model for trend in RStudio and then cut and paste the resulting model into Azure Machine Learning. RStudio provides an interactive environment for this type of interactive analysis.

As a first attempt, I will try a polynomial regression with powers up to 3. There is a real danger of over-fitting these kinds of models. Therefore, it is best to avoid high order terms. The `I()` function inhibits interpretation of the contents (interprets the contents 'as is') and allows you to write a literally interpreted function in a regression equation.

Copy

```

milk.lm <- lm(Milk.Prod ~ Time + I(Month.Count^2) + I(Month.Count^3), data =
cadairytrain)

```

```
summary(milk.lm)
```

This generates the following.

Copy

```
##
## Call:
## lm(formula = Milk.Prod ~ Time + I(Month.Count^2) + I(Month.Count^3),
##     data = cadairytrain)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.12667 -0.02730  0.00236  0.02943  0.10586
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   6.33e+00   1.45e-01  43.60  <2e-16 ***
## Time          1.63e-09   1.72e-10   9.47  <2e-16 ***
## I(Month.Count^2) -1.71e-06   4.89e-06  -0.35    0.726
## I(Month.Count^3) -3.24e-08   1.49e-08  -2.17    0.031 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0418 on 212 degrees of freedom
## Multiple R-squared:  0.941, Adjusted R-squared:  0.94
## F-statistic: 1.12e+03 on 3 and 212 DF, p-value: <2e-16
```

From P values ($\text{Pr}(>|t|)$) in this output, we can see that the squared term may not be significant. I will use the `update()` function to modify this model by dropping the squared term.

Copy

```
milk.lm <- update(milk.lm, . ~ . - I(Month.Count^2))
summary(milk.lm)
```

This generates the following.

Copy

```
##
## Call:
## lm(formula = Milk.Prod ~ Time + I(Month.Count^3), data = cadairytrain)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
```

```
## -0.12597 -0.02659 0.00185 0.02963 0.10696
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   6.38e+00  4.07e-02   156.6  <2e-16 ***
## Time          1.57e-09  4.32e-11    36.3  <2e-16 ***
## I(Month.Count^3) -3.76e-08  2.50e-09   -15.1  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0417 on 213 degrees of freedom
## Multiple R-squared:  0.941, Adjusted R-squared:  0.94
## F-statistic: 1.69e+03 on 2 and 213 DF, p-value: <2e-16
```

This looks better. All of the terms are significant. However, the 2e-16 value is a default value, and should not be taken too seriously.

As a sanity test, let's make a time series plot of the California dairy production data with the trend curve shown. I have added the following code in the Azure Machine Learning [Execute R Script](#) model (not RStudio) to create the model and make a plot. The result is shown in Figure 23.

Copy

```
milk.lm <- lm(Milk.Prod ~ Time + I(Month.Count^3), data = cadairytrain)

plot(cadairytrain$Time, cadairytrain$Milk.Prod, xlab = "Time", ylab = "Log CA
Milk Production 1000s lb", type = "l")
lines(cadairytrain$Time, predict(milk.lm, cadairytrain), lty = 2, col = 2)
```

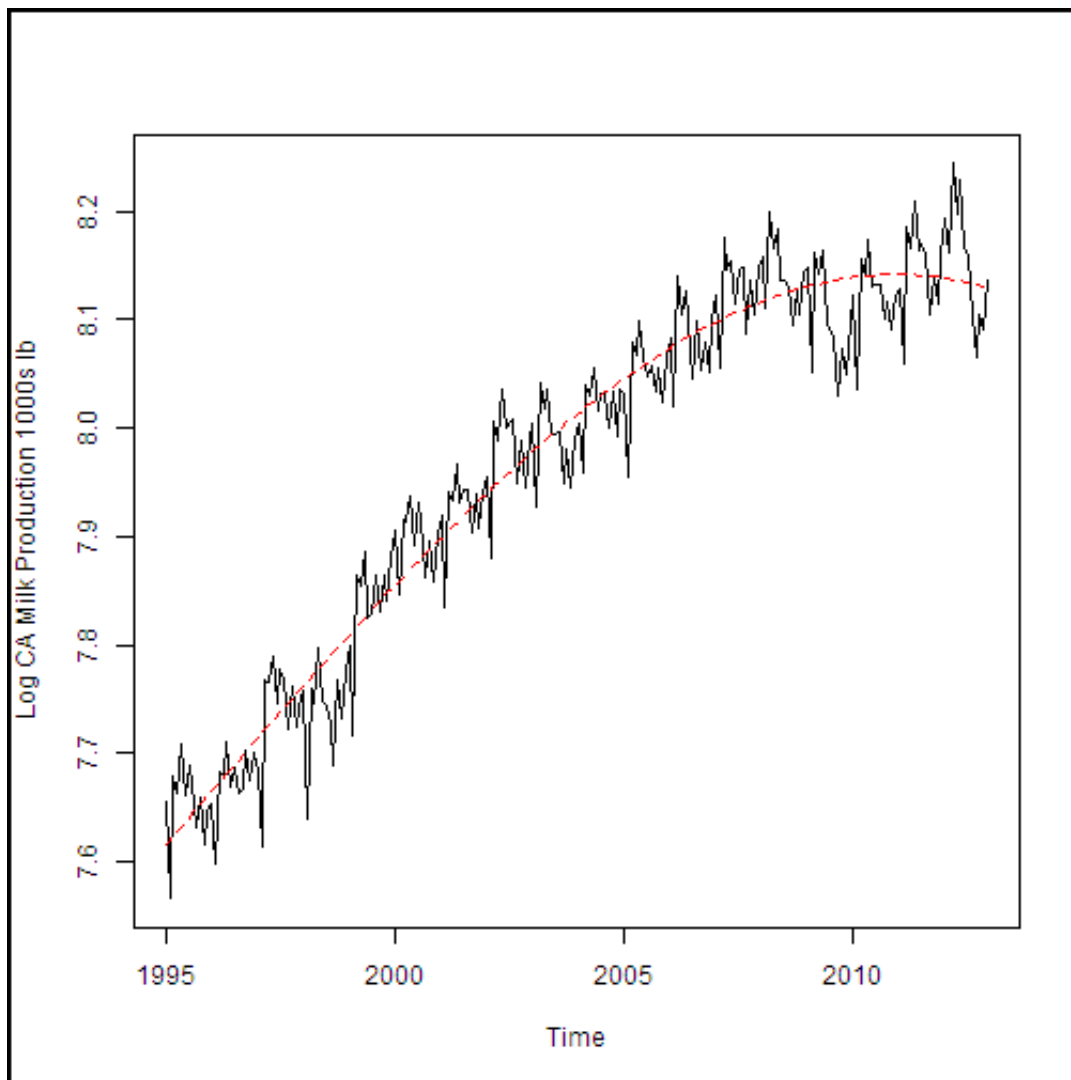



Figure 23. California milk production data with trend model shown.

It looks like the trend model fits the data fairly well. Further, there does not seem to be evidence of over-fitting, such as odd wiggles in the model curve.

Seasonal model

With a trend model in hand, we need to push on and include the seasonal effects. We will use the month of the year as a dummy variable in the linear model to capture the month-by-month effect. Note that when you introduce factor variables into a model, the intercept must not be computed. If you do not do this, the formula is over-specified and R will drop one of the desired factors but keep the intercept term.

Since we have a satisfactory trend model we can use the `update()` function to add the new terms to the existing model. The `-1` in the update formula drops the intercept term. Continuing in RStudio for the moment:

Copy

```
milkm2 <- update(milkm1, . ~ . + Month - 1)
summary(milkm2)
```

This generates the following.

Copy

```
##
## Call:
## lm(formula = Milk.Prod ~ Time + I(Month.Count^3) + Month - 1,
##     data = cadairytrain)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.06879 -0.01693  0.00346  0.01543  0.08726
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## Time              1.57e-09   2.72e-11    57.7  <2e-16 ***
## I(Month.Count^3) -3.74e-08   1.57e-09   -23.8  <2e-16 ***
## MonthApr           6.40e+00   2.63e-02   243.3  <2e-16 ***
## MonthAug           6.38e+00   2.63e-02   242.2  <2e-16 ***
## MonthDec           6.38e+00   2.64e-02   241.9  <2e-16 ***
## MonthFeb           6.31e+00   2.63e-02   240.1  <2e-16 ***
## MonthJan           6.39e+00   2.63e-02   243.1  <2e-16 ***
## MonthJul           6.39e+00   2.63e-02   242.6  <2e-16 ***
## MonthJun           6.38e+00   2.63e-02   242.4  <2e-16 ***
## MonthMar           6.42e+00   2.63e-02   244.2  <2e-16 ***
## MonthMay           6.43e+00   2.63e-02   244.3  <2e-16 ***
## MonthNov           6.34e+00   2.63e-02   240.6  <2e-16 ***
## MonthOct           6.37e+00   2.63e-02   241.8  <2e-16 ***
## MonthSep           6.34e+00   2.63e-02   240.6  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0263 on 202 degrees of freedom
## Multiple R-squared:  1, Adjusted R-squared:  1
## F-statistic: 1.42e+06 on 14 and 202 DF, p-value: <2e-16
```

We see that the model no longer has an intercept term and has 12 significant month factors. This is exactly what we wanted to see.

Let's make another time series plot of the California dairy production data to see how well the seasonal model is working. I have added the following code in the Azure Machine Learning [Execute R Script](#) to create the model and make a plot.

Copy

```
milk.lm2 <- lm(Milk.Prod ~ Time + I(Month.Count^3) + Month - 1, data =
cadairytrain)
```

```
plot(cadairytrain$Time, cadairytrain$Milk.Prod, xlab = "Time", ylab = "Log CA
Milk Production 1000s lb", type = "l")
lines(cadairytrain$Time, predict(milk.lm2, cadairytrain), lty = 2, col = 2)
```

Running this code in Azure Machine Learning produces the plot shown in Figure 24.

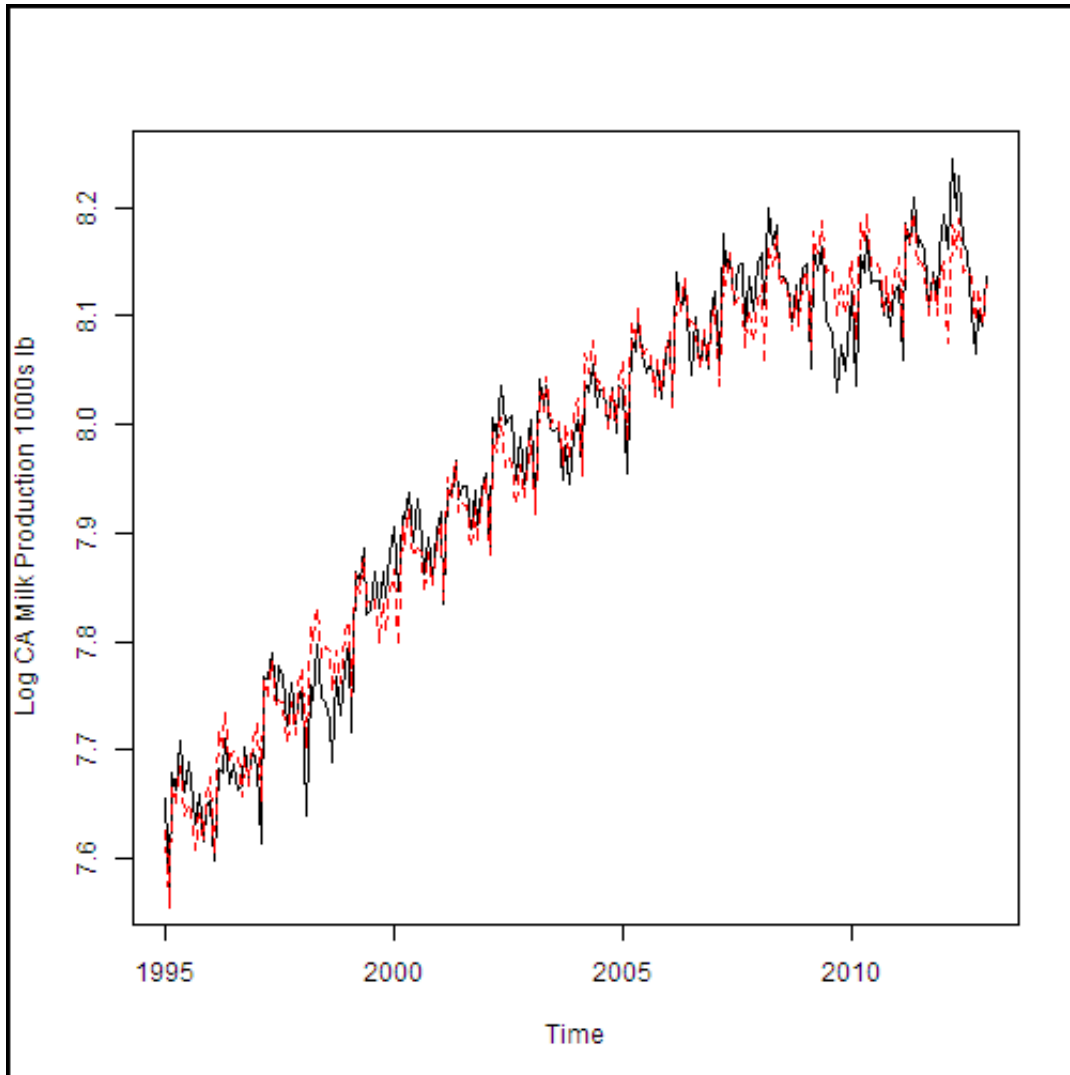


Figure 24. California milk production with model including seasonal effects.

The fit to the data shown in Figure 24 is rather encouraging. Both the trend and the seasonal effect (monthly variation) look reasonable.

As another check on our model, let's have a look at the residuals. The following code computes the predicted values from our two models, computes the residuals for the seasonal model, and then plots these residuals for the training data.

Copy

```
## Compute predictions from our models
predict1 <- predict(milk.lm, cadairydata)
predict2 <- predict(milk.lm2, cadairydata)

## Compute and plot the residuals
```

```
residuals <- cadata$Milk.Prod - predict2
plot(cadata$Time, residuals[1:216], xlab = "Time", ylab = "Residuals of
Seasonal Model")
```

The residual plot is shown in Figure 25.

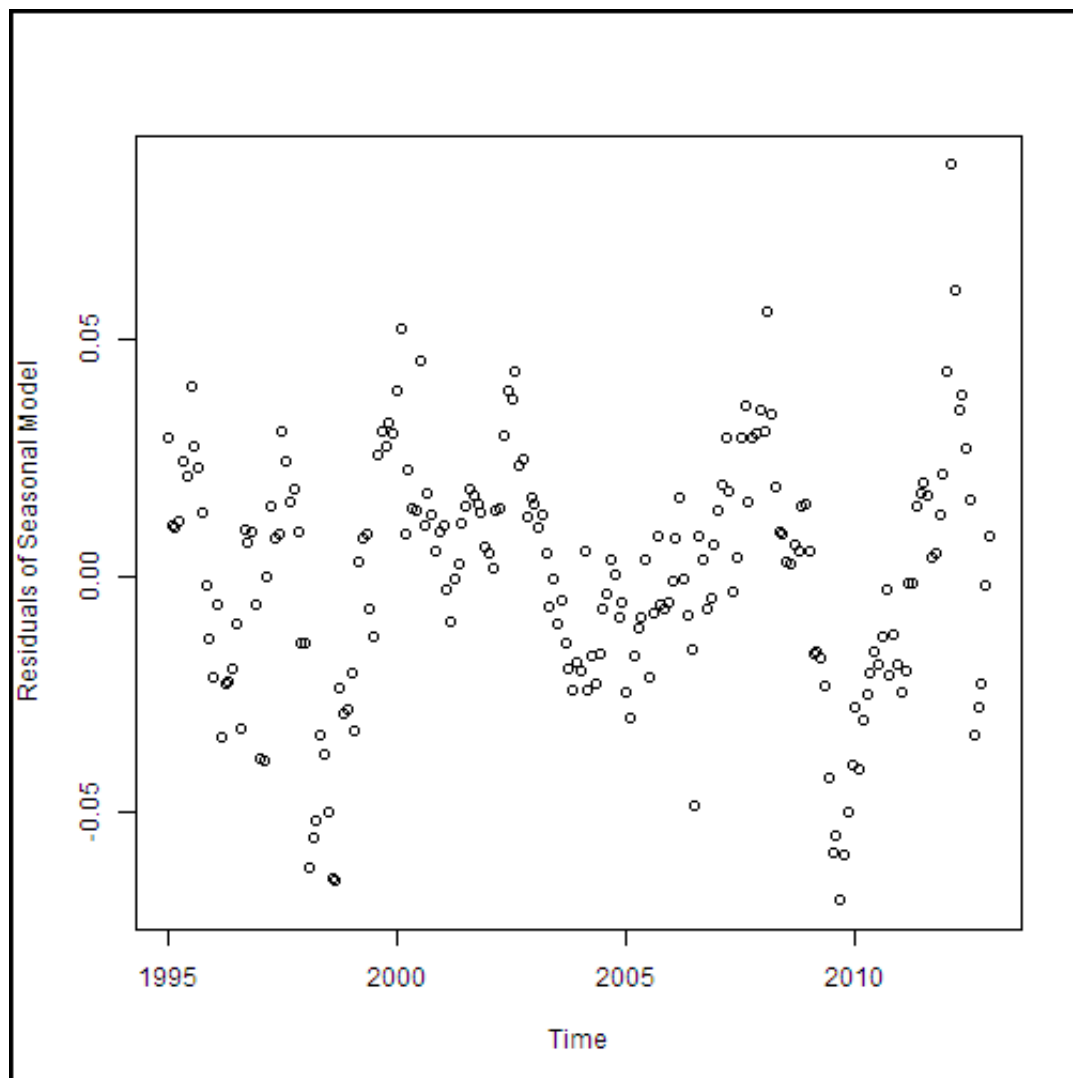


Figure 25. Residuals of the seasonal model for the training data.

These residuals look reasonable. There is no particular structure, except the effect of the 2008-2009 recession, which our model does not account for particularly well.

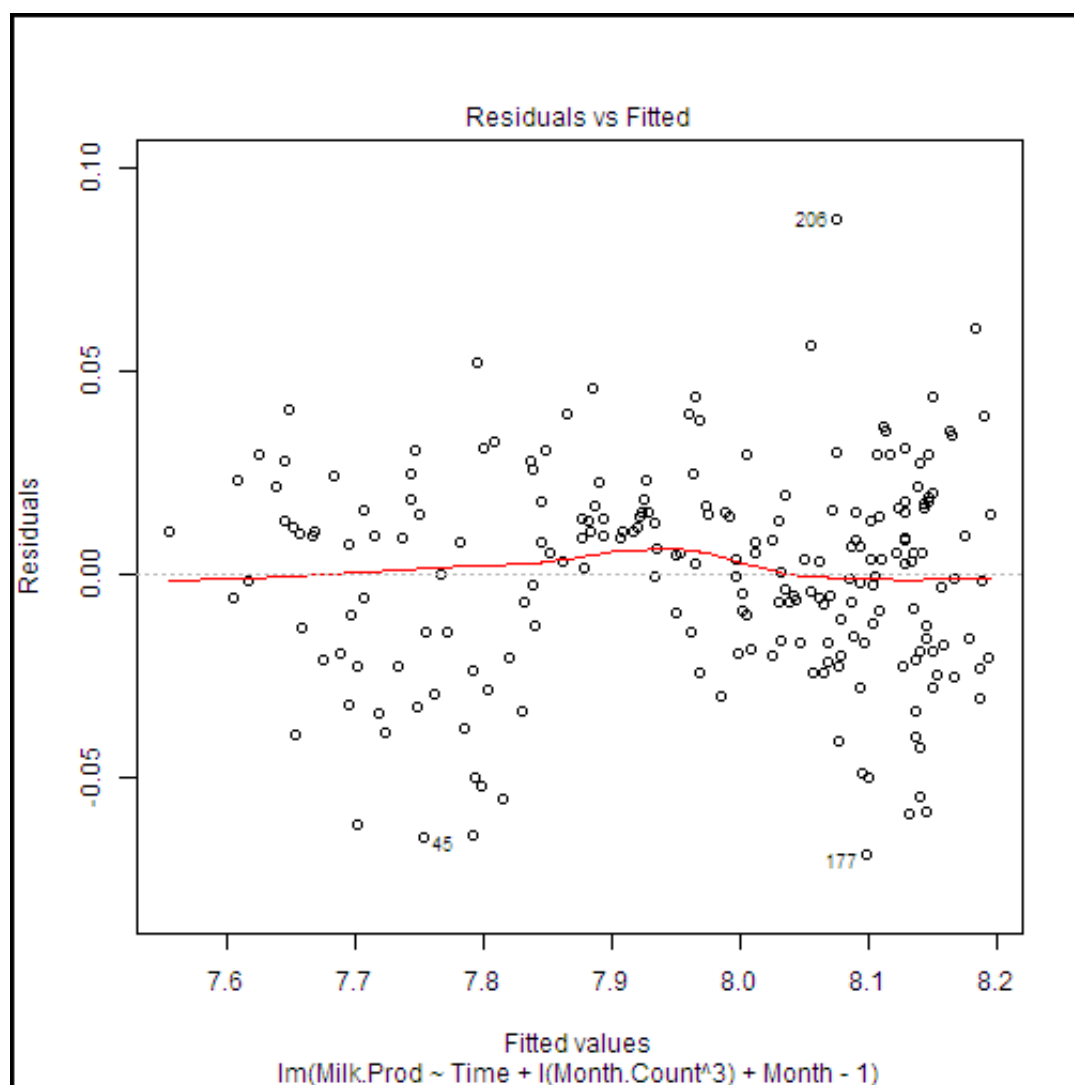
The plot shown in Figure 25 is useful for detecting any time-dependent patterns in the residuals. The explicit approach of computing and plotting the residuals I used places the residuals in time order on the plot. If, on the other hand, I had plotted `milk.lm$residuals`, the plot would not have been in time order.

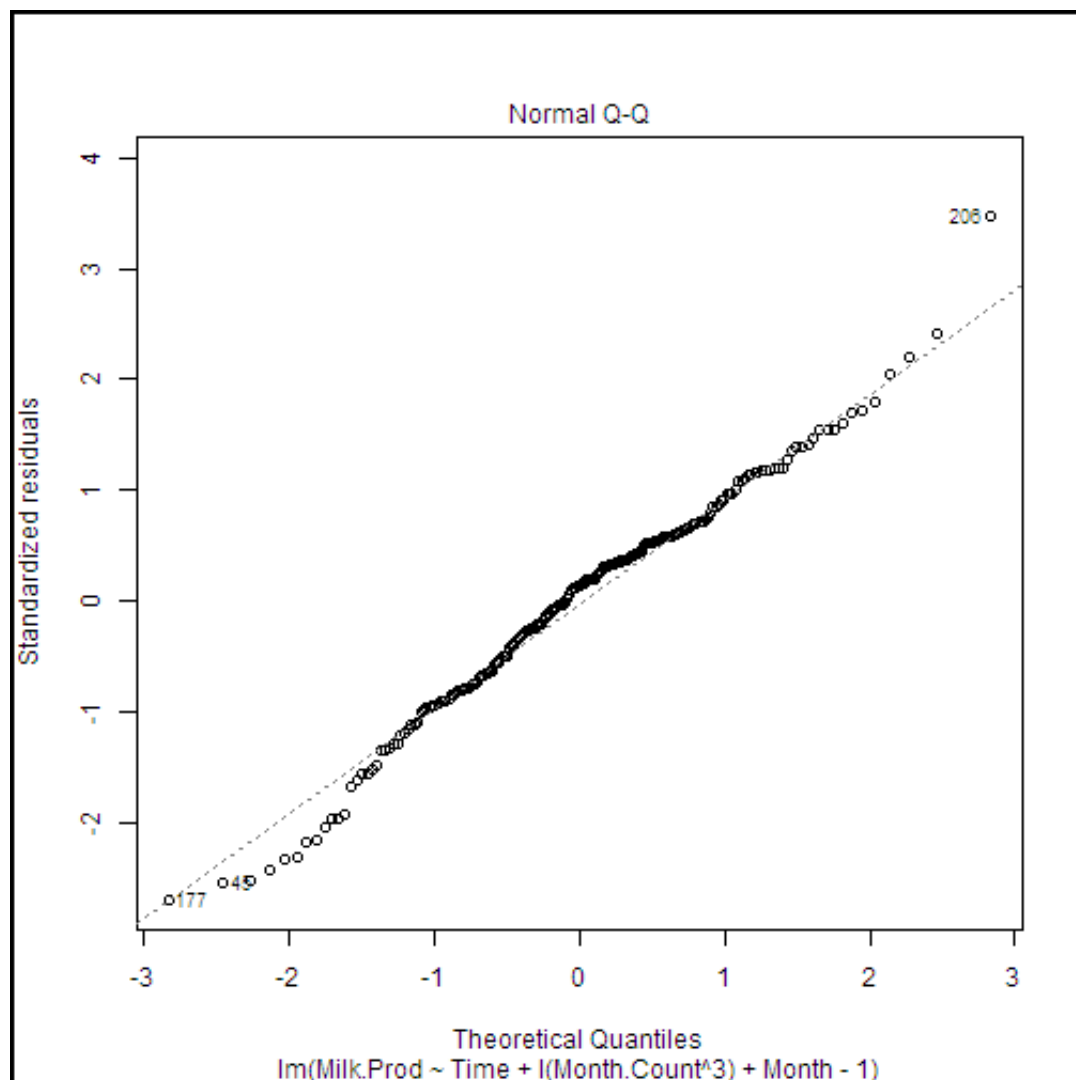
You can also use `plot.lm()` to produce a series of diagnostic plots.

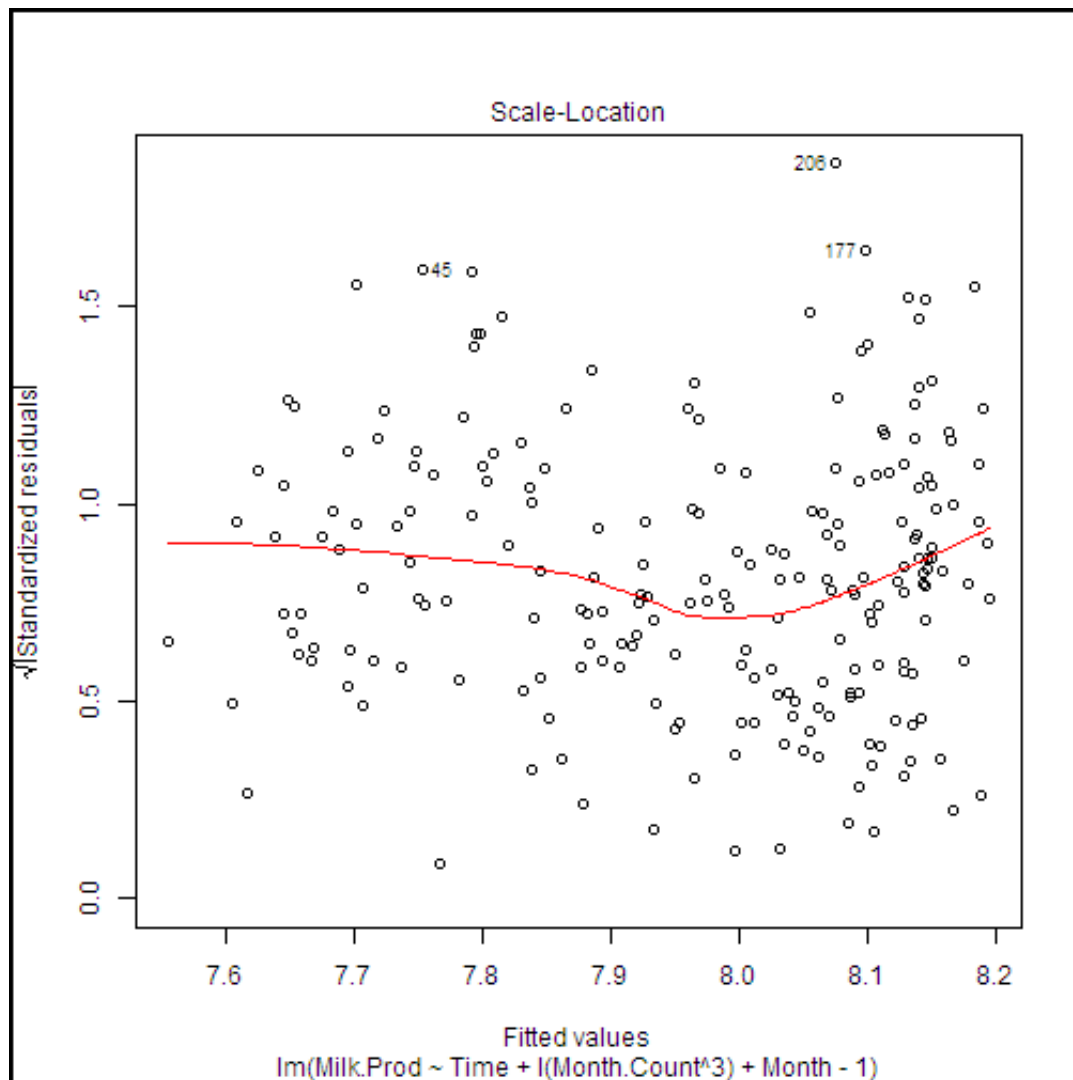
Copy

```
## Show the diagnostic plots for the model
plot(milk.lm2, ask = FALSE)
```

This code produces a series of diagnostic plots shown in Figure 26.







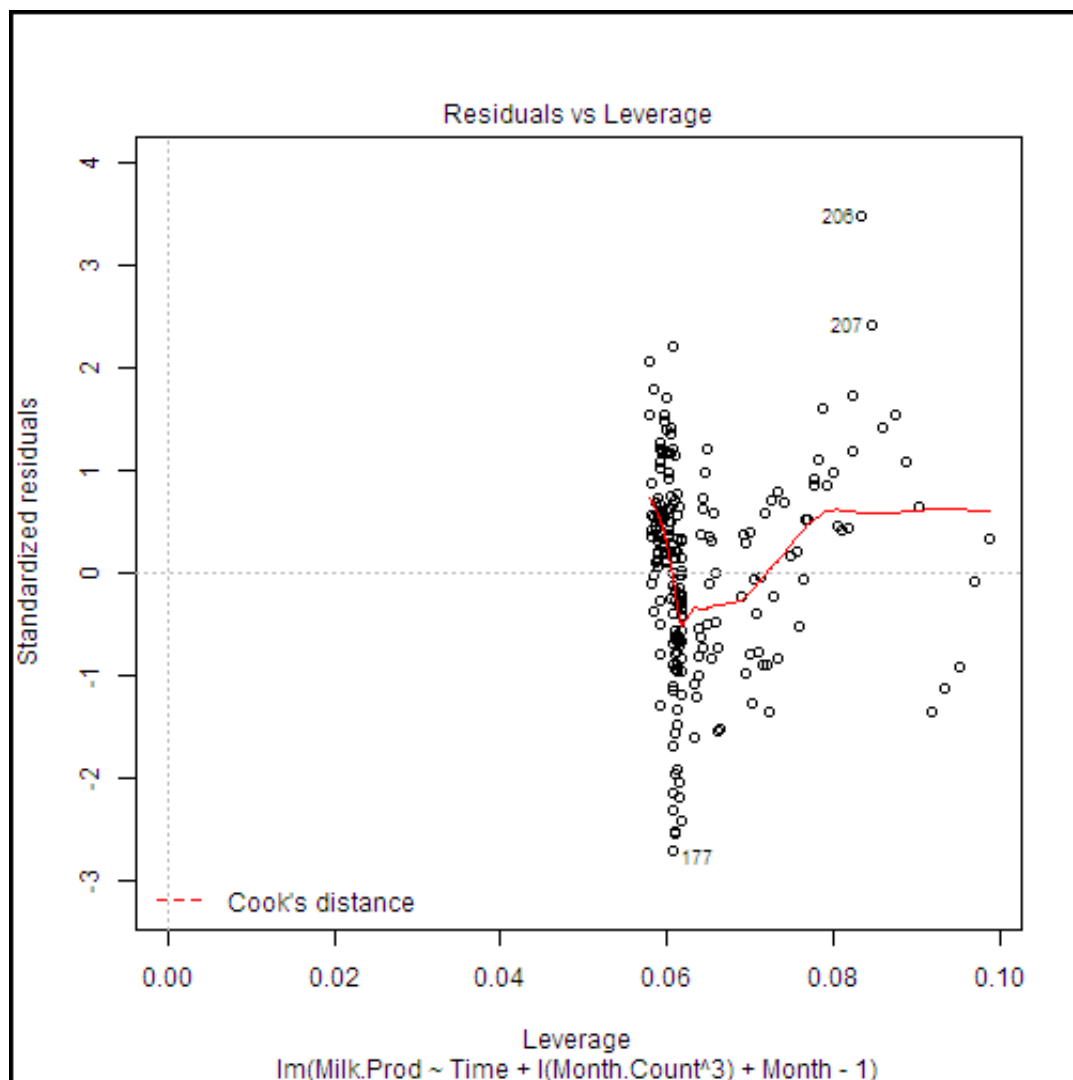


Figure 26. Diagnostic plots for the seasonal model.

There are a few highly influential points identified in these plots, but nothing to cause great concern. Further, we can see from the Normal Q-Q plot that the residuals are close to normally distributed, an important assumption for linear models.

Forecasting and model evaluation

There is just one more thing to do to complete our example. We need to compute forecasts and measure the error against the actual data. Our forecast will be for the 12 months of 2013. We can compute an error measure for this forecast to the actual data that is not part of our training dataset. Additionally, we can compare performance on the 18 years of training data to the 12 months of test data.

A number of metrics are used to measure the performance of time series models. In our case we will use the root mean square (RMS) error. The following function computes the RMS error between two series.

Copy

```
RMS.error <- function(series1, series2, is.log = TRUE, min.length = 2){
  ## Function to compute the RMS error or difference between two
  ## series or vectors
```



```
messages <- c("ERROR: Input arguments to function RMS.error of wrong type
encountered",
              "ERROR: Input vector to function RMS.error is too short",
              "ERROR: Input vectors to function RMS.error must be of same
length",
              "WARNING: Funtion rms.error has received invald input time
series.")

## Check the arguments
if(!is.numeric(series1) | !is.numeric(series2) | !is.logical(is.log) |
!is.numeric(min.length)) {
  warning(messages[1])
  return(NA)}

if(length(series1) < min.length) {
  warning(messages[2])
  return(NA)}

if((length(series1) != length(series2))) {
  warning(messages[3])
  return(NA)}

## If is.log is TRUE exponentiate the values, else just copy
if(is.log) {
  tryCatch( {
    temp1 <- exp(series1)
    temp2 <- exp(series2) },
    error = function(e){warning(messages[4]); NA}
  )
} else {
  temp1 <- series1
  temp2 <- series2
}

## Compute predictions from our models
predict1 <- predict(milk.lm, cadairydata)
predict2 <- predict(milk.lm2, cadairydata)

## Compute the RMS error in a dataframe
tryCatch( {
  sqrt(sum((temp1 - temp2)^2) / length(temp1))),
  error = function(e){warning(messages[4]); NA}
}
```

As with the `log.transform()` function we discussed in the "Value transformations" section, there is quite a lot of error checking and exception recovery code in this function. The principles employed are the same. The work is done in two places wrapped in `tryCatch()`. First, the time series are exponentiated, since we have been working with the logs of the values. Second, the actual RMS error is computed.

Equipped with a function to measure the RMS error, let's build and output a dataframe containing the RMS errors. We will include terms for the trend model alone and the complete model with seasonal factors. The following code does the job by using the two linear models we have constructed.

Copy

```
## Compute the RMS error in a dataframe
## Include the row names in the first column so they will
## appear in the output of the Execute R Script
RMS.df <- data.frame(
  rowNames = c("Trend Model", "Seasonal Model"),
  Traing = c(
    RMS.error(predict1[1:216], cadairydata$Milk.Prod[1:216]),
    RMS.error(predict2[1:216], cadairydata$Milk.Prod[1:216])),
  Forecast = c(
    RMS.error(predict1[217:228], cadairydata$Milk.Prod[217:228]),
    RMS.error(predict2[217:228], cadairydata$Milk.Prod[217:228]))
)
RMS.df

## The following line should be executed only when running in
## Azure Machine Learning Studio
maml.mapOutputPort('RMS.df')
```

Running this code produces the output shown in Figure 27 at the Result Dataset output port.

CA Dairy Analysis > Execute R Script > Result Dataset



rows	columns			
2	3			
		rowNames	Traing	Forecast
view as				
				
		Trend Model	122.238008	174.088492
		Seasonal Model	75.115958	94.725325

Figure 27. Comparison of RMS errors for the models.

From these results, we see that adding the seasonal factors to the model reduces the RMS error significantly. Not too surprisingly, the RMS error for the training data is a bit less than for the forecast.

APPENDIX A: Guide to RStudio

RStudio is quite well documented, so in this appendix I will provide some links to the key sections of the RStudio documentation to get you started.

1. Creating projects

You can organize and manage your R code into projects by using RStudio. The documentation that uses projects can be found at <https://support.rstudio.com/hc/articles/200526207-Using-Projects>.

I recommend that you follow these directions and create a project for the R code examples in this document.

2. Editing and executing R code

RStudio provides an integrated environment for editing and executing R code. Documentation can be found at <https://support.rstudio.com/hc/articles/200484448-Editing-and-Executing-Code>.

3. Debugging

RStudio includes powerful debugging capabilities. Documentation for these features is at <https://support.rstudio.com/hc/articles/200713843-Debugging-with-RStudio>.

The breakpoint troubleshooting features are documented at <https://support.rstudio.com/hc/articles/200534337-Breakpoint-Troubleshooting>.

APPENDIX B: Further reading

This R programming tutorial covers the basics of what you need to use the R language with Azure Machine Learning Studio. If you are not familiar with R, two introductions are available on CRAN:

- R for Beginners by Emmanuel Paradis is a good place to start at http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf.
- An Introduction to R by W. N. Venables et. al. goes into a bit more depth, at <http://cran.r-project.org/doc/manuals/R-intro.html>.

There are many books on R that can help you get started. Here are a few I find useful:

- The Art of R Programming: A Tour of Statistical Software Design by Norman Matloff is an excellent introduction to programming in R.

- R Cookbook by Paul Teetor provides a problem and solution approach to using R.
- R in Action by Robert Kabacoff is another useful introductory book. The companion Quick R website is a useful resource at <http://www.statmethods.net/>.
- R Inferno by Patrick Burns is a surprisingly humorous book that deals with a number of tricky and difficult topics that can be encountered when programming in R. The book is available for free at <http://www.burns-stat.com/documents/books/the-r-inferno/>.
- If you want a deep dive into advanced topics in R, have a look at the book Advanced R by Hadley Wickham. The online version of this book is available for free at <http://adv-r.had.co.nz/>.

A catalogue of R time series packages can be found in the CRAN Task View for time series analysis: <http://cran.r-project.org/web/views/TimeSeries.html>. For information on specific time series object packages, you should refer to the documentation for that package.

The book Introductory Time Series with R by Paul Cowpertwait and Andrew Metcalfe provides an introduction to using R for time series analysis. Many more theoretical texts provide R examples.

Some great internet resources:

- DataCamp: DataCamp teaches R in the comfort of your browser with video lessons and coding exercises. There are interactive tutorials on the latest R techniques and packages. Take the free interactive R tutorial at <https://www.datacamp.com/courses/introduction-to-r>
- A quick R tutorial by Kelly Black from Clarkson University <http://www.cyclismo.org/tutorial/R/>
- 60+ R resources listed at <http://www.computerworld.com/article/2497464/business-intelligence-60-r-resources-to-improve-your-data-skills.html>

In this article: